

Oracle9i Application Server

Web Services Developer's Guide

Release 2 (9.0.3)

August 2002

Part No. B10004-01

ORACLE®

Oracle9i Application Server Web Services Developer's Guide, Release 2 (9.0.3)

Part No. B10004-01

Copyright © 2002 Oracle Corporation. All rights reserved.

Primary Authors: Thomas Van Raalte and Rodney Ward

Contributors: Marco Carrer, Anirban Chatterjee, Daxin Cheng, David Clay, Tony D'Silva, Neil Evans, Bert Feldman, Kathryn Gruenefeldt, Steven Harris, Anish Karmarkar, Prabha Krishna, Sunil Kunisetty, Wai-Kwong (Sam) Lee, Gary Moyer, Steve Muench, Giuseppe Panciera, Wei Qian, Eric Rajkovic, Venkata Ravipati, Susan Shepard, Alok Srivastava, Zhe (Alan) Wu, Joyce Yang, Chen Zhou

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

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

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle is a registered trademark, and Oracle*MetaLink*, Oracle Store, Oracle9i, Oracle9iAS Discoverer, and PL/SQL are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

Contents

Send Us Your Comments	xiii
Preface.....	xv
1 Web Services Overview	
What Are Web Services?.....	1-2
Understanding Web Services.....	1-2
Benefits of Web Services.....	1-3
About the Web Services e-Business Transformation.....	1-3
About Business Transformation with Web Services.....	1-4
About Technology Transformation with Web Services.....	1-4
Overview of Web Services Standards.....	1-5
Simple Object Access Protocol (SOAP)	1-6
Web Services Description Language (WSDL)	1-6
Universal Description, Discovery, and Integration (UDDI).....	1-7
Scenario: Web Services with a Currency Converter Application.....	1-7
Understanding the Publisher’s Role	1-8
Understanding the Caller’s Role	1-8
2 Oracle9iAS Web Services	
Oracle9iAS OC4J (J2EE) and Oracle SOAP Based Web Services	2-2
Oracle9iAS Web Services Features.....	2-2
Developing End-to-End Web Services	2-3
Deploying and Managing Web Services	2-4

Using Oracle9i JDeveloper with Web Services.....	2-4
Securing Web Services	2-5
Aggregating Web Services	2-5
Oracle9iAS Web Services Architecture	2-6
About Servlet Entry Points for Web Services	2-8
What Are the Packaging and Deployment Options for Web Services	2-10
About Server Skeleton Code Generation for Web Services.....	2-10
Understanding WSDL and Client Proxy Stubs for Web Services	2-11
Overview of a WSDL Based Web Service Client.....	2-12
Overview of a Client-Side Proxy Stubs Based Web Service Client	2-12
Web Services Home Page	2-13
About Universal Description, Discovery, and Integration Registry	2-14
Oracle Enterprise Manager Features to Register Web Services	2-15

3 Developing and Deploying Java Class Web Services

Using Oracle9iAS Web Services With Java Classes	3-2
Writing Java Class Based Web Services	3-2
Writing Stateless and Stateful Java Web Services	3-3
Building a Sample Java Class Implementation	3-3
Defining a Java Class Containing Methods for the Web Service.....	3-3
Defining an Interface for Explicit Method Exposure	3-5
Writing a WSDL File (Optional).....	3-6
Using Supported Data Types for Java Web Services.....	3-7
Preparing and Deploying Java Class Based Web Services	3-9
Creating a Configuration File to Assemble Java Class Web Services	3-9
Adding Web Service Top Level Tags	3-9
Adding Java Stateless Service Tags.....	3-10
Adding Java Stateful Service Tags	3-11
Adding WSDL and Client-Side Proxy Generation Tags.....	3-14
Running WebServicesAssembler To Prepare Java Class Web Services.....	3-14
Deploying Java Class Based Web Services.....	3-14
Serializing and Encoding Parameters and Results for Web Services	3-15

4 Developing and Deploying EJB Web Services

Using Oracle9iAS Web Services With Stateless Session EJBs	4-2
Writing Stateless Session EJB Web Services	4-2
Defining a Stateless Session Remote Interface	4-3
Defining a Stateless Session Home Interface	4-3
Defining a Stateless Session EJB Bean	4-4
Returning Results From EJB Web Services	4-5
Error Handling for EJB Web Services	4-5
Serializing and Encoding Parameters and Results for EJB Web Services	4-5
Using Supported Data Types for Stateless Session EJB Web Services.....	4-6
Writing a WSDL File for EJB Web Services (Optional)	4-7
Preparing and Deploying Stateless Session EJB Based Web Services	4-8
Creating a Configuration File to Assemble Stateless Session EJB Web Services.....	4-8
Adding Web Service Top Level Tags	4-8
Adding Stateless Session EJB Service Tags.....	4-9
Adding WSDL and Client-Side Proxy Generation Tags	4-11
Running WebServicesAssembler To Prepare Stateless Session EJB Web Services.....	4-11
Deploying Web Services Implemented as EJBs	4-11

5 Developing and Deploying Stored Procedure Web Services

Using Oracle9iAS Web Services with Stored Procedures	5-2
Writing Stored Procedure Web Services	5-2
Preparing Stored Procedure Web Services	5-3
Creating a Configuration File to Assemble Stored Procedure Web Services	5-4
Adding Web Service Top Level Tags	5-4
Adding Stateless Stored Procedure Java Service Tags.....	5-5
Adding WSDL and Client-Side Proxy Generation Tags	5-10
Running WebServicesAssembler With Stored Procedure Web Services	5-10
Setting Up Datasources in Oracle9iAS Web Services (OC4J).....	5-11
Deploying Stored Procedure Web Services	5-12
Limitations for Stored Procedures Running as Web Services	5-12
Supported Stored Procedure Features for Web Services.....	5-12
Unsupported Stored Procedure Features for Web Services.....	5-13
Database Server Release Limitation for Boolean Use in Oracle PLSQL Web Services	5-14

6 Developing and Deploying Document Style Web Services

Using Document Style Web Services	6-2
Writing Document Style Web Services	6-3
Supported Method Signatures for Document Style Web Services	6-3
Passing Null Values for Document Style Web Services.....	6-3
Arrays of Elements	6-4
Writing Stateless and Stateful Document Style Web Services	6-4
Writing Classes and Interfaces for Document Style Web Services.....	6-5
Defining Methods in a Document Style Web Service	6-5
Defining an Interface for Explicit Method Exposure	6-7
Handling Messages for Document Style Web Services	6-9
Preparing Document Style Web Services	6-9
Creating a Configuration File to Assemble Document Style Web Services.....	6-10
Adding Web Service Top Level Tags	6-10
Adding Java Service Tags with Document Message Style Specified	6-11
Adding WSDL and Client-Side Proxy Generation Tags.....	6-13
Running WebServicesAssembler With Document Style Web Services.....	6-16
Deploying Document Style Web Services	6-16

7 Developing and Deploying JMS Web Services

JMS Web Services Overview	7-2
Using JMS Web Services	7-2
JMS Web Services Backend Message Processing	7-3
Using an MDB for Message Processing.....	7-3
Using a JMS Client for Message Processing	7-4
Writing JMS Web Services and Handling Messages	7-6
Using an MDB for Backend Message Processing.....	7-6
Developing the MDB that Processes Incoming Messages	7-7
Developing the MDB that Generates Outgoing Messages	7-8
Compiling and Preparing the MDB EJB.jar File.....	7-8
Assembling the JMS Web Service With the MDB.....	7-8
Defining the Server-Side Resource References	7-9
Using a JMS Standalone Program for Backend Message Processing.....	7-9
Message Processing and Reply Messages	7-10

Preparing and Configuring JMS Web Services	7-11
Creating a Configuration File to Assemble JMS Web Services	7-12
Adding Web Service Top Level Tags	7-12
Adding JMS Doc Service Tags	7-13
Adding WSDL and Client-Side Proxy Generation Tags	7-16
Running WebServicesAssembler With JMS Web Services	7-17
Deploying JMS Web Services	7-17
Limitations for JMS Web Services	7-18

8 Building Clients that Use Web Services

Locating Web Services	8-2
Getting WSDL Files and Client-Side Proxy Jars for Web Services	8-2
Using the Web Service Home Page to Save WSDL and Client Side Proxies	8-3
Limitations for Web Service Test Pages	8-4
Getting Web Service WSDL and Client-Side Proxies Directly	8-5
Getting WSDL Service Descriptions	8-5
Getting Client-Side Proxy Jar and Client-Side Proxy Source Jar	8-5
Getting Client-Side Proxy Jar and Client-Side Proxy Source by Package	8-6
Generating Client-Side Proxies With WebServicesAssembler	8-8
Working with Client-Side Proxy Jar to Use Web Services	8-9
Using Web Services Security Features	8-11
Working with WSDL Files and JDeveloper to Use Web Services	8-14

9 Web Services Assembly Tool

Running the Web Services Assembly Tool	9-2
Web Services Assembly Tool Configuration File Sample	9-2
Web Services Assembly Tool Configuration File Sample Output	9-4
Generating WSDL Files and Client Side Proxies	9-5
Generating and Assembling WSDL Files	9-5
Manually Producing a WSDL File	9-7
Generating Client-Side Proxies with WSDL	9-8
Web Services Assembly Tool Configuration File Specification	9-9
Web Services Assembly Tool Limitations	9-11

10 Discovering and Publishing Web Services

UDDI Registration	10-2
UDDI Registry	10-2
Oracle UDDI Enterprise Web Services Registry	10-4
Web Services Discovery	10-5
Using Tools.....	10-5
Using the Inquiry API.....	10-5
Web Services Publishing	10-8
Using Oracle Enterprise Manager.....	10-8
Oracle UDDI Registry.....	10-9
Using the Oracle Enterprise Manager Deploy Applications Wizard.....	10-9
Using the Oracle Enterprise Manager Web Services Details Window.....	10-11
Updating Published Web Services in the UDDI Registry.....	10-12
Using the Publishing API.....	10-14
UDDI Registry Administration.....	10-19
User Management.....	10-19
Performance Monitoring and Tuning.....	10-20
Data Backup and Restore Operations.....	10-20
Using the Command-Line Tool uddiadmin.jar.....	10-20
Server Configuration.....	10-20
User Account Management.....	10-21
Administrative Entity Management.....	10-21
Import Operation.....	10-22
Database Configuration.....	10-22
Built-in Validated Category Management.....	10-23
Transport Security.....	10-25
Additional Information.....	10-26
Server Configuration Parameters Reference Information.....	10-26
Database Character Set and Built-in ISO-3166 Classification	10-29
Recommended Configuration for a Production Environment	10-30

11 Consuming Web Services in J2EE Applications

Consuming XML or HTML Streams in J2EE Applications	11-2
Web Service HTML/XML Stream Processing Wizard	11-2
Sample Use Scenarios.....	11-3
Handling an XML or HTML Stream Accessed Through a Static URL.....	11-3
Handling an XML or HTML Stream Accessed Through a Form	11-13
Advanced Section -- Editing Changes You Can Make to Generated Files.....	11-31
Editing the Generated XSLT Stylesheet	11-32
Modifying Environment Options in the Generated ejb-jar.xml File.....	11-32
Consuming SOAP-Based Web Services Using WSDL	11-33
Advanced Configuration.....	11-35
Known Limitations of the wsdl2ejb Utility	11-40
Running the Demonstration	11-41
RPC and Document Style with Simple Types Example	11-42
Round 2 Interop Services: Base Test Suite Example.....	11-45
Dynamic Invocation of Web Services	11-52
Dynamic Invocation API	11-52
WebServiceProxy Client.....	11-56
Known Limitations.....	11-60

A Using Oracle SOAP

Understanding Oracle9iAS SOAP	A-2
Apache SOAP Documentation	A-3
Configuring the SOAP Request Handler Servlet	A-3
Using Oracle9iAS SOAP Management Utilities and Scripts	A-6
Managing Providers.....	A-6
Using the Service Manager to Deploy and Undeploy Java Services	A-7
Generating Client Proxies from WSDL Documents	A-8
Generating WSDL Documents from Java Service Implementations	A-9
Deploying Oracle9iAS SOAP Services	A-10
Creating Deployment Descriptors	A-10
Installing a SOAP Web Service in OC4J	A-12
Disabling an Installed SOAP Web Service.....	A-12
Installing a SOAP Web Service in an OC4J Cluster.....	A-12

Using Oracle9iAS SOAP Handlers	A-13
Request Handlers.....	A-13
Response Handlers	A-13
Error Handlers	A-13
Configuring Handlers	A-13
Using Oracle9iAS SOAP Audit Logging	A-14
Audit Logging Information.....	A-15
Audit Logging Output.....	A-15
Auditable Events.....	A-15
Audit Logging Filters.....	A-16
Configuring the Audit Logger	A-18
Using Oracle9iAS SOAP Pluggable Configuration Managers	A-20
Working With Oracle9iAS SOAP Transport Security	A-21
Apache Listener and Servlet Engine Configuration for SSL	A-25
Using JSSE with Oracle9iAS SOAP Client	A-25
Using Oracle9iAS SOAP Sample Services	A-28
The Xmethods Sample	A-28
The AddressBook Sample.....	A-28
The StockQuote Sample	A-28
The Company Sample	A-29
The Provider Sample.....	A-29
The AddressBook2 Sample.....	A-29
The Messaging Sample	A-29
The Mime Sample	A-29
Using the Oracle9iAS SOAP EJB Provider	A-29
Stateless Session EJB Provider	A-29
Stateful Session EJB Provider in Apache SOAP	A-30
Stateful Session EJB Provider in Oracle9iAS SOAP.....	A-30
Entity EJB Provider in Oracle9iAS SOAP.....	A-30
Deployment and Use of the Oracle9iAS SOAP EJB Provider	A-31
Current Known EJB Provider Limitations	A-32
Using PL/SQL Stored Procedures With the SP Provider	A-32
SP Provider Supported Functionality	A-32
SP Provider Unsupported Functionality.....	A-33
SP Provider Supported Simple PL/SQL Types.....	A-33

Using Object Types.....	A-34
Deploying a Stored Procedure Provider	A-34
Translating PL/SQL Stored Procedures into Java.....	A-34
Deploying a Stored Procedure Service	A-35
Invoking a SOAP Service that is a Stored Procedure	A-36
SOAP Troubleshooting and Limitations	A-36
Tunneling Using the TcpTunnelGui Command	A-37
Setting Configuration Options for Debugging.....	A-37
Using DMS to Display Runtime Information.....	A-38
SOAP Limitations for Java Type Precedence with Overloaded Methods	A-38
Oracle9iAS SOAP Differences From Apache SOAP	A-39
Service Installation Differences	A-39
Optional Provider Enhancements	A-39
Oracle Transport libraries.....	A-40
Modifications to Apache EJB Provider.....	A-40
Stored Procedure Provider	A-40
Utility Enhancements.....	A-40
Modifications to Sample Code.....	A-40
Handling the mustUnderstand Attribute in the SOAP Header	A-40
Setting the mustUnderstand Check.....	A-41
How the mustUnderstand Check Works.....	A-41
Differences Between Apache SOAP and Oracle SOAP for mustUnderstand	A-41

Glossary

Index

Send Us Your Comments

Oracle9i Application Server Web Services Developer's Guide, Release 2 (9.0.3)

Part No. B10004-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: iasdocs_us@oracle.com
- FAX: 650-506-7407 Attn: Oracle9i Application Server Documentation Manager

- Postal service:

Oracle Corporation
Oracle9i Application Server Web Services Developer's Guide
500 Oracle Parkway M/S 2op3
Redwood Shores, CA 94065
USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This guide describes Oracle*9i*AS Web Services.

This preface contains these topics:

- [Intended Audience](#)
- [Documentation Accessibility](#)
- [Organization](#)
- [Related Documentation](#)
- [Conventions](#)

Intended Audience

Oracle9i Application Server Web Services Developer's Guide is intended for application programmers, system administrators, and other users who perform the following tasks:

- Configure software installed on the Oracle9i Application Server
- Create programs that implement Web Services
- Create Java programs that run as Web Services clients

To use this document, you need a working knowledge of Java programming language fundamentals.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation 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 additional information, visit the Oracle Accessibility Program Web site at

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, 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, JAWS 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 Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Organization

This document contains:

Chapter 1, "Web Services Overview"

This chapter provides an overview of Web Services.

Chapter 2, "Oracle9iAS Web Services"

This chapter describes the Oracle9iAS Web Services features, architecture, and implementation.

Chapter 3, "Developing and Deploying Java Class Web Services"

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services that are implemented as Java classes.

Chapter 4, "Developing and Deploying EJB Web Services"

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services that are implemented as stateless session Enterprise Java Beans (EJBs).

Chapter 5, "Developing and Deploying Stored Procedure Web Services"

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services implemented as stateless PL/SQL Stored Procedures or Functions.

Chapter 6, "Developing and Deploying Document Style Web Services"

This chapter describes the procedures you use to write and deploy Document Style Oracle9iAS Web Services implemented as Java classes.

Chapter 7, "Developing and Deploying JMS Web Services"

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services that expose JMS destinations as Web Services.

Chapter 8, "Building Clients that Use Web Services"

This chapter describes the steps required to build a client application that uses Oracle9iAS Web Services.

Chapter 9, "Web Services Assembly Tool"

This chapter describes the Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, that assists in assembling Oracle9iAS Web Services. The

Web Services assembly tool takes a configuration file which describes the location of the Java classes or J2EE/EJB Jar files and produces a J2EE EAR file that can be deployed under Oracle9iAS Web Services.

Chapter 10, "Discovering and Publishing Web Services"

This chapter provides a description of the Universal Discovery Description and Integration (UDDI)-compliant Web services registry in which business Web Service providers in an enterprise environment can publish and describe their Web Services.

Chapter 11, "Consuming Web Services in J2EE Applications"

This chapter describes how to consume Web Services in J2EE applications.

Appendix A, "Using Oracle SOAP"

This appendix describes Oracle SOAP and covers the differences between Apache SOAP and Oracle SOAP.

Glossary

The glossary contains the Web Services glossary terms and descriptions.

Related Documentation

For more information, see these Oracle resources:

- *Overview Guide* in the Oracle9i Application Server Documentation Library.
- *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library.

In North America, printed documentation is available for sale in the Oracle Store at

<http://oraclestore.oracle.com/>

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

<http://www.oraclebookshop.com/>

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

<http://otn.oracle.com/membership/>

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

<http://otn.oracle.com/docs/index.htm>

For additional information, see:

- <http://www.w3.org/TR/SOAP/> for information on Simple Object Access Protocol (SOAP) 1.1 specification
- <http://www.uddi.org> for information on Universal Description, Discovery and Integration specifications.
- <http://www.w3.org/TR/wsdl> for information on the Web Services Description Language (WSDL) format.
- *Java 2 Platform Enterprise Edition Specification, v1.3* at <http://java.sun.com/j2ee/docs.html>

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- [Conventions in Text](#)
- [Conventions in Code Examples](#)
- [Conventions for Microsoft Windows Operating Systems](#)

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an index-organized table .
<i>Italics</i>	Italic typeface indicates book titles or emphasis.	<i>Oracle9i Database Concepts</i> Ensure that the recovery catalog and target database do <i>not</i> reside on the same disk.
UPPERCASE monospace (fixed-width) font	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the datafiles and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JReplUtil class implements these methods.
<i>lowercase italic monospace (fixed-width) font</i>	Lowercase italic monospace font represents placeholders or variables.	You can specify the <i>parallel_clause</i> . Run <i>Uold_release</i> .SQL where <i>old_release</i> refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[]	Brackets enclose one or more optional items. Do not enter the brackets.	DECIMAL (<i>digits</i> [, <i>precision</i>])
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	{ENABLE DISABLE}
	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE DISABLE} [COMPRESS NOCOMPRESS]
...	Horizontal ellipsis points indicate either: <ul style="list-style-type: none"> That we have omitted parts of the code that are not directly related to the example That you can repeat a portion of the code 	CREATE TABLE ... AS <i>subquery</i> ; SELECT <i>col1</i> , <i>col2</i> , ... , <i>coln</i> FROM employees;
. . . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
<i>Italics</i>	Italicized text indicates placeholders or variables for which you must supply particular values.	CONNECT SYSTEM/ <i>system_password</i> DB_NAME = <i>database_name</i>

Convention	Meaning	Example
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	<pre>SELECT last_name, employee_id FROM employees;</pre> <pre>SELECT * FROM USER_TABLES;</pre> <pre>DROP TABLE hr.employees;</pre>
lowercase	<p>Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.</p> <p>Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.</p>	<pre>SELECT last_name, employee_id FROM employees;</pre> <pre>sqlplus hr/hr</pre> <pre>CREATE USER mjones IDENTIFIED BY ty3MUJ9;</pre>

Conventions for Microsoft Windows Operating Systems

The following table describes conventions for Microsoft Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Oracle Database Configuration Assistant, choose Start > Programs > Oracle - <i>HOME_NAME</i> > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	<pre>c:\winnt\"system32 is the same as</pre> <pre>C:\WINNT\SYSTEM32</pre>

Convention	Meaning	Example
C:\>	<p>Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the <i>command prompt</i> in this manual.</p>	C:\oracle\oradata>
	<p>The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.</p>	<pre>C:\>exp scott/tiger TABLES=emp QUERY=\ "WHERE job='SALESMAN' and sal<1600\" C:\>imp SYSTEM/password FROMUSER=scott TABLES=(emp, dept)</pre>
<i>HOME_NAME</i>	<p>Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.</p>	C:\> net start OracleHOME_ NAME_TNSListener

Convention	Meaning	Example
<i>ORACLE_HOME</i> and <i>ORACLE_BASE</i>	<p>In releases prior to Oracle8i release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level <i>ORACLE_HOME</i> directory that by default used one of the following names:</p> <ul style="list-style-type: none"> ■ C:\orant for Windows NT ■ C:\orawin95 for Windows 95 ■ C:\orawin98 for Windows 98 <p>This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level <i>ORACLE_HOME</i> directory. There is a top level directory called <i>ORACLE_BASE</i> that by default is C:\oracle. If you install Oracle9i release 1 (9.0.1) on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is C:\oracle\ora90. The Oracle home directory is located directly under <i>ORACLE_BASE</i>.</p> <p>All directory path examples in this guide follow OFA conventions.</p> <p>Refer to <i>Oracle9i Database Getting Starting for Windows</i> for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.</p>	Go to the <i>ORACLE_BASE\ORACLE_HOME\rdms\admin</i> directory.

Web Services Overview

This chapter provides an overview of Web Services. [Chapter 2, "Oracle9iAS Web Services"](#) describes the Oracle9iAS Web Services features, architecture, and implementation.

This chapter covers the following topics:

- [What Are Web Services?](#)
- [Overview of Web Services Standards](#)
- [Scenario: Web Services with a Currency Converter Application](#)

What Are Web Services?

Web Services consist of a set of messaging protocols, programming standards, and network registration and discovery facilities that expose business functions to authorized parties over the Internet from any web-connected device.

This section covers the following topics:

- [Understanding Web Services](#)
- [Benefits of Web Services](#)
- [About the Web Services e-Business Transformation](#)

Understanding Web Services

A **Web Service** is a discrete business process that does the following:

- Exposes and describes itself – A Web Service defines its functionality and attributes so that other applications can understand it. A Web Service makes this functionality available to other applications.
- Allows other services to locate it on the web – A Web Service can be registered in an electronic *Yellow Pages*, so that applications can easily 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.
- Returns a response – When a Web Service is invoked, the results are passed back to the requesting application over the same Internet standard protocol that is used to invoke the service.

Web Services provide 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.

Benefits of Web Services

The benefits for enterprises seeking to develop and use Web Services to streamline their business processes include the following:

- Support for open Internet standards. Oracle supports SOAP, WSDL, and UDDI as the primary standards to develop Web Services. Web Services developed with Oracle's products can inter-operate with those developed to Microsoft's .NET architecture.
- Simple and productive development facilities. Oracle provides developers with an easy-to-use and productive environment for developing Web Services using a programming model that is identical to that for J2EE applications.
- Mission critical deployment facilities. Oracle provides a mission-critical platform to deploy Web Services by unifying the Web Services and J2EE runtime infrastructure. Oracle9iAS Web Services provide optimizations to speed up Web Services responses, to scale Web Services on single CPUs or multiple CPUs, and to provide high availability through fault tolerant design and clustering.

See Also: ["Overview of Web Services Standards"](#) on page 1-5

About the Web Services e-Business Transformation

The move to transform businesses to e-Businesses has driven organizations around the world to begin to use the Internet to manage corporate business processes. Despite this transformation, business on the Internet still functions as a set of local nodes, or Web sites, with point-to-point communications between them. As more business moves online, the Internet should no longer be used in such a static manner, but rather should be used as a universal business network through which services can flow freely, and over which applications can interact and negotiate among themselves.

To enable this transformation, the Internet needs to support a standards-based infrastructure that enables companies and their enterprise applications to communicate with other companies and their applications more efficiently. These standards should allow discrete business processes to expose and describe themselves on the Internet, allow other services to locate them, to invoke them once they have been located, and to provide a predictable response.

Web Services drive this transformation by promising a fundamental change in the way businesses function and enterprise applications are developed and deployed.

This e-Business transformation is occurring in the following two areas:

- Business Transformation with Web Services
- Technology Transformation with Web Services

About Business Transformation with Web Services

Web Services enables the next-generation of e-business, a customer-centric, agile enterprise that does the following:

- Expands Markets - Offers business processes to existing and new customers as services over the Internet, opening new global channels and capturing new revenue opportunities.
- Improves Efficiencies - Streamlines business processes across the entire enterprise and with business partners, taking action in real-time with up-to-date information.
- Reaches Suppliers and Partners - Creates and maintains pre-defined, systematic, contractually negotiated relationships and dynamic, spot partnerships with business partners who are tightly linked within supply chains.

About Technology Transformation with Web Services

Web Services enables enterprise applications with the following technology transformations:

- Development and Deployment – Web Services can be developed and deployed quickly and productively.
- Locating Services – Web Services allow applications to be aggregated and discovered within Internet portals, enterprise portals, or service registries which serve as Internet *Yellow Pages*.
- Integrating Services – Web Services allow applications to locate and electronically communicate with other applications within an enterprise and outside the enterprise boundaries.
- Inter-Operating Services – Web Services allow applications to inter-operate with applications that are developed using different programming languages and following different component paradigms.

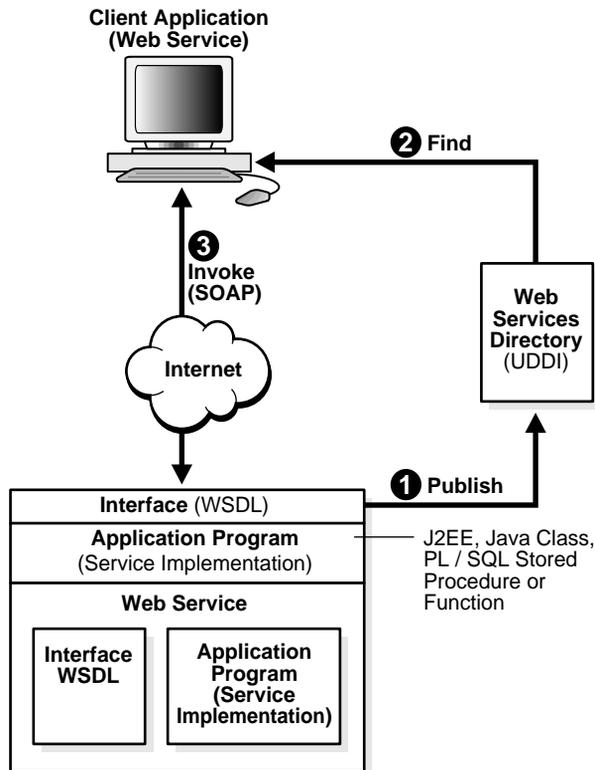
Overview of Web Services Standards

This section describes the Internet standards that comprise Web Services, including:

- [Simple Object Access Protocol \(SOAP\)](#)
- [Web Services Description Language \(WSDL\)](#)
- [Universal Description, Discovery, and Integration \(UDDI\)](#)

Figure 1–1 shows a conceptual architecture for Web Services using these standards.

Figure 1–1 Web Services Standards



Simple Object Access Protocol (SOAP)

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 request-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 need to exchange business or other types of documents where a message is sent but the sender may not expect or wait for an immediate response. Message-oriented information exchange is also called **Document style** exchange.

SOAP has the following features:

- Protocol independence
- Language independence
- Platform and operating system independence
- Support for SOAP XML messages incorporating attachments (using the multipart MIME structure)

See Also: <http://www.w3.org/TR/SOAP/> for information on Simple Object Access Protocol (SOAP) 1.1 specification

Web Services Description Language (WSDL)

The 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 the 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 proxies or program templates that access available services.

See Also: <http://www.w3.org/TR/wsdl> for information on the Web Services Description Language (WSDL) format.

Universal Description, Discovery, and Integration (UDDI)

The Universal Description, Discovery, and Integration (UDDI) specification is an online electronic registry that serves as electronic *Yellow Pages*, providing an information structure where various business entities register themselves and the services they offer through their WSDL definitions.

There are two types of UDDI registries, public UDDI registries that serve as aggregation points for a variety of businesses to publish their services, and private UDDI registries that serve a similar role within organizations.

See Also: <http://www.uddi.org> for information on Universal Description, Discovery and Integration specifications.

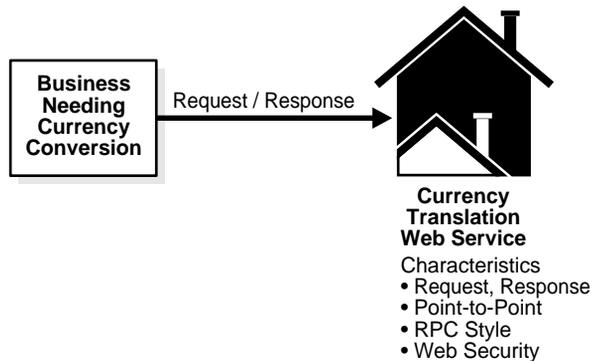
Scenario: Web Services with a Currency Converter Application

To understand how Web Services work, consider a currency translation service that provides businesses with up-to-the-instant currency conversion information. [Figure 1–2](#) shows the characteristics of such a Web Service.

A business has a financial management application which needs to check the conversion rate from one currency to another currency before completing a transaction. The financial management application sends a request to the currency conversion Web Service, it is processed, and a response is returned in real-time.

Using Web Services, there are two roles to consider, the role of the publisher that develops the currency conversion Web Service, and the role of the caller, the financial management application that uses the Web Service.

Figure 1–2 Currency Conversion Web Service



Understanding the Publisher's Role

The publisher develops the currency conversion Web Service; the publisher's role includes the following:

- Develop the application - The currency conversion company develops a currency conversion application. The currency conversion application is developed in Java/J2EE or any other programming language.
- Publish the application interfaces - The currency conversion application has a set of formalized interfaces. These interfaces are published in WSDL.
- Register with a Web Service registry - The currency conversion company registers itself as a business entity and publishes its WSDL interface in a UDDI registry.

Understanding the Caller's Role

The caller is the financial management application that uses the currency conversion Web Service; the caller's role includes the following:

- Search UDDI Registry - The Web Service caller, the invoking business' enterprise application, searches the UDDI registry and locates the currency conversion service.
- Invoke the Currency Conversion Service - The invoking business invokes the currency conversion service using the information stored in the UDDI registry. This includes the URL for the service to locate the currency conversion service,

and the WSDL interface to define the available methods in the currency conversion service.

- Communicate the Response - The caller and the Web Service communicate following a simple request/response pattern.

Oracle9iAS Web Services

This chapter describes the Oracle9iAS Web Services features, architecture, and implementation.

This chapter covers the following topics:

- [Oracle9iAS OC4J \(J2EE\) and Oracle SOAP Based Web Services](#)
- [Oracle9iAS Web Services Features](#)
- [Oracle9iAS Web Services Architecture](#)
- [Understanding WSDL and Client Proxy Stubs for Web Services](#)
- [Web Services Home Page](#)
- [About Universal Description, Discovery, and Integration Registry](#)

Oracle9iAS OC4J (J2EE) and Oracle SOAP Based Web Services

Oracle9i Application Server (Oracle9iAS) supports two different Web Services options, a J2EE based Web Services environment built into Oracle9iAS OC4J, and an Apache SOAP (Oracle SOAP) based Web Services environment.

The chapters in this manual describe the Oracle9iAS OC4J (J2EE) Web Services environment. This environment makes it easy to develop and deploy services using J2EE artifacts, and is moving the Oracle Web Services features toward the evolving Web Services standards included in the next release of J2EE (J2EE 1.4). The J2EE based Web Services environment includes many development and deployment features that are integrated with the advanced Oracle9iAS features.

[Appendix A, "Using Oracle SOAP"](#) describes the Oracle9iAS support for Apache SOAP (Oracle SOAP). Oracle9iAS includes support for Apache SOAP because this implementation was one of the earliest SOAP implementations and it supports existing Web Services applications.

Note: Oracle recommends using the Oracle9iAS OC4J (J2EE) Web Services environment for developing Web Services. The Apache SOAP (Oracle SOAP) implementation is currently in maintenance mode.

Oracle9iAS Web Services Features

Oracle9iAS provides advanced runtime features and comprehensive support for developing and deploying Web Services. The Oracle9iAS infrastructure includes support for the following:

- [Developing End-to-End Web Services](#)
- [Deploying and Managing Web Services](#)
- [Using Oracle9i JDeveloper with Web Services](#)
- [Securing Web Services](#)
- [Aggregating Web Services](#)

Developing End-to-End Web Services

Oracle9iAS Web Services provides comprehensive support for developing Web Services, including:

- **Development Environment** – Oracle9iAS Web Services allows application developers to implement Web Services using J2EE components. In addition, you can use Java Classes or PL/SQL Stored Procedures to implement Web Services. Web Services inherit all the runtime and lifecycle management elements of J2EE Applications.
- **Development Tools and Wizards** – Oracle9iAS Web Services Developers can use the same set of command line utilities to create, package, and deploy Web Services as other Oracle9iAS Containers for J2EE (OC4J) applications. In addition Oracle9iAS Web Services provides the Web Service HTML/XML Streams Processing Wizard that assists developers in creating an EJB whose methods access and process XML or HTML streams.
- **Automatically Generating WSDL** – Oracle9iAS Web Services can generate WSDL and client-side proxy stubs. This generation occurs when the Web Service is assembled using the WebServices Assembly tool or alternatively, for a deployed Web Service, the first time the WSDL or the client-side proxy stubs are requested (after the first request, the previously generated WSDL or client-side proxy stubs are sent when requested).
- **Registration, Publishing, and Discovery** – Oracle9iAS Web Services provides a standards-compliant UDDI registry where Web Services can be published and discovered. The Oracle UDDI registry supports both a private and public UDDI registry and can also synchronize information with other UDDI nodes.
- **Developer Simplicity** – Using Oracle9iAS Web Services, developers do not need to learn a completely new set of concepts – Web Services are developed, deployed and managed using the same programming concepts and tools as with J2EE Applications.
- **Business Logic Reuse** – Application developers can transparently publish their J2EE Applications to new Web Services clients with no change in the application itself. Their existing business logic developed in J2EE can be transparently accessed from existing J2EE/EJB clients.
- **Common Runtime Services** – Oracle9iAS has a common runtime and brokering environment for J2EE Applications and Web Services. As a result, Web Services transparently inherit various services available with the J2EE Container including Transaction Management, Messaging, Naming, Logging, and Security Services.

Deploying and Managing Web Services

Oracle Enterprise Manager and the Web Services Assembly Tool assist with deploying and managing Oracle9iAS Web Services. These tools provide the following support for Web Services:

- Packaging and Assembly - The Web Services Assembly Tool assists with assembling Web Services and producing a J2EE .ear file.
- Deployment – Oracle Enterprise Manager provides a comprehensive set of facilities to deploy Web Services to Oracle9iAS. Oracle Enterprise Manager provides a single, consistent *Deploy Applications* wizard for deploying Web Services to Oracle9iAS. It accepts a J2EE .ear file, and walks you through a set of steps to get information about the application to be deployed, and then deploys the application.
- Register Web Service - The *Deploy Applications* wizard is only available when deploying Web Services. This step provides access to facilities for registering Web Services in the UDDI Registry.
- Browse the UDDI Registry - Oracle's UDDI Registry provides the UDDI standards compliant pre-defined, hierarchical categorization schemes. Oracle Enterprise Manager can drill-down through these categories and look up specific Web Services registered in any category.
- Monitoring and Administration – Once deployed, Oracle Enterprise Manager provides facilities to de-install a Web Service and also to monitor Web Service performance, as measured by response-time and throughput, and status, as measured by up-time, CPU, and memory consumption. Oracle Enterprise Manager also provides facilities to identify and list all the Web Services deployed to a specific Oracle9iAS instance.

Using Oracle9i JDeveloper with Web Services

The Oracle9i JDeveloper IDE supports Oracle9iAS Web Services. Oracle9i JDeveloper is the industry's most advanced Java and XML IDE and provides unparalleled productivity and end-to-end J2EE and integrated Web Services standards compliance.

JDeveloper supports Oracle9iAS Web Services with the following features:

- Allows developers to create Java stubs from Web Services WSDL descriptions to programmatically use existing Web Services.
- Allows developers to create a new Web Service from Java or EJB classes, automatically producing the required deployment descriptor, web.xml, and WSDL file for you.
- Provides schema-driven WSDL file editing.
- Offers significant J2EE deployment support for Web Services J2EE .ear files, with automatic deployment to OC4J.

Securing Web Services

Oracle Enterprise Manager secures Oracle9iAS Web Services in the same way that it secures J2EE Servlets running under OC4J. This provides a comprehensive set of security facilities, including:

- Complete, standards-based security architecture for encryption, authentication, and authorization of Web Services.
- Single Sign-on to enable users to access several Web Services with a single password.
- Single Point of administration to enable users to centrally manage the security for Web Services.

Aggregating Web Services

Oracle9iAS Portal facility provides the ability to aggregate Oracle9iAS Web Services within an organization into a Portal. Additionally, portlets in the Oracle9iAS Portal framework can be published as Web Services.

Oracle9iAS Web Services Architecture

Oracle9iAS Containers for J2EE (OC4J) provides the foundation for building applications as components and supports Oracle9iAS Web Services. Oracle9iAS Web Services supports both RPC and Document Style web services.

Oracle9iAS Web Services supports the following RPC Web Services:

- Java Classes
- Stateless Session Enterprise Java Beans (EJBs)
- Stateless PL/SQL Stored Procedures or Functions

Oracle9iAS Web Services supports the following Document Style web services:

- Java Class Document Style Web Services
- JMS Document Style Web Services

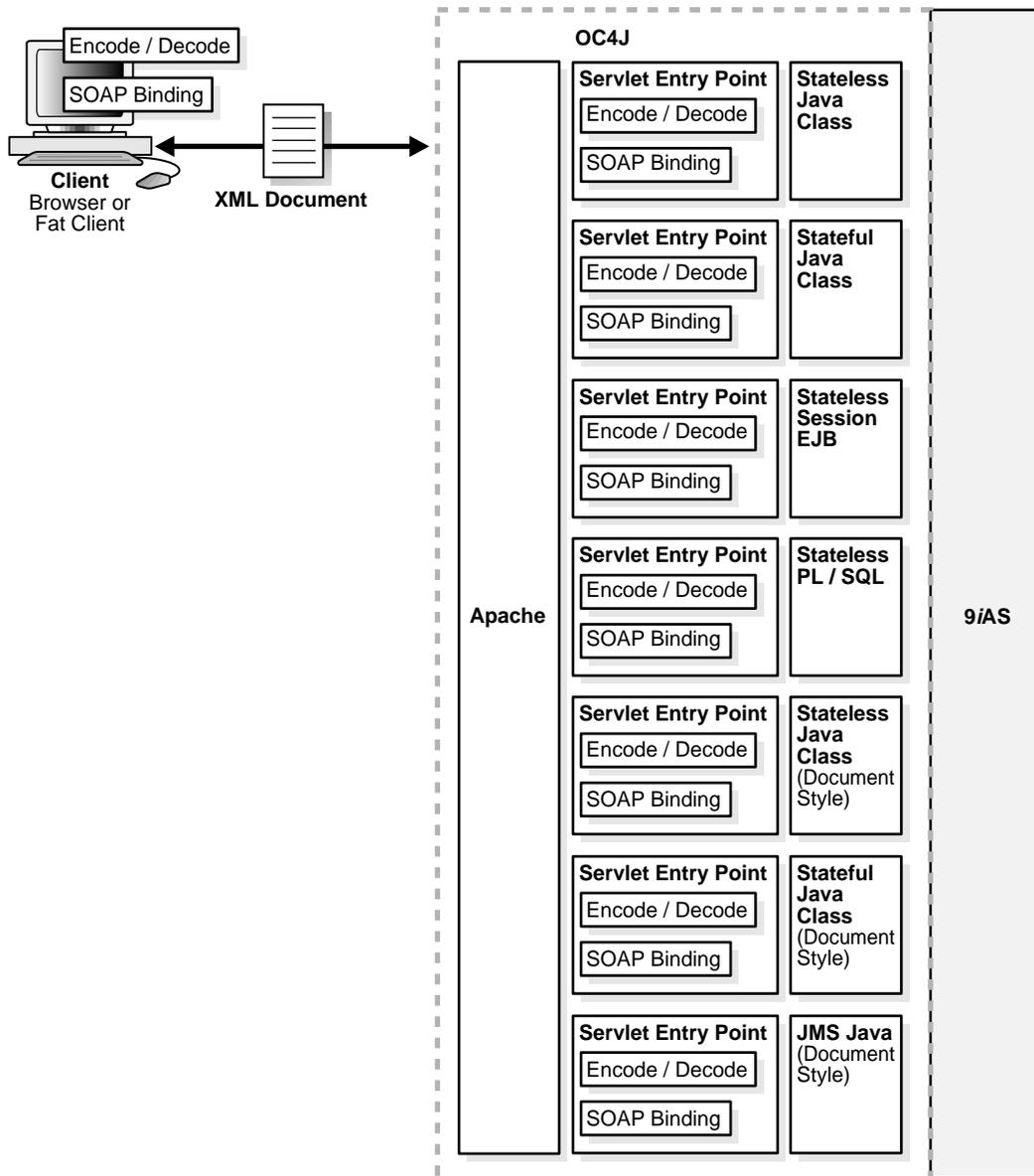
For each implementation type, Oracle9iAS Web Services uses a different Servlet that conforms to J2EE standards to provide an entry point to a Web Service implementation. [Figure 2–1](#) shows the Web Services runtime architecture, including the Servlet entry points.

The Oracle9iAS Web Services runtime architecture discussion includes the following:

- [About Servlet Entry Points for Web Services](#)
- [What Are the Packaging and Deployment Options for Web Services](#)
- [About Server Skeleton Code Generation for Web Services](#)

See Also: "[Simple Object Access Protocol \(SOAP\)](#)" on page 1-6 for information on RPC Style and Document Style Web Services.

Figure 2-1 Web Services Runtime Architecture (RPC and Document Style with Servlet Entry Points)



About Servlet Entry Points for Web Services

To use Oracle9iAS Web Services, you need to deploy a J2EE .ear file to Oracle9iAS. The J2EE .ear file contains a Web Services Servlet configuration and includes an implementation of the Web Service. Oracle9iAS Web Services supplies the Servlet classes, one for each supported implementation type. At runtime, Oracle9iAS uses the Servlet classes to access the user supplied Web Service implementation.

The Oracle9iAS Web Services Servlet classes support the following Web Services implementation types:

- Java Class (Stateless) - The object implementing the Web Service is any arbitrary Java class. The Web Service is stateless.
- Java Class (Stateful) -The object implementing the Web Service is any arbitrary Java class. The Web Service is considered stateful. A Servlet `HttpSession` maintains the object state between requests from the same client.
- Stateless Session EJBs - Stateless Session EJBs can be exposed as Web Services. The Web Service is considered to be stateless.
- PL/SQL Stored Procedure or Function - The object implementing the Web Service is a Java class that accesses the PL/SQL stored procedure or function. The Web Service is considered to be stateless. The Oracle JPublisher tool generates the Java access class for the PL/SQL stored procedure or function.
- Java Class Document Style Web Service (Stateless) - The object implementing the Web Service is a Java class using a supported method signature. The Web Service is stateless.
- Java Class Document Style Web Service (Stateful) -The object implementing the Web Service is a Java class using a supported method signature. The Web Service is considered stateful. A Servlet `HttpSession` maintains the object state between requests from the same client.
- Java JMS Web Service - Supports sending and receiving messages to or from JMS destinations. Using the JMS Web Service you can include an MDB to handle or generate messages.

When a Web Service is deployed, a unique instance of the Servlet class manages the Web Service. The Servlet class is implemented as part of Oracle9iAS Web Services runtime support. To make Web Services accessible, you deploy the Web Service implementation with the corresponding Web Services Servlet.

Note: Using Oracle9iAS SOAP, based on Apache SOAP 2.2, there is only a single instance of a single Servlet entry point for all the Web Services in the entire system. The Oracle9iAS Web Services architecture differs; under Oracle9iAS Web Services, a unique Servlet instance supports each Web Service.

RPC Style Web Service implementations under Oracle9iAS Web Services that take values as parameters or that return values to a client need to restrict the types passed. This restriction allows the types passed to be converted between XML and Java objects (and between Java objects and XML). [Table 2–1](#) lists the supported types for passing to or from Oracle9iAS Web Services.

Document Style Web Service implementations under Oracle9iAS Web Services restrict the signature of the Java methods that implement the Web Service. Only `org.w3c.dom.Element` can be passed to or sent from these Web Services.

Table 2–1 Web Services Supported Data Types (for RPC Parameters and Return Values)

Primitive Type	Object Type
Boolean	<code>java.lang.Boolean</code>
byte	<code>java.lang.Byte</code>
double	<code>java.lang.Double</code>
float	<code>java.lang.Float</code>
int	<code>java.lang.Integer</code>
long	<code>java.lang.Long</code>
short	<code>java.lang.Short</code>
string	<code>java.lang.String</code>
	<code>java.util.Date</code>
	<code>org.w3c.dom.Element</code>
	<code>org.w3c.dom.Document</code>
	<code>org.w3c.dom.DocumentFragment</code>
	Java Beans (whose property types are listed in this table or are another supported Java Bean)
	Single-dimensional arrays of types listed in this table

What Are the Packaging and Deployment Options for Web Services

Oracle9iAS Web Services are accessed as Servlets, thus, Web Services need to be assembled. The `WebServicesAssembler` tool prepares J2EE .ear files for Web Services by configuring a `web.xml` file that is a component of a J2EE .war file, and including the required resources and the implementation and support classes.

To build a Web Service with the assembly tool, you can supply a Jar file, .war file, `obj.jar`, or .ear file that includes your Web Service implementation. The assembly tool then build the Web Service using configuration information specified in its XML configuration file.

See Also:

- [Chapter 3, "Developing and Deploying Java Class Web Services"](#)
- [Chapter 4, "Developing and Deploying EJB Web Services"](#)
- [Chapter 5, "Developing and Deploying Stored Procedure Web Services"](#)
- [Chapter 6, "Developing and Deploying Document Style Web Services"](#)
- [Chapter 7, "Developing and Deploying JMS Web Services"](#)

About Server Skeleton Code Generation for Web Services

The first time Oracle9iAS Web Services receives a request for a service, the Servlet entry point automatically does the following (this discussion does not apply for JMS Web Services, which are handled differently):

- Validates the class loading. All the classes that are required for the Web Service implementation must conform to standard J2EE class loading norms.
- Validates the data types. All the Java classes or EJBs must conform to the restrictions on supported parameter and return types as shown in [Table 2-1](#).
- Generates server skeleton code. The server skeleton code is only generated the first time the Web Service is accessed or when the ear file is redeployed (when an application is redeployed, the server skeleton code and other Web Services support files are regenerated). The generated code is stored in the temporary directory associated with the Servlet context. The server skeleton code controls the lifecycle of the EJB (for Stateless Session EJB implementations), handles the marshaling of the parameters and return types (for SOAP RPC based Web Services), and dispatches to the actual Java class or EJB methods that implement the service.

After the server skeleton class is generated, when subsequent requests for a service are received, the server skeleton directly handles marshalling and then invokes the method that implements the service (for Web Services implemented with PL/SQL stored procedures or functions, the server skeleton invokes the Java class that accesses the Database containing the PL/SQL stored procedure or function).

For document style Web Services, the server skeleton passes the DOM element to the method that implements the service.

Understanding WSDL and Client Proxy Stubs for Web Services

Oracle9iAS Web Services provides a tool to generate a WSDL file that can be packaged with a Web Service at assembly time, or the WSDL file can be generated at runtime. This tool also supports generating client-side proxy stubs, given a WSDL file.

There are several elements to Oracle9iAS Web Services WSDL support. First, RPC style Web Services are based on interoperable XML data representations and arbitrary Java objects do not in general map to XML. Oracle9iAS Web Services supports a set of XML types corresponding to a set of Java types (see [Table 2-1](#) for the list of supported Java types).

Second, using Oracle9iAS Web Services, an application developer can either statically generate the WSDL interfaces for a Web Service or the Oracle9iAS Web Services runtime can generate WSDL and client-side proxy stubs if they are not provided when a Web Service is deployed. These files can be generated by the runtime on the server-side and delivered when they are requested by a Web Services client.

Oracle9iAS also provides a client-side tool to statically generate WSDL given a Java class or a J2EE application. Likewise, the Web Services Assembly tool can generate the client-side proxy given a generated WSDL file or a known WSDL endpoint.

See Also:

- ["Generating Client-Side Proxies With WebServicesAssembler"](#) on page 8-8
- ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Overview of a WSDL Based Web Service Client

Using Web Services, a client application sends a SOAP request that invokes a Web Service and handles the SOAP response from the service. To facilitate client application development, the Oracle9iAS Web Services runtime can generate WSDL to describe a Web Service. Using the WSDL, development tools can assist developers in building applications that invoke Web Services.

See Also:

- ["Using Oracle9i JDeveloper with Web Services"](#) on page 2-4
- [Chapter 8, "Building Clients that Use Web Services"](#)

Overview of a Client-Side Proxy Stubs Based Web Service Client

Using Web Services, a client application sends a SOAP request that invokes a Web Service and handles the SOAP response from the service. To facilitate client-side application development, Oracle9iAS Web Services can generate client-side proxy stubs. The client-side proxy stubs hide the details of composing a SOAP request and decomposing the SOAP response. The generated client-side proxy stubs support a synchronous invocation model for requests and responses. The generated stubs make it easier to write a Java client application to make a Web Service (SOAP) request and handle the response.

See Also: [Chapter 8, "Building Clients that Use Web Services"](#)

Web Services Home Page

Oracle9iAS Web Services provides a Web Service Home Page for each deployed Web Service.

A Web Service Home Page provides the following:

- A Link to the WSDL file - To obtain the WSDL file for a Web Service, select the Service Description link and save the file.
- Links to Web Service Test Pages for each supported operation-To test the available Web Service operations enter the parameter values for the operation, if any, and select the Invoke button.
- Links to the Web Service client-side proxy Jar and the client-side proxy source - To obtain the client-side proxy Jar or the client-side proxy source, select the appropriate link, Proxy Jar or Proxy Source, and save the file.

Figure 2–2 shows a sample Web Service Home Page.

Figure 2–2 *Web Service Home Page*

StatefulExample endpoint

WSDL for Service: StatefulExample, generated by Oracle WSDL toolkit (version: 1.1)
For a formal definition, please review the [Service Description](#) (*rpc style*).

StatefulExample service

The following operations are supported.

- [count](#)
- [helloWorld](#)

oc4j client

The java proxy is packaged in a .jar either as classes or sources files.

- [Proxy Jar](#)
- [Proxy Source](#)

About Universal Description, Discovery, and Integration Registry

The Universal Description, Discovery, and Integration (UDDI) specification consists of a four-tier hierarchical XML schema that provides the base information model to publish, validate, and invoke information about Web Services. The four types of information that the UDDI XML schema defines are:

- **Business Entity** - The top level XML element in a UDDI entry captures the starting set of information required by partners seeking to locate information about a business' services including its name, its industry or product category, its geographic location, and optional categorization and contact information. This includes support for *Yellow Pages* taxonomies to search for businesses by industry, product, or geography.
- **Business Service** - The `businessService` structure groups a series of related Web Services together so that they can be related to either a business process or a category of services. An example of a business process could be a logistics/delivery process which could include several Web Services including shipping, routing, warehousing, and last-mile delivery services. By organizing Web Services into groups associated with categories or business processes, UDDI allows more efficient search and discovery of Web Services.
- **Binding Information** - Each `businessService` has one or more technical Web Service Descriptions captured in an XML element called a binding template. The binding template contains the information that is relevant for application programs that need to invoke or to bind to a specific Web Service. This information includes the Web Service URL address, and other information describing hosted services, routing and load balancing facilities.
- **Compliance Information** - While the `bindingTemplate` contains the information required to invoke a service, it is not always enough to simply know where to contact a particular Web Service. For instance, to send a business partner's Web Service a purchase order, the invoking service must not only know the location/URL for the service, but what format the purchase order should be sent in, what protocols are appropriate, what security required, and what form of a response will result after sending the purchase order. Before invoking a Web Service, it is useful to determine whether the specific service being invoked complies with a particular behavior or programming interface. Each `bindingTemplate` element, therefore, contains an element called a `tModel` that contains information which enables a client to determine whether a specific Web Service is a compliant implementation.

Oracle Enterprise Manager Features to Register Web Services

When a Web Service is deployed on Oracle9iAS, you can use Oracle Enterprise Manager to register the specific Web Service and publish its WSDL to the UDDI registry and to discover published Web Services.

See Also: [Chapter 10, "Discovering and Publishing Web Services"](#)

Developing and Deploying Java Class Web Services

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services that are implemented as Java classes.

This chapter covers the following topics:

- [Using Oracle9iAS Web Services With Java Classes](#)
- [Writing Java Class Based Web Services](#)
- [Preparing and Deploying Java Class Based Web Services](#)
- [Serializing and Encoding Parameters and Results for Web Services](#)

Using Oracle9iAS Web Services With Java Classes

This chapter shows sample code for writing Web Services implemented with Java classes and describes the difference between writing stateful and stateless Java Web Services.

Oracle9iAS supplies Servlets to access the Java classes which implement a Web Service. The Servlets handle requests generated by a Web Service client, run the Java method that implements the Web Service and returns results back to Web Services clients.

See Also:

- [Chapter 2, "Oracle9iAS Web Services"](#)
- [Chapter 4, "Developing and Deploying EJB Web Services"](#)
- [Chapter 5, "Developing and Deploying Stored Procedure Web Services"](#)
- [Chapter 8, "Building Clients that Use Web Services"](#)

Writing Java Class Based Web Services

Writing Java class based Web Services involves building a Java class that includes one or more methods. When a Web Services client makes a service request, Oracle9iAS Web Services invokes a Web Services Servlet that runs the method that implements the service request. There are very few restrictions on what actions Web Services can perform. At a minimum, Web Services generate some data that is sent to a client or perform an action as specified by a Web Service request.

This section shows how to write a stateful and a stateless Java Web Service that returns a string, "Hello World". The stateful service also returns an integer running count of the number of method calls to the service. This Java Web Service receives a client request and generates a response that is returned to the Web Service client.

The sample code is supplied with Oracle9iAS Web Services in the directory `$ORACLE_HOME/webservices/demo/basic/java_services` on UNIX or in `%ORACLE_HOME%\webservices\demo\basic\java_services` on Windows.

Writing Stateless and Stateful Java Web Services

Oracle9iAS Web Services supports stateful and stateless implementations for Java classes running as Web Services, as follows:

- For a stateful Java implementation, Oracle9iAS Web Services uses a single Java instance to serve the Web Service requests from an individual client.
- For a stateless Java implementation, Oracle9iAS Web Services creates multiple instances of the Java class in a pool, any one of which may be used to service a request. After servicing the request, the object is returned to the pool for use by a subsequent request.

Note: It is the job of the Web Services developer to make the design decision to implement a stateful or stateless Web Service. When packaging Web Services, stateless and stateful Web Services are handled slightly differently. This chapter describes these differences in the section, "[Preparing and Deploying Java Class Based Web Services](#)" on page 3-9.

Building a Sample Java Class Implementation

Developing a Java Web Service consists of the following steps:

- [Defining a Java Class Containing Methods for the Web Service](#)
- [Defining an Interface for Explicit Method Exposure](#)
- [Writing a WSDL File \(Optional\)](#)

Defining a Java Class Containing Methods for the Web Service

Create a Java Web Service by writing or supplying a Java class with methods that are deployed as a Web Service. In the sample supplied in the `java_services` sample directory, the `.ear` file, `ws_example.ear` contains the Web Service source, class, and configuration files. In the expanded `.ear` file, the class `StatefulExampleImpl` provides the stateful Java service and `StatelessExampleImpl` provides the stateless Java service.

When writing a Java Web Service, if you want to place the Java service in a package, use the Java package specification to name the package. The first line of `StatefulExampleImpl.java` specifies the package name, as follows:

```
package oracle.j2ee.ws_example;
```

The stateless sample Web Service is implemented with `StatelessExampleImpl`, a public class. The class defines a public method, `helloWorld()`. In general, a Java class for a Web Service defines one or more public methods.

[Example 3-1](#) shows `StatelessExampleImpl`.

The stateful sample Web Service is implemented with `StatefulExampleImpl`, a public class. The class initializes the count and defines two public methods, `count()` and `helloWorld()`.

[Example 3-2](#) shows `StatefulExampleImpl`.

Example 3-1 Defining A Public Class with Java Methods for a Stateless Web Service

```
package oracle.j2ee.ws_example;

public class StatelessExampleImpl {
    public StatelessExampleImpl() {
    }
    public String helloWorld(String param) {
        return "Hello World, " + param;
    }
}
```

Example 3-2 Defining a Public Class with Java Methods for a Stateful Web Service

```
package oracle.j2ee.ws_example;

public class StatefulExampleImpl {
    int count = 0;
    public StatefulExampleImpl() {
    }
    public int count() {
        return count++;
    }
    public String helloWorld(String param) {
        return "Hello World, " + param;
    }
}
```

A Java class implementation for a Web Service must include a public constructor that takes no arguments. [Example 3-1](#) shows the public constructor `StatelessExampleImpl()` and [Example 3-2](#) shows `StatefulExampleImpl()`.

When an error occurs while running a Web Service implemented as a Java class, the Java class should throw an exception. When an exception is thrown, the Web Services Servlet returns a Web Services (SOAP) fault. Use the standard J2EE and OC4J administration facilities to view the logs of Servlet 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 method's parameters and return values must use supported types, or you need to use an interface class to limit the methods exposed to those methods using only supported types. [Table 3-1](#) lists the supported types for parameters and return values for Java methods that implement Web Services.

Note: See [Table 3-1](#) for the list of supported types for parameters and return values.

Defining an Interface for Explicit Method Exposure

Oracle9iAS Web Services allows you to limit the methods you expose as Web Services by supplying a public interface. To limit the methods exposed in a Web Service, include a public interface that lists the method signatures for the methods that you want to expose. [Example 3-3](#) shows an interface to the method in the class `StatelessExampleImpl`. [Example 3-4](#) shows an interface to the methods in the class `StatefulExampleImpl`.

Example 3-3 Using a Public Interface to Expose Stateless Web Services Methods

```
package oracle.j2ee.ws_example;

public interface StatelessExample {
    String helloWorld(String param);
}
```

Example 3–4 Using a Public Interface to Expose Stateful Web Services Methods

```
package oracle.j2ee.ws_example;

public interface StatefulExample {
    int count();
    String helloWorld(String param);
}
```

When an interface class is not included with a Web Service, the Web Services deployment exposes all public methods defined in the Java class. Using an interface, for example `StatelessExample` shown in [Example 3–3](#) or `StatefulExample` shown in [Example 3–4](#), exposes only the methods listed in the interface.

Note: Using an interface, only the methods with the specified method signatures are exposed when the Java class is prepared and deployed as a Web Service.

Use a Web Services interface for the following purposes:

1. To limit the exposure of methods to a subset of the public methods within a class.
2. To expand the set of methods that are exposed as Web Services to include methods within the superclass of a class.
3. To limit the exposure of methods to a subset of the public methods within a class, where the subset contains only the methods that use supported types for parameters or return values. [Table 3–1](#) lists the supported types for parameters and return values for Java methods that implement Web Services.

See Also: ["Using Supported Data Types for Java Web Services"](#) on page 3-7

Writing a WSDL File (Optional)

The `WebServicesAssembler` supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate WSDL files and client-side proxy files. You can use these tags to control whether the WSDL file and the client-side proxy are generated. Using these tags you can also specify that the generated WSDL file or a WSDL file that you write is packaged with the Web Service J2EE .ear.

A client-side developer either uses the WSDL file that is obtained from a deployed Web Service, or the client-side proxy that is generated from the WSDL to build an application that uses the Web Service.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Using Supported Data Types for Java Web Services

[Table 3–1](#) lists the supported data types for parameters and return values for Oracle9iAS Web Services.

Table 3–1 *Web Services Supported Data Types*

Primitive Type	Object Type
Boolean	<code>java.lang.Boolean</code>
byte	<code>java.lang.Byte</code>
double	<code>java.lang.Double</code>
float	<code>java.lang.Float</code>
int	<code>java.lang.Integer</code>
long	<code>java.lang.Long</code>
short	<code>java.lang.Short</code>
string	<code>java.lang.String</code>
	<code>java.util.Date</code>
	<code>org.w3c.dom.Element</code>
	<code>org.w3c.dom.Document</code>
	<code>org.w3c.dom.DocumentFragment</code>
	Java Beans (whose property types are listed in this table or are another supported Java Bean)
	Single-dimensional arrays of types listed in this table.

Note: Oracle9iAS Web Services does not support `Element []`, (arrays of `org.w3c.dom.Element`).

A Bean, for purposes of Web Services, is any Java class which conforms to the following restrictions:

- It must have a constructor taking no arguments.
- It must expose all interesting state through properties.
- It must not matter what order the accessors for the properties, for example, the `setX` or `getX` methods, are in.

Oracle9iAS Web Services allows Beans to be returned or passed in as arguments to J2EE Web Service methods, as long as the Bean only consists of property types that are listed in [Table 3-1](#) or are another supported Java Bean.

When Java Beans are used as parameters to Oracle9iAS Web Services, the client-side code should use the generated Bean included with the downloaded client-side proxy. This is because the generated client-side proxy code translates Simple Object Access Protocol (SOAP) structures to and from Java Beans by translating SOAP structure namespaces to and from fully qualified Bean class names. If a Bean with the specified name does not exist in the specified package, the generated client code will fail.

However, there is no special requirement for clients using Web Services Description Language (WSDL) to form calls to Oracle9iAS Web Services, rather than the client-side proxy. The generated WSDL document describes SOAP structures in a standard way. Application development environments, such as JDeveloper, which work directly from WSDL documents can correctly call Oracle9iAS Web Services with Java Beans as parameters.

Note: When Web Service proxy classes and WSDL are generated, all Java primitive types in the service implementation on the server-side are mapped to Object types in the proxy code or in the WSDL. For example, when the Web Service implementation includes parameters of primitive Java type `int`, the equivalent parameter in the proxy is of type `java.lang.Integer`. This mapping occurs for all primitive types.

See Also: [Chapter 8, "Building Clients that Use Web Services"](#)

Preparing and Deploying Java Class Based Web Services

To deploy a Java class as a Web Service you need to assemble a J2EE .ear file that includes the deployment descriptors for the Oracle9iAS Web Services Servlet and includes the Java class that supplies the Java implementation. This section describes how to use the Oracle9iAS Web Services tool, `WebServicesAssembler`. `WebServicesAssembler` takes an XML configuration file that describes the Java Class Web Service and produces a J2EE .ear file that can be deployed under Oracle9iAS Web Services.

This section contains the following topics.

- [Creating a Configuration File to Assemble Java Class Web Services](#)
- [Running WebServicesAssembler To Prepare Java Class Web Services](#)

Creating a Configuration File to Assemble Java Class Web Services

The Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9iAS Web Services. This section describes how to create a configuration file to use with Java Class Web Services.

Create a `WebServicesAssembler` configuration file by adding the following:

- [Adding Web Service Top Level Tags](#)
- [Adding Java Stateless Service Tags](#)
- [Adding Java Stateful Service Tags](#)
- [Adding WSDL and Client-Side Proxy Generation Tags](#)

Adding Web Service Top Level Tags

[Table 3–2](#) describes the top level `WebServicesAssembler` configuration file tags. Add these tags to provide top level information describing the Java Stateless Web Service or a Java Stateful Web Service. These tags are included within a `<web-service>` tag in the configuration file.

[Example 3–5](#) shows a complete `config.xml` file, including the top level tags.

Table 3–2 Top Level WebServicesAssembler Configuration Tags

Tag	Description
<code><context></code> <i>context</i> <code></context></code>	Specifies the context root of the Web Service. This tag is required.
<code><datasource-JNDI-name></code> <code></datasource-JNDI-name></code>	Specifies the datasource associated with the Web Service.
<code><description></code> <i>description</i> <code></description></code>	Provides a simple description of the Web Service. This tag is optional.
<code><destination-path></code> <i>dest_path</i> <code></destination-path></code>	Specifies the name of the generated J2EE .ear file output. The <i>dest_path</i> specifies the complete path for the output file. This tag is required.
<code><display-name></code> <i>disp_name</i> <code></display-name></code>	Specifies the Web Service display name. This tag is optional.
<code><option</code> <code>name="source-path"></code> <i>path</i> <code></option></code>	Includes a specified file in the output .ear file. Use this option to include java resources. This resource is added to the <code>lib</code> directory in the generated WAR component of the J2EE .ear file. The <i>path</i> specifies the path to the file to include.
<code><stateless-java-service></code> <i>sub-tags</i> <code></stateless-java-service></code>	Use this tag to add a Java Web Services that defines a stateless service. See Table 3–3 for a description of valid <i>sub-tags</i> .
<code><stateful-java-service></code> <i>sub-tags</i> <code></stateful-java-service></code>	Use this tag to add a Java Web Services that defines a stateful service. See Table 3–3 for a description of valid <i>sub-tags</i> .
<code><temporary-directory></code> <i>temp_dir</i> <code></temporary-directory></code>	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding Java Stateless Service Tags

Prepare Java Stateless Web Services using the `WebServicesAssembler` `<stateless-java-service>` tag. This tag is included within a `<web-service>` tag in the configuration file. Add this tag to provide information required for generating a Stateless Java Web Service.

Table 3–3 shows the `<stateless-java-service>` sub-tags and the `<stateful-java-service>` sub-tags. As noted in Table 3–3, some of the sub-tags listed only apply when using a `<stateful-java-service>`.

Example 3–5 shows a complete `config.xml` file, including `<stateless-java-service>`.

Note: It is the job of the Web Services developer to make the design decision to implement a stateful or stateless Web Service. When packaging Web Services, stateless and stateful Web Services are handled slightly differently.

Adding Java Stateful Service Tags

Prepare Java Stateful Web Services using the `WebServicesAssembler` `<stateful-java-service>` tag. This tag is included within a `<web-service>` tag in the configuration file. Add this tag to provide information required for generating a Stateful Java Web Service.

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's generated `J2EE.ear` file.

Table 3–3 shows the `<stateful-java-service>` sub-tags.

Example 3–5 shows a complete `config.xml` file, including `<stateful-java-service>`.

Table 3–3 *Stateless and Stateful Java Service Sub-Tags*

Tag	Description
<code><class-name></code> <i>class</i> <code></class-name></code>	Specifies the fully qualified class name for the class that supplies the Web Service implementation. This tag is required
<code><interface-name></code> <i>interface</i> <code></interface-name></code>	Specifies the fully qualified name of the interface that tells the Web Service Servlet generation code which methods should be exposed as Web Services. This tag is optional
<code><ejb-resource></code> <i>ejb-resource</i> <code></ejb-resource></code>	This is a backwards compatibility tag. See Also: the top level <code><option name="source-path"></code> tag in Table 3–2. This tag is optional

Table 3–3 (Cont.) Stateless and Stateful Java Service Sub-Tags

Tag	Description
<code><java-resource></code> <i>resource</i> <code></java-resource></code>	<p>This is a backwards compatibility tag.</p> <p>See Also: the top level <code><option name="source-path"></code> tag in Table 3–2.</p> <p>This tag is optional</p>
<code><message-style></code> <i>rpc</i> <code></message-style></code>	<p>Sets the message style. When defining a Java Web Service, if you include the <code><message-style></code> tag you must specify the value <code>rpc</code>.</p> <p>Valid Values: <code>doc</code>, <code>rpc</code></p> <p>This tag is optional</p> <p>Default value: <code>rpc</code> (when the <code><message-style></code> tag is not supplied)</p>
<code><scope></code> <i>scope</i> <code></scope></code>	<p>Sets the scope of the session for stateful services.</p> <p>The <code><scope></code> tag only applies for stateful services. Use this tag only within the <code><stateful-java-service></code> tag.</p> <p>This tag is optional</p> <p>Valid Values: <code>application</code>, <code>session</code></p> <p>Default Value: <code>session</code></p>
<code><session-timeout></code> <i>value</i> <code></session-timeout></code>	<p>Sets the session timeout for a stateful session.</p> <p>The <code><session-timeout></code> tag only applies for stateful services. Use this tag only within the <code><stateful-java-service></code> tag.</p> <p>Specify <i>value</i> with an integer that defines the timeout for the session in seconds. The default value for the session timeout for stateful Java sessions where no session timeout is specified is 60 seconds.</p> <p>This tag is optional</p>
<code><uri></code> <i>URI</i> <code></uri></code>	<p>Specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the <i>URI</i> is appended to the <code><context></code> to specify the Web Service location.</p> <p>This tag is required</p>

Example 3-5 Sample WebServicesAssembler Configuration File

```

<web-service>
  <display-name>Web Services Example</display-name>
  <description>Java Web Service Example</description>
  <!-- Specifies the resulting web service archive will be stored in
        ./ws_example.ear -->
  <destination-path>./ws_example.ear</destination-path>
  <!-- Specifies the temporary directory that web service assembly
        tool can create temporary files. -->
  <temporary-directory>./tmp</temporary-directory>
  <!-- Specifies the web service will be accessed in the servlet context
        named "/webservices". -->
  <context>/webservices</context>

  <!-- Specifies the web service will be stateless -->
  <stateless-java-service>
    <interface-name>oracle.j2ee.ws_example.StatelessExample</interface-name>
    <class-name>oracle.j2ee.ws_example.StatelessExampleImpl</class-name>
    <!-- Specifies the web service will be accessed in the uri named
          "statelessTest" within the servlet context. -->
    <uri>/statelessTest</uri>
    <!-- Specifies the location of Java class files are under
          ./src -->
    <java-resource>./src</java-resource>
  </stateless-java-service>

  <stateful-java-service>
    <interface-name>oracle.j2ee.ws_example.StatefulExample</interface-name>
    <class-name>oracle.j2ee.ws_example.StatefulExampleImpl</class-name>
    <!-- Specifies the web service will be accessed in the uri named
          "statefullTest" within the servlet context. -->
    <uri>/statefulTest</uri>
    <!-- Specifies the location of Java class files are under
          ./src -->
    <java-resource>./src</java-resource>
  </stateful-java-service>
</web-service>

```

Adding WSDL and Client-Side Proxy Generation Tags

The `WebServicesAssembler` supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate WSDL files and client-side proxy files. You can use these tags to control whether the WSDL file and the client-side proxy are generated. Using these tags you can also specify that the generated WSDL file or a WSDL file that you supply is packaged with the Web Service J2EE .ear.

A client-side developer can use the WSDL file that is obtained from a deployed Web Service, or the client-side proxy that is generated from the WSDL to build an application that uses the Web Service.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Running WebServicesAssembler To Prepare Java Class Web Services

After you create the `WebServicesAssembler` configuration file, you can generate a J2EE .ear file for the Web Service. The J2EE .ear file includes the Java Web Service servlet configuration information, including the file `web.xml`, and the Java classes and interfaces that you supply.

Run the Oracle9iAS Web Services assembly tool, `WebServicesAssembler` as follows:

```
java -jar WebServicesAssembler.jar -config config_file
```

Where: *config_file* is the configuration file that contains the `<stateless-java-service>` or the `<stateful-java-service>` tags.

See Also:

- ["Creating a Configuration File to Assemble Java Class Web Services"](#) on page 3-9
- ["Running the Web Services Assembly Tool"](#) on page 9-2

Deploying Java Class Based Web Services

After creating the J2EE .ear file containing the Java classes and the Web Services Servlet deployment descriptors you can deploy the Web Service as you would any standard J2EE application stored in an .ear file (to run under OC4J).

See Also: *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library

Serializing and Encoding Parameters and Results for Web Services

Parameters and results sent between Web Service clients and a Web Service implementation go through the following steps:

1. Parameters are serialized and encoded in XML when sent from the Web Service client.
2. Parameters are deserialized and decoded from XML when the Web Service receives a request on the server side.
3. Parameters or results are serialized and encoded in XML when a request is returned from a Web Service to a Web Service client.
4. Parameters or results must be deserialized and decoded from XML when the Web Service client receives a reply.

Oracle*9iAS* Web Services supports a prepackaged implementation for handling these four steps for serialization and encoding, and deserialization and decoding. The prepackaged mechanism makes the four serialization and encoding steps transparent both for the Web Services client-side application, and for the Java service writer that is implementing a Web Service. Using the prepackaged mechanism, Oracle*9iAS* Web Services supports the following encoding mechanisms:

- Standard SOAP v.1.1 encoding: Using standard SOAP v.1.1 encoding, the server side Web Services Servlet that calls the Java class implementation handles serialization and encoding internally for the types supported by Oracle*9iAS* Web Services. [Table 3–1](#) lists the supported Web Services parameter and return value types when using standard SOAP v.1.1 encoding.
- Literal XML encoding. Using Literal XML encoding, a Web Service client can pass as a parameter, or a Java service can return as a result, a value that is encoded as a conforming W3C Document Object Model (DOM) `org.w3c.dom.Element`. When an `Element` passes as a parameter to a Web Service, the server side Java implementation processes the `org.w3c.dom.Element`. For return values sent from a Web Service, the Web Services client parses or processes the `org.w3c.dom.Element`.

Note: For parameters to a Web Service or results that the Web Service generates and returns to Web Services clients, the Oracle*9iAS* Web Services implementation supports either the Standard SOAP encoding or Literal XML encoding but not both, for any given Web Service (Java method).

See Also: [Chapter 8, "Building Clients that Use Web Services"](#)

Developing and Deploying EJB Web Services

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services that are implemented as stateless session Enterprise Java Beans (EJBs).

This chapter covers the following topics:

- [Using Oracle9iAS Web Services With Stateless Session EJBs](#)
- [Writing Stateless Session EJB Web Services](#)
- [Preparing and Deploying Stateless Session EJB Based Web Services](#)

Using Oracle9iAS Web Services With Stateless Session EJBs

This chapter shows sample code for writing Web Services implemented with stateless session EJBs.

Oracle9iAS supplies Servlets to access the EJBs which implement a Web Service. A Servlets handle requests generated by a Web Service client, locates the EJB home and remote interfaces, runs the EJB that implements the Web Service, and returns results back to the Web Service client.

See Also:

- [Chapter 2, "Oracle9iAS Web Services"](#)
- [Chapter 3, "Developing and Deploying Java Class Web Services"](#)
- [Chapter 5, "Developing and Deploying Stored Procedure Web Services"](#)
- [Chapter 8, "Building Clients that Use Web Services"](#)

Writing Stateless Session EJB Web Services

Writing EJB based Web Services involves obtaining or building an EJB that implements a service. The EJB should contain one or more methods that a Web Services Servlet running under Oracle9iAS invokes when a client makes a Web Services request. There are very few restrictions on what actions Web Services can perform. At a minimum, Web Services usually generate data that is sent to a Web Services client or perform an action as specified by a Web Services method request.

This section shows how to write a simple stateless session EJB Web Service, `HelloService` that returns a string, "Hello World", to a client. This EJB Web Service receives a client request with a single `String` parameter and generates a response that it returns to the Web Service client.

The sample code for the complete Web Service is supplied with Oracle9iAS Web Services installation in the following directory:

```
$ORACLE_HOME/webservices/demo/basic/stateless_ejb on UNIX  
%ORACLE_HOME%\webservices\demo\basic\stateless_ejb on Windows.
```

Create a stateless session EJB Web Service by writing a standard J2EE stateless session EJB containing a remote interface, a home interface, and an enterprise bean class. Oracle9iAS Web Services runs EJBs that are deployed as Oracle9iAS Web Services in response to a request issued by a Web Service client.

Developing a stateless session EJB consists of the following steps:

- [Defining a Stateless Session Remote Interface](#)
- [Defining a Stateless Session Home Interface](#)
- [Defining a Stateless Session EJB Bean](#)
- [Returning Results From EJB Web Services](#)
- [Error Handling for EJB Web Services](#)
- [Serializing and Encoding Parameters and Results for EJB Web Services](#)
- [Using Supported Data Types for Stateless Session EJB Web Services](#)
- [Writing a WSDL File for EJB Web Services \(Optional\)](#)

See Also: "Preparing and Deploying Stateless Session EJB Based Web Services" on page 4-8

Defining a Stateless Session Remote Interface

When looking at the `HelloService` EJB Web Service, note that the `.ear` file, `HelloService.ear` defines the Web Service and its configuration files. In the sample directory, the file `HelloService.java` provides the remote interface for the `HelloService` EJB.

[Example 4-1](#) shows the Remote interface for the sample stateless session EJB.

Example 4-1 Stateless Session EJB Remote Interface for Web Service

```
package demo;

public interface HelloService extends javax.ejb.EJBObject {
    java.lang.String hello(java.lang.String phrase) throws java.rmi.RemoteException;
}
```

Defining a Stateless Session Home Interface

The sample file `HelloServiceHome.java` provides the home interface for the `HelloService` EJB.

[Example 4-2](#) shows the `EJBHome` interface for the sample stateless session EJB.

Example 4–2 Stateless Session EJB Home Interface for Web Service

```
package demo;
/**
 * This is a Home interface for the Session Bean
 */
public interface HelloServiceHome extends javax.ejb.EJBHome {

    HelloService create() throws javax.ejb.CreateException, java.rmi.RemoteException
    ;
}
```

Defining a Stateless Session EJB Bean

The sample file `HelloServiceBean.java` provides the Bean logic for the `HelloService` EJB. When you create a Bean to implement a Web Service, the parameters and return values must be of supported types. [Table 4–1](#) lists the supported types for parameters and return values for stateless session EJBs that implement Web Services.

[Example 4–3](#) shows the source code for the `HelloService` Bean.

Example 4–3 Stateless Session EJB Bean Class for Web Services

```
package demo;

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

/**
 * This is a Session Bean Class.
 */
public class HelloServiceBean implements SessionBean {
    private javax.ejb.SessionContext mySessionCtx = null;

    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 {}
    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }
}
```

```
}  
public String hello(String phrase)  
{  
    return "HELLO!! You just said :" + phrase;  
}  
public void setSessionContext(javax.ejb.SessionContext ctx) throws  
java.rmi.RemoteException {  
    mySessionCtx = ctx;  
}  
}
```

Returning Results From EJB Web Services

The `hello()` method shown in [Example 4-3](#) returns a `String`. An Oracle9iAS Web Services server-side Servlet runs the Bean that calls the `hello()` method when the Servlet receives a Web Services request from a client. After executing the `hello()` method, the Servlet returns a result to the Web Service client.

[Example 4-3](#) shows that the EJB Bean writer only needs to return values of supported types to create Web Services implemented as stateless session EJBs.

See Also: ["Using Supported Data Types for Stateless Session EJB Web Services"](#) on page 4-6

Error Handling for EJB Web Services

When an error occurs while running a Web Service implemented as an EJB, the EJB should throw an exception. When an exception is thrown, the Web Services Servlet returns a Web Services (SOAP) fault. Use the standard J2EE and OC4J administration facilities for logging Servlet errors for a Web Service that uses stateless session EJBs for its implementation.

Serializing and Encoding Parameters and Results for EJB Web Services

Parameters and results sent between Web Service clients and a Web Service implementation need to be encoded and serialized. This allows the call and return values to be passed as XML documents using SOAP.

See Also: ["Serializing and Encoding Parameters and Results for Web Services"](#) on page 3-15

Using Supported Data Types for Stateless Session EJB Web Services

Table 4–1 lists the supported data types for parameters and return values for Oracle9iAS Web Services.

Table 4–1 Web Services Supported Data Types

Primitive Type	Object Type
Boolean	<code>java.lang.Boolean</code>
byte	<code>java.lang.Byte</code>
double	<code>java.lang.Double</code>
float	<code>java.lang.Float</code>
int	<code>java.lang.Integer</code>
long	<code>java.lang.Long</code>
short	<code>java.lang.Short</code>
string	<code>java.lang.String</code>
	<code>java.util.Date</code>
	<code>org.w3c.dom.Element</code>
	<code>org.w3c.dom.Document</code>
	<code>org.w3c.dom.DocumentFragment</code>
	Java Beans (whose property types are listed in this table or are another supported Java Bean)
	Single-dimensional arrays of types listed in this table.

Note: Oracle9iAS Web Services does not support `Element[]`, (arrays of `org.w3c.dom.Element`).

A Bean, for purposes of Web Services, is any Java class which conforms to the following restrictions:

- It must have a constructor taking no arguments.
- It must expose all interesting state through properties.
- It must not matter what order the accessors for the properties, for example, the `setX` or `getX` methods, are in.

Oracle9iAS Web Services allows Beans to be returned or passed in as arguments to J2EE Web Service methods, as long as the Bean only consists of property types that are listed in [Table 4-1](#) or are another supported Java Bean.

When Java Beans are used as parameters to Oracle9iAS Web Services, the client-side code should use the generated Bean included with the downloaded client-side proxy. This is because the generated client-side proxy code translates Simple Object Access Protocol (SOAP) structures to and from Java Beans by translating SOAP structure namespaces to and from fully qualified Bean class names. If a Bean with the specified name does not exist in the specified package, the generated client code will fail.

However, there is no special requirement for clients using Web Services Description Language (WSDL) to form calls to Oracle9iAS Web Services, rather than the client-side proxy. The generated WSDL document describes SOAP structures in a standard way. Application development environments, such as JDeveloper, which work directly from WSDL documents can correctly call Oracle9iAS Web Services with Java Beans as parameters.

Note: When Web Service proxy classes and WSDL are generated, all Java primitive types in the service implementation on the server-side are mapped to Object types in the proxy code or in the WSDL. For example, when the Web Service implementation includes parameters of primitive Java type `int`, the equivalent parameter in the proxy is of type `java.lang.Integer`. This mapping occurs for all primitive types.

See Also: [Chapter 8, "Building Clients that Use Web Services"](#)

Writing a WSDL File for EJB Web Services (Optional)

The `WebServicesAssembler` supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate WSDL files and client-side proxy files. You can use these tags to control whether the WSDL file and the client-side proxy are generated. Using these tags you can also specify that the generated WSDL file or a WSDL file that you write is packaged with the Web Service J2EE `.ear`.

A client-side developer either uses the WSDL file that is obtained from a deployed Web Service, or the client-side proxy that is generated from the WSDL to build an application that uses the Web Service.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Preparing and Deploying Stateless Session EJB Based Web Services

To deploy a stateless session EJB as a Web Service you need to assemble a J2EE .ear file that includes the deployment descriptors for the Oracle9iAS Web Services Servlet and includes the `ejb.jar` that supplies the Java implementation. This section describes how to use the Oracle9iAS Web Services tool, `WebServicesAssembler`. `WebServicesAssembler` takes an XML configuration file that describes the stateless session EJB Web Service and produces a J2EE .ear file that can be deployed under Oracle9iAS Web Services.

This section contains the following topics.

- [Creating a Configuration File to Assemble Stateless Session EJB Web Services](#)
- [Running WebServicesAssembler To Prepare Stateless Session EJB Web Services](#)

Creating a Configuration File to Assemble Stateless Session EJB Web Services

The Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9iAS Web Services. This section describes how to create a configuration file to use with stateless session EJB Web Services.

Create `WebServicesAssembler` configuration file by adding the following:

- [Adding Web Service Top Level Tags](#)
- [Adding Stateless Session EJB Service Tags](#)
- [Adding WSDL and Client-Side Proxy Generation Tags](#)

Adding Web Service Top Level Tags

[Table 4–2](#) describes the top level `WebServicesAssembler` configuration file tags. Add these tags to provide top level information describing the Java Stateless Web Service or a Java Stateful Web Service. These tags are included within a `<web-service>` tag in the configuration file.

[Example 4–4](#) shows a complete `config.xml` file, including the top level tags.

Table 4–2 Top Level WebServicesAssembler Configuration Tags

Tag	Description
<code><context></code> <i>context</i> <code></context></code>	Specifies the context root of the Web Service. This tag is required.
<code><datasource-JNDI-name></code> <code></datasource-JNDI-name></code>	Specifies the datasource associated with the Web Service.
<code><description></code> <i>description</i> <code></description></code>	Provides a simple description of the Web Service. This tag is optional.
<code><destination-path></code> <i>dest_path</i> <code></destination-path></code>	Specifies the name of the generated J2EE .ear file output. The <i>dest_path</i> specifies the complete path for the output file. This tag is required.
<code><display-name></code> <i>disp_name</i> <code></display-name></code>	Specifies the Web Service display name. This tag is optional.
<code><option name="source-path"></code> <i>path</i> <code></option></code>	Includes a specified file in the output .ear file. Use this option to include Java resources. The <i>path</i> specifies the path to the file to include. This tag is optional.
<code><stateless-session-ejb-service></code> <i>sub-tags</i> <code></stateless-session-ejb-service></code>	Use this tag to add a stateless session EJB Web Service. See Table 4–3 for a description of the valid <i>sub-tags</i> .
<code><temporary-directory></code> <i>temp_dir</i> <code></temporary-directory></code>	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding Stateless Session EJB Service Tags

Prepare Stateless Session EJB Web Services using the `WebServicesAssembler` `<stateless-session-ejb-service>` tag. This tag is included within a `<web-service>` tag in the configuration file. Add this tag to provide information required for generating a stateless session EJB Web Service.

[Table 4–3](#) shows the `<stateless-session-ejb-service>` sub-tags.

[Example 4–4](#) shows a complete `config.xml` file, including `<stateless-session-ejb-service>`.

Table 4–3 Stateless Session EJB Web Service Sub-Tags

Tag	Description
<code><ejb-name></code> <i>name</i> <code></ejb-name></code>	Specifies the <i>name</i> of the stateless session EJB. This tag is required
<code><ejb-resource></code> <i>resource</i> <code></ejb-resource></code>	This is a backwards compatibility tag. See Also: the top level <code><option name="source-path"></code> tag in Table 4–2 . This tag is optional
<code><path></code> <i>path</i> <code></path></code>	This is a backwards compatibility tag. See Also: the top level <code><option name="source-path"></code> tag in Table 4–2 . This tag is optional
<code><uri></code> <i>URI</i> <code></uri></code>	Specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the <i>URI</i> is appended to the <code><context></code> to specify the Web Service location. This tag is required.

Example 4–4 Sample Stateless Session EJB WebServicesAssembler Configuration File

```

<web-service>
  <display-name>EJB Web Services Demo</display-name>
  <destination-path>tmp/HelloService.ear</destination-path>
  <temporary-directory>tmp</temporary-directory>
  <context>/sejb_webservices</context>

  <stateless-session-ejb-service>
    <path>tmp/Hello.jar</path>
    <uri>/HelloService</uri>
    <ejb-name>HelloService</ejb-name>
  </stateless-session-ejb-service>
</web-service>

```

Adding WSDL and Client-Side Proxy Generation Tags

The `WebServicesAssembler` supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate WSDL files and client-side proxy files. You can use these tags to control whether the WSDL file and the client-side proxy are generated. Using these tags you can also specify that the generated WSDL file or a WSDL file that you write is packaged with the Web Service J2EE .ear.

A client-side developer either uses the WSDL file that is obtained from a deployed Web Service, or the client-side proxy that is generated from the WSDL to build an application that uses the Web Service.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Running WebServicesAssembler To Prepare Stateless Session EJB Web Services

After you create the `WebServicesAssembler` configuration file, you can generate a J2EE .ear file for the Web Service. The J2EE .ear file includes the stateless session EJB Web Service servlet configuration information.

Run the Oracle9iAS Web Services assembly tool, `WebServicesAssembler` as follows:

```
java -jar WebServicesAssembler.jar -config config_file
```

Where: *config_file* is the configuration file that contains the `<stateless-session-ejb-service>` tag.

See Also:

- ["Creating a Configuration File to Assemble Stateless Session EJB Web Services"](#) on page 4-8
- ["Running the Web Services Assembly Tool"](#) on page 9-2

Deploying Web Services Implemented as EJBs

After creating the .ear file containing a stateless session EJB, you can deploy the Web Service as you would any standard J2EE application stored in an .ear file (to run under OC4J).

See Also: *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library

Developing and Deploying Stored Procedure Web Services

This chapter describes how to write and deploy Oracle9iAS Web Services implemented as stateless PL/SQL Stored Procedures or Functions (**Stored Procedure Web Services**). Stored Procedure Web Services enable you to export, as services running under Oracle9iAS Web Services, PL/SQL procedures and functions that run on an Oracle database server.

This chapter covers the following topics:

- [Using Oracle9iAS Web Services with Stored Procedures](#)
- [Writing Stored Procedure Web Services](#)
- [Preparing Stored Procedure Web Services](#)
- [Deploying Stored Procedure Web Services](#)
- [Limitations for Stored Procedures Running as Web Services](#)

Using Oracle9iAS Web Services with Stored Procedures

This chapter shows sample code for writing Web Services implemented with stateless PL/SQL stored procedures or functions. The sample is based on a PL/SQL package representing a company that manages employees.

Oracle9iAS Web Services supplies a Servlet to access Java classes that support PL/SQL Stored Procedure Web Services. The Servlet handles requests generated by a Web Service client, runs the Java method that accesses the stored procedure that implements the Web Service, and returns results back to the Web Service client.

The Oracle database server supports procedures implemented in languages other than PL/SQL, including Java and C/C++. These stored procedures can be exposed as Web Services using PL/SQL interfaces.

See Also:

- [Chapter 2, "Oracle9iAS Web Services"](#)
- [Chapter 3, "Developing and Deploying Java Class Web Services"](#)
- [Chapter 6, "Developing and Deploying Document Style Web Services"](#)

Writing Stored Procedure Web Services

Writing Stored Procedure Web Services involves creating and installing a PL/SQL package on an Oracle database server that is available as a datasource to Oracle9iAS and generating a Java class that includes one or more methods to access the Stored Procedure.

The code for the sample Stored Procedure Web Service is supplied in the directory `$ORACLE_HOME/webservices/demo/basic/stored_procedure` on UNIX or in `%ORACLE_HOME%\webservices\demo\basic\stored_procedure` on Windows.

Create a Stored Procedure Web Service by writing and installing a PL/SQL Stored Procedure. To write and install a PL/SQL Stored Procedure, you need to use facilities independent of Oracle9iAS Web Services.

For example, to use the sample `COMPANY` package, first create and load the supplied package on the database server using the `create.sql` script. This script, along with several other required `.sql` scripts are in the `stored_procedure` directory. These scripts create several database tables and the sample `COMPANY` package.

When the Oracle database server is running on the local system, use the following command to create the sample PL/SQL package:

```
sqlplus scott/tiger @create
```

When the Oracle database server is not the local system, use the following command and include a connect identifier to create the sample PL/SQL package:

```
sqlplus scott/tiger@db_service_name @create
```

where *db_service_name* is the net service name for the Oracle database server.

See Also:

- ["Limitations for Stored Procedures Running as Web Services"](#) on page 5-12
- *PL/SQL User's Guide and Reference* in the Oracle Database Documentation Library
- *Oracle Net Services Administrator's Guide* in the Oracle Documentation Library

Preparing Stored Procedure Web Services

This section describes how to use the Oracle9iAS Web Services tool `WebServicesAssembler` to prepare a J2EE .ear file that supports using a PL/SQL procedure or function as a Stored Procedure Web Service.

This section contains the following topics:

- [Creating a Configuration File to Assemble Stored Procedure Web Services](#)
- [Running WebServicesAssembler With Stored Procedure Web Services](#)
- [Setting Up Datasources in Oracle9iAS Web Services \(OC4J\)](#)

Creating a Configuration File to Assemble Stored Procedure Web Services

The Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9iAS Web Services. This section describes how to create a configuration file to use to assemble a Stored Procedure Web Service. The Web Services assembly tool uses an XML configuration file that describes the Stored Procedure Web Service and produces a J2EE `.ear` file that can be deployed under Oracle9iAS Web Services.

Create `WebServicesAssembler` configuration file by adding the following:

- [Adding Web Service Top Level Tags](#)
- [Adding Stateless Stored Procedure Java Service Tags](#)
- [Adding WSDL and Client-Side Proxy Generation Tags](#)

Adding Web Service Top Level Tags

[Table 5–1](#) describes the top level `WebServicesAssembler` configuration file tags. Add these tags to provide top level information describing the PL/SQL Stored Procedure Web Service.

[Example 5–1](#) shows a complete `config.xml` file, including the top level tags.

Table 5–1 Top Level `WebServicesAssembler` Configuration Tags

Tag	Description
<code><context></code> <i>context</i> <code></context></code>	Specifies the context root of the Web Service. This tag is required.
<code><datasource-JNDI-name></code> <code></datasource-JNDI-name></code>	Specifies the datasource associated with the Web Service.
<code><description></code> <i>description</i> <code></description></code>	Provides a simple description of the Web Service. This tag is optional.
<code><destination-path></code> <i>dest_path</i> <code></destination-path></code>	Specifies the name of the generated J2EE <code>.ear</code> file output. The <i>dest_path</i> specifies the complete path for the output file. This tag is required.
<code><display-name></code> <i>disp_name</i> <code></display-name></code>	Specifies the Web Service display name. This tag is optional.

Table 5–1 (Cont.) Top Level `WebServicesAssembler` Configuration Tags

Tag	Description
<code><option name="source-path"> path </option></code>	Includes a specified file in the output .ear file. Use this option to include Java resources. The <i>path</i> specifies the path to the file to include.
<code><stateless-stored-procedure-java-service sub-tags </stateless-stored-procedure-java-service></code>	Use this tag to add stateless stored procedure Web Services. See Table 5–2 and Table 5–4 for a description of valid <i>sub-tags</i> .
<code><temporary-directory temp_dir </temporary-directory></code>	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding Stateless Stored Procedure Java Service Tags

There are two ways to develop Stored Procedure Web Services using the `WebServicesAssembler`:

- [Adding Stateless Stored Procedure Java Service Using Jar Generation](#)
- [Adding Stateless Stored Procedure Java Services Using a Pre-generated Jar](#)

Note: Most Stored Procedure Web Service developers use the Jar generation technique for assembling the Web Service J2EE .ear file. Only use the pre-generated Jar technique for creating a J2EE .ear when you have a pre-generated Jar file containing JPublisher generated classes.

Adding Stateless Stored Procedure Java Service Using Jar Generation

Using a configuration file that includes the `<jar-generation>` tag specifies Oracle Database Server connection information that allows the `WebServicesAssembler` to run JPublisher to generate the classes to support the Stored Procedure Web Service. The JPublisher generated classes support accessing

the PL/SQL procedure or function and also includes classes for mapping Java types to PL/SQL types. The `WebServicesAssembler` packages the generated classes into a Jar file that is assembled with the Stored Procedure Web Service.

[Table 5–2](#) describes the `<stateless-stored-procedure-java-service>` `WebServicesAssembler` configuration file tags used when creating a configuration file that uses Jar generation to create a Stored Procedure Web Service. The `<stateless-stored-procedure-java-service>` tag is included within a `<web-service>` tag in the configuration file. Add this tag to provide information required for generating the Stored Procedure Web Service J2EE .ear file.

[Table 5–3](#) describes the sub-tags for `<jar-generation>` within the `<stateless-stored-procedure-java-service>` tag. The `<jar-generation>` tags provide information to the `WebServicesAssembler` so that it can run `JPublisher` to generate the Java classes for the Stored Procedure Web Service. The `WebServicesAssembler` then uses these classes to generate the Jar file that provides Java mappings for the stored procedure or function.

[Example 5–1](#) shows a complete `config.xml` file, including the Stored Procedure Web Service tags shown in [Table 5–2](#) and [Table 5–3](#).

Table 5–2 Stateless Stored Procedure Sub-Tags (Using Jar Generation)

Tag	Description
<code><database-JNDI-name></code> <code>source_JNDI_name</code> <code></database-JNDI-name></code>	This tag specifies the JNDI name of the backend database. The <code>data-sources.xml</code> OC4J configuration file describes the database server source associated with the specified <code>source_JNDI_name</code> .
<code><jar-generation></code> <code>sub-tags</code> <code></jar-generation></code>	Table 5–3 describes the supported <i>sub-tags</i> for <code><jar-generation></code> . Example: <pre> <jar-generation> <schema>scott/tiger</schema> <db-url>jdbc:oracle:thin:@system1:1521:orcl</db-url> <prefix>sp.company</prefix> <db-pkg-name>Company</db-pkg-name> </jar-generation> </pre>
<code><uri></code> <code>URI</code> <code></uri></code>	This tag specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the <code>URI</code> is appended to the <code><context></code> to specify the Web Service location.

Table 5–3 Stateless Stored Procedure <jar-generation> Sub-Tags

Tag	Description
<pre><db-pkg-name> pkg_name </db-pkg-name></pre>	<p>Where <i>pkg_name</i> is the name of the PL/SQL package to export.</p> <p>This is required when <jar-generation> is included.</p>
<pre><db-url> url_path </db-url></pre>	<p>Where <i>url_path</i> is the database connect string for the Oracle database server with the specified package to export. The <schema> and <db-url> are combined to connect to the database which contains the stored procedures to be exported.</p> <p>This is required when <jar-generation> is included.</p> <p>Example:</p> <pre><db-url>jdbc:oracle:thin:@system1.us.oracle.com:1521:tv1</db-url></pre>
<pre><method-name> method </method-name></pre>	<p>Where <i>method</i> is the name of the PL/SQL method to export.</p> <p>This tag is optional. Including multiple <method> tags is valid. In this case the specified methods are exported.</p> <p>Without this tag, all methods within the package are exported. If the specified method is overloaded, then all variations of the method are exported.</p>
<pre><prefix> prefix </prefix></pre>	<p>Where <i>prefix</i> is the Java package prefix for generated classes.</p> <p>By default, the PL/SQL package is generated into a Java class in the default Java package.</p> <p>This tag is optional.</p> <p>Example:</p> <pre><prefix>sp.company</prefix></pre>
<pre><schema> user_name/password </schema></pre>	<p>This tag includes the Database Server <i>user_name/password</i>:</p> <p>where:</p> <p><i>user_name</i> is the database user name.</p> <p><i>password</i> is the database password for the specified user name.</p> <p>This tag is required when <jar-generation> is included.</p> <p>Example:</p> <pre><schema>scott/tiger</schema></pre>

Example 5–1 Sample WebServicesAssembler Configuration File For Stored Procedure Using <jar-generation> Tag

```

<web-service>
  <display-name>Web Services Example</display-name>
  <description>Java Web Service Example</description>
  <!-- Specifies the resulting web service archive will be stored in ./spexample.ear -->
  <destination-path>./spexample.ear</destination-path>
  <!-- Specifies the temporary directory that web service assembly tool can create temporary files. -->
  <temporary-directory>/tmp</temporary-directory>
  <!-- Specifies the web service will be accessed in the servlet context named "/webservices". -->
  <context>/webservices</context>
  <!-- Specifies the web service will be stateless -->

  <stateless-stored-procedure-java-service>
    <jar-generation>
      <schema>scott/tiger</schema>
      <db-url>jdbc:oracle:thin:@system1:1521:orcl</db-url>
      <prefix>sp.company</prefix>
      <db-pkg-name>Company</db-pkg-name>
    </jar-generation>
    <!-- Specifies the web service will be accessed in the uri named
         "statelessSP" within the servlet context. -->
    <uri>/statelessSP</uri>
    <database-JNDI-name>/jdbc/OracleDataSource</database-JNDI-name>
  </stateless-stored-procedure-java-service>
</web-service>

<wsdl-gen>
  <wsdl-dir>wsdl</wsdl-dir>
  <!--force 'true' will write over existing wsdl -->
  <option name="force">true</option>
  <!-- change this to point to your soap servers http listener -->
  <option name="httpServerURL">http://localhost:8888</option>
</wsdl-gen>

<proxy-gen>
  <proxy-dir>proxy</proxy-dir>
  <!-- include-source 'true' will create an additional jar with only the proxy source-->
  <option name="include-source">true</option>
</proxy-gen>
</web-service>

```

Adding Stateless Stored Procedure Java Services Using a Pre-generated Jar

Using a configuration file that specifies the stored procedure <class-name> and <interface-name> assembly options when a pre-generated Jar file that includes the required classes to support the Web Service is available. The <class-name>

and `<interface-name>` tags specified in a configuration file support using a previously generated Jar file that contains the Java classes that provide a mapping between the PL/SQL procedure or function and the Web Service.

Table 5-4 describes the `<stateless-stored-procedure-java-service>` `WebServicesAssembler` configuration file tags used when creating a configuration file that uses a pre-generated Jar file to create a Stored Procedure Web Service. The `<stateless-stored-procedure-java-service>` tag is included within a `<web-service>` tag in the configuration file. Add this tag to provide information required for generating the Stored Procedure Web Service J2EE .ear file.

The `<class>` and `<interface>` tags that are added to the `<stateless-stored-procedure-java-service>` only when using a pre-generated Jar file.

Table 5-4 Stateless Stored Procedure Sub-Tags (Using Pre-generated Jar File)

Tag	Description
<code><class-name></code> <i>class</i> <code></class-name></code>	<p>The Stored Procedure Web Services Servlet definition requires a <code><param-name></code> with the value class-name and a corresponding <code><param-value></code> set to the fully qualified name of the Java class that accesses the PL/SQL Web Service implementation.</p> <p>You need to use the configuration file <code><class-name></code> tag to supply the class name for this parameter; you can find the class name in the Jar file you provide that is specified in the top level <code><option name="source-path"></code> tag.</p>
<code><database-JNDI-name></code> <i>source_JNDI_name</i> <code></database-JNDI-name></code>	<p>This tag specifies the JNDI name of the backend database.</p> <p>The <code>data-sources.xml</code> OC4J configuration file describes the database server source associated with the specified <i>source_JNDI_name</i>.</p>
<code><interface-name></code> <i>interface</i> <code></interface-name></code>	<p>A Stored Procedure Web Services Servlet definition requires a <code><param-name></code> with the value interface-name, and a corresponding <code><param-value></code> set to the fully qualified name of the Java interface that specifies the methods to include in the stored procedure Web Service.</p> <p>The <code><interface-name></code> tag provides the name of the interface that tells the Web Service Servlet generation code which methods should be exposed as Web Services. You can find the interface name in the Jar file you provide that is specified in the top level <code><option name="source-path"></code> tag.</p>

Table 5-4 (Cont.) Stateless Stored Procedure Sub-Tags (Using Pre-generated Jar File)

Tag	Description
<code><java-resource></code> <i>resource</i> <code></java-resource></code>	<p>This is a backwards compatibility tag.</p> <p>See Also: the top level <code><option name="source-path"></code> tag in Table 5-1.</p> <p>This tag is optional.</p> <p>The Stored Procedure pre-generated Jar file should be specified using the <code><java-resource></code> tag. The class specified with the <code><class-name></code> tag and the interface specified with the <code><interface-name></code> tag must exist in the resource specified in the <code><java-resource></code> tag(s).</p>
<code><uri></code> <i>URI</i> <code></uri></code>	<p>This tag specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the <i>URI</i> is appended to the <code><context></code> to specify the Web Service location.</p>

See Also:

- ["Adding Stateless Stored Procedure Java Service Using Jar Generation"](#) on page 5-5
- *Oracle9i JPublisher User's Guide* in the Oracle Database Documentation Library

Adding WSDL and Client-Side Proxy Generation Tags

The `WebServicesAssembler` configuration file supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate Web Service description WSDL files and client-side proxy files. You can add these tags to control whether the WSDL file and the client-side proxy are generated. You can also specify that the WSDL file be assembled with the Stored Procedure Style Web Service J2EE .ear. A client-side developer can then use the WSDL file that is obtained from the deployed Web Service to build an application that uses the Web Service.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Running WebServicesAssembler With Stored Procedure Web Services

After you create the `WebServicesAssembler` configuration file, you can generate a J2EE .ear file for the Stored Procedure Web Service. The J2EE .ear file includes Stored Procedure Web Service servlet configuration information, including the file `web.xml`, and JPublisher generated classes (the `WebServicesAssembler` collects

the JPublisher generated classes into a single Jar file that it includes in the generated J2EE .ear).

Run the Oracle9iAS Web Services assembly tool, `WebServicesAssembler` as follows:

```
java -jar WebServicesAssembler.jar -config my_pl_service_config
```

Where: `my_pl_service_config` is the configuration file that contains the `<stateless-stored-procedure-java-service>` tag.

See Also:

- ["Creating a Configuration File to Assemble Stored Procedure Web Services"](#) on page 5-4
- ["Running the Web Services Assembly Tool"](#) on page 9-2

Setting Up Datasources in Oracle9iAS Web Services (OC4J)

To add Web Services based on PL/SQL Stored Procedures you need to set up data sources in OC4J by configuring `data-sources.xml`. Configuring the `data-sources.xml` file points OC4J to a database. The database should contain PL/SQL Stored Procedure packages that implement a Stored Procedure Web Service.

A single database connection is created when OC4J initializes a Web Services Servlet instance. The resulting database connection is destroyed when OC4J removes the Web Services Servlet instance. Each Stored Procedure Web Services Servlet implements a single threaded model. As a result, any Web Services Servlet instance can only service a single client's database connection requests at any given time. OC4J pools the Web Services Servlet instances and assigns instances to Oracle9iAS Web Services clients.

Every invocation of a PL/SQL Web Service is implicitly a separate database transaction. It is not possible to have multiple service method invocations run within a single database transaction. When such semantics are required, the user must write a PL/SQL procedure that internally invokes other procedures and functions, and then expose the new procedure as another method in a Stored Procedure Web Service (but Oracle9iAS Web Services does not provide explicit support or tools to do this).

See Also: *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library

Deploying Stored Procedure Web Services

After creating the J2EE .ear file containing the Stored Procedure Web Service configuration, class, Jar, and support files you can deploy the Web Service as you would any standard J2EE application stored in a J2EE .ear file (to run under OC4J).

See Also: *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library

Limitations for Stored Procedures Running as Web Services

This section covers the following topics:

- [Supported Stored Procedure Features for Web Services](#)
- [Unsupported Stored Procedure Features for Web Services](#)
- [Database Server Release Limitation for Boolean Use in Oracle PLSQL Web Services](#)

Supported Stored Procedure Features for Web Services

Stored Procedure Web Services support the following PL/SQL features:

1. PL/SQL stored procedures, including both procedures and functions
2. IN parameter modes
3. Packaged procedures only (top-level procedures must be wrapped in a package before they can be exported as a Web Service)
4. Overloaded procedures. However, if two different PL/SQL types map to the same Java type during the Java class generation step, there may be errors reported when the PL/SQL package is exported; these errors may be fixed by avoiding the overloading in the PL/SQL parameters, or by writing a new dummy package which does not contain the offending overloaded procedures.

JPublisher may map multiple PL/SQL types into the same Java type. For example, different PL/SQL number types may all map to Java int. This means that methods that were considered overloaded in PL/SQL are no longer overloaded in Java. If this is an issue, the user should wrap their PL/SQL code in a new PL/SQL package that does not contain these ambiguity problems.

5. Simple PL/SQL types

The following simple types are supported. NULL values are supported for all of the simple types listed below, except NATURALN and POSITIVEN.

The JPublisher documentation provides full details on the mappings for these simple types.

VARCHAR2 (STRING, VARCHAR), LONG, CHAR (CHARACTER), NUMBER (DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT), PLS_INTEGER, BINARY_INTEGER (NATURAL, NATURALN, POSITIVE, POSITIVEN), BOOLEAN

6. User-defined Object Types.

See Also:

Oracle9i JPublisher User's Guide in the Oracle Database Documentation Library

Unsupported Stored Procedure Features for Web Services

Stored Procedure Web Services impose the following limitations on PL/SQL functions and procedures:

1. Only procedures and functions within a PL/SQL package are exported as Web Services. Top-level stored procedures must be wrapped inside a package; ADT methods must be wrapped into package-level methods with a default "this" reference.
2. OUT and IN OUT parameter modes are not supported.
3. NCHAR and related types are not supported.
4. JPublisher and Oracle9iAS Web Services does not provide comprehensive support for LOB types.

If your PL/SQL procedures use LOB types as input/output types, then the generated Java translation may not work in all cases. If you see an error, the offending procedures will have to be rewritten before the PL/SQL package can be exported as Stored Procedure Web Services.

5. Due to a bug in JPublisher, many integer numeric types are translated into `java.math.BigDecimal` instead of the Java scalar types--the workaround for this bug is to temporarily use `java.math.BigDecimal` as the argument and return types.

6. JPublisher translates almost all PL/SQL types to Java types. The deployment tools for Stored Procedure Web Services generate "jdbc" style for builtin, number, and lob types, and the "oracle" style for user types (in the "customdatum" compatibility mode). Check the JPublisher documentation for full details of these styles, and for the caveats associated with them.

See Also:

Oracle9i JPublisher User's Guide in the Oracle Database Documentation Library

Database Server Release Limitation for Boolean Use in Oracle PLSQL Web Services

Using a Oracle Database Server of Release 9.2.0.1 or earlier, or with a Database Server that is not Java-enabled, then you must install the SYS.SQLJUTIL package into the SYS schema to support PL/SQL BOOLEAN arguments.

The PLSQL script that defines this package is located at the following location on Unix:

```
${ORACLE_HOME}/sqlj/lib/sqljutil.sql
```

On Windows systems, this script is located at the following location:

```
%ORACLE_HOME%\sqlj\lib\sqljutil.sql
```

Developing and Deploying Document Style Web Services

This chapter describes the procedures you use to write and deploy Oracle9iAS Web Services that handle document style messages and are implemented as Java classes.

This chapter covers the following topics:

- [Using Document Style Web Services](#)
- [Writing Document Style Web Services](#)
- [Preparing Document Style Web Services](#)
- [Deploying Document Style Web Services](#)

Using Document Style Web Services

This chapter describes Document Style Web Services that are implemented with Java classes and describes the difference between writing stateful and stateless Document Style Java Web Services. The sample code for Document Style Web Services is supplied with Oracle9iAS Web Services in the `stateless` and `stateful` directories in the directory `$ORACLE_HOME/webservices/demo/basic/java_doc_services` on UNIX or in `%ORACLE_HOME%\webservices\demo\basic\java_doc_services` on Windows.

Oracle9iAS supplies Servlets to access the Java classes which you write to implement a Web Service. The Servlets handle messages generated by Web Services clients and dispatch them to run the Java methods that implement Document Style Web Services. After a Web Service is deployed, when a client makes a service request (uses a service) the Oracle9iAS Web Services runtime, using an automatically generated Web Services Servlet invokes the methods that you implement to support the Document Style Web Service.

See Also:

- [Chapter 3, "Developing and Deploying Java Class Web Services"](#)
- [Chapter 4, "Developing and Deploying EJB Web Services"](#)
- [Chapter 7, "Developing and Deploying JMS Web Services"](#)
- [Chapter 8, "Building Clients that Use Web Services"](#)

Writing Document Style Web Services

Writing Document Style Java Web Services involves building a Java class that includes one or more methods using supported method signatures; the java class includes methods that either handle an incoming message or return an outgoing message.

This section covers the following topics:

- [Supported Method Signatures for Document Style Web Services](#)
- [Writing Stateless and Stateful Document Style Web Services](#)
- [Writing Classes and Interfaces for Document Style Web Services](#)

Supported Method Signatures for Document Style Web Services

Table 6–1 shows the supported method signatures for Document Style Web Services. The Oracle9iAS Web Services runtime verifier rejects Document Style Web Services that do not conform to the method signatures listed in Table 6–1.

The `Element` input parameter and `Element` return value shown in the method signatures in Table 6–1 must conform to the Document Object Model (DOM) as specified by the W3C (`org.w3c.dom.Element`).

Table 6–1 Supported Method Signatures for Document Style Java Web Services

Method Signature	Description
<code>public Element op_Name(Element e_name)</code>	The method <code>op_Name</code> is a Document Style Web Service operation implemented as a Java method that takes an <code>Element e_name</code> as an input parameter and returns an <code>Element</code> .
<code>public Element get_Name()</code>	The method <code>get_Name</code> is a Document Style Web Service operation implemented as a Java method that takes no input parameters and returns an <code>Element</code> .
<code>public void set_Name(Element e_name)</code>	The method <code>set_Name</code> is a Document Style Web Service operation implemented as a Java method that takes an <code>Element e_name</code> as an input parameter and returns nothing.

Passing Null Values for Document Style Web Services

A `null` could be passed as an input `Element` or as the `Element` that the Document Style Web Service returns.

Arrays of Elements

Oracle9iAS Web Services does not support `Element []` (arrays of `org.w3c.dom.Element`).

See Also:

- ["Handling Messages for Document Style Web Services"](#) on page 6-9
- <http://www.w3.org/DOM/> for information on the W3C Document Object Model (DOM)

Writing Stateless and Stateful Document Style Web Services

Oracle9iAS Web Services supports stateful and stateless implementations for Document Style Java classes running as Web Services. For a stateful Java implementation, Oracle9iAS Web Services allows a single Java instance to serve the Web Service requests from an individual client.

For a stateless Java implementation, Oracle9iAS Web Services creates multiple instances of the Java class in a pool, any one of which may be used to service a request. After servicing the request, the object is returned to the pool for use by a subsequent request.

Note: It is the job of the Web Services developer to make the design decision to implement a stateful or stateless Web Service. When packaging Web Services, stateless and stateful Web Services are handled slightly differently. This chapter describes these differences in the section, ["Preparing Document Style Web Services"](#) on page 6-9.

Note: Deploying a stateful Java implementation class as a stateless Document Style Web Service could yield unpredictable results.

Writing Classes and Interfaces for Document Style Web Services

Developing a Document Style Java Web Service consists of the following steps:

- [Defining Methods in a Document Style Web Service](#)
- [Defining an Interface for Explicit Method Exposure](#)
- [Handling Messages for Document Style Web Services](#)

Defining Methods in a Document Style Web Service

Create a Document Style Web Service by writing or supplying a Java class with methods that are deployed as a Document Style Web Service. The `stateful` and `stateless` sample directories contain sample stateless and stateful Document Style Web Services. In the `src` directories, the file `StatefulDocImpl.java` provides the implementation of the sample stateful Java service and `StatelessDocImpl.java` provides the implementation of the stateless Document Style Web Service. These examples use interface classes; the use of interface classes is optional when implementing Document Style Web Services.

A Java class that implements a Document Style Web Service has the following limitations:

- The Java class should define public methods that conform to the method signatures shown in [Table 6-1](#). If you use an interface, then only the public methods specified in the interface need to conform to the method signature restrictions. If you do not include an interface, then all the public methods in the class must conform to the method signature restrictions shown in [Table 6-1](#).
- The Java class implementation must include a public constructor that takes no arguments.

There are very few restrictions on what actions a Document Style Java class based web service can perform. At a minimum, the service performs some action to handle an incoming message (`Element`) or to generate an outgoing message (`Element`).

The `StatelessDoc` Web Service sample is implemented with `StatelessDocImpl`, a public class and the interface `StatelessDoc`. The `StatelessDocImpl` class defines two public methods: `displayElement()`, that displays the incoming message on the server where the web service runs, and

`processElement()`, that takes an incoming message and returns a transformed message to the client. The private method `applyXSLtoXML()` is a helper method that transforms the incoming message, as specified in the `converter.xml` file.

[Example 6-1](#) shows the method signatures for the `StatelessDocImpl` class (see the `src` directory to view the complete source code for `StatelessDocImpl`).

Example 6-1 Defining Java Methods for a Stateless Document Style Web Service

```
import org.w3c.dom.*;
import oracle.xml.parser.v2.*;
import java.io.*;

public class StatelessDocImpl implements StatelessDoc
{
    public StatelessDocImpl()
    { }

    // Display the Element that was sent
    public void displayElement(Element e)
    { }

    //method to process the input xml doc
    public Element processElement(Element e)
    { }

    /**
     * This Method Transforms an XML Document into another using the provided
     * Style Sheet: converter.xml. Note : This Method makes use of XSL
     * Transformation capabilities of Oracle XML Parser Version 2.0
     */
    private Element applyXSLtoXML(Element e)
        throws Exception
    {}
}
```

The `StatfulDoc` Web Service sample is implemented with `StatefulDocImpl`, a public class and the interface `StatefulDoc`. The `StatefulDocImpl` class defines two public methods: `startShopping()` that initializes the state of the customer information and `makePurchase()`, that modifies the state of the customer

information and returns the updated information to the client. The private method `processElement()` is a helper method that processes the customer's XML element representing a purchase and returns the updated XML element.

[Example 6-2](#) shows the method signatures for the `StatefulDoc` class (see the `src` directory to view the complete source code for `StatefulDocImpl`).

Example 6-2 Defining Java Methods for a Stateful Document Style Web Service

```
import org.w3c.dom.*;
import oracle.xml.parser.v2.*;

public class StatefulDocImpl implements StatefulDoc
{
    private Element e ;
    public void startShopping(Element e)
    {
    }
    public Element makePurchase()
    {
    }
    private void processElement(Element e) {
    }
}
```

Defining an Interface for Explicit Method Exposure

Oracle9iAS Web Services allows you to limit the methods you expose as Document Style Web Services by supplying a public interface. To limit the methods exposed in a Web Service, include a public interface that lists the method signatures for the methods that you want to expose. [Example 6-3](#) shows an interface for the methods in the class `StatelessDocImpl`. [Example 6-4](#) shows an interface for the methods in the class `StatefulDocImpl`.

When an interface is included with a Document Style Web Service, then only the public methods specified in the interface need to conform to the method signature restrictions shown in [Table 6-1](#). If you do not include an interface, then all the public methods in the class must conform to the method signature restrictions. Using an interface, for example `StatelessDoc` shown in [Example 6-3](#), only the methods with the specified method signatures are exposed when the Java class is prepared and deployed as a Document Style Web Service.

Use a Document Style Web Service interface for the following purposes:

1. To limit the exposure of methods to a subset of the public methods within a class.
2. To expand the set of methods that are exposed to include methods within the superclass of a class.
3. To limit the exposure of methods to a subset of the public methods within a class, where the subset contains only the methods that use supported method signatures. [Table 6–1](#) lists the supported signatures for Java methods that implement Document Style Web Services.

Example 6–3 Using a Public Interface to Expose Stateless Java Services

```
import org.w3c.dom.*;

public interface StatelessDoc
{
    //method to display the element
    public void displayElement(Element e) ;

    //method to process the input xml doc
    public Element processElement(Element e) ;
}
```

Example 6–4 Using a Public Interface to Expose Stateful Java Services

```
import org.w3c.dom.Element;

// Interface that implements getElement and setElement
public interface StatefulDoc {

    // Set the Element
    public void startShopping(Element e);

    // Retrieve the element that was set
    public Element makePurchase();
}
```

Handling Messages for Document Style Web Services

It is entirely up to the Web Service developer to determine the processing that occurs for messages associated with a Document Style Web Service.

The message associated with a Document Style Web Service is specified in the `Element` parameter or the `Element` return value associated with the Document Style Web Service. It is the Document Style Web Service developer's job to process or generate messages. The only limitation on Document Style Web Service messages is that the `Element` must conform to must conform to the Document Object Model (DOM) as specified by the W3C (`org.w3c.dom.Element`).

A Document Style Web Service implementation or the client that uses a service may need to supports null values, since a null could be passed as an input `Element` or as the `Element` that is returned.

For example, the following is valid for a Document Style Web Service implementation:

```
Element get_op () {  
    return null;  
}
```

Preparing Document Style Web Services

This section describes how to use the Oracle9iAS Web Services tool `WebServicesAssembler` to prepare a J2EE .ear file for a stateless and stateful Document Style Web Service implemented as Java classes.

To deploy a Java class that implements a Document Style Web Service, you need to assemble a J2EE .ear file that includes the deployment descriptors for the Oracle9iAS Web Services Servlet and the Java classes that supply the Java implementation. A Web Service implemented with Java classes includes a .war file that provides configuration information for the Web Services Servlet running under Oracle9iAS Containers for J2EE (OC4J). This section describes the procedures you use to create a configuration file to use with the `WebServicesAssembler`.

This section contains the following topics:

- [Creating a Configuration File to Assemble Document Style Web Services](#)
- [Running WebServicesAssembler With Document Style Web Services](#)

Creating a Configuration File to Assemble Document Style Web Services

The Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9iAS Web Services. This section describes how to create a configuration file to use to assemble a Document Style Web Service. The Web Services assembly tool uses an XML configuration file that describes the Document Style Web Service. The `WebServicesAssembler` uses the configuration file to produce a J2EE .ear file that can be deployed under Oracle9iAS Web Services.

Create `WebServicesAssembler` configuration file by adding the following:

- [Adding Web Service Top Level Tags](#)
- [Adding Java Service Tags with Document Message Style Specified](#)
- [Adding WSDL and Client-Side Proxy Generation Tags](#)

Adding Web Service Top Level Tags

[Table 6–2](#) describes the top level `WebServicesAssembler` configuration file tags. Add these tags to provide top level information describing the Document Style Web Service.

[Example 6–5](#) shows a complete stateless sample configuration file. [Example 6–6](#) shows a complete stateful sample configuration file. The `stateless` and `stateful` directories in the `java_doc_services` demo directory contain the sample `config.xml` files.

Table 6–2 Top Level `WebServicesAssembler` Configuration Tags

Tag	Description
<code><context></code>	Specifies the context root of the Web Service.
<code>context</code>	This tag is required.
<code></context></code>	
<code><datasource-JNDI-name></code>	Specifies the datasource associated with the Web Service.
<code></datasource-JNDI-name></code>	
<code><description></code>	Provides a simple description of the Web Service.
<code>description</code>	This tag is optional.
<code></description></code>	
<code><destination-path></code>	Specifies the name of the generated J2EE .ear file output. The <i>dest_path</i> specifies the complete path for the output file.
<code>dest_path</code>	This tag is required.
<code></destination-path></code>	

Table 6–2 (Cont.) Top Level `WebServicesAssembler` Configuration Tags

Tag	Description
<code><display-name></code> <i>disp_name</i> <code></display-name></code>	Specifies the Web Service display name. This tag is optional.
<code><option</code> <code>name=source-path"></code> <i>path</i> <code><option></code>	Includes a specified file in the output .ear file. The <i>path</i> specifies the path to the file to include.
<code><stateless-java-service></code> <i>sub-tags</i> <code></stateless-java-service></code>	Use this tag to add a Document Style Web Services that defines a stateless service. See Table 6–3 for a description of valid <i>sub-tags</i> .
<code><stateful-java-service></code> <i>sub-tags</i> <code></stateful-java-service></code>	Use this tag to add a Document Style Web Services that defines a stateful service. See Table 6–3 for a description of valid <i>sub-tags</i> .
<code><temporary-directory></code> <i>temp_dir</i> <code></temporary-directory></code>	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding Java Service Tags with Document Message Style Specified

The Document Style Web Service developer determines if the service is stateful or stateless. The configuration file includes different tags depending on the type of the service. This section covers the tags for both cases, including:

- [Adding Stateful Document Style Java Service Tags](#)
- [Adding Stateless Document Style Java Service Tags](#)

Table 6–3 Java Service `WebServicesAssembler` Configuration Tags - Document Style

Tag	Description
<code><class-name></code> <i>class</i> <code></class-name></code>	<p>The Document Style Web Service definition requires at least one <code><class-name></code> tag. The <i>value</i> specifies the name of the Java class that provides the Document Style Web Service implementation.</p> <p>This tag is required.</p>
<code><interface-name></code> <i>interface</i> <code></interface-name></code>	<p>A Document Style Web Service configuration file supports the optional <code><interface-name></code> tag. The corresponding <i>interface</i> value supplied specifies the name of the Java interface that lists the methods to include in the Document Style Web Service.</p> <p>This tag is optional.</p>
<code><java-resource></code> <i>resource</i> <code></java-resource></code>	<p>This tag supports adding a Java <i>resource</i>. This specifies the location of the java resources to include in the Document Style Web Service.</p> <p>Include multiple <code><java-resource></code> tags to include multiple Java resources.</p> <p>This tag is optional</p>
<code><message-style></code> <i>doc</i> <code></message-style></code>	<p>When defining a Document Style Web Service, you must include the <code><message-style></code> tag and specify the value <i>doc</i>.</p> <p>Valid Values: <i>doc</i>, <i>rpc</i></p> <p>This tag is required for Document Style Web Services.</p> <p>Default value: <i>rpc</i> (when the <code><message-style></code> tag is not supplied)</p>
<code><scope></code> <code></scope></code>	<p>The <code><scope></code> tag only applies for stateful services. Use this tag only within the <code><stateful-java-service></code> tag.</p> <p>This tag is optional.</p> <p>Valid Values: <i>application</i>, <i>session</i></p> <p>Default Value: <i>session</i></p>
<code><session-timeout></code> <i>value</i> <code></session-timeout></code>	<p>This optional parameter only applies for stateful services. Use this tag only within the <code><stateful-java-service></code> tag.</p> <p>Specify <i>value</i> with an integer that defines the timeout for the session timeout. session. The default value for the session timeout for stateful Java sessions where no session timeout is specified is 60 seconds.</p> <p>This tag is optional.</p>
<code><uri></code> <i>URI</i> <code></uri></code>	<p>This tag specifies servlet mapping pattern for the Servlet that implements the Document Style Web Service. The path specified as the <i>URI</i> is appended to the <code><context></code> to specify the Document Style Web Service location.</p> <p>This tag is optional.</p>

Adding Stateful Document Style Java Service Tags

[Table 6–3](#) describes the `<stateful-java-service>` `WebServicesAssembler` configuration file tags. Use these tags when creating a configuration file for a stateful Document Style Web Service.

[Example 6–5](#) shows a complete `config.xml` file, including the stateful Document Style Web Service tags.

Adding Stateless Document Style Java Service Tags

[Table 6–3](#) describes the `<stateless-java-service>` `WebServicesAssembler` configuration file tags to use when creating a stateful Document Style Web Service. The `<stateless-java-service>` tag is included within a `<web-service>` tag in the configuration file. Add this tag to provide information required for generating a stateless Document Style Web Service J2EE .ear file.

[Example 6–6](#) shows a complete `config.xml` file, including the stateless Document Style Web Service tags.

Note: Deploying a stateful Java implementation class as a stateless Document Style Web Service could yield unpredictable results.

Adding WSDL and Client-Side Proxy Generation Tags

The `WebServicesAssembler` configuration file supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate Web Service description WSDL files and client-side proxy files. You can add these tags to control whether the WSDL file and the client-side proxy are generated. You can also specify that the WSDL file be assembled with the Document Style Web Service .ear. A client-side developer can then obtain the WSDL file from the deployed Web Service and use it to build an application.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Example 6-5 Sample Stateful Java WebServicesAssembler Configuration File for a Document Style Web Service

```

<web-service>
  <display-name>Stateful Java Document Web Service</display-name>
  <description>Stateful Java Document Web Service Example</description>
  <!-- Specifies the resulting web service archive will be stored in ./docws.ear -->
  <destination-path>./docws.ear</destination-path>
  <!-- Specifies the temporary directory that web service assembly tool can create temporary files. -->
  <temporary-directory>./temp</temporary-directory>
  <!-- Specifies the web service will be accessed in the servlet context named "/docws". -->
  <context>/statefuldocws</context>

  <!-- Specifies the web service will be stateful -->

  <stateful-java-service>
    <interface-name>StatefulDoc</interface-name>
    <class-name>StatefulDocImpl</class-name>
    <!-- Specifies the web service will be accessed in the uri named "/docService" within the servlet
context. -->
    <uri>/docservice</uri>
    <!-- Specifies the location of Java class files ./classes -->
    <java-resource>./classes</java-resource>
    <!-- Specifies that it uses document style SOAP messaging -->
    <message-style>doc</message-style>
  </stateful-java-service>

  <!-- generate the wsdl -->
  <wsdl-gen>
<wsdl-dir>wsdl</wsdl-dir>
  <!-- over-write a pregenerated wsdl , turn it 'false' to use the pregenerated wsdl-->
  <option name="force">true</option>
  <option name="httpServerURL">http://localhost:8888</option>
  </wsdl-gen>

  <!-- generate the proxy -->

  <proxy-gen>
<proxy-dir>proxy</proxy-dir>
  <option name="include-source">true</option>
  </proxy-gen>
</web-service>

```

Example 6-6 Sample Stateless Java WebServicesAssembler Configuration File for a Document Style Web Service

```

<web-service>
  <display-name>Stateless Java Document Web Service</display-name>
  <description>Stateless Java Document Web Service Example</description>
  <!-- Specifies the resulting web service archive will be stored in ./statelessdocws.ear -->
  <destination-path>./statelessdocws.ear</destination-path>
  <!-- Specifies the temporary directory that web service assembly tool can create temporary files. -->
  <temporary-directory>./temp</temporary-directory>
  <!-- Specifies the web service will be accessed in the servlet context named "/statelessdocws". -->
  <context>/statelessdocws</context>
  <!-- to package the stylesheet to format input xml -->
  <option name="source-path">converter.xml</option>

  <!-- Specifies the web service will be stateless -->

  <stateless-java-service>
    <interface-name>StatelessDoc</interface-name>
    <class-name>StatelessDocImpl</class-name>
    <!-- Specifies the web service will be accessed in the uri named "/docService" within the servlet
context. -->
    <uri>/docservice</uri>
    <!-- Specifies the location of Java class files ./classes -->
    <java-resource>./classes</java-resource>
    <!-- Specifies that it uses document style SOAP messaging -->
    <message-style>doc</message-style>
  </stateless-java-service>

  <!-- generate the wsdl -->
  <wsdl-gen>
<wsdl-dir>wsdl</wsdl-dir>
    <!-- over-write a pregenerated wsdl , turn it 'false' to use the pregenerated wsdl-->
    <option name="force">true</option>
    <option name="httpServerURL">http://localhost:8888</option>
  </wsdl-gen>

  <!-- generate the proxy -->
  <proxy-gen>
<proxy-dir>proxy</proxy-dir>
    <option name="include-source">true</option>
  </proxy-gen>
</web-service>

```

Running WebServicesAssembler With Document Style Web Services

After you create the `WebServicesAssembler` configuration file, you can generate a J2EE `.ear` file for the Document Style Web Service. The J2EE EAR file includes Document Style Web Service servlet configuration information, including the generated file `web.xml`, and the implementation classes.

Run the Oracle9iAS Web Services assembly tool, `WebServicesAssembler` as follows:

```
java -jar WebServicesAssembler.jar -config my_service_config
```

Where: `my_service_config` is the configuration file that contains the `<stateless-java-service>` or the `<stateful-java-service>` tag.

See Also:

- ["Creating a Configuration File to Assemble Document Style Web Services"](#) on page 6-10
- ["Running the Web Services Assembly Tool"](#) on page 9-2

Deploying Document Style Web Services

After creating the `.ear` file containing Java classes and the Web Services Servlet deployment descriptors, you can deploy the Web Service as you would any standard J2EE application stored in an `.ear` file (to run under OC4J).

See Also: *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library.

Developing and Deploying JMS Web Services

This chapter describes the procedures you use to configure, deploy, and build Oracle9iAS Web Services that expose JMS destinations, including JMS Queues and JMS Topics as Web Services. This chapter also covers writing a backend JMS message processor to consume incoming JMS messages and to generate outgoing JMS messages.

Oracle9iAS Web Services supports asynchronous message facilities with JMS Web Services.

This chapter covers the following topics:

- [JMS Web Services Overview](#)
- [Writing JMS Web Services and Handling Messages](#)
- [Preparing and Configuring JMS Web Services](#)
- [Deploying JMS Web Services](#)
- [Limitations for JMS Web Services](#)

JMS Web Services Overview

This section covers the following topics:

- [Using JMS Web Services](#)
- [JMS Web Services Backend Message Processing](#)

Using JMS Web Services

The sample code for JMS Web Services is supplied in the `demo1` and `demo2` directories in `$ORACLE_HOME/webservices/demo/basic/jms_service` on UNIX or in `%ORACLE_HOME%\webservices\demo\basic\jms_service` on Windows.

JMS Web Services examples show both Orion JMS and Oracle JMS. In the samples, `demo1` uses Orion JMS and `demo2` uses Oracle JMS. Oracle JMS is recommended for use with JMS Web Services. Orion JMS examples are provided for demonstration purposes only.

Using JMS Web Services, Oracle9iAS supplies a Servlet that supports two operations on messages: a `send` operation and a `receive` operation. Using these two operations, if the destination is a JMS Queue, `send` means enqueue, and `receive` means dequeue. If the destination is a topic, `send` means publish and `receive` means subscribe. An individual JMS Web Service can support just the `send` operation, just the `receive` operation, or both operations, as determined by the service developer.

The JMS Web Service determines how to handle incoming and outgoing messages for JMS destinations based on the configuration of the JMS Web Service and on the operation specified by the client-side program that uses the JMS Web Service. The Oracle9iAS Web Services runtime verifier throws an exception if the operation supplied by a JMS Web Service client is invalid. For example, if the deployment operation is `send`, and the request is `receive`, an exception is thrown.

The client-side message associated with a JMS Web Service is an XML document that conforms to the Document Object Model (DOM) as specified by the W3C (`org.w3c.dom.Element`). For a `send` operation, it is the client-side developer's job to deliver a message of the correct form to a JMS Web Service. And likewise, for a `receive` operation, the client must handle the message it receives from a JMS Web Service.

See Also: <http://java.sun.com/products/jms/> for information on JMS

JMS Web Services Backend Message Processing

A JMS Web Service consists of configuration information that defines the Web Service, and, in addition the server-side developer provides code that consumes the messages that a JMS Web Service client sends, or generates the messages that the client receives.

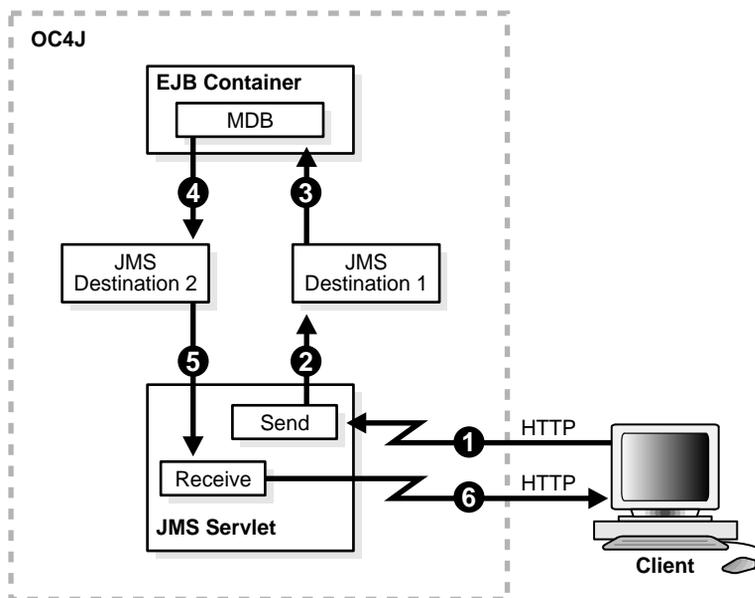
This section describes the architecture for processing JMS messages associated with a JMS Web Service and covers the following topics:

- [Using an MDB for Message Processing](#)
- [Using a JMS Client for Message Processing](#)

Using an MDB for Message Processing

A JMS Web Service either sends messages to a JMS destination or receives messages from a JMS destination and can use an MDB on the backend for generating and consuming messages. For example, [Figure 7-1](#) shows an MDB based JMS Web Service that, from the JMS Web Service client's view, handles both the message send and the message receive operations.

Figure 7-1 *MDB Based JMS Web Service*



[Figure 7-1](#) includes an MDB that is configured to listen to a JMS destination. The MDB based JMS Web Service works with the following steps:

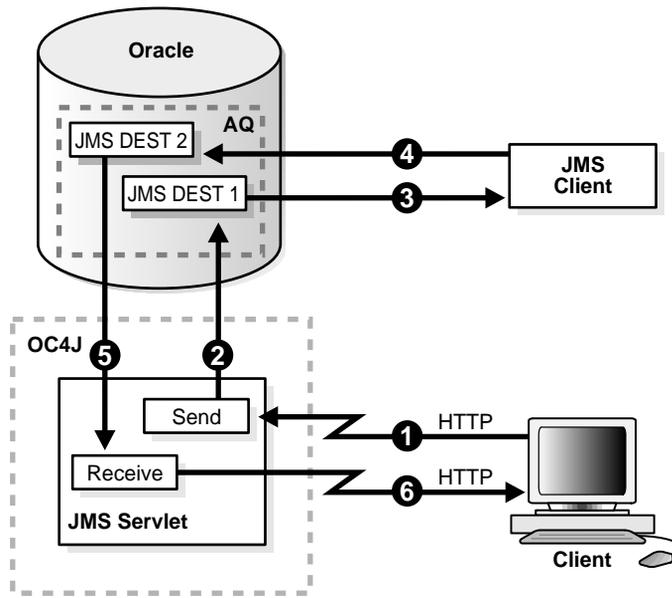
1. A JMS Web Service client performs a `send` operation on the JMS Web Service to send a message.
2. The JMS Web Service processes the incoming message and directs it to a JMS destination, JMS Destination 1.
3. The EJB container invokes the MDB listening on JMS Destination 1.
4. After processing the 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 listening on JMS Destination 1 and the same MDB could also send the message to JMS Destination 2.
5. (Arrows 5 and 6) A JMS Web Service client performs a `receive` operation on the JMS Web Service to receive a message. The JMS Web Service consumes a message from the JMS destination, processes it, and passes the outgoing message to the client.

Using a JMS Client for Message Processing

Using a JMS client for message processing, the JMS Web Service does not assemble, deploy, or run the JMS code on the backend. A separate JMS program that runs outside of the JMS Web Service, as a standalone JMS client, is responsible for generating and consuming the JMS messages that are associated with the JMS Web Service.

For example, [Figure 7-2](#) shows a JMS Web Service that use a server-side JMS client for message processing.

Figure 7-2 JMS Client Based JMS Web Service



The JMS Web service includes only configuration information that supports handling messages and using JMS destinations. The JMS client based JMS Web Service works with the following steps:

1. A JMS Web Service client performs a `send` operation on the JMS Web Service to send a message.
2. The JMS Web Service then processes the incoming message and directs it to JMS DEST 1.
3. The JMS client processes the incoming message on JMS DEST 1. The incoming message could be identified using a message listener, or by other means.
4. After processing the incoming message the JMS client may produce a new message on JMS DEST 2. The message on JMS DEST 2 could be produced by another JMS client or by the same JMS client.
5. (Arrows 5 and 6) A JMS Web Service client performs a `receive` operation on the JMS Web Service to receive a message. The JMS Web Service consumes an outgoing message from the JMS destination and passes the message to the client.

Writing JMS Web Services and Handling Messages

Writing a JMS Web Service presents a server-side developer with two tasks:

1. Building the backend message processing program for a JMS Web Service.
2. Preparing and configuring a JMS Web Service.

This section covers the following:

- [Using an MDB for Backend Message Processing](#)
- [Using a JMS Standalone Program for Backend Message Processing](#)
- [Message Processing and Reply Messages](#)

See Also:

- ["Preparing and Configuring JMS Web Services" on page 7-11](#)
- [Chapter 4, "Developing and Deploying EJB Web Services"](#)

Using an MDB for Backend Message Processing

When a JMS Web Service uses an MDB for generating or consuming messages, the MDB must be assembled with the JMS Web Service. In this case, the MDB is packaged as part of the J2EE .ear file that is deployed as a JMS Web Service.

Using an MDB with a JMS Web Service, the server-side developer is responsible for performing the following steps:

- [Developing the MDB that Processes Incoming Messages](#)
- [Developing the MDB that Generates Outgoing Messages](#)
- [Compiling and Preparing the MDB EJB.jar File](#)
- [Assembling the JMS Web Service With the MDB](#)
- [Defining the Server-Side Resource References](#)

Note: A given JMS Web Service may process incoming messages, generate outgoing messages, or do both.

Developing the MDB that Processes Incoming Messages

The MDB that processes incoming messages, generated from a JMS Web Service send operation, must include an `onMessage()` method with the following characteristics:

- The `onMessage()` method should be declared as `public`, but not `final` or `static`
- The `onMessage()` method should have a return type of `void`
- The `onMessage()` method should have one argument of type `javax.jms.Message`. The JMS Web Service only supports messages of type `ObjectMessage`, so the MDB developer should cast the incoming JMS Web Service message to an `ObjectMessage`.
- The message payload is available from the message using the `getObject()` method on the incoming JMS message and casting to the `Element` type.

[Example 7-1](#) shows an MDB method that handles an incoming JMS Message. Also see `MessageBean.java` in the `demo1` directory for the complete code.

Example 7-1 Sample Incoming `onMessage()` Method for JMS Web Service

```
public void onMessage(Message inMessage) {
    ObjectMessage msg = null;
    Element e;
    try {
        // Message should be of type objectMessage
        if (inMessage instanceof ObjectMessage) {
            // retrieve the object
            msg = (ObjectMessage) inMessage;
            e = (Element)msg.getObject();
            processElement(e);
            this.send2Queue(e);
        } else {
            System.out.println("MessageBean::onMessage() => Message of wrong type: "
+ inMessage.getClass().getName());
        }
    } catch (JMSEException ex) {
        ex.printStackTrace();
        mdc.setRollbackOnly();
    } catch (Throwable te) {
        te.printStackTrace();
    }
}
```

Developing the MDB that Generates Outgoing Messages

An MDB that generates an outgoing message, consumed by a JMS Web Service receive operation, must include code that produces a message on a JMS destination with the following characteristics:

- The message placed on the JMS destination should be of type: `javax.jms.Message.ObjectMessage`.
- Set the payload of the message using the `setObject()` method on the outgoing JMS message and casting to the `java.io.Serializable` type.

[Example 7-2](#) shows a code fragment that creates an outgoing message of the correct type. For the complete code for this example, see `MessageBean2.java` in the `demo2` directory.

Example 7-2 Sample Outgoing Message for JMS Web Service

```
// Create an Object Message
message = queueSession.createObjectMessage();
// Stuff the result into the ObjectMessage
((ObjectMessage)message).setObject((java.io.Serializable)ee);
// Send the Message
queueSender.send(message);
```

Compiling and Preparing the MDB EJB.jar File

After compiling the MDB classes, create an EJB .jar file that includes the MDB and its required deployment information.

Assembling the JMS Web Service With the MDB

Assemble the MDB's EJB.jar file with the JMS Web Service .ear file using the `WebServicesAssembler` tool and a configuration file containing the top-level tag `<option name=source-path">` that specifies the EJB .jar, and the `<jms-doc-service>` that defines the JMS Web Service configuration.

See Also:

- ["Preparing and Configuring JMS Web Services"](#) on page 7-11
- ["Deploying JMS Web Services"](#) on page 7-17

Defining the Server-Side Resource References

Define the resource references associated with the JMS destinations that the JMS Web Service uses:

- If the MDB uses Orion JMS, define the resource references in the OC4J `jms.xml` configuration file.
- If the MDB uses Oracle JMS, then run the `sql` files that support access to the Oracle JMS destinations.

See Also:

Chapter 3, "AQ Programmatic Environments" in the *Oracle9i Application Developer's Guide - Advanced Queuing* in the Oracle9i Documentation library

Using a JMS Standalone Program for Backend Message Processing

Using a JMS standalone program on the backend for the JMS Web Service, the server-side developer is responsible for performing the following steps:

1. Developing the JMS client that defines the JMS destinations, handles incoming messages, processes them, and produces the outgoing messages. The JMS client can also perform processing that uses a JMS destination that triggers an MDB.
2. Assembling the JMS Web Service `.ear` file using the `WebServicesAssembler` tool and a configuration file containing the top-level tag `<jms-doc-service>`.
3. Defining the resource references associated with JMS destinations in the OC4J `jms.xml` configuration file. If the JMS destinations are defined in Oracle JMS, then the developer must run the `sql` files that initialize the access to the Oracle JMS destinations.

See Also:

- ["Using an MDB for Backend Message Processing"](#) on page 7-6
- ["Deploying JMS Web Services"](#) on page 7-17

Note: When a JMS Web Service uses standalone a JMS client to consume or generate messages, the standalone client cannot be assembled with the JMS Web Service.

Message Processing and Reply Messages

The JMS Web Service processes an incoming message, a JMS Web Service send operation message, and places the 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 Web Service.

The client-side message associated with a JMS Web Service is an XML document that conforms to the Document Object Model (DOM) as specified by the W3C (`org.w3c.dom.Element`). When a JMS Web Service is sent an `Element` from a Web Service client, it creates a `JMSObjectMessage` that contains the `Element`. The JMS Web Service may set certain header values before it places the message on a JMS destination. Depending on the values of optional configuration tags specified when the JMS Web Service is assembled, the JMS Web Service sets the following JMS Message Headers:

```
JMSType  
JMSReplyTo  
JMSExpiration  
JMSPriority  
JMSDeliveryMode
```

When the JMS Web Service sets the `JMSReplyTo` header, it uses either the value specified with the `<reply-to-topic-resource-ref>` or the `<reply-to-queue-resource-ref>` (only one of these should be configured for any given JMS Web Service). The value specified with the `<reply-to-connection-factory-resource-ref>` tag is set on the message as a standard string property. The property name is `OC4J_REPLY_TO_FACTORY_NAME`.

[Example 7-3](#) provides a code segment that shows where the `onMessage()` method gets the `ReplyTo` information for message generated from a JMS Web Service send operation:

Example 7-3

```
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();

```

.
.
.

See Also:

- ["Developing the MDB that Processes Incoming Messages" on page 7-7](#)
- ["Adding JMS Doc Service Tags" on page 7-13](#)

Preparing and Configuring JMS Web Services

This section describes how to use the Oracle9iAS Web Services tool `WebServicesAssembler` to prepare a J2EE .ear file for a JMS Web Service.

To deploy a JMS Web Service, you need to assemble a J2EE .ear file. The J2EE .ear file can include the following:

- The deployment descriptors for the Oracle9iAS Web Services Servlet.
- If the JMS Web Service also includes an MDB, then the J2EE .ear also includes a Jar file that supplies the MDB implementation. This component is optional. To expose JMS Queues or Topics as JMS Web Services, you are not required to include an MDB Jar file with the JMS Web Service.

This section describes the procedures you use to create a configuration file to use with the `WebServicesAssembler`.

This section contains the following topics:

- [Creating a Configuration File to Assemble JMS Web Services](#)
- [Running WebServicesAssembler With JMS Web Services](#)

Creating a Configuration File to Assemble JMS Web Services

The Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9iAS Web Services. This section describes how to create an XML configuration file that describes the JMS Web Service to be assembled.

Create `WebServicesAssembler` configuration file by adding the following:

- [Adding Web Service Top Level Tags](#)
- [Adding JMS Doc Service Tags](#)
- [Adding WSDL and Client-Side Proxy Generation Tags](#)

Adding Web Service Top Level Tags

[Table 7-1](#) describes the top level `WebServicesAssembler` configuration file tags. Add these tags to provide top level information describing the JMS Web Service.

[Example 7-4](#) shows a complete JMS Web Service sample configuration file. The `demo1` and `demo2` directories in the `jms_service` directory contain complete `config.xml` files for JMS Web Services.

Table 7-1 Top Level `WebServicesAssembler` Configuration Tags

Tag	Description
<code><context></code>	Specifies the context root of the Web Service.
<code>context</code>	This tag is required.
<code></context></code>	
<code><datasource-JNDI-name></code>	Specifies the datasource associated with the Web Service.
<code></datasource-JNDI-name></code>	
<code><description></code>	Provides a simple description of the Web Service.
<code>description</code>	This tag is optional.
<code></description></code>	
<code><destination-path></code>	Specifies the name of the generated J2EE .ear file output. The <i>dest_path</i> specifies the complete path for the output file.
<code>dest_path</code>	This tag is required.
<code></destination-path></code>	
<code><display-name></code>	Specifies the Web Service display name.
<code>disp_name</code>	This tag is optional.
<code></display-name></code>	

Table 7-1 (Cont.) Top Level WebServicesAssembler Configuration Tags

Tag	Description
<code><option name="source-path" ></code> <code>path</code> <code></option></code>	Includes a specified file in the output .ear file. For a JMS Web Service, use this tag to specify the MDB source .jar file if the Web Service uses an MDB to handle the JMS messages. The <i>path</i> specifies the path to the file to include.
<code><jms-doc-service></code> <code>sub-tags</code> <code></jms-doc-service></code>	Use this tag to add a JMS Web Service. See Table 7-2 for a description of the valid <i>sub-tags</i> .
<code><temporary-directory></code> <code>temp_dir</code> <code></temporary-directory></code>	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding JMS Doc Service Tags

The `<jms-doc-service>` defines the configuration information for a JMS Web Service. The JMS Web Service developer determines if the service supports send operations, receive operations, or both send and receive, based on the value of the `<operation>` sub-tag. Some of the configuration file tags are only valid, depending on the operation selected for the Web Service. [Table 7-2](#) lists all the supported `<jms-doc-service>` sub-tags, and includes information on whether each is valid, based on the operation specified.

Table 7-2 JMS Service WebServicesAssembler Configuration Tags

Tag	Description
<code><connection-factory- resource-ref></code> <code>resource-ref</code> <code></connection-factory- resource-ref></code>	Specifies the Topic Connection Factory or Queue Connection Factory resource reference <i>resource-ref</i> for the JMS destination associated with the JMS Web Service. This tag is required.
<code><jms-delivery-mode></code> <code>delivery-mode</code> <code></jms-delivery-mode></code>	Sets the <code>JMSDeliveryMode</code> message header to the specified <i>delivery-mode</i> value for the JMS message that is created with a send operation. This tag is valid when the <code><operation></code> value is: send or both This tag is optional.
<code><jms-expiration></code> <code>expiration</code> <code></jms-expiration></code>	Sets the <code>JMSExpiration</code> message header to the specified <i>expiration</i> value for the JMS message that is created with a send operation. This tag is valid when the <code><operation></code> value is: send or both This tag is optional.

Table 7–2 (Cont.) JMS Service WebServicesAssembler Configuration Tags

Tag	Description
<pre><jms-message-type> message-type </jms-message-type></pre>	<p>Sets the JMSType for the message to the specified <i>message-type</i> for the JMS message that is created with a send operation</p> <p>This tag is valid when the <operation> value is: send or both</p> <p>This tag is optional.</p>
<pre><jms-priority> priority </jms-priority></pre>	<p>Sets the JMSPriority message header to the specified <i>priority</i> value for the JMS message that is created with a send operation.</p> <p>This tag is valid when the <operation> value is: send or both</p> <p>This tag is optional.</p>
<pre><operation> op </operation></pre>	<p>Specifies the operation <i>op</i> that the JMS Web Service supports.</p> <p>Using the send and receive operation:</p> <ul style="list-style-type: none"> ■ If the destination is a JMS Queue, send means enqueue, and receive means dequeue. ■ If the destination is a topic, send means publish and receive means subscribe. <p>The send operation uses the <connection-factory-resource-ref> and the corresponding JMS destination <queue-resource-ref> or <topic-resource-ref> to determine the JMS destination for a send operation on the service.</p> <p>With the receive operation, when the <reply-to-connection-factory-resource-ref> tag is not set, then the receive operation uses the <connection-factory-resource-ref> and the corresponding JMS destination <queue-resource-ref> or <topic-resource-ref>. When the <reply-to-connection-factory-resource-ref> tag is set, then the <reply-to-*> tags specify the JMS destination for receive operations.</p> <p>Valid values: send, receive, both</p> <p>Default value: both</p> <p>This tag is optional.</p>
<pre><queue-resource-ref> queue-ref </queue-resource-ref></pre>	<p>Specifies the resource reference <i>queue-ref</i> of the destination JMS queue.</p> <p>Either a <topic-resource-ref> or a <queue-resource-ref> must be specified, but not both. When a <queue-resource-ref> is specified, the <connection-factory-resource-ref> must refer to a corresponding Queue connection factory.</p>

Table 7-2 (Cont.) JMS Service `WebServicesAssembler` **Configuration Tags**

Tag	Description
<pre><reply-to-connection- factory-resource-ref> reply-to-conn-factory-res-ref </reply-to-connection- factory-resource-ref></pre>	<p>If the <code><operation></code> specified is both, then receive operations use the <code><reply-to-connection-factory-resource-ref></code>. The specified <code>reply-to-conn-factory-res-ref</code> value specifies the JMS destination connection factory for receive operations. Also, if the MDB, or any JMS consumer, expects to send results back, then the name of the destination connection factory to which the reply message will be sent has to be specified in this parameter.</p> <p>See Also: "Message Processing and Reply Messages" on page 7-10.</p> <p>This tag is optional.</p>
<pre><reply-to-queue- resource-ref> reply-to-queue-res-ref </reply-to-queue- resource-ref></pre>	<p>Specifies the resource reference <code>reply-to-queue-res-ref</code> of the destination JMS queue.</p> <p>When a <code><reply-to-queue-resource-ref></code> is specified, the <code><reply-to-connection-factory-resource-ref></code> must refer to a corresponding Queue connection factory.</p> <p>If the <code><reply-to-connection-factory-resource-ref></code> tag is set, then either a <code><reply-to-topic-resource-ref></code> or a <code><reply-to-queue-resource-ref></code> must be specified, but not both.</p> <p>This tag is optional.</p>
<pre><reply-to-topic- resource-ref> reply-to-topic-res-ref </reply-to-topic- resource-ref></pre>	<p>Specifies the resource reference <code>reply-to-topic-res-ref</code> of the destination JMS Topic.</p> <p>When a <code><reply-to-topic-resource-ref></code> is specified, the <code><reply-to-connection-factory-resource-ref></code> must refer to a corresponding Topic connection factory.</p> <p>If the <code><reply-to-connection-factory-resource-ref></code> tag is set, then either a <code><reply-to-topic-resource-ref></code> or a <code><reply-to-queue-resource-ref></code> must be specified, but not both.</p> <p>This tag is optional.</p>
<pre><topic-resource-ref> topic-ref </topic-resource-ref></pre>	<p>Specifies the resource reference <code>topic-ref</code> of the destination JMS Topic.</p> <p>Either a <code><topic-resource-ref></code> or a <code><queue-resource-ref></code> must be specified, but not both. When a <code><topic-resource-ref></code> is specified, the <code><connection-factory-resource-ref></code> must refer to a corresponding Topic connection factory.</p>
<pre><uri> URI </uri></pre>	<p>This tag specifies servlet mapping pattern for the Servlet that implements the JMS Web Service. The path specified as the <code>URI</code> is appended to the <code><context></code> to specify the JMS Web Service location.</p> <p>This tag is optional.</p>

Adding WSDL and Client-Side Proxy Generation Tags

The `WebServicesAssembler` supports the `<wsdl-gen>` and `<proxy-gen>` tags to allow a Web Service developer to generate WSDL files and client-side proxy files. You can use these tags to control whether the WSDL file and the client-side proxy are generated. Using these tags you can also specify that the generated WSDL file or a WSDL file that you write is packaged with the Web Service J2EE `.ear`.

A client-side developer either uses the WSDL file that is obtained from a deployed Web Service, or the client-side proxy that is generated from the WSDL to build an application that uses the Web Service.

See Also: ["Generating WSDL Files and Client Side Proxies"](#) on page 9-5

Example 7-4 Sample `WebServicesAssembler` Configuration File for JMS Web Service

```
<web-service>
  <display-name>JMS Web Service Example</display-name>
  <description>JMS Web Service Example</description>
  <!-- Name of the destination -->
  <destination-path>./jmswsl.ear</destination-path>
  <temporary-directory>./tmp</temporary-directory>
  <!-- Context root of the application -->
  <context>/jmswsl</context>
  <!-- Path of the jar file with MDBs defined/implemented in it -->
  <option name="source-path">MDB/mdb_service1.jar</option>

  <!-- tags for jms doc service -->
  <jms-doc-service>
    <uri>JmsSend</uri>
    <connection-factory-resource-ref>jms/theQueueConnectionFactory</connection-factory-resource-ref>
    <queue-resource-ref>jms/theQueue</queue-resource-ref>
    <operation>send</operation>x
  </jms-doc-service>

  <jms-doc-service>
    <uri>JmsReceive</uri>
    <connection-factory-resource-ref>jms/logQueueConnectionFactory</connection-factory-resource-ref>
    <queue-resource-ref>jms/logQueue</queue-resource-ref>
    <operation>receive</operation>
  </jms-doc-service>
  <!-- generate the wsdl -->
  <wsdl-gen>
    <wsdl-dir>wsdl</wsdl-dir>
    <!-- over-write a pregenerated wsdl , turn it 'false' to use the pregenerated wsdl-->
    <option name="force">true</option>
    <option name="httpServerURL">http://localhost:8888</option>
```

```
<!-- do not package the wsdl -generate it again on teh server-->
<option name="packageIt">false</option>
</wsdl-gen>
<!-- generate the proxy -->
<proxy-gen>
  <proxy-dir>proxy</proxy-dir>
  <option name="include-source">>true</option>
</proxy-gen>
</web-service>
```

Running WebServicesAssembler With JMS Web Services

After you create the `WebServicesAssembler` configuration file, you can generate a J2EE .ear file for the JMS Web Service. The J2EE EAR file includes Web Service servlet configuration information, including the generated file `web.xml`, and if the service includes MDBs, the `ejb.jar` file containing the implementation classes.

Run the Oracle9iAS Web Services assembly tool, `WebServicesAssembler` as follows:

```
java -jar WebServicesAssembler.jar -config my_jms_service_config
```

Where: `my_jms_service_config` is the configuration file that contains the `<jms-doc-service>` tag.

See Also:

- ["Creating a Configuration File to Assemble JMS Web Services"](#) on page 7-12
- ["Running the Web Services Assembly Tool"](#) on page 9-2

Deploying JMS Web Services

After creating the .ear file containing Java classes and the Web Services Servlet deployment descriptors, you can deploy the Web Service as you would any standard J2EE application stored in an .ear file (to run under OC4J).

See Also: *Oracle9iAS Containers for J2EE User's Guide* in the Oracle9iAS Documentation Library.

Limitations for JMS Web Services

The JMS Web Service only supports messages of type `ObjectMessage` (`javax.jms.Message.ObjectMessage`).

Building Clients that Use Web Services

This chapter describes the Oracle9iAS Web Services features that allow you to easily create and run a client application that uses Oracle9iAS Web Services.

This chapter contains the following topics:

- [Locating Web Services](#)
- [Getting WSDL Files and Client-Side Proxy Jars for Web Services](#)
- [Working with Client-Side Proxy Jar to Use Web Services](#)
- [Working with WSDL Files and JDeveloper to Use Web Services](#)

Locating Web Services

When you want to use Web Services you need to develop a client application. There are two types of Web Services clients: static web service clients and dynamic web service clients. A **static web service client** knows where a Web Service is located without looking up the service in a UDDI registry. A **dynamic web service client** performs a lookup to find the Web Service's location in a UDDI registry before accessing the service. [Chapter 10, "Discovering and Publishing Web Services"](#) provides detailed information on looking up Web Services in a UDDI registry.

Using a static client Oracle9iAS Web Services provides several options for locating Oracle9iAS Web Services, including:

- Using a known Web Service located at a known URL.
- Using Oracle9iAS Web Services and a known service URL to obtain a client-side proxy Jar, or by other means obtaining a client-side proxy Jar for a Web Service. The client-side proxy Jar that Oracle9iAS Web Services generates includes the URL to locate the associated Web Service.
- Using Oracle9iAS Web Services and a known service URL to obtain a WSDL file, or by other means obtaining a WSDL file that describes a Web Service. The WSDL files that Oracle9iAS Web Services generates includes the URL to locate the associated Web Service.

After you locate a Web Service or after you obtain either the WSDL or client-side proxy Jar, you can build a client-side application that uses the Web Service.

See Also: [Chapter 10, "Discovering and Publishing Web Services"](#)

Getting WSDL Files and Client-Side Proxy Jars for Web Services

This section covers the following:

- [Using the Web Service Home Page to Save WSDL and Client Side Proxies](#)
- [Getting Web Service WSDL and Client-Side Proxies Directly](#)
- [Generating Client-Side Proxies With WebServicesAssembler](#)

Using the Web Service Home Page to Save WSDL and Client Side Proxies

To use Oracle9iAS Web Services you need to create a client-side application that accesses a Web Service. Oracle9iAS Web Services supplies the following files for deployed Web Services:

- WSDL service descriptions
- Client-side proxy Jar (class files)
- Client-side proxy source

Oracle9iAS Web Services provides a Web Service Home Page for each deployed Web Service. To access a Home Page, enter a service endpoint of the form,

`http://host:port/context-root/service`

Figure 8–1 shows the Web Service Home Page for StatefulExample, at the following endpoint,

`http://system1.us.oracle.com/webservices/statefulTest`

A Web Service Home Page provides the following:

- A Link to the WSDL file - To obtain the WSDL file for a Web Service, select the Service Description link and save the file.
- Links to Web Service Test Pages for each supported operation-To test the available Web Service operations enter the parameter values for the operation, if any, and select the Invoke button.
- Links to the Web Service client-side proxy Jar and the client-side proxy source - To obtain the client-side proxy Jar or the client-side proxy source, select the appropriate link, Proxy Jar or Proxy Source, and save the file.

Figure 8–1 Web Service Home Page

StatefulExample endpoint

WSDL for Service: StatefulExample, generated by Oracle WSDL toolkit (version: 1.1)

For a formal definition, please review the [Service Description](#) (*rpc style*).

StatefulExample service

The following operations are supported.

- [count](#)
- [helloWorld](#)

oc4j client

The java proxy is packaged in a .jar either as classes or sources files.

- [Proxy Jar](#)
- [Proxy Source](#)

Limitations for Web Service Test Pages

Web Service Test Pages have the following limitations:

- There is no support for complex input parameters for RPC style Web Services. Such pages do not support the Invoke button.
- There is no support for Document Style Web Services. Such pages do not support the Invoke button.

Getting Web Service WSDL and Client-Side Proxies Directly

If you do not use the Web Service Home Page to get the WSDL file or client-side proxy for a Web Service, you can obtain these files directly.

This section covers the following:

- [Getting WSDL Service Descriptions](#)
- [Getting Client-Side Proxy Jar and Client-Side Proxy Source Jar](#)
- [Getting Client-Side Proxy Jar and Client-Side Proxy Source by Package](#)

Getting WSDL Service Descriptions

To obtain the WSDL service description for a Web Service, use the Web Service URL and append a query string. The format for the URL to obtain the WSDL service description is as follows (see [Table 8-1](#) for a description of the URL components):

```
http://host:port/context-root/service?WSDL
```

or

```
http://host:port/context-root/service?wsdl
```

This command 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 that you obtain, you can build a client application to access the Web Service.

Getting Client-Side Proxy Jar and Client-Side Proxy Source Jar

To obtain the client-side proxy Jar for a Web Service, use the Web Service URL and append a query string. The client-side proxy Jar file contains the proxy stubs class that supports building an application that communicates using SOAP to access the Web Service. The proxy class does the following:

- Provides a static location for the Web Service (the service does not need to be looked up in a UDDI registry).
- Provides proxy methods for each method exposed as part of the Web Service.
- Performs all of the work to construct the SOAP request, including marshalling and unmarshalling parameters, and handling the response.

The format for the URL to obtain the client-side proxy Jar is as follows (see [Table 8–1](#) for a description of the URL components):

```
http://host:port/context-root/service?PROXY_JAR  
or
```

```
http://host:port/context-root/service?proxy_jar
```

This command returns the file `service_proxy.jar`. The `service_proxy.jar` is a Jar file that contains the client-side proxy classes that you can use to build a client-side application to access the Web Service.

To obtain the client-side proxy source Jar for a Web Service, use the Web Service URL and append a query string. The format for the URL to obtain the client-side proxy source Jar is as follows (see [Table 8–1](#) for a description of the URL components):

```
http://host:port/context-root/service?PROXY_SOURCE  
or
```

```
http://host:port/context-root/service?proxy_source
```

This command returns the file `service_proxysrc.jar`. The file `service_proxysrc.jar` is a Jar file that contains the client-side proxy source files. This file represents the source code for the file `service_proxy.jar` associated with the service.

Getting Client-Side Proxy Jar and Client-Side Proxy Source by Package

When you obtain the client-side proxy Jar file or the client-side proxy source Jar, you have the option of including a request parameter that specifies a package name for the generated client-side proxy classes or source files. If the Web Service's client-side Java class is part of a particular package, then you should specify the package name to match the client-side application's package name.

The format for the URL to obtain the client-side proxy Jar and specify the package name is as follows (see [Table 8–1](#) for a description of the URL components):

```
http://host:port/context-root/service?PROXY_JAR&packageName=mypackage  
or
```

```
http://host:port/context-root/service?proxy_jar&packageName=mypackage
```

This command returns the file `service_proxy.jar`. The `service_proxy.jar` is a Jar file that contains the client-side proxy classes, using the specified package, `mypackage` for the Java package statement.

The format for the URL to obtain the client-side proxy source Jar and specify the package name is as follows (see [Table 8–1](#) for a description of the URL components):

```
http://host:port/context-root/service?PROXY_SOURCE&packageName=mypackage
```

or

```
http://host:port/context-root/service?proxy_source&packageName=mypackage
```

This command returns the file `service_proxysrc.jar`. As for the `proxy_jar`, you have the option of specifying a request parameter with a supplied package name by include a `packageName=name` option. The `service_proxysrc.jar` is a Jar file that contains the client-side source files for the client-side proxy that accesses the Web Service.

Table 8–1 URL for Accessing Client Side Proxy Stubs

URL Component	Description
<i>context-root</i>	The context-root is the value specified in the <code><context-root></code> tag for the web module associated with the Web Service. See the <code>META-INF/application.xml</code> in the Web Service's <code>.ear</code> file to determine this value.
<i>host</i>	This is the host of the Web Service's server running Oracle9iAS Web Services.
<i>mypackage</i>	This specifies the value that you want to use for the package name in the generated proxy Jar or proxy source.
<i>port</i>	This is the port of the Web Service's server running Oracle9iAS Web Services.
<i>service</i>	The service is the value specified in the <code><url-pattern></code> tag for the servlet associated with the Web Service. This is the service name. See the <code>WEB-INF/web.xml</code> in the Web Service's <code>.war</code> file to determine this value.

See Also:

- [Chapter 3, "Developing and Deploying Java Class Web Services"](#)
- [Chapter 4, "Developing and Deploying EJB Web Services"](#)
- [Chapter 5, "Developing and Deploying Stored Procedure Web Services"](#)

Generating Client-Side Proxies With WebServicesAssembler

The Oracle9iAS Web Services `WebServicesAssembler` tool allows you to generate client-side proxies. A client-side proxy can access a Web Service that is deployed either on an Oracle9iAS Web Services endpoint or on a third party Web Service endpoint.

To generate a client-side proxy with `WebServicesAssembler`, specify a `<proxy-gen>` tag in the configuration file. [Table 8-2](#) describes the `<proxy-gen>` `WebServicesAssembler` configuration file sub-tags.

Note: When you are generating client-side proxies and you are accessing an external WSDL file from behind a firewall, make sure to set the appropriate security properties shown in [Table 8-3](#), such as `http.proxyHost` and `http.proxyPort`.

[Example 8-1](#) shows a sample `WebServicesAssembler` that includes a `<proxy-gen>` tag.

Example 8-1 *WebServicesAssembler Proxy Gen Configuration File*

```
<?xml version="1.0"?>
<web-service>
  <proxy-gen>
    <proxy-dir>/TestArea/Hotel/proxy/outside</proxy-dir>
    <option name="include-source">true</option>
    <option name="wsdl-location" package-name="myPackage.proxy">
      http://terraservice.net/TerraService.asmx?WSDL</option>
    <option name="wsdl-location">
      http://ws.serviceobjects.net/sq/FastQuote.asmx?WSDL</option>
  </proxy-gen>
</web-service>
```

Table 8-2 Proxy Generation <proxy-gen> Sub-Tags

Tag	Description
<pre><proxy-dir> directory </proxy-dir></pre>	<p>Specifies the directory for the generated client-side proxy stubs Jar file that is included in the generated Web Service .ear file.</p> <p>This tag is required.</p>
<pre><option name="include-source"> value </option></pre>	<p>Setting <i>value</i> to true tells <code>WebServicesAssembler</code> to include the classes and the source in the generated client-side proxy. When the value is false, the source is not included in the generated Jar.</p> <p>This tag is optional.</p> <p>Valid values: true, false</p> <p>Default value: false</p>
<pre><option name="wsdl-location"> URL </option></pre> <p>or</p> <pre><option name="wsdl-location" package-name="package"> URL </option></pre>	<p>This tag sets the <i>URL</i> to use for the source WSDL to use to generate the client-side proxy.</p> <p>This option also supports the optional attribute <code>package-name</code>. The <code>package-name</code> can specify the name <i>package</i> for the generated client-side proxy.</p> <p>This tag is optional.</p> <p>Examples:</p> <pre><option name="wsdl-location"> http://system1:8888/webservice3/TestService?WSDL </option></pre> <pre><option name="wsdl-location" package-name="myPackage.proxy"> http://system1:8888/webservice3/TestService?WSDL </option></pre>

See Also: [Chapter 9, "Web Services Assembly Tool"](#)

Working with Client-Side Proxy Jar to Use Web Services

This section describes how to use the client-side proxy Jar when you are building the client-side application to access a Web Service. The client-side proxy Jar class allows you to easily build an application that uses a Web Service.

The client side proxy Jar file contains a Java class to serve as a proxy to the Web Service implementation. The client-side proxy code constructs a SOAP request and marshalls and unmarshalls parameters for you. Using the proxy classes saves you

the work of creating SOAP requests for accessing a Web Service or processing Web Service responses.

[Example 8-2](#) shows a source code sample client-side proxy extracted from a Web Service. For each operation available on the Web Service, there is a corresponding method in the proxy class. The example shows the method `helloWorld(String)` that serves as a proxy to the `helloWorld(String)` method in the associated Web Service implementation.

[Example 8-3](#) shows client-side application code that uses the `helloWorld()` method from the supplied client-side proxy shown in [Example 8-2](#).

Note: When you are accessing an external Web Service from behind a firewall, make sure to set the appropriate security properties shown in [Table 8-3](#), such as `http.proxyHost` and `http.proxyPort`.

Example 8-2 Sample Client-side Proxy Method for Web Services

```
public class StatefulExampleProxy {  
  
    public java.lang.String helloWorld(java.lang.String param0) throws Exception  
    {  
        .  
        .  
        .  
    }  
    .  
    .  
    .  
}
```

Example 8–3 Sample Client-side Application Using a Proxy Class for Web Services

```
import oracle.j2ee.ws_example.proxy.*;

public class Client
{
    public static void main(String[] argv) throws Exception
    {
        StatefulExampleProxy proxy = new StatefulExampleProxy();
        System.out.println(proxy.helloWorld("Scott"));
        System.out.println(proxy.count());
        System.out.println(proxy.count());
        System.out.println(proxy.count());
    }
}
```

When Java Beans are used as parameters to Oracle9iAS Web Services, the client-side code should use the generated Bean included with the downloaded client-side proxy. This is because the generated client-side proxy code translates Simple Object Access Protocol (SOAP) structures to and from Java Beans by translating SOAP structure namespaces to and from fully qualified Bean class names. If a Bean with the specified name does not exist in the specified package, the generated client code will fail.

However, there is no special requirement for clients using Web Services Description Language (WSDL) to form calls to Oracle9iAS Web Services, rather than the client-side proxy. The generated WSDL document describes SOAP structures in a standard way. Application development environments, such as JDeveloper, which work directly from WSDL documents can correctly call Oracle9iAS Web Services with Java Beans as parameters.

Using Web Services Security Features

When you run a client-side application that uses Oracle9iAS Web Services, you can access secure Web Services by setting properties in the client application. [Table 8–3](#) shows the available properties that provide credentials and other security information for Web Services clients.

In a Web Services client application, you can set the security properties shown in [Table 8–3](#) as system properties by using the `-D` flag at the Java command line, or you can also set security properties in the Java program by adding these properties to the system properties (use `System.setProperty()` to add properties). In addition, the client side stubs include the `_setTransportProperties` method that

is a public method in the client proxy stubs. This method enables you to set the appropriate values for security properties by supplying a `Properties` argument.

Table 8–3 Web Services HTTP Transport Security Properties

Property	Description
<code>http.authType</code>	Specifies the HTTP authentication type. The case of the value specified is ignored. Valid values: <code>basic</code> , <code>digest</code> The value <code>basic</code> specifies HTTP basic authentication. Specifying any value other than <code>basic</code> or <code>digest</code> is the same as not setting the property.
<code>http.password</code>	Specifies the HTTP authentication password.
<code>http.proxyAuthType</code>	Specifies the proxy authentication type. The case of the value specified is ignored. Valid values: <code>basic</code> , <code>digest</code> Specifying any value other than <code>basic</code> or <code>digest</code> is the same as not setting the property.
<code>http.proxyHost</code>	Specifies the hostname or IP address of the proxy host.
<code>http.proxyPassword</code>	Specifies the HTTP proxy authentication password.
<code>http.proxyPort</code>	Specifies the proxy port. The specified value must be an integer. This property is only used when <code>http.proxyHost</code> is defined; otherwise this value is ignored. Default value: 80
<code>http.proxyRealm</code>	Specifies the realm for which the proxy authentication username/password is specified.
<code>http.proxyUsername</code>	Specifies the HTTP proxy authentication username.
<code>http.realm</code>	Specifies the realm for which the HTTP authentication username/password is specified.
<code>http.username</code>	Specifies the HTTP authentication username.

Table 8–3 (Cont.) Web Services HTTP Transport Security Properties

Property	Description
<code>java.protocol.handler.pkgs</code>	<p>Specifies a list of package prefixes for <code>java.net.URLStreamHandlerFactory</code>. The prefixes should be separated by " " vertical bar characters.</p> <p>This value should contain: <code>HTTPClient</code></p> <p>This value is required by the Java protocol handler framework; it is not defined by Oracle9i Application Server. This property must be set when using HTTPS. If this property is not set using HTTPS, a <code>java.net.MalformedURLException</code> is thrown.</p> <p>Note: This property must be set as a system property.</p> <p>For example, set this property as shown in either of the following:</p> <ul style="list-style-type: none"> ■ <code>java.protocol.handler.pkgs=HTTPClient</code> ■ <code>java.protocol.handler.pkgs=sun.net.www.protocol.HTTPClient</code>
<code>oracle.soap.transport.allowUserInteraction</code>	<p>Specifies the allows user interaction parameter. The case of the value specified is ignored. When this property is set to <code>true</code> and either of the following are true, the user is prompted for a username and password:</p> <ol style="list-style-type: none"> 1. If any of properties <code>http.authType</code>, <code>http.username</code>, or <code>http.password</code> is not set, and a 401 HTTP status is returned by the HTTP server. 2. If either of properties <code>http.proxyAuthType</code>, <code>http.proxyUsername</code>, or <code>http.proxyPassword</code> is not set and a 407 HTTP response is returned by the HTTP proxy. <p>Valid values: <code>true</code>, <code>false</code></p> <p>Specifying any value other than <code>true</code> is considered as <code>false</code>.</p>
<code>oracle.ssl.ciphers</code>	<p>Specifies a list of: separated cipher suites that are enabled.</p> <p>Default value: The list of all cipher suites supported with Oracle SSL.</p>
<code>oracle.wallet.location</code>	<p>Specifies the location of an exported Oracle wallet or exported trustpoints.</p> <p>Note: The value used is not a URL but a file location, for example:</p> <p><code>/etc/ORACLE/Wallets/system1/exported_wallet</code> (on UNIX)</p> <p><code>d:\oracle\system1\exported_wallet</code> (on Windows)</p> <p>This property must be set when HTTPS is used with SSL authentication, server or mutual, as the transport.</p>
<code>oracle.wallet.password</code>	<p>Specifies the password of an exported wallet. Setting this property is required when HTTPS is used with client, mutual authentication as the transport.</p>

Working with WSDL Files and JDeveloper to Use Web Services

The Web Services WSDL allows you to manually, or using Oracle9i JDeveloper or another IDE, build client applications that use Web Services.

The Oracle9i JDeveloper IDE supports Oracle9iAS Web Services with WSDL features and provides unparalleled productivity for building end-to-end J2EE and integrated Web Services applications.

JDeveloper supports Oracle9iAS Web Services with the following features:

- Allows developers to create Java stubs from Web Services WSDL descriptions to programmatically use existing Web Services.
- Allows developers to create a new Web Service from Java or EJB classes, automatically producing the required deployment descriptor, web.xml, and WSDL file for you.
- Provides schema-driven WSDL file editing.
- Offers significant J2EE deployment support for Web Services J2EE .ear files, with automatic deployment to OC4J.

Non-Oracle Web Services IDEs or client development tools can use the supplied WSDL file to generate Web Services requests for services running under Oracle9iAS Web Services. Currently, many IDEs have the capability to create SOAP requests, given a WSDL description for the service.

Web Services Assembly Tool

The Oracle9iAS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9iAS Web Services. The Web Services assembly tool takes a configuration file which describes a Web Service, including the location of the Java classes, PL/SQL stored procedures or functions, or J2EE EAR, WAR, or JAR files and produces a J2EE EAR file that can be deployed under Oracle9iAS Web Services.

This chapter contains the following topics:

- [Running the Web Services Assembly Tool](#)
- [Web Services Assembly Tool Configuration File Sample](#)
- [Generating WSDL Files and Client Side Proxies](#)
- [Web Services Assembly Tool Configuration File Specification](#)
- [Web Services Assembly Tool Limitations](#)

Running the Web Services Assembly Tool

Run the Web Services assembly tool as follows:

```
java -jar WebServicesAssembler.jar [-debug] -config [file]  
or  
java -jar WebServicesAssembler.jar [-debug]
```

Where *file* is a Web Services assembly tool configuration file. Without the `-config` option, a file named `config.xml` must be present in the same directory where `WebServicesAssembler.jar` is invoked.

With the `-debug` option, `WebServicesAssembler` displays verbose debugging comments.

Web Services Assembly Tool Configuration File Sample

The sample configuration file shown in [Example 9-1](#) defines two services to be wrapped in an Enterprise ARchive file (EAR). The sample includes configuration information for services defined with `<stateless-java-service>` and `<stateful-java-service>` tags.

See Also:

- ["Preparing and Deploying Java Class Based Web Services"](#) on page 3-9
- ["Preparing and Deploying Stateless Session EJB Based Web Services"](#) on page 4-8
- ["Preparing Stored Procedure Web Services"](#) on page 5-3
- ["Preparing Document Style Web Services"](#) on page 6-9
- ["Preparing and Configuring JMS Web Services"](#) on page 7-11

Example 9-1 Sample Web Services Assembly Tool Configuration File

```
<web-service>

  <display-name>Web Services Example</display-name>
  <description>Java Web Service Example</description>
  <!-- Specifies the resulting web service archive will be stored in ./ws_example.ear -->
  <destination-path>./ws_example.ear</destination-path>
  <!-- Specifies the temporary directory that web service assembly
       tool can create temporary files. -->
  <temporary-directory>./tmp</temporary-directory>
  <!-- Specifies the web service will be accessed in the servlet context
       named "/webservices". -->
  <context>/webservices</context>

  <!-- Specifies the web service will be stateless -->
  <stateless-java-service>
    <interface-name>oracle.j2ee.ws_example.StatelessExample</interface-name>
    <class-name>oracle.j2ee.ws_example.StatelessExampleImpl</class-name>
    <!-- Specifies the web service will be accessed in the uri named
         "statelessTest" within the servlet context. -->
    <uri>/statelessTest</uri>
    <!-- Specifies the location of Java class files are under ./src -->
    <java-resource>./src</java-resource>
  </stateless-java-service>

  <stateful-java-service>
    <interface-name>oracle.j2ee.ws_example.StatefulExample</interface-name>
    <class-name>oracle.j2ee.ws_example.StatefulExampleImpl</class-name>
    <!-- Specifies the web service will be accessed in the uri named
         "statefullTest" within the servlet context. -->
    <uri>/statefullTest</uri>
    <!-- Specifies the location of Java class files are under ./src -->
    <java-resource>./src</java-resource>
  </stateful-java-service>
</web-service>
```

Web Services Assembly Tool Configuration File Sample Output

After running the Web Services Assembly tool with the sample input file shown in [Example 9-1](#), the generated output is an EAR file (/tmp/ws_example.ear) The generated J2EE .ear file, ws_example.ear, has the structure shown in [Example 9-2](#).

Example 9-2 Structure of Web Services Assembly Tool Sample Ear File

```
ws_example.ear
|---META-INF
|   `---application.xml
`---ws_example_web.war
    |---index.html
    `---WEB-INF
        |-----web.xml
        `-----classes
            `-----oracle
                `-----j2ee
                    `---ws_example
                        |---StatefulExample.java
                        |---StatefulExample.class
                        |---StatefulExampleImpl.java
                        `---StatefulExampleImpl.class
                        |---StatelessExample.java
                        |---StatelessExample.class
                        |---StatelessExampleImpl.java
                        `---StatelessExampleImpl.class
```

Generating WSDL Files and Client Side Proxies

This section describes using the `<wsdl-gen>` and `<proxy-gen>` tags in a `WebServicesAssembler` configuration file. These tags controls the options for generating WSDL files and client-side proxies for Web Services. A client-side developer can obtain and use the WSDL file or the client-side proxies to build an application that uses a Web Service. A server-side developer that is assembling Web Services can use these file for testing Web Services.

This section covers the following topics:

- [Generating and Assembling WSDL Files](#)
- [Generating Client-Side Proxies with WSDL](#)

Generating and Assembling WSDL Files

Using Oracle9iAS Web Services, a Web Service developer has several choices for deciding how the WSDL file that is associated with a Web Service is generated:

1. Using the `<wsdl-gen>` tag, you can specify that `WebServicesAssembler` create the WSDL file. At assembly time when the Web Service is prepared, the `WebServicesAssembler` generates and packages the WSDL file with the Web Service.

[Example 9-3](#) shows a configuration file that includes the `<wsdl-gen>` tag.

2. Allowing the Oracle9iAS Web Services runtime to generate the WSDL file when the WSDL is requested by a Web Service client (after the WEB Service is deployed). In this case, you do not specify the `<wsdl-gen>` tag in the configuration file.
3. Creating a WSDL file manually. In this case, use the `<wsdl-gen>` tag during assembly of the J2EE .ear file to specify the path to the WSDL file. At assembly time when the Web Service is prepared, the `WebServicesAssembler` packages the WSDL file with the Web Service.

[Table 9-1](#) describes the `<wsdl-gen>` `WebServicesAssembler` configuration file sub-tags.

Note: Using the `<wsdl-gen>` tag, the default behavior is to package the WSDL into the J2EE .ear file. To exclude the generated WSDL from the J2EE .ear file, use `<option name="packageIt">` tag and set the value to `false`.

Table 9–1 WSDL Generation <wsdl-gen> Sub-Tags

Tag	Description
<pre><option name="force"> value </option></pre>	<p>Setting <i>value</i> to true forces <code>WebServicesAssembler</code> to overwrite any existing WSDL file in the WSDL directory specified with the <code><wsdl-dir></code> tag.</p> <p>Valid values: true, false</p> <p>Default value: true</p>
<pre><option name="httpServerURL"> URL </option></pre>	<p>This tag sets the value for the HTTP server listener endpoint in the generated WSDL. Set the <i>URL</i> to point to the Web Service HTTP listener.</p> <p>Example:</p> <pre><option name="httpServerURL">http://localhost:8888</option></pre>
<pre><option name="packageIt"> value </option></pre>	<p>Setting <i>value</i> to true tells <code>WebServicesAssembler</code> to include the generated WSDL in the assembled .ear file. When the value is false, the generated WSDL file is not included in the assembled .ear file.</p> <p>Valid values: true, false</p> <p>Default value: true</p>
<pre><wsdl-dir> directory </wsdl-dir></pre>	<p>Specifies the <i>directory</i> for the WSDL file source that is included in the generated Web Service .ear file.</p> <p>When you are manually supplying the WSDL file, place a copy of the WSDL file in the specified directory and use the <code><option name="force"></code> tag with the value false.</p>

Example 9–3 WebServicesAssembler Configuration File Including <wsdl-gen>

```
<web-service>

  <display-name>Stateless Java Document Web Service</display-name>
  <description>Stateless Java Document Web Service Example</description>
  <destination-path>./statelessdocws.ear</destination-path>
  <temporary-directory>./temp</temporary-directory>
  <context>/statelessdocws</context>
  <option name="source-path">converter.xsl</option>
```

```

<stateless-java-service>
  <interface-name>StatelessDoc</interface-name>
  <class-name>StatelessDocImpl</class-name>
  <uri>/docservice</uri>
  <java-resource>./classes</java-resource>
  <message-style>doc</message-style>
</stateless-java-service>

<!-- generate the wsdl -->
<wsdl-gen>
  <wsdl-dir>wsdl</wsdl-dir>
  <!-- over-write a pregenerated wsdl , turn it 'false'
       to use the pregenerated wsdl-->
  <option name="force">true</option>
  <option name="httpServerURL">http://localhost:8888</option>
</wsdl-gen>

</web-service>

```

Manually Producing a WSDL File

When you do not want to use either the `WebServicesAssembler` tool generated WSDL or the Oracle*9iAS* Web Services runtime generated WSDL file, and you want to supply your own version of the Web Service WSDL file, perform the following steps:

1. Manually create the WSDL file for your service.
2. Name the WSDL file with a name using the `.wsdl` extension placed after the service name. For example, `service1.wsdl` for a service named `service1`.
3. Create a configuration file that includes the `<wsdl-gen>` tag, including `<option name="force"> set to false` and `<option name="packageIt"> set to true`.
4. Place the WSDL file that you create in the directory specified with the `<wsdl-dir>` tag.
5. Run the `WebServicesAssembler` with the specified configuration file.

Generating Client-Side Proxies with WSDL

When the `<proxy-gen>` tag is included in a configuration file with the `<wsdl-gen>`, the generated WSDL is used to generate the proxy that is placed in the specified directory (this occurs when `WebServicesAssembler` runs during the Web Service assembly process).

Table 8–2 lists the `<proxy-gen>` sub-tags.

Note: Using `<proxy-gen>`, the generated proxy is not assembled in the J2EE .ear file.

Example 9–4 shows a sample configuration file that includes both the `<wsdl-gen>` and the `<proxy-gen>` tags.

Example 9–4 *WebServicesAssembler Configuration File Including `<wsdl-gen>`*

```
<web-service>
  <display-name>Test</display-name>
  <description>Test program</description>
  <destination-path>test.ear</destination-path>
  <temporary-directory>temp</temporary-directory><context>/HotelService</context>
  <option name="source-path">Workspace1/common/classes</option>

  <stateless-java-service>
    <interface-name>com.mypackage1.Itest</interface-name>
    <uri>/main</uri>
    <class-name>com.mypackage1.test</class-name>
  </stateless-java-service>

  <wsdl-gen>
    <wsdl-dir>wsdl</wsdl-dir>
    <option name="force">true</option>
    <option name="httpServerURL">http://localhost:8888</option>
    <option name="packageIt">false</option>
  </wsdl-gen>

  <proxy-gen>
    <proxy-dir>proxy</proxy-dir>
    <option name="include-source">true</option>
  </proxy-gen>
</web-service>
```

Web Services Assembly Tool Configuration File Specification

The input file for `WebServicesAssembler` is an XML file conforming to the Web Services Assembly Tool configuration file DTD.

[Example 9-5](#) shows the Web Services Assembly Tool Configuration file DTD.

Example 9-5 Assembly Tool Input File DTD

```
<?xml version="1.0" encoding="UCS-2"?>
<!-- Specify the properties of the web services to be assembled. -->
<!ELEMENT web-service
((display-name)?,(description)?,destination-path,temporary-directory,context,(datasource-JNDI-name)?,(stateful-
java-service)*,(stateless-java-service)*,(stateless-stored-procedure-java-service)*,(stateless-session-ejb-
service)*,(jms-doc-service)*,(option)*,(wsdl-gen)?,(proxy-gen)?)>
<!ELEMENT display-name (#PCDATA)*>
<!ELEMENT description (#PCDATA)*>
<!-- Specify the full path of the resulting EAR file. For example,
"/home/demo/webservices.ear" -->
<!ELEMENT destination-path (#PCDATA)*>
<!-- Specify a directory where the assembly tool can create temporary
directories and files. -->
<!ELEMENT temporary-directory (#PCDATA)*>
<!-- Specify the context root of the web services. For example, "/webservices". -->
<!ELEMENT context (#PCDATA)*>
<!-- for specifying database resource refs -->
<!ELEMENT datasource-JNDI-name (#PCDATA)*>

<!-- Specify the properties of a stateful Java service -->
<!ELEMENT stateful-java-service
((interface-name)?,class-name,uri,(java-resource)*,(ejb-resource)*,(scope)*,(session-timeout)*,(message-style
)?)>
<!-- Specify the properties of a stateless Java service -->
<!ELEMENT stateless-java-service
((interface-name)?,class-name,uri,(java-resource)*,(ejb-resource)*,(message-style)?)>
<!-- Specify the properties of a stateless stored procedure Java service -->
<!ELEMENT stateless-stored-procedure-java-service
((interface-name)?,(class-name)?,uri,database-JNDI-name,(java-resource)?,(jar-generation)?)>
<!-- Specify the properties of a stateless session ejb service -->
<!ELEMENT stateless-session-ejb-service (path,uri,ejb-name,(ejb-resource)*)>

<!-- Specify the java interface which defines the public methods to be exposed
in the web service. For example, "com.foo.myproject.helloWorld". -->
<!ELEMENT interface-name (#PCDATA)*>
<!-- Specify the java class to be exposed as a web service. If interface-name is
not specified, all the public methods in this class will be exposed. For example,
"com.foo.myproject.helloWorldImpl". -->
<!ELEMENT class-name (#PCDATA)*>
<!-- Specify the uri of this service. This uri is used in the URL to access the
```

WSDL and client jar, and invoke the web service. For example, "/myService". -->

```
<!ELEMENT uri (#PCDATA)*>
<!--
Specify the java resources used in this service.
The value can be a directory or a file that implements the web services. If it
is a directory, all the files and subdirectories under the directory are copied
and packaged in the Enterprise ARchive. If the java resource should belong to a
java package, you should either package it as a jar file and specify it as a
java resource, or create the necessary directory and specify the directory which
contains this directory structure as java resource. For example, you want to
include "com.mycompany.mypackage.foo" class as a java resource of the web
services, you can either package this class file in foo.jar and specify
<java-resource>c:/mydir/foo.jar</java-resource>, or place the class under
d:/mydir/com/mycompany/mypackage/foo.class and specify the java resource as
<java-resource>c:/mydir/</java-resource>.
-->
<!ELEMENT java-resource (#PCDATA)*>
<!-- Specify the ejb resources used in this service. ejb-resource should be a
jar file that implements a enterprise java bean. -->
<!ELEMENT ejb-resource (#PCDATA)*>
<!-- Specify the database JNDI name for stateless PL/SQL web service. -->
<!ELEMENT database-jndi-name (#PCDATA)*>
<!-- Specifies the path of the EJB jar file to exposed as web services. -->
<!ELEMENT path (#PCDATA)*>
<!-- Specify the ejb-name of the session bean to be exposed as web services.
ejb-name should match the <ejb-name> value in the META-INF/ejb-jar.xml of the bean. -->
<!ELEMENT ejb-name (#PCDATA)*>
<!-- Specify scope of Statefull Java service -->
<!ELEMENT scope (#PCDATA)*>
<!-- Specify session timeout of Statefull Java service -->
<!ELEMENT session-timeout (#PCDATA)*>
<!-- Specify the directory location of the generated wsdl-->
<!ELEMENT wsdl-dir (#PCDATA)*>
<!-- Specify that wsdl generation is to happen 'force' 'httpServerURL' 'packageIt'-->
<!ELEMENT wsdl-gen (wsdl-dir,(option)*>
<!-- Specify the directory location of the generated proxy-->
<!ELEMENT proxy-dir (#PCDATA)*>
<!ELEMENT option (#PCDATA)*>
<ATTLIST option name CDATA #REQUIRED>

<!-- Specifying that proxy generation is asked for , it can have optional tags as
'include-source' 'wsdl-location' -->
<!ELEMENT proxy-gen (proxy-dir,(option)*>
<!ELEMENT jar-generation (db-package-name,db-schema,db-url,prefix,(method-name)*>
<!ELEMENT database-JNDI-name (#PCDATA)*>
<!ELEMENT db-package-name (#PCDATA)*>
<!ELEMENT db-url (#PCDATA)*>
<!ELEMENT db-schema (#PCDATA)*>
<!ELEMENT prefix (#PCDATA)*>
```

```

<!ELEMENT method-name (#PCDATA)*>
  <!-- specify the message style ,if this tag is not present it is considered to have 'rpc' ..it can have
values of 'rpc' or 'doc' or 'document' -->
<!ELEMENT message-style (#PCDATA)*>

<!ELEMENT connection-factory-resource-ref (#PCDATA)*>
<!ELEMENT topic-resource-ref (#PCDATA)*>
<!ELEMENT queue-resource-ref (#PCDATA)*>
<!--Resource ref of the return destination factory-->
<!ELEMENT reply-to-connection-factory-resource-ref (#PCDATA)*>
<!--Resource ref of the return destination Topic. -->
<!ELEMENT reply-to-topic-resource-ref (#PCDATA)*>
<!--Resource ref of the return destination Queue. -->
<!ELEMENT reply-to-queue-resource-ref (#PCDATA)*>
<!--jms-priority ,jms-message-type,jms-delvery-mode ,jms-expiration The JMS properties are only set for
enqueuing operations, i..e, for send operations only. -->
<!ELEMENT jms-priority (#PCDATA)*>
<!ELEMENT jms-message-type (#PCDATA)*>
<!ELEMENT jms-delivery-mode (#PCDATA)*>
<!ELEMENT jms-expiration (#PCDATA)*>
<!-- operation property is optional. Possible values for this parameter are: send, receive, and both. If not
provided, the value defaults to both. -->
<!ELEMENT operation (#PCDATA)*>
<!ELEMENT jms-doc-service
(uri,connection-factory-resource-ref,(topic-resource-ref)?,(queue-resource-ref)?,(reply-to-connection-factory
-resource-ref)?,(reply-to-topic-resource-ref)?,(reply-to-queue-resource-ref)?,(jms-priority)?,(jms-message-ty
pe)?,(jms-delivery-mode)?,(jms-expiration)?,(operation)?>

```

Web Services Assembly Tool Limitations

The `WebServicesAssembler` tool has the following limitations:

- No Upload/download capabilities: the Web Services Assembly tool does not upload Java classes from a client system to a server or download a generated EAR file back to a client system.
- Does not support advanced configuration tasks: for example, the Web Services Assembly tool is not able to control the security options for a Web Services Servlet, cannot secure an EJB, secure welcome files, or perform other administrative tasks.

Discovering and Publishing Web Services

Oracle9iAS Containers for Java2 Enterprise Edition (J2EE), or OC4J, provides a Universal Discovery Description and Integration (UDDI) Web Services registry in which Web Services provider administrators in an enterprise environment can publish their Web Services for use by Web Services consumers (programmers). Web Services consumers can use the UDDI inquiry interface to discover these published Web Services by browsing, searching, and drilling down in the UDDI registry to select one or more Web Services from among those registered to be used in their applications for a particular enterprise process.

For example, a Web Services provider administrator working with programmers who have completed a Web Services implementation using the J2EE stack (either EJBs, JavaBeans, JSP, or servlets) and exposing the implementation as a Simple Open Access Protocol (SOAP)-based Web Services, can publish the Web Services by providing all the metadata and pointers to the interface specification in the UDDI registry. In this way, the Web Services provider administrator publishes the availability of these Web Services for the Web Services consumer to discover and select for use in their own applications.

UDDI Registration

The information provided in a UDDI registration can be used to perform three types of searches:

- White pages search -- containing address, contact, and known identifiers. For example, search for a business that you already know something about, such as its name or some unique ID.
- Yellow pages topical search -- containing industrial categorizations based on standard classifications, such as NAICS, ISO-3166, and UNSPSC.
- Green pages service search -- containing technical information about Web Services that are exposed by a business, including references to specifications of interfaces for Web Services, as well as support for pointers to various file and URL-based discovery mechanisms.

UDDI uses standards-based technologies, such as common Internet protocols (TCP/IP and HTTP), XML, and SOAP, which is a specification for using XML in simple message-based exchanges. UDDI is a standard Web Services description format and Web Services discovery protocol; a UDDI registry can contain metadata for any type of service, with best practices already defined for those described by Web Services Description Language (WSDL).

UDDI Registry

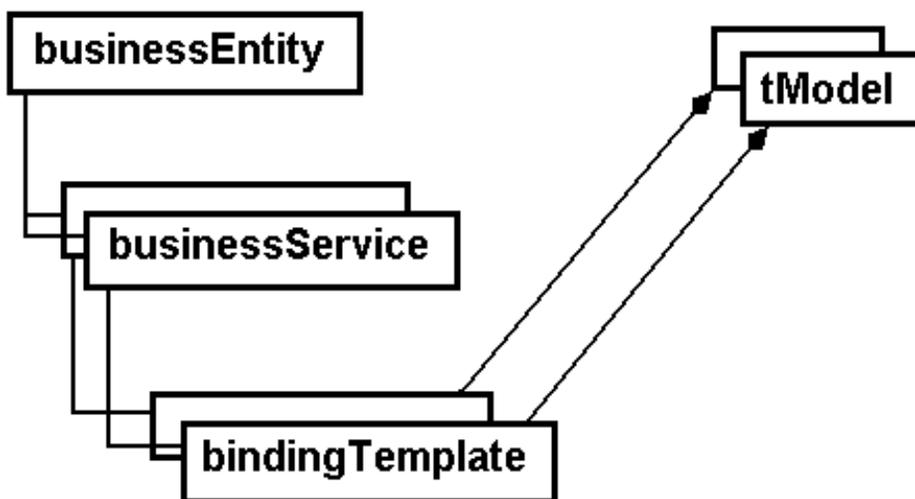
The UDDI registry consists of the following four data structure types that group information to facilitate rapid location and understanding of registration information:

- `businessEntity` -- the top-level, logical parent data structure; contains descriptive information about the business that publishes information about Web Services, such as business services, categories, contacts, discovery URLs, and identifier and category information that is useful for performing searches.
- `businessService` -- the logical child of a single `businessEntity` data structure as well as the logical parent of a `bindingTemplate` structure; contains descriptive business service information about a particular family of technical services including its name, brief description, technical service description, and category information that is useful for performing searches.
- `bindingTemplate` -- the logical child of a single `businessService` data structure; contains technical information about a Web Services entry point and references to interface specifications.

tModel -- descriptions of specifications for Web Services or classifications that form the basis for technical fingerprints; represents the technical specification of the Web Services, in order to facilitate Web Services consumer searching for registered Web Services that are compatible with a particular technical specification. That is, based on the descriptions of the specifications for Web Services in the tModel structure, Web Services consumers can easily identify other compatible Web Services.

Figure 10-1 shows the UDDI information model and the relationships among its four data structure types.

Figure 10-1 UDDI Information Model Showing the Relationship Among the Four Main Data Structure Types



Because UDDI makes use of XML and SOAP, each of these data structure types contains a number of elements and attributes that further serve to describe a business or have a technical purpose. See *UDDI Data Structure Reference V1.0 Open Draft Specification 30 September 2000* and *UDDI Programmer's API 1.0 Open Draft Specification 30 September 2000* for a complete description of the UDDI service description framework, including its XML schema, and the approximately 20 request messages and 10 response messages that comprise the request/response XML SOAP message interface that is used to perform publishing and inquiry functions against the UDDI business registry.

Oracle UDDI Enterprise Web Services Registry

This section describes a subset of features that provide UDDI support for Web Services deployed in OC4J as the Oracle 9iAS release 2 UDDI enterprise implementation of OC4J Web Services and the UDDI enterprise-wide Web Services registry.

The Oracle UDDI registry support for Web Services deployed in OC4J is composed of two parts:

- Web Services discovery -- consumers can use the Inquiry API available for Java programmers to implement their own Web Services discovery tool to search, locate, and drill down to discover OC4J Web Services in the Oracle UDDI registry, as well as in any other accessible UDDI Version 1.0 Web Services registry. See [Using the Inquiry API](#) on page 10-5 for more information about using the Inquiry API and locating the Javadoc documentation that describes the Inquiry API.
- Web Services publishing -- Web Services provider administrators can publish OC4J Web Services into the enterprise-wide Oracle UDDI Web Services registry using the **Application Server: iAS: OC4J home: Deployed Applications: Deploy Application Wizard** provided through Oracle Enterprise Manager. This wizard takes you through the steps necessary to deploy a J2EE application on the OC4J container, and in this process, there is a step where you can publish Web Services (Web Services servlets contained in the EAR file) to the Oracle UDDI registry.

Web Services provider administrators can also update published Web Services by searching, locating, and drilling down to OC4J Web Services using the **Application Server: iAS: OC4J home: Administration: Related Links: UDDI Registry** link provided through Oracle Enterprise Manager.

Web Services Discovery

Web Services are discovered in the Oracle UDDI Registry by browsing the registry using tools or using the Inquiry API.

Using Tools

Consumers can create their own inquiry browse tool or use third-party tools to browse and drill down for information about Web Services from the Oracle UDDI registry as well as from any other accessible UDDI Version 1.0 Web Services registry. Consumers can use the Inquiry API available for Java programmers to implement their own Web Services discovery interface.

Using the Inquiry API

The Inquiry API lets consumers search for the available registered Web Services by providing find (browse and drill-down) calls and get calls for locating and getting information in each of the four data structures shown in [Figure 10-1](#).

The Inquiry API allows consumers to discover and use Web Services using the Java language. Programs can be written in any language and can use the SOAP protocol to discover Web Services. The Java API is provided as a convenience for Java programmers. The URL for the UDDI registry is `http://<ias-http-server-host-name><ias-port-number>/uddi/inquiry`, where `<ias-http-server-host-name>` is where the Oracle HTTP Server powered by Apache is installed and `<ias-port-number>` is the port number for the Oracle HTTP Server.

The Inquiry API is located in the Oracle9iAS installation directory, `<ORACLE_HOME>/ds/uddi/` for UNIX and `<ORACLE_HOME>\ds\uddi\` for Windows. The API documentation that describes how to use this Inquiry API can be found on the Oracle9iAS Documentation Library as UDDI Client API Reference (Javadoc) under Oracle9iAS Web Services, which is located under the J2EE and Internet applications tab.

A set of sample demo files are located in the following directory:

```
<ORACLE_HOME>/ds/uddi/demo/demo.zip for UNIX  
<ORACLE_HOME>\ds\uddi\demo\demo.zip for Windows
```

Within the `demo.zip` file is a Java program file, `UddiInquiryExample.java`, that provides Java programmers with a starting point that demonstrates the key constructs and the sequence in using the Oracle UDDI client library.

The program example does the following:

- Gets an instance of a `SoapTransportLiaison`. This is an implementation that handles the details of communication between the UDDI client and server using the SOAP protocol and some underlying transport protocol (in this case HTTP).

```
SoapTransportLiaison transport = new OracleSoapHttpTransportLiaison();
```

- Calls a helper method to set up proxy information, if necessary. You can specify HTTP proxy information for accessing the UDDI registry on the command line, using parameters, such as `-Dhttp.proxyHost=<hostname>` `-Dhttp.proxyPort=<portnum>`.

```
setHttpProxy((SoapHttpTransportLiaison)transport);
```

- Uses the `SoapTransportLiaison` and the URL of a UDDI inquiry registry to initialize an instance of the `UddiClient`, which connects to the specified UDDI registry. The `UddiClient` instance is the primary interface by which clients send requests to the UDDI registry.

```
UddiClient uddiClient = new UddiClient(szInquiryUrl, null, transport);
```

- Uses the `UddiClient` to perform a find business request. Specifically, it finds all business entities that start with the letter *T* and prints out the response. Note that input parameters and return values are objects that precisely mimic the XML elements defined in the UDDI specification.

```
// Find a business with a name that starts with "T"
String szBizToFind = "T";
System.out.println("\nListing businesses starting with " + szBizToFind);
// Actual find business operation:
// First null means no specialized FindQualifier.
// Second null means no max number of entries in response.
// (For example, maxRows attribute is absent.)
BusinessList bl = uddiClient.findBusiness(szBizToFind, null, null);
// Print the response.
System.out.println("The response is: ");
List listBusinessInfo = bl.getBusinessInfos().getUddiElementList();
for (int i = 0; i < listBusinessInfo.size(); i++) {
    BusinessInfo businessInfo = (BusinessInfo)listBusinessInfo.get(i);
    System.out.println(businessInfo.getName());
    System.out.println(businessInfo.getFirstDescription());
}
```

- Uses the `UddiClient` to get a `UddiElementFactory` instance. This factory should always be used to create any UDDI objects needed for inquiries.

```
UddiElementFactory uddiEltFactory = uddiClient.getUddiElementFactory();
```

- Uses the `UddiElementFactory` instance to create a `CategoryBag` and its `KeyedReference`, which will be used for searching.

```
CategoryBag cb = (CategoryBag)uddiEltFactory.createCategoryBag();
KeyedReference kr =
    (KeyedReference)uddiEltFactory.createKeyedReference();
kr.setTModelKey(szCategoryTModelKey);
kr.setKeyValue(szCategoryKeyValue);
kr.setKeyName("");
cb.addUddiElement(kr);
```

- Uses the `UddiClient` to perform a find service request. Specifically, it finds a maximum of 30 services, which are classified as application service providers (code 81.11.21.06.00) under the UNSPSC classification in any business entities (no `businessKey` is specified).

```
ServiceList serviceList =
    uddiClient.findService("", cb, null, new Integer(30));
```

- Uses the `UddiElementFactory` instance to retrieve an `XmlWriter` object. To view the raw XML data represented by an object, which extends `UddiElement`, *marshall* the element content to the writer and then flush and close the writer.

```
XmlWriter writerXmlWriter = uddiEltFactory.createWriterXmlWriter(
    new PrintWriter(System.out));
serviceList.marshall(writerXmlWriter);
writerXmlWriter.flush();
writerXmlWriter.close();
```

- Closes the `UddiClient` instance when finished to release resources.

```
uddiClient.close();
```

- Provides URLs (in comments) to the Oracle UDDI registry and four public UDDI registries.

Web Services Publishing

Web Services are published in the Oracle UDDI Registry by using Oracle Enterprise Manager or using the Publishing API.

Using Oracle Enterprise Manager

Using Oracle Enterprise Manager, Web Services provider administrators can publish Web Services in the Oracle UDDI Registry in two ways:

- Navigate to the **Application Server: iAS: OC4J home: Deployed Applications: Deploy Application Wizard**. The **Deploy Application Wizard** takes you through the process of deploying a J2EE application on the OC4J container by assembling the needed application and module deployment descriptors as an Enterprise Archive (EAR) file. See *Oracle9iAS Containers for J2EE User's Guide* for information about EAR file-based deployment of J2EE Web applications.

The second-to-last step, the **Publish Web Services** step, of the **Deploy Applications Wizard** lets Web Services provider administrators publish Web Services (servlets) that are found in the EAR file. Any Web Services servlet in an application that you want to access must be published to the Oracle UDDI Registry to one or more desired categories within one or more of the classifications provided. Any unpublished Web Services in an application appears with the status of `Not Published` and when the Web Services is published, the status changes to `Published`.

- Navigate to the **Application Server: iAS: OC4J home: UDDI Registry: Web Services Details** window. The **Web Services Details** window lets Web Services provider administrators publish J2EE applications to the UDDI Registry after entering all required Service Details and tModel Details information.

Web Services provider administrators can update the discovered published Web Services. They find these published Web Services through the Oracle Enterprise Manager Discovery tool using the **UDDI Registry** link in the **Related Links** column within the **Administration** section of the **OC4J: home** window from the **Application Server: iAS:** window.

Oracle UDDI Registry

The Oracle UDDI Registry uses the following three standard classifications:

- North American Industry Classification System (NAICS)

This is a classification system for each industry and corresponding code. For more information about NAICS, see the Web site at

<http://www.census.gov/epcd/www/naics.html>

- Universal Standard Products and Services Codes (UNSPSC)

This is the first coding system to classify both products and services for use throughout the global marketplace. For more information about UNSPSC, see the Web site at

<http://eccma.org/unspsc/>

- ISO-3166 Geographic classification (ISO-3166)

This a list of all country names and each corresponding two-character code element. For more information about ISO-3166, see the Web site at

<http://www.din.de/gremien/nas/nabd/iso3166ma/>

When Web Services provider administrators publishes Web Services, they can select the classification and the category to which they want to register the Web Services. They have the option of publishing their Web Services to any or all three of these classifications and to as many categories and subcategories as they wish within each classification.

See Also: "[Database Character Set and Built-in ISO-3166 Classification](#)" on page 10-29

Using the Oracle Enterprise Manager Deploy Applications Wizard

Web Services provider administrators can publish Web Services using the Oracle Enterprise Manager **Deploy Applications Wizard**. They do this as follows:

1. Invoke Oracle Enterprise Manager and navigate to the **Application Server: iAS** window and then to the **OC4J: home** window. Locate the **Deployed Applications** section within the **OC4J: home** window and click **Deploy Application** to invoke the **Deploy Application** wizard.
2. Step through each window of the **Deploy Application** wizard and provide the essential information for each step.

3. At the **Publish Web Services** window, select the desired Web Services to register from the list of Web Services known to the application whose status is `Not Published` by clicking its corresponding radio button in the `Select` column, and click **Publish** to continue to the **Web Services Details** window.
4. At the **Web Services Details** window, review, edit, or enter the information as needed in each of the fields in the `Service Details` section and in the `tModel Details` section.
 - a. To add categories for either the Web Services or the tModel sections, click **Browse UDDI Registry** and browse to the desired classification and drill down as needed through each desired category noting all desired category names and values.
 - b. Click **Add Category** to add an empty row of category information.
 - c. Select the desired classification, then enter the value code and its corresponding category name for the desired category.
 - d. Click **Add Category** again to create another empty category row.
 - e. Select the desired classification, enter the value code and its corresponding category name for the desired category.
 - f. Repeat this process (Steps d and e) as many times as it takes to add all the categories to which to register this Web Services.
 - g. After entering all the necessary information on the **Web Services Details** window they are ready to publish the Web Services to the Oracle UDDI Registry, click **OK**. You return to the **Publish Web Services** window.
5. Back at the **Publish Web Services** window, select another Web Services to publish and repeat this entire process again as described in Steps 3 and 4.
6. After publishing all Web Services for this application, click **Next** to continue to the **Summary** window where all the application deployment information can be reviewed.
7. If there are no further changes, click **Deploy** to deploy the J2EE application on the OC4J container. Doing this returns you to the Oracle Enterprise Manager OC4J Home page. Then, to repeat the process of deploying another J2EE application on the OC4J container, click **Deploy Application**.

After deployment, metadata describing the Web Services that you chose to publish has been added to the UDDI registry.

Using the Oracle Enterprise Manager Web Services Details Window

Web Services provider administrators can publish Web Services using the Oracle Enterprise Manager **Web Services Details** window. They do this as follows:

1. Invoke Oracle Enterprise Manager and navigate to the **Application Server: iAS** window and then to the **OC4J: home** window. Locate the **UDDI Registry** link in the **Related Links** column within the **Administration** section of the **OC4J: home** window.

Click the **UDDI Registry** link.

2. The **UDDI Registry** window lets the administrator select one of the three standard classifications: NAICS, UNSPSC, or ISO-3166 by clicking its link or lets you publish Web Services by selecting the **Administration** link.

Click the **Administration** link.

3. At the **Web Services Details** window, enter the required information in each of the fields in the **Service Details** section and in the **tModel Details** section.
 - a. Enter the service name, service description, and service URL to the Servlet in the **Service Details** section.
 - b. Enter the tModel name, tModel description, and the URL to the WSDL document in the **tModel Details** section.
 - c. To add categories for either the Web Services or the tModel sections, click **Browse UDDI Registry** and browse to the desired classification and drill down as needed through each desired category, noting all desired category names and values.
 - d. Click **Add Category** to add an empty row of category information.
 - e. Select the desired classification, then enter the value code and its corresponding category name for the desired category.
 - f. Click **Add Category** again to create another empty category row.
 - g. Select the desired classification, enter the value code and its corresponding category name for the desired category.
 - h. Repeat this process (Steps d and e) as many times as needed to add all the categories to which to register this Web Services.
 - i. After entering all required information on the **Web Services Details** window, publish the Web Services to the Oracle UDDI Registry by clicking **Apply**. This returns you to the **UDDI Registry** window where you can

choose to publish another J2EE application to the UDDI registry by following the same steps again, beginning at Step 2.

Updating Published Web Services in the UDDI Registry

Oracle Enterprise Manager provides a user interface for Web Services provider administrators to browse, drill down, and get information about Web Services published for categories in the Oracle UDDI Registry. Web Services provider administrators can update the discovered published Web Services. They find these published Web Services through the Oracle Enterprise Manager Discovery tool using the **UDDI Registry** link within the **Administration** section of the **OC4J: home** window from the **Application Server: iAS** window.

To update published Web Services using Oracle Enterprise Manager to discover that Web Services, do the following:

1. Invoke Oracle Enterprise Manager and navigate to the **Application Server: iAS** window and then to the **OC4J: home** window. Locate the **UDDI Registry** link in the **Related Links** column within the **Administration** section of the **OC4J: home** window.

Click the **UDDI Registry** link.

2. The **UDDI Registry** window lets the administrator select one of the three standard classification: NAICS, UNSPSC, or ISO-3166 by clicking its link. The **UDDI Registry** window lets the administrator browse any of the three classifications and discover published Web Services associated with any category or subcategory.

Click the desired classification link.

3. The **UDDI Registry: <Classification Name>** window lets the administrator drill down from category to subcategory to discover published Web Services associated with any category or subcategory. Each classification is organized in a hierarchical tree. Navigate down a particular branch by clicking the category name to determine all its subcategory names, and so forth. As you navigate down a branch, also note the change in the category code value.

Navigate to the desired category or subcategory by successively clicking the desired category links.

4. The **Web Services: <Category Name>** window lets the administrator continue to drill down through the categories or you can view all Web Services published in a particular category by selecting the corresponding radio button in the Select column for that category and clicking **View Services**.

Select the corresponding radio button in the Select column for the desired category and click **View Services**.

5. The **Web Services** window lists all Web Services published for that category name. For each Web Services listed for the selected category, its corresponding service name, service key, and business key are also listed. If no Web Services is published for a selected category or subcategory, none is listed.

To view the complete details of a particular published Web Services listed for a category, either click its service name link or select its corresponding radio button in the Select column and click **View Details**.

Click the desired service name link.

6. The **Web Services Details** window displays detailed information for the selected Web Services published in the Oracle UDDI Registry. This information includes:

- **Service Details**

Service details include information such as the Web Services name, Web Services description, and the URL of the Web Services access point.

- Category

- Category information includes the classification and the corresponding code value and its category name.

- **tModel Details**

tModel details include information that describes the interface that the Web Services implements, such as the tModel name, tModel description, and URL to the interface specification, typically a WSDL document.

- Category

- Category information includes the classification and the corresponding code value and its category name.

Category information can be added or deleted for both the **Service Details** and **tModel Details** sections. You can browse the Oracle UDDI Registry (click **Browse UDDI Registry**) looking for categories in which to register this Web Services. You can add categories (click **Add Category**) to which both this Web Services and tModel are to be registered. You can remove categories (click **Delete**) to which this Web Services and tModel are registered.

Service and tModel detail information can be modified by moving the cursor to the appropriate field and making the necessary changes.

After making all selections or completing all changes for this Web Services, click **Apply** to save your changes.

If you have made changes to any field and you decide you want to return to the original set of values for all selections, click **Revert**. The window refreshes with the original set of values for all selections as if you had just begun your current session.

Make your modifications and click **Apply** to save your changes.

7. To discover and update other published Web Services for the same category, at the top of the **Web Services Details** window, select the desired **Web Services:<Classification Name>** link to return to the desired **Web Services:<Classification Name>** window. At this window, select another Web Services to view in more detail, make any necessary changes, and finally click **Apply** to save your changes.

Alternatively, you can select the **UDDI Registry** link at the top of the **Web Services Details** window to return to the **UDDI Registry** window where you can navigate to another classification to discover Web Services for other categories. At each desired category, select the desired Web Services to view its details, make any necessary changes, and finally click **Apply** to save your changes.

Using the Publishing API

Note: The publishing API was released as the UDDI v1.0 Compliance Patch Kit available on Metalink as Patch number 2367149. This Patch Kit must be installed for Oracle9iAS release 2 (9.0.2) and Oracle9iAS release 3 (9.0.3) for server-side support of the publishing API.

The UDDI publishing API lets programmers, following authentication, publish Web Services by providing save and delete calls for each of the four key UDDI data structures (businessEntity, businessService, bindingTemplate, and tModel).

The publishing API allows programmers to publish Web Services using the Java language. Programs can be written in any language and use the SOAP protocol to publish Web Services. The Java API is provided as a convenience for Java programmers.

The Publishing API is located in the Oracle9iAS installation directory, <ORACLE_HOME>/ds/uddi/ for UNIX and <ORACLE_HOME>\ds\uddi\ for Windows. The API documentation that describes how to use this publishing API can be found on the Oracle9iAS Documentation Library CD-ROM as UDDI Client API Reference (Javadoc) under Oracle9iAS Web Services, which is located under the J2EE and Internet Applications tab.

A set of sample demo files are located in the <ORACLE_HOME>/ds/uddi/demo.zip file for UNIX and the ORACLE_HOME\ds\uddi\demo.zip file for Windows.

Within the demo.zip file is a Java program file, UddiPublishingExample.java, that provides Java programmers with a starting point that demonstrates the key constructs and the sequence in using the Oracle UDDI client library.

The program example does the following:

- Gets an instance of a SoapTransportLiaison. This is an implementation that handles the details of communication between the UDDI client and server using the SOAP protocol and some underlying transport protocol (in this case HTTP).

```
SoapTransportLiaison transport =  
    new OracleSoapHttpTransportLiaison();
```

- Uses the SoapTransportLiaison and the URL of a UDDI publishing registry to initialize an instance of the UddiClient, which connects to the specified UDDI registry. The UddiClient instance is the primary interface by which clients send requests to the UDDI registry. Authentication is done by the transport layer (HTTP BASIC in this example).

```
TransportAuthenticationLiason auth =  
    new TransportAuthenticationLiason();  
UddiClient uddiClient =  
    new UddiClient(null, szPublishingUrl, transport, auth);
```

- Performs authentication. You should make this call before doing any publishing.

```
UddiClient.authenticate();
```

- Uses the UddiClient to get a UddiElementFactory instance. This factory should always be used to create any UDDI objects needed.

```
UddiElementFactory uddiEltFactory =  
    uddiClient.getUddiElementFactory();
```

- Performs various publishing operations that include creating and saving a `tModel`, a `businessEntity`, a `businessService`, and a `bindingTemplate` data structure for the purpose of creating a business that provides a Google-interface-compatible service.
- Creates a `tModel` data structure that represents a Google-compatible service by using the `UddiElementFactory` instance.

```
TModel tModel = (TModel)uddiEltFactory.createTModel();
tModel.setName("urn:google.com:search-interface");
```

- Creates and includes the `OverviewDoc` data structure in the `tModel` data structure by using the `UddiElementFactory` instance.

```
OverviewDoc overviewDocTm =
    (OverviewDoc)uddiEltFactory.createOverviewDoc();
    tModel.setOverviewDoc(overviewDocTm);
overviewDocTm.setOverviewURL("http://api.google.com/GoogleSearch.wsdl");
```

- In the `tModel` data structure, uses the `UddiElementFactory` instance to create a `CategoryBag` data structure and its `KeyedReference`, which will be used for searching. Classify the `tModel` data structure as a SOAP/WSDL-based interface and put it under the "applicable service providers" category.

```
CategoryBag catBagTm =
    (CategoryBag)uddiEltFactory.createCategoryBag();
tModel.setCategoryBag(catBagTm);
```

```
KeyedReference krTm1 =
    (KeyedReference)uddiEltFactory.createKeyedReference();
```

```
catBagTm.addUddiElement(krTm1);
krTm1.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UDDI_TYPE);
krTm1.setKeyName("wsdlSpec");
krTm1.setKeyValue("wsdlSpec");
```

```
KeyedReference krTm2 =
    (KeyedReference)uddiEltFactory.createKeyedReference();
catBagTm.addUddiElement(krTm2);
krTm2.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UDDI_TYPE);
krTm2.setKeyName("wsdlSpec");
krTm2.setKeyValue("wsdlSpec");
```

```
KeyedReference krTm3 =
```

```

(KeyedReference)uddiEltFactory.createKeyedReference();
catBagTm.addUddiElement(krTm3);
krTm3.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UNSPSC_7_3);
krTm3.setKeyName("application service providers");
krTm3.setKeyValue("81.11.21.06.00");

```

- Publishes the Google search interface tModel business operation.

```

System.out.println("\nPublish the google search interface tModel.");
TModel tMSaved = uddiClient.saveTModel(tModel);
String szGoogleTModelKey = tMSaved.getTModelKey();
System.out.println("The tModel is saved with tModelKey assigned to be " +
    szGoogleTModelKey);

```

- Creates a businessEntity data structure that represents a Google-compatible service by using the UddiElementFactory instance.

```

BusinessEntity businessEntity =
    (BusinessEntity)uddiEltFactory.createBusinessEntity();
businessEntity.setName("ACME search Inc.");

```

In the businessEntity data structure, uses the UddiElementFactory instance to create a CategoryBag data structure and its KeyedReference data structure, which will be used for searching. Classify the businessEntity data structure under the "information services and data processing services" category.

```

KeyedReference krBel =
    (KeyedReference)uddiEltFactory.createKeyedReference();
catBagBe.addUddiElement(krBel);
krBel.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_NAICS_1997);
krBel.setKeyName("Information Services and Data Processing Services");
krBel.setKeyValue("514");

```

- Creates a businessService data structure that represents a Google-compatible service by using the UddiElementFactory instance.

```

BusinessServices businessServices =
    (BusinessServices)uddiEltFactory.createBusinessServices();
businessEntity.setBusinessServices(businessServices);
BusinessService businessService =
    (BusinessService)uddiEltFactory.createBusinessService();
businessServices.addUddiElement(businessService);
businessService.setName("ACME Web Search service");

```

In the businessService data structure, uses the UddiElementFactory instance to create a CategoryBag data structure and its KeyedReference data structure,

which will be used for searching. Classify the `businessService` data structure under the "application service providers" category.

```
CategoryBag catBagBs =
    (CategoryBag)uddiEltFactory.createCategoryBag();
businessService.setCategoryBag(catBagBs);
KeyedReference krBs1 =
    (KeyedReference)uddiEltFactory.createKeyedReference();
catBagBs.addUddiElement(krBs1);
krBs1.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UNSPSC_7_3);
krBs1.setKeyName("application service
providers");krBs1.setKeyValue("81.11.21.06.00");
```

- Creates the `bindingTemplates` data structure that represent a Google-compatible service by using the `UddiElementFactory` instance.

```
BindingTemplates bindingTemplates =
    (BindingTemplates)uddiEltFactory.createBindingTemplates();
businessService.setBindingTemplates(bindingTemplates);
BindingTemplate bindingTemplate =
    (BindingTemplate)uddiEltFactory.createBindingTemplate();
bindingTemplates.addUddiElement(bindingTemplate);
```

- Creates and includes the access point in the `bindingTemplates` data structure by using the `UddiElementFactory` instance.

```
AccessPoint accessPoint =
    (AccessPoint)uddiEltFactory.createAccessPoint();
bindingTemplate.setAccessPoint(accessPoint);
accessPoint.setUrlType("http");
accessPoint.setContent("http://foobar.net/search-g");
```

- Creates and includes the `tModel` instance details in the `bindingTemplates` data structure by using the `UddiElementFactory` instance.

```
TModelInstanceDetails tModelInstanceDetails =
    (TModelInstanceDetails)uddiEltFactory.createTModelInstanceDetails();
bindingTemplate.setTModelInstanceDetails(tModelInstanceDetails);
```

- Declares that the `bindingTemplate` data structure implements the Google search interface.

```
TModelInstanceInfo tModelInstanceInfo =
    (TModelInstanceInfo)uddiEltFactory.createTModelInstanceInfo();
tModelInstanceDetails.addUddiElement(tModelInstanceInfo);
tModelInstanceInfo.setTModelKey(szGoogleTModelKey);
```

- Publishes the `businessEntity` data structure and its containing `businessService` and `bindingTemplate` data structures.

```
System.out.println("Publish the ACME Search Inc. businessEntity...");
BusinessEntity bESaved = uddiClient.saveBusiness(businessEntity);
System.out.println("The saved businessEntity (in XML) is:");
```

- Uses the `UddiElementFactory` instance to retrieve an `XmlWriter` object. To view the raw XML data represented by an object, which extends `UddiElement`, *marshall* the element content to the writer and then flush and close the writer.

```
XmlWriter writerXmlWriter =
    uddiEltFactory.createWriterXmlWriter(new PrintWriter(System.out));
bESaved.marshall(writerXmlWriter);
writerXmlWriter.flush();
writerXmlWriter.close();
```

- Closes the `UddiClient` instance when finished to release resources and to log out from the registry.

```
uddiClient.close();
```

UDDI Registry Administration

The following sections describe new UDDI registry administration features.

User Management

Oracle9iAS Release 2 UDDI Registry has two types of users, as defined by two different J2EE security roles.

- **uddipublisher:** Users with the `uddipublisher` security role can access the publishing end point and save UDDI entities in the registry.
- **uddiadmin:** Users with the `uddiadmin` security role can access the publishing end point and perform administrative activities.

User management, including operations such as creation, deletion, suspension, role management, and so forth, is handled by OC4J Java Authentication and Authorization (JAAS) service. Refer to *Oracle9iAS Containers for J2EE Services Guide* for more information.

There is a set of additional UDDI-specific user operations. See [User Account Management](#) on page 10-21 for more information about UDDI registry administration.

Performance Monitoring and Tuning

On the back end of an Oracle database, UDDI servlets, and the associated JDBC connection pools can all be monitored using Oracle Enterprise Manager and other standard database monitoring and tuning utilities.

In an OC4J standalone environment, performance information is typically available at

```
http://<oc4j-host-name>:<port-number>/dmsoc4j/Spy
```

Data Backup and Restore Operations

Registry data backup and restore operations can be done by using the standard Oracle database backup and restore operations.

Using the Command-Line Tool `uddiadmin.jar`

The command-line tool `uddiadmin.jar` is located in the `ds/uddi/lib/uddiadmin.jar` file for UNIX and in the `ds\uddi\lib\uddiadmin.jar` file for Windows. Administrators can use this tool for various administrative activities. In general, the command-line tool takes the command-line parameters of the following form:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>  
[-verbose] <action to perform and additional parameters>
```

The user name is `ias_admin` and the password is `ias_admin`.

Note that the `-verbose` option will cause stack trace information to be printed out when an exception is encountered.

[Server Configuration](#) on page 10-20 through [Import Operation](#) on page 10-22 and [Built-in Validated Category Management](#) on page 10-23 describe the administrative uses of this command-line tool.

Server Configuration

The following parameters are used for server configuration operations. See [Server Configuration Parameters Reference Information](#) on page 10-26 for more information about these configuration parameters.

```
Parameter: <registry publishing URL> <username> <password>  
[-verbose] -getProperties
```

Description: Lists the current registry configuration parameters.

For example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
[-verbose] -getProperties
```

```
Parameter: <registry publishing URL> <username> <password>
[-verbose] -setProperty <name>=<value>
```

Description: Changes the value of the named configuration parameter. The UDDI registry J2EE application needs to be restarted for the parameters to take effect.

User Account Management

In general, user management is handled by the OC4J JAAS service. This section describes UDDI-registry-specific operations that are not handled by the OC4J JAAS service. The following parameters are used for user account management:

```
Parameter: <registry publishing URL> <username> <password>
[-verbose] -getUsers
```

Description: Lists all existing users who have entities in the registry.

For example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
[-verbose] -getUsers
```

```
Parameter: <registry publishing URL> <username> <password>
[-verbose] -getUserDetail <username_to_retrieve>
```

Description: Retrieves the details of the named user, currently the authorizedName of each user.

Administrative Entity Management

The following parameters are used for administrative entity management.

```
Parameter: <registry publishing URL> <username> <password>
[-verbose] -deleteEntity [-businessKey <businessKey> |
-serviceKey <serviceKey> | -bindingKey <bindingKey> |
-tModelKey <tModelKey>]
```

Description: Deletes the named entity irrespective of the owner of the entity. Note that this operation performs a nonpermanent delete (hide) operation in the case of a tModel entity.

```
Parameter: <registry publishing URL> <username> <password>
[-verbose] -destroyTModel <tModelKey>
```

Description: Permanently deletes the named tModel from the registry (as opposed to the UDDI-defined delete_tModel call, which is just hiding the tModel).

Parameter: <registry publishing URL> <username> <password>
[-verbose] -changeOwner <new username> [-businessKey
<businessKey> | -tModelKey <tModelKey>]

Description: Changes the ownership of the named entity to the new specified user.

Import Operation

The following parameter is used for importing entities:

Parameter: <registry publishing URL> <username> <password>
[-verbose] -import [-businesses <filename> | -tmodels
<filename>]

Description: Imports all businessEntity or tModel data structures in the named file. For importing the businessEntity data structure, the named file (<filename>) for importing should contain a UDDI businessDetail XML document. For importing tModels, the named file should contain a UDDI tModelDetail XML document. By importing them, the entityKeys (such as, businessKey, serviceKey, bindingKey, tModelKey) are preserved. The operatorName and authorizedName fields, however, are not preserved. The operatorName field will be replaced by the operatorName configuration parameter of the registry. The owner of the imported entities is the administrator; hence, the authorizedName field will be the authorizedName of the administrator.

The import parameter is particularly useful in importing the well-known service interface specification tModel and classification tModel data structures from some authoritative sources.

Because the entity keys are preserved, administrators should be careful in evaluating the source of the entities to ensure that there will not be a collision in entity keys.

Database Configuration

The following sections describe some database-specific configuration information.

Database Character Set Should Be UTF-8 The database character set should be UTF-8 to accommodate all possible characters. However, if a customer knows for sure that the data to be stored in the registry contains characters of a specific country or region (such as western Europe), the customer may use the appropriate database character set.

Functional Index Should Be Enabled The functional index must be enabled to support index-based case-insensitive search. The following `init.ora` parameters are involved: `query_rewrite_enabled=true`

In addition, the cost-based optimizer must be turned on for analyzing all tables or indexes in the UDDISYS schema. For example:

```
execute dbms_stats.gather_schema_stats(ownname=>'UDDISYS',cascade=>true);
```

Accuracy of Modified Timestamps of UDDI Entities The accuracy of modified timestamps of UDDI entities is dependent on the version and compatibility of the database. If the database compatibility is release 9.0.1 or higher, the modified timestamps are of SQL type `TIMESTAMP` with an accuracy up to microseconds. If the database compatibility is below release 9.0.1, the modified timestamps are of SQL type `DATE` with an accuracy up to seconds.

Built-in Validated Category Management

Oracle9iAS Release 2 UDDI Registry can perform a spell-check form of category value validation, increasing the data accuracy in the registry. An administrator can add or remove the set of categories that will be validated by the registry.

Adding a New Category for Registry-based Validation To add a new category, you must load the category values into the database and register the category with the registry. Perform the following steps:

1. Load the category values into the database. To do this, the category values of the entire category should be in a file using the following format:
 - Each line of the file describes one category value in the category. It should be in the following format:


```
<category value> | <description of category value>
  | <category value of the parent>
```
 - If a category value is a root value, for example, it has no parent, the category value of the parent should be set to itself.
 - The line of a category value should occur before all of its descendants.

Examples can be found in the `ds/uddi/taxonomy` directory for UNIX and in the `ds\uddi\taxonomy` directory for Windows. Excerpts from the NAICS file are as follows:

```
22|Utilities|22
221|Utilities|22
```

2211|Electric Power Generation, Transmission|221

It is recommended that you save the file with UTF-8 encoding.

2. Create a SQL*Loader control file to load the category file. An example is `ds/uddi/admin/naics-97.ctl` for UNIX and `ds\uddi\admin\naics-97.ctl` for Windows. Copy the file and replace the name of the category file in the control file with the one you create.
3. Load the category file to the database using SQL*Loader. Refer to the SQL*Loader sections of *Oracle9i Database Utilities* for more information about using SQL*Loader.
4. Register the category with the registry as follows:
 - a. Register the category by saving a new tModel for it in the registry. For example, look at the tModel named `ntis-gov:naics:1997`. You can use the included sample Web applications link `http://<ias-web-server-host>:<ias-web-server-port>/uddi/` or a third-party UDDI v1.0-compliant tool. If the tModel data structure has been defined in some other registry, you can also import it (instead of creating a new one, which results in different tModelKeys entities) using the `uddiadmin.jar` utility. See [Import Operation](#) on page 10-22 for more information.
 - b. Configure the registry so that it recognizes the category that must be validated using the command-line administrative tool, `uddiadmin.jar`. For example, to add a new tModel entity with key `UUID:FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF0`, issue the `setProperty` command for the property `oracle.uddi.server.categoryValidationTModelKeys` as follows:

```
java -jar uddiadmin.jar <registry publishing URL> <username>
  <password> -setProperty
"oracle.uddi.server.categoryValidationTModelKeys=
'UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4' ,
'UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88' ,
'UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2' ,
'UUID:CD153257-086A-4237-B336-6BDCBDC6634' ,
'UUID:FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF0' "
```

Notice that because the `setProperty` command defines all categories that need to be validated, to add a new category, you should set the property with all the existing tModelKeys values plus the new tModelKey value that needs to be added.

Removing a Category from Registry-based Validation To remove a category from registry-based validation, you should deregister the category with the registry and remove the category values in the database. Perform the following steps:

1. To deregister the category with the registry, you should remove it from the list of validated categories using the `uddiadmin.jar setProperty` command for the property `oracle.uddi.server.categoryValidationTModelKeys`.

You do not have to (and in general should not) delete the `tModel` entity from the registry.

2. To remove the category values from the database, use the SQL*Plus script `udivcrm.sql` in the `ds/uddi/admin` directory for UNIX and in the `ds\uddi\admin` directory for Windows. For example:

```
sqlplus uddisys/uddisys @udivcrm.sql
```

When running this script, you will be prompted for the `tModelKey` value of the category to be removed. You should see that a set of rows is deleted. If the result shows that 0 rows are deleted, you have entered an invalid `tModelKey` value.

Transport Security

The Inquiry API in general does not require authentication. However, if the inquiry end point needs to be protected, transport level authentication, such as HTTP BASIC authentication and HTTPS SSL client authentication, can be enabled by configuring the `web.xml` file. A security role, `uddiguest`, is reserved for accessing the protected inquiry end point. Refer to *Oracle9iAS Containers for J2EE Services Guide* and *Oracle9iAS Containers for J2EE User's Guide* for more information about security roles and related security configuration.

For the Publishing API, you may want to allow HTTPS access only. To disable HTTP access, edit the `web.xml` file of the `orauddi` application to enforce data confidentiality and make adjustments to HTTP servers accordingly. Refer to Chapter 8 Security in *Oracle9iAS Containers for J2EE User's Guide* and *Oracle9iAS Containers for J2EE Services Guide* for more information. For example, to disable HTTP access in the `web.xml` file, use the following code:

```
<user-data-constraint>  
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>  
</user-data-constraint>
```

Additional Information

The UUID generation algorithm used generates version 4 UUID, which creates UUIDs from random numbers.

All built-in tModel data structures as specified in the UDDI v1 specification are included. An additional tModel data structure `uddi-org:operators`, defined in the UDDI v2 specification, is also included to classify the bootstrap node `businessEntity` that represents the UDDI registry itself.

Server Configuration Parameters Reference Information

This section describes reference information for some UDDI server configuration parameters. It is divided into two sections:

- Advanced configuration parameters
- Installation or first-use configuration parameters

These server configuration parameters are referenced in [Server Configuration](#) on page 10-20. As each example shows, these configuration parameters can be changed only by using the command-line administration tool, `uddiadmin.jar`, which is described in [Using the Command-Line Tool `uddiadmin.jar`](#) on page 10-20.

Advanced Configuration Parameters The following UDDI server parameters can be configured for advanced use.

Parameter name: `identifierValidation` (Advanced use parameter)

Description: Determine if the registry internally validates `identifierBag` upon `save_XXX` calls. In a typical case, there is no reason to change it.

Parameter Type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>  
  [-verbose] -setProperty oracle.uddi.server.identifierValidation=true
```

Parameter name: `operatorCategory` (Advanced use parameter)

Description: If `categoryValidation` is true, this property determines whether or not the `uddi-org:operators` category scheme is validated. If `uddi-org:operators` is

validated, in a single-node scenario, it implies no additional businessEntity data structures can be classified in uddi-org:operators

Parameter type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
[-verbose] -setProperty
    oracle.uddi.server.categoryValidation.operatorCategory=true
```

Parameter name: categoryValidation (Advanced use parameter)

Description: Determine if the registry internally validates categoryBag upon save_XXX calls. In a typical case, there is no reason to change it.

Parameter type/allowable values: boolean (true, false)

Initial value: true

Typical value: true

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
[-verbose] -setProperty oracle.uddi.server.categoryValidation=true
```

Parameter name: categoryValidationTModelKeys (Advanced use parameter)

Description: If categoryValidation is true, this property defines the list of categories to be validated internally by the registry. The validation is essentially a spell-check: compare the value with the set of valid values to make sure it is one.

Parameter type/allowable values: A list in the form of '<tModelKey1>', '<tModelKey2>', '<tModelKey3>'.

Initial value: 'UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4', which represents (uddi-org:types classification). The pre-installed value, however, is UDDI types classification plus the three classifications defined in UDDI v1 specification: (uddi-org:types, uddi-org:iso-ch:3166-1999, ntis-gov:naics:1997, unspsc-org:unspsc).

Note: The uddi-org:types classification *should not* be removed from the list.

Typical value: The pre-installed value.

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
  [-verbose] -setProperty
"oracle.uddi.server.categoryValidationTModelKeys=
'UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4' ,
'UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88' ,
'UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2' ,
'UUID:CD153257-086A-4237-B336-6BDCBDCC6634' "
```

Installation or First-Use Parameters The following two parameters `operatorName` and `businessEntityURLPrefix` should be changed immediately after an installation, but should not be changed afterward.

Parameter Name: `operatorName`

Description: The name of the operator, appearing in the `operator` attribute of UDDI responses.

Parameter type/allowable values: A non-null string.

Initial value: `OracleUddiServer`

Typical value: `<domain of the UDDI registry>/uddi`

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
  [-verbose] -setProperty oracle.uddi.server.operatorName=OracleUddiServer
```

Parameter Name: `businessEntityURLPrefix`

Description: This parameter customizes the URL prefix for the generated `businessEntity` URL for `discoveryURL[@useType='businessEntity']`. The actual URL is in the form of `<businessEntityURLPrefix>?businessKey=<businessKey>`.

Parameter type/allowable values: A valid URL.

Initial value: The UDDI registry will prompt an administrator for an initial value upon server initialization.

Typical value: The host name and port should be the host name and port of the Web server (which may or may not be the same as the servlet container).

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
[-verbose] -setProperty oracle.uddi.server.businessEntityURLPrefix=
```

Parameter name: defaultLang

Description: The default language of the registry, used when a description element given in a save_XXX call is not qualified by the xml:lang attribute.

Parameter type/allowable values: Values of xml:lang

Initial value: en

Typical value: The locale of the primary region the registry serves.

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry publishing URL> <username> <password>
[-verbose] -setProperty oracle.uddi.server.defaultLang=en
```

Database Character Set and Built-in ISO-3166 Classification

The UDDI specification mandates that the registry support the full UTF-8 character set. Oracle recommends, though does not require, using UTF-8 as the character set for the Oracle9iAS infrastructure database if the UDDI registry is used.

If the database is not configured with the UTF-8 character set or its equivalent or superset, there could be data corruption and error due to loss in character set conversion to or from UTF-8. Refer to *Oracle9i Globalization Support Guide* for details.

In particular, the descriptions in UDDI built-in ISO-3166 classification contains descriptions with non-ASCII characters, such as some Western European characters and some Eastern European characters for the names of cities or regions. In order to support the non-UTF-8 database, all non-ASCII characters in the descriptions are replaced with ASCII characters as an approximation.

If you do have a UTF-8 database, you can upgrade the built-in ISO-3166 classification to the one with accurate descriptions using the following instructions:

- Delete the existing ISO-3166 classification by running the SQL script, `clrISO.sql`, for example:

```
cd <ORACLE_HOME>/ds/uddi/admin
sqlplus system/manager @clrISO.sql
```

- Load the ISO-3166 classification with accurate descriptions by using SQL*Loader control file `iso3166-99.ct1`, for example:

```
cd <ORACLE_HOME>/ds/uddi/admin
sqlldr userid=system/manager control=iso3166-99.ct1
```

Recommended Configuration for a Production Environment

The following information describes some post-installation configuration steps that you should do immediately after the installation. These steps are not mandatory, but are highly recommended in a production environment.

- **User repository setup:** By default, a file-based JAAS repository (`jazn-data.xml`) is deployed with two users: `ias_admin` and `publisher`. You should customize the repository. For example, LDAP directory or a centralized `jazn-data.xml` should be your user repository. Refer to Chapter 7 Managing the JAAS Provider in *Oracle9iAS Containers for J2EE Services Guide* for more information. Examples of using command-line utilities are as follow:

```
java -jar jazn.jar -listusers jazn.com
java -jar jazn.jar -listroles jazn.com
java -jar jazn.jar -adduser jazn.com myname mypassword
java -jar jazn.jar -grantrole webserviceuserrole jazn.com myname
java -jar jazn.jar -revokerole webserviceuserrole jazn.com myname
java -jar jazn.jar -remuser jazn.com myname
```

- **Security for publishing the end point:** By default, HTTP access is enabled. However, HTTPS access is recommended for security concerns. See [Transport Security](#) on page 10-25 for more information about disabling HTTP access.

Consuming Web Services in J2EE Applications

This chapter describes how to consume Web Services in J2EE applications. Two types of Web-based information or services are supported:

- HTML/XML streams accessed through HTTP, see [Consuming XML or HTML Streams in J2EE Applications](#).
- SOAP-based Web Services described using WSDL, see [Consuming SOAP-Based Web Services Using WSDL](#).

In addition, when a Java2 Enterprise Edition (J2EE) application acquires a WSDL document at runtime, the dynamic invocation API is used to invoke any SOAP operation described in the WSDL document. See [Dynamic Invocation of Web Services](#) for information about how to use the dynamic invocation API.

Consuming XML or HTML Streams in J2EE Applications

Oracle9iAS Containers for Java2 Enterprise Edition (J2EE), also referred to as OC4J, provides support for processing XML or HTML streams accessible through the HTTP/S protocols for consuming into J2EE applications. The Web Service HTML/XML Stream Processing Wizard assists developers in creating an Enterprise JavaBean (EJB) whose methods will access and process the desired XML or HTML streams.

In the simplest case, suppose a developer wants programmatic access to an XML news feed accessible through a static URL. In another case, a developer wants programmatic access to a dynamic stream accessed through the submission of an HTML form. Now, suppose HTTP/S basic authentication is required to access either of these two types of resources. In either case, developers must be able to quickly and easily process XML or HTML streams, thus consuming these Web Services in their own specific J2EE applications.

Web Service HTML/XML Stream Processing Wizard

Developers using the Web Service HTML/XML Stream Processing Wizard first specify how the XML/HTML stream should be accessed and then define the desired processing actions on the stream.

Developers can choose among the following options when specifying their XML/HTML stream access:

1. Supply a *static* URL that has no parameters.
2. Define an HTML form to be submitted, its *action* URL, and its parameters.
3. Supply the URL of an HTML page where the form to be submitted is defined.

Additional HTTP-related settings can also be specified. They include HTTP proxy settings, authentication, and HTTPS Oracle Wallet information.

To assist developers in defining the processing to be applied to the stream, the wizard accesses the XML/HTML stream (prompting the developer for sample form values if necessary). The resulting sample XML/HTML stream is shown in a searchable XML tree. Through the wizard, the developer can perform the following actions:

1. Leave the XML stream unprocessed and have the service response be the original stream.
2. Select a node in the XML tree and have the service response be an XML Element corresponding to that node.

3. Select a node in the XML tree and define through the wizard a simple transformation for it. The service response will be the result of that transformation. Optionally, the same transformation can be applied to all the siblings of the selected node.

The wizard allows developers to create multi-operation services by repeating the steps described previously for each operation.

Note: JavaScript code contained in HTML streams will be ignored and not processed.

Upon completion of the steps described previously, the Web Service HTML/XML Stream Processing Wizard generates a JavaBean and an EJB whose methods perform the appropriate HTTP request and processing of the XML or HTML response. If it is necessary to support multi-operation services, then the generated stub keeps the HTTP session information in its state, and the generated stub is modeled as a stateful session EJB user option. The resulting Java code is then compiled and archived, creating the required .ear file that the developer can immediately deploy in Oracle9iAS.

Sample Use Scenarios

This section describes two sample use cases for a better understanding of how to use the Web Service HTML/XML Stream Processing Wizard.

Handling an XML or HTML Stream Accessed Through a Static URL

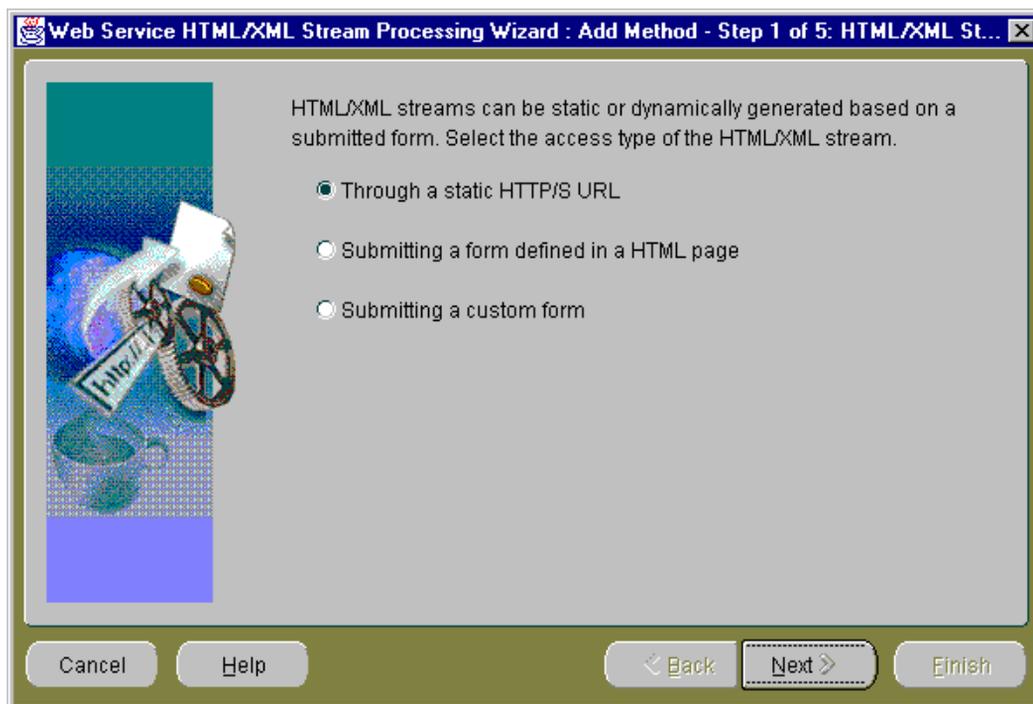
The following steps generate the Java stubs that consume a static XML or HTML stream.

1. Invoke the Web Service HTML/XML Stream Processing Wizard using the following command:

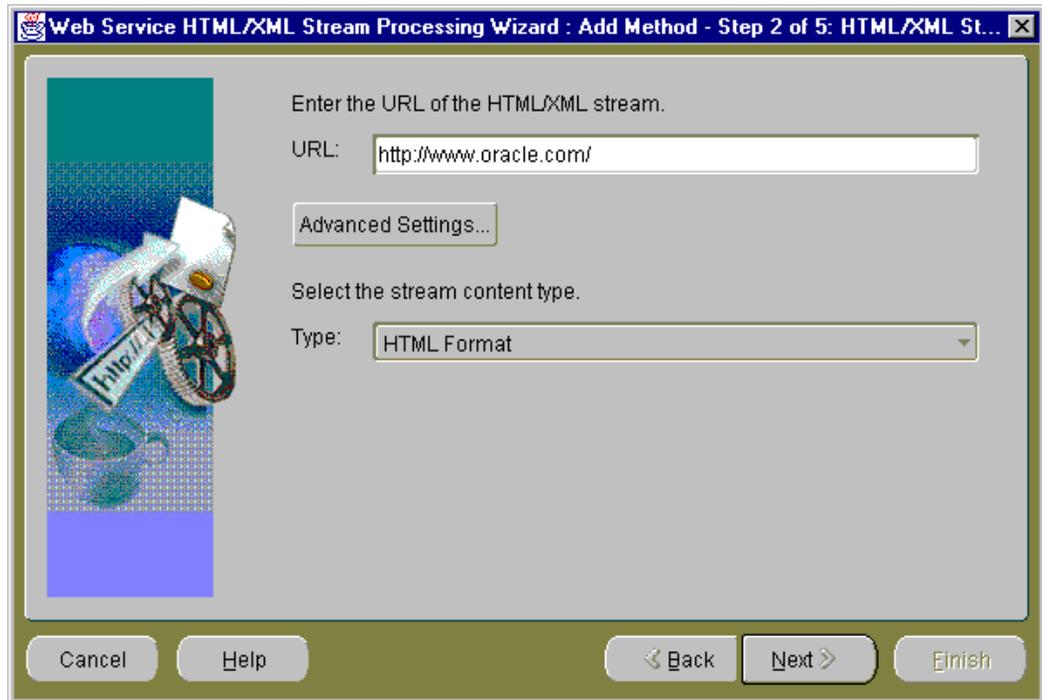
```
java -jar WebServicesHtmlXmlWizard.jar
```

Note: The WebServicesHtmlXmlWizard.jar file is located in your `$ORACLE_HOME/webservices/lib` Oracle9iAS installation directory for UNIX or `%ORACLE_HOME%\webservices\lib` Oracle9iAS installation directory for Windows.

2. In **Step 1 of 5: HTML/XML Stream Type**, select the first option **Through a static HTTP/S URL**, then click **Next** to continue to the next step.

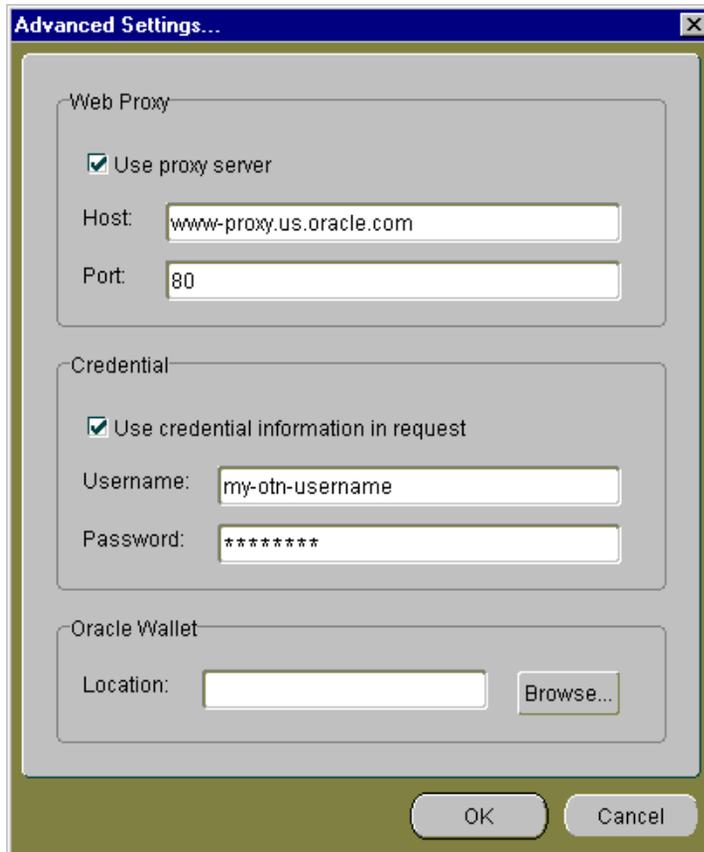


3. In **Step 2 of 5: HTML/XML Stream URL**, enter the URL of the HTML page in which you want to access the resource. Accept the default stream content type, HTML Format. If the stream content type is XML, then select the XML Format content type.



If you must access the URL from outside a firewall, click **Advanced Settings**. For this example, assume you must go through a firewall to access the desired URL.

4. At the **Advanced Settings** pop-up window, select **Use proxy server** and place a checkmark in the box, then enter the host address and port number for your proxy server. Click **OK** to return to the **HTML/XML Stream URL** window. Click **Next** to continue to the next step.



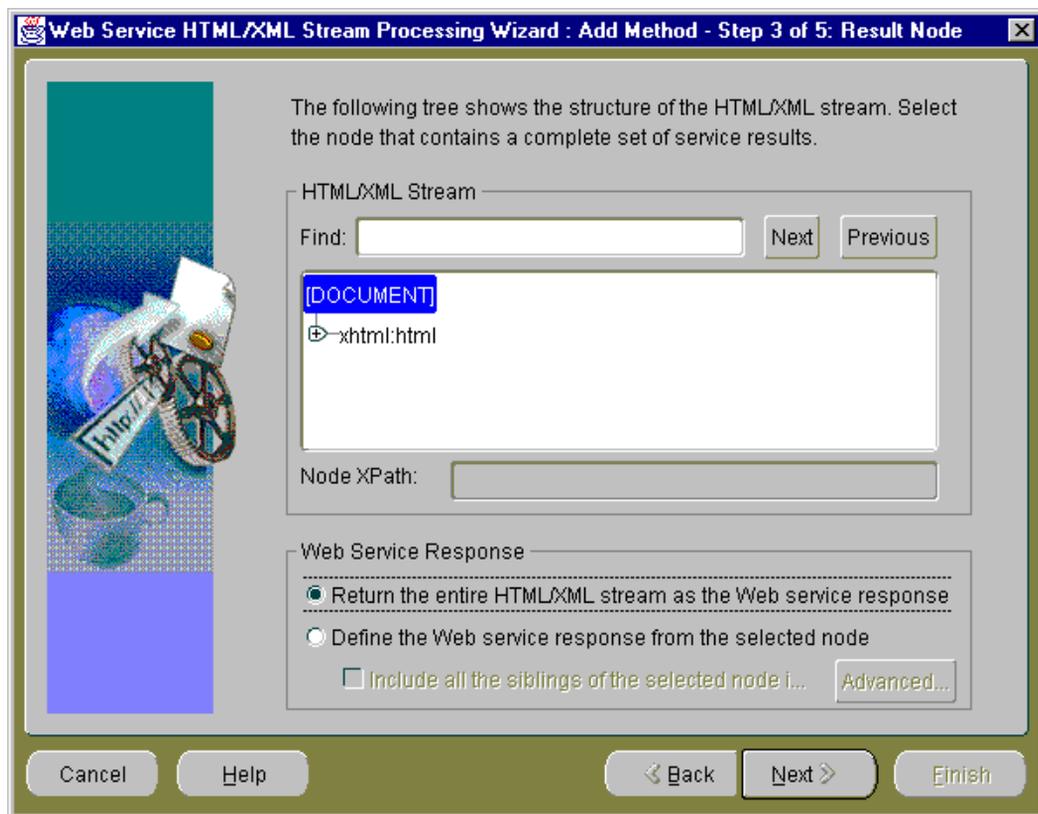
Note: If the URL you are accessing requires basic HTTP authentication, select **Use credential information in request**, then enter the user name and password in the **Credential** section of the **Advanced Settings** pop-up window.

If the URL you are accessing requires basic HTTPS authentication, use the **Oracle Wallet** section of the **Advanced Settings** pop-up window to enter the Wallet location.

5. In **Step 3 of 5: Result Node**, the HTML/XML Stream tree is shown in the **HTML/XML Stream** section. Ignore this HTML/XML stream tree for now.

Note: You may need to move your mouse to the bottom of the wizard window, grab the edge (note the double-headed, vertically oriented arrow), and pull the window down to expand it so you can see the **Service Response Tree** pane.

Note: If the original HTML/XML stream was in HTML, the wizard first converts it into XHTML (making it a valid XML document), and then shows its structure in the tree.



Then, for the **Web Service Response** section, select how you want to build the Web Service response; you can select one of two options:

- **Return the entire HTML/XML stream as the Web service response**
- **Define the Web service response from the selected node**

For this sample use, you want to take the entire page content as the Web page content, therefore, select the first option, **Return the entire HTML/XML stream as the Web service response**.

Note: If you select the **Define the Web service response from the selected node** option, a **Service Response Fields** window displays. This option lets you finalize the output extracting process by letting you select elements of interest to be outputs and assign names to the output fields. See list item number 8 on page 11-24 for more information about the **Service Response Fields** window.

Click **Next** to continue to the next step.

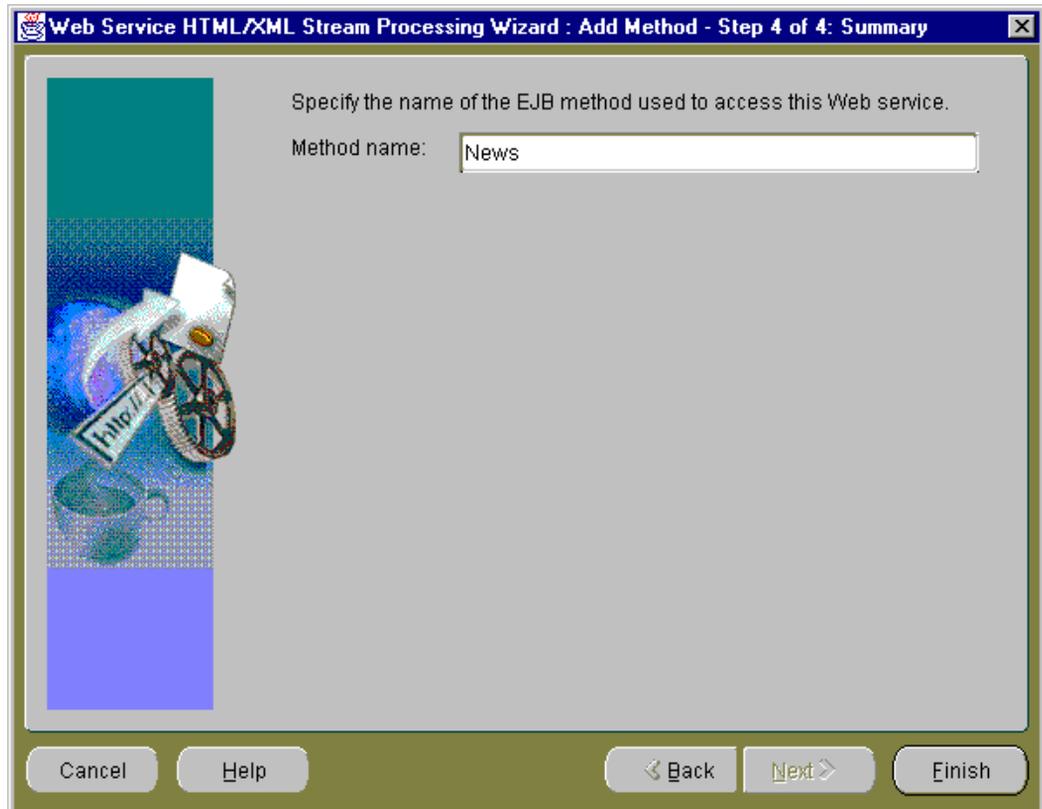
6. In **Step 4 of 4: Summary**, you must specify your EJB method name.

If this is the first HTML or XML stream you are processing in this session, then you will see only the EJB method name. You need to enter only the EJB method name and click **Finish** to complete the operation of creating your EJB method.

If this is the second or subsequent HTML or XML stream you are processing in this session, then the suggested EJB method information is displayed for your EJB method, describing the name for the J2EE application, the EJB name, the name of the service package, and the name of the service class. By default, the names are preselected based on the information that is already known.

If you want to retain this suggested EJB method information and display it in the next step, the **Console** window, then leave the option **Use the method information to define EJB as follows** selected (checkmarked). If not, deselect this option and the EJB method information that appeared previously will be displayed in the **Console** window.

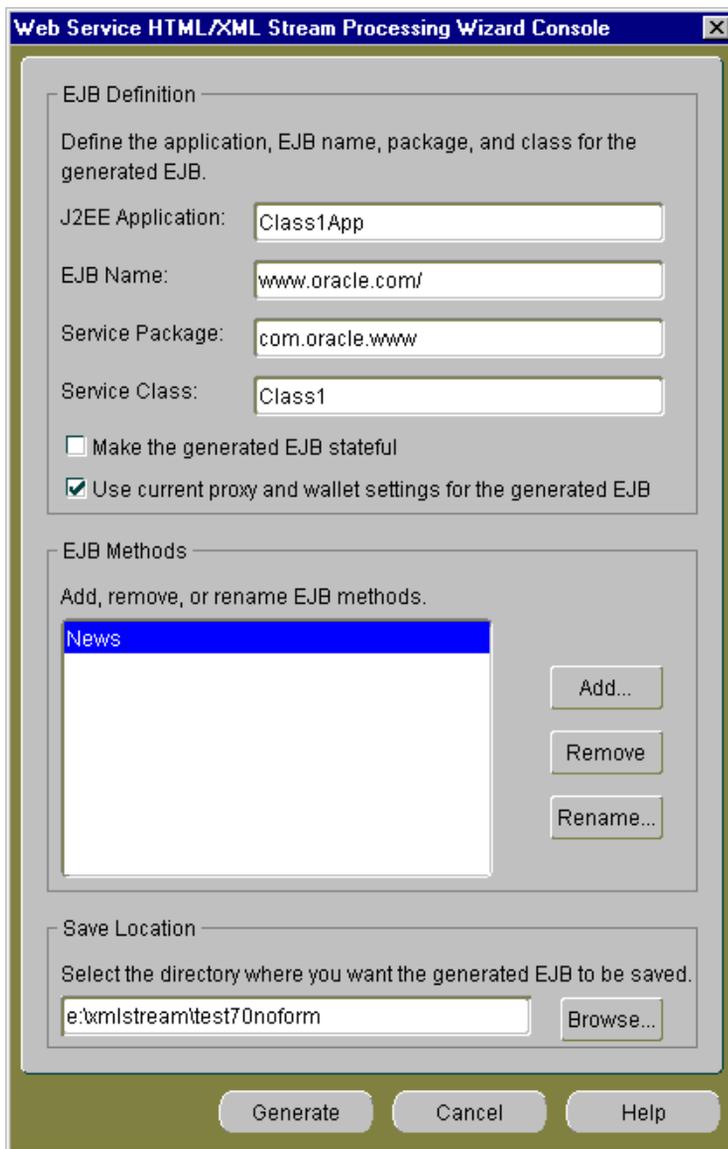
You cannot change the values for any of these EJB definition fields in this step; however, in the final step (**Console** window), you can change these names.



Enter an EJB method name, then click **Finish** to continue to the next step.

Note: Once you click **Finish** on the **Summary** window, you cannot return to a previous step. You really are finished with the process of creating an EJB method that will access and process the specified XML or HTML stream.

7. In the final step, the **Console** window, you see the main window of the Web Service HTML/XML Stream Processing Wizard that always remains in view once you reach this step of creating an EJB method.



The **Console** window is divided into three sections: EJB Definition, EJB Methods, and Save Location.

EJB Definition Section

The **EJB Definition** section contains the default EJB definition for your current EJB consisting of the J2EE application name, the EJB name, the service package name, and the service class name. You can change any of these definition names by placing the cursor in the field and editing the name.

You can make the generated EJB stateful by selecting the **Make the generated EJB stateful** option. By default this option is not selected.

You can choose to use the current proxy and wallet settings for the generated EJB by selecting the **Use current proxy and wallet settings for the generated EJB** option. By default this option is already selected.

EJB Methods Section

The **EJB Methods** section lets you add, remove, or rename EJB methods.

If you click **Add**, you return to Step 1 of this wizard, the **Step 1 of 5: HTML/XML Stream Type** window where you can begin again the process of adding another EJB method definition that accesses an HTML or XML stream through the HTTP/S protocol.

If you select an EJB method and click **Remove**, the highlighted EJB method is removed. Note that there is a confirmation window that pops up as part of this operation.

If you select an EJB method name and click **Rename**, a **Rename** pop-up window lets you rename the EJB method. You can click **OK** to complete the rename operation and return to the **Console** window, or you can click **Cancel** to cancel this rename operation and return to the **Console** window.

Save Location Section

The **Save Location** section lets you specify where you want the generated EJB method to be saved. You can either enter a drive and directory name or browse to the desired location by clicking **Browse**.

If you want, edit the EJB definition names in the **EJB Definition** section, then enter the directory name where you want to save your generated EJB. You can optionally browse to this directory location and select it, or browse to the desired directory and create a new directory name.

Select the **Make the generated EJB stateful** option if you are creating a multi-operational service. When you create a multi-operational service, which needs to maintain a conversational state with the remote HTTP server across

method calls, you must access other site content and perform the defined processing. In addition, keep the HTTP/S session information in its state so other method calls can share the same session information. The generated Java stub will then be modeled as a stateful session EJB.

An example of a multioperational service would be one operation that includes the login methods for HTTP or HTTPS authentication. A second operation would include the methods that scrape the Web site to which you were granted access through login authentication. In this case, method calls for both operations share the same session information.

For this sample use, leave the **Make the generated EJB stateful** box without a checkmark because this is a single operational service.

Click **Generate** to save your generated EJB.

At this point, you can quit from the wizard by clicking **Cancel** and at the **Warning** confirmation pop-up window, click **OK**.

You can add another EJB method by clicking **Add** the **EJB Methods** section, which starts you again at Step 1 of the wizard, the **HTML/XML Stream Type** window.

The Web Service HTML/XML Stream Processing Wizard generates the following sets of files located within the destination directory name you specified in the **Console** window. The wizard will save the generated files using the following directory layout:

```
Root /
+ app.ear
+ src/
  + ... generated java sources ...
+ classes/
  + META-INF/
    + ejb-jar.xml
  + ... compiled classes and xml resources ....
+ deploy/
  + ejb.jar
  + META-INF/
    + application.xml
```

- An .ear file (which is a JAR containing the J2EE application that can be deployed in Oracle9iAS) is located within the parent directory you specified in Step 7. The .ear file contains the generated EJB, JAR, and XML files for your application, where the application.xml file located in the /deploy/META-INF directory for UNIX or the \deploy\META-INF directory for Windows serves as the EAR manifest file.

- A JAR file, containing your EJB application class files is located within the `/deploy` directory for UNIX or the `\deploy` directory for Windows. The JAR file includes all EJB application class files and the deployment descriptor file.
- A standard J2EE EJB deployment descriptor (`ejb-jar.xml`), for all the beans in the module, is located within the `/classes/META-INF` directory for UNIX or the `\classes\META-INF` directory for Windows. The XML deployment descriptor describes the application components and provides additional information to enable the container to manage the application.
- The source code of a set of Java classes that you can use in your Java applications is located within the `/src` directory for UNIX or the `\src` directory for Windows. The generated JavaBean and EJB Java source code is contained in subdirectories according to their Java package names.
- The `/classes` directory for UNIX or the `\classes` directory for Windows contains the compiled generated classes and additional XML resources used by the generated code.

The following code is generated in the `src/com/oracle/www/Class1.java` file on UNIX or the `src\com\oracle\www\Class1.java` file on Windows showing the remote interface (`Class1`) of the generated EJB. In this case, a method (`news`) with no parameters that return an `org.wc3.dom.Element` is generated because the HTML stream was selected as a static HTML page.

```
public interface Class1 extends EJBObject
{
    public org.w3c.dom.Element news()
        throws RemoteException;
}
```

Handling an XML or HTML Stream Accessed Through a Form

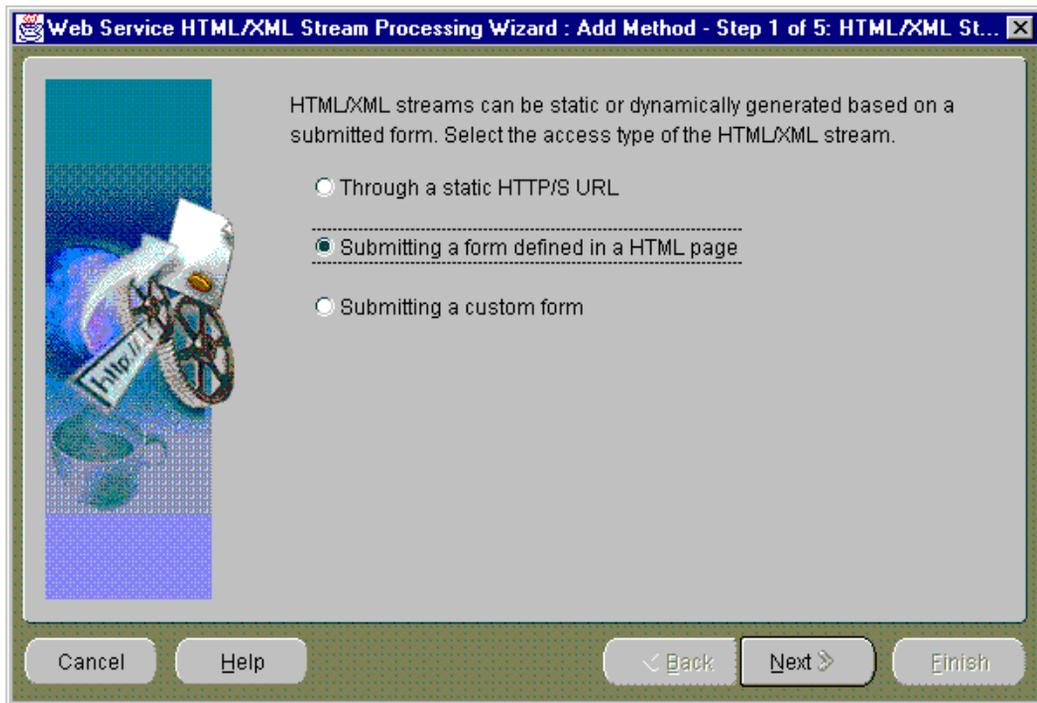
The following steps generate the Java stubs that consume a dynamic XML or HTML stream requiring a form to be submitted.

1. Invoke the Web Service HTML/XML Stream Processing Wizard using the following command:

```
java -jar WebServicesHtmlXmlWizard.jar
```

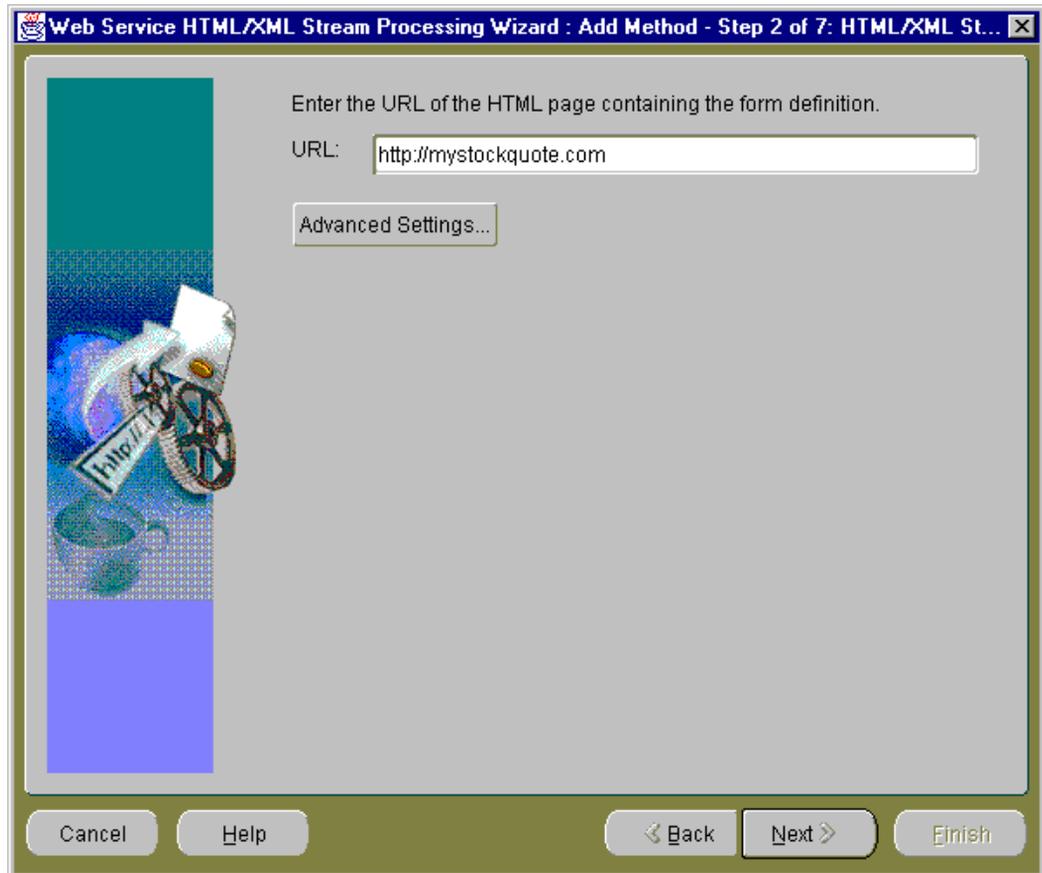
Note: The WebServicesHtmlXmlWizard.jar file is located in your `ORACLE_HOME/webservices/lib` Oracle9iAS installation directory for UNIX or `%ORACLE_HOME%\webervices\lib` Oracle9iAS installation directory for Windows.

2. In **Step 1 of 5: HTML/XML Stream Type**, select the second option, **Submitting a form defined in an HTML page**, then click **Next** to continue to the next step.



Note that you can optionally select the **Submitting a custom form** option if you must customize the form to allow for variables such as where the Web server offers a certain action, but the corresponding form is not provided in the HTML page.

3. In **Step 2 of 7: HTML/XML Stream URL**, enter the URL of the HTML page from which you want to access the resource.



If you must access the URL from outside a firewall, click **Advanced Settings**. For this example, assume you must go through a firewall to access the desired URL.

4. At the **Advanced Settings** pop-up window, select **Use proxy server** and place a checkmark in the box, then enter the host address and port number for you proxy server. Click **OK** to return to the **HTML/XML Stream URL** window. Click **Next** to continue to the next step.

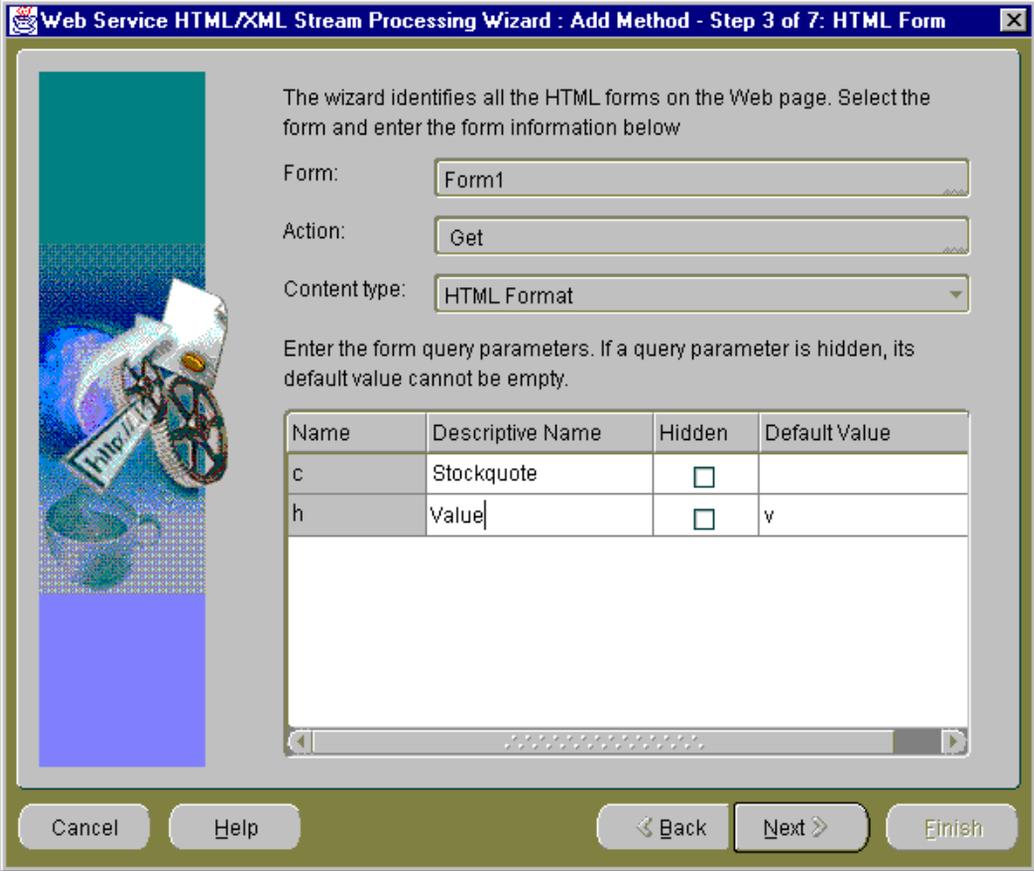


Note: If the URL you are accessing requires basic HTTP authentication, select **Use credential information in request**, then enter the user name and password in the **Credential** section of the **Advanced Settings** pop-up window.

If the URL you are accessing requires basic HTTPS authentication, use the **Oracle Wallet** section of the **Advanced Settings** pop-up window to enter the Wallet location.

5. In **Step 3 of 7: HTML Form**, the Web Service HTML/XML Stream Processing Wizard identifies all HTML forms on the Web page. For this sample use, the

Form field shows just one form, the default form name, Form1 and the **Action** field shows the HTML form action. In the **Content Type** field, the default is HTML Format. This specifies the content type of the page returned by the remote server upon the submission of the form. If the content type is XML, then select XML Format. Accept the default content type as HTML format.



The wizard identifies all the HTML forms on the Web page. Select the form and enter the form information below

Form:

Action:

Content type:

Enter the form query parameters. If a query parameter is hidden, its default value cannot be empty.

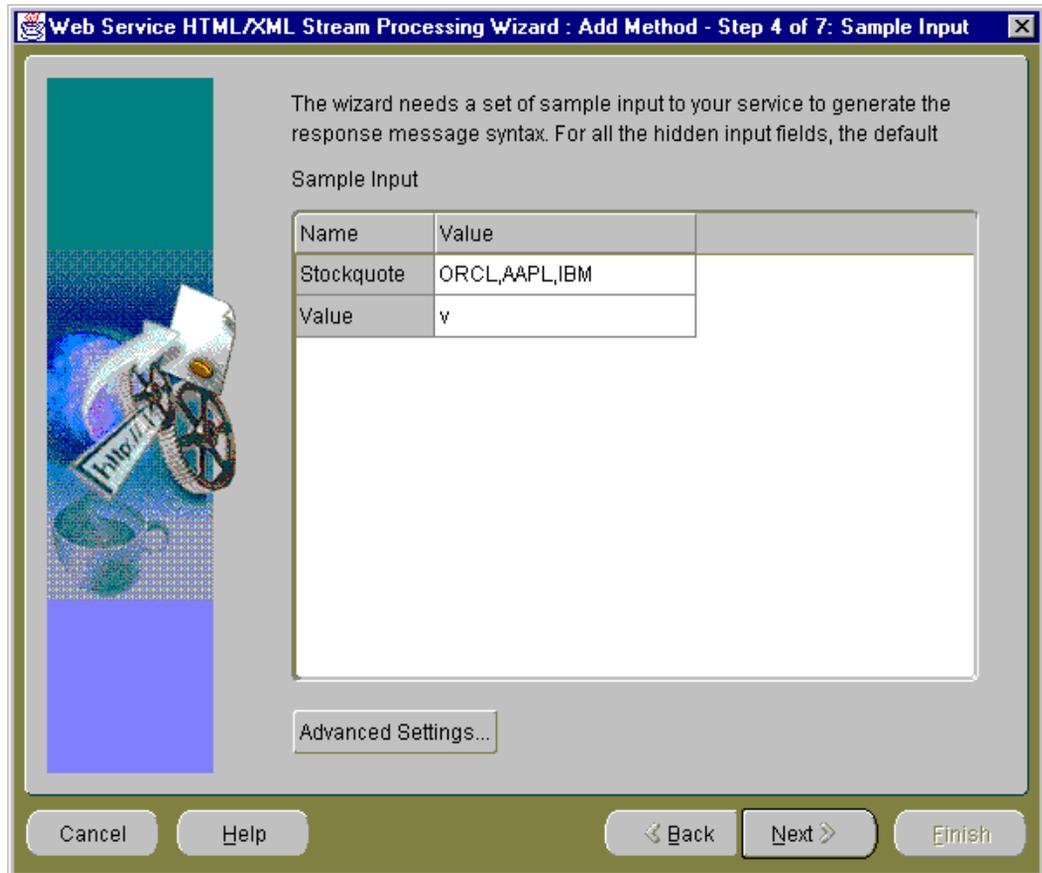
Name	Descriptive Name	Hidden	Default Value
c	Stockquote	<input type="checkbox"/>	
h	Value	<input type="checkbox"/>	v

Buttons: Cancel, Help, < Back, Next >, Finish

Note: If you are submitting a custom form, there is no need to specify an action.

In the form query parameters section, checkmark the names of the query parameters and add descriptive names as needed in the **Descriptive Names** column for each query parameter. Descriptive names are used as the name of the parameter in the signature of the method being defined. For query parameters that should remain hidden, click the appropriate row and column to change the default value from unchecked to checked. Note that for each hidden query parameter, you must also enter a default value. Hidden parameters are not exposed as Java parameters in the signature of the method being defined. When you have made all the necessary changes, click **Next** to continue to the next step.

6. In **Step 4 of 7: Sample Input**, you must enter sample input to your service in order to generate the response message syntax. The default values for all the hidden query form parameters specified in the previous step, **Step 3 of 7 HTML Form**, are used as sample input. Add or edit the sample input values for all required query form parameters in the **Value** fields for each parameter.

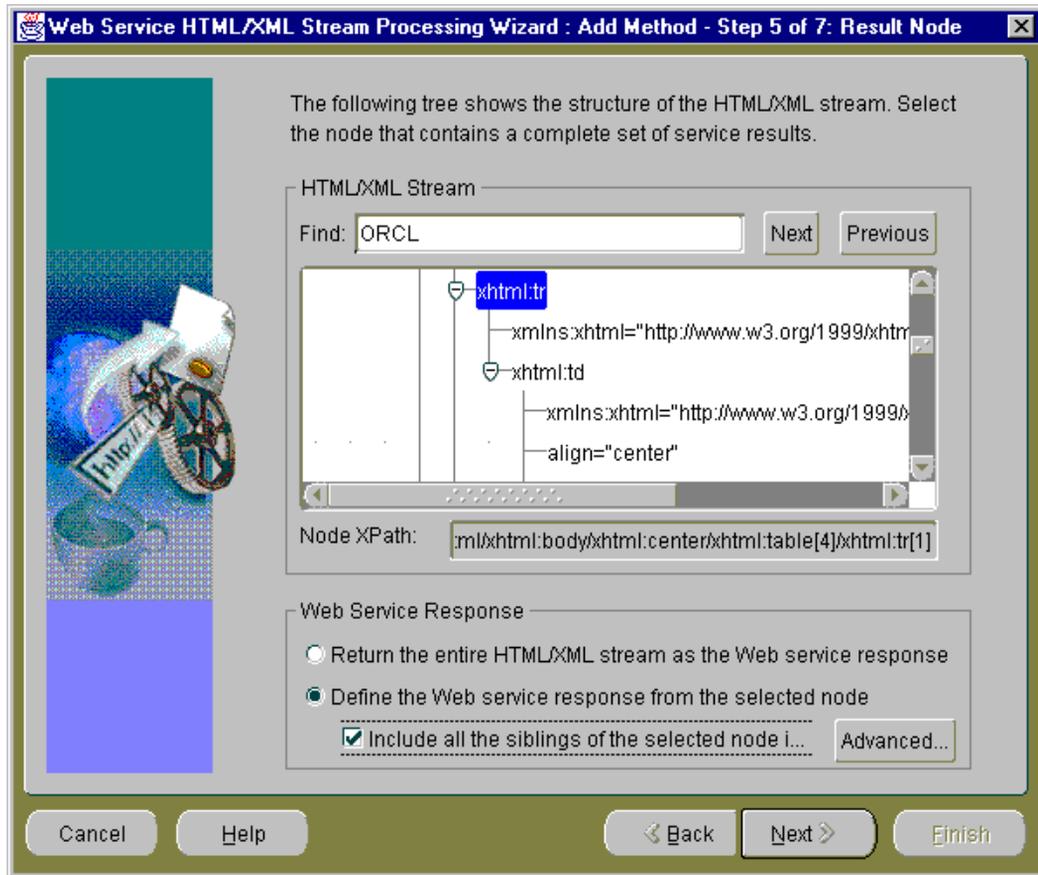


If you want to check your Web proxy information, enter basic HTTP authentication information, or enter basic HTTPS authentication information, click **Advanced Settings** and enter or edit the desired information.

Click **Next** to continue to the next step.

7. In **Step 5 of 7: Result Node**, the HTML/XML stream tree is shown in the **HTML/XML Stream** section.

Note: You may need to resize the window vertically so you can see the **HTML/XML Stream Tree** pane.



The **Result Node** window shows the structure of the HTML or XML stream as an HTML/XML stream tree and lets you define your Web Service response based on the contents of the HTML/XML stream.

You have two options in defining your Web Service response:

- To select the entire HTML/XML stream to be part of your Web Service response.
- To select just the node that contains the complete set of service results in the HTML/XML stream and define this to be the Web Service response. Optionally, you can also include in the Web Service response all siblings of the selected node.

The **Web Service Response** section lets you define the Web Service response as either the entire HTML/XML stream or as the parent node you selected in the **HTML/XML Stream** section. If the parent node contains siblings, you can optionally select them all to be included in the Web Service response. If you choose to include all the siblings, you can click **Advanced Settings** to display the **Advanced Settings** pop-up window where you can enter a predicate that filters the set of sibling nodes, view the resulting Xpath, and view or edit the Response element name.

If you want to select the entire HTML/XML stream to be part of your Web Service response, select the first option **Return the entire HTML/XML Stream as the Web service response**, then click **Next** at the bottom of the window to continue to the next step.

If you want to select just the node that contains the complete set of information you are interested in, select the second option **Define the Web service response from the selected node**. Then, navigate to the node you want by moving down the HTML/XML stream tree.

You can quickly locate the desired element in the HTML/XML stream tree by entering its name in the **Find** field and clicking **Next** at the end of this field. The name of the element is highlighted in the HTML/XML stream tree. You can go to the next or previous occurrence of this element by clicking **Next** or **Previous** the end of the **Find** field.

From the highlighted element, navigate toward the root of the tree to the node that contains the complete set of information in which you are interested. The node of interest is usually the next lowest table row node (`xhtml:tr`) that is within a different table; it is usually located one level lower toward the root of the tree.

[Figure 11-1](#) and [Figure 11-2](#) together show an excerpt of what the `xhtml` tree would appear like when expanded. The selected node `xhtml:tr` is located in the next lower table node, which is one level lower than the `xhtml:tr` nodes for ORCL and its two siblings AAPL and IBM.

Figure 11-1 Expanded xhtml Tree Showing the Selected Node of Interest Relative to the Nodes for ORCL and Sibling Nodes AAPL and IBM (Part1)

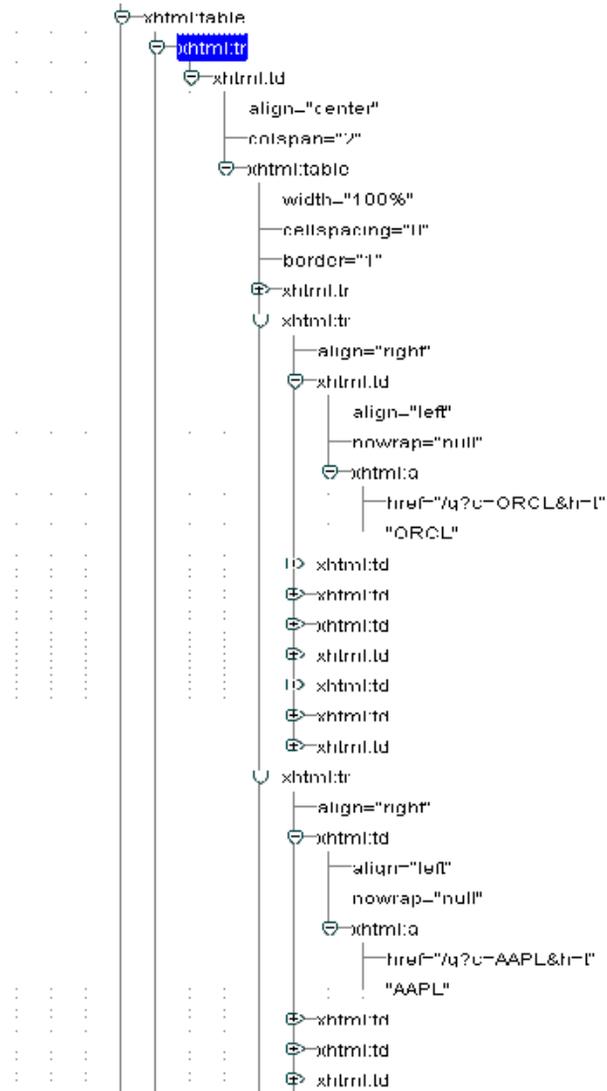
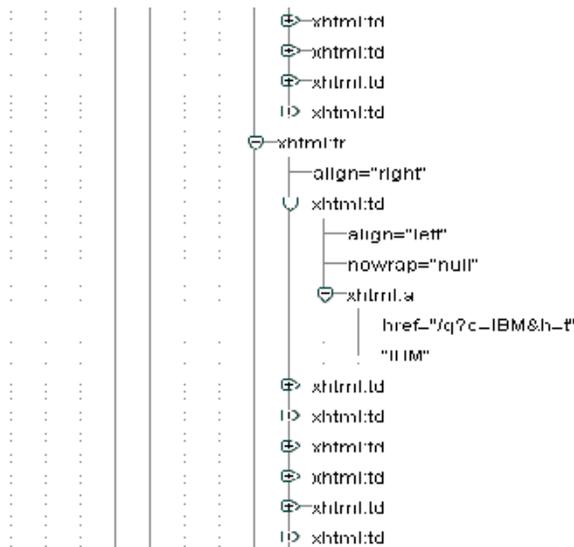


Figure 11–2 Expanded xhtml Tree Showing the Selected Node of Interest Relative to the Nodes for ORCL and Sibling Nodes AAPL and IBM (Part 2)



Note that the **Node Location** field contains the complete name of the node you selected.

When you select the option **Define the Web service response from the selected node**, another option is now available and that is whether or not to include all the siblings of the selected node in the response.

If the node you selected has siblings that you want to include in the Web Service response, select the option **Include all the siblings of the selected node in the response**. When you make this selection, an **Advanced Settings** button enables. Click **Advanced Settings** to display the **Advanced Settings** pop-up window where you can enter a predicate that filters the set of sibling nodes, view the resulting Xpath, and view or edit the Response element name.

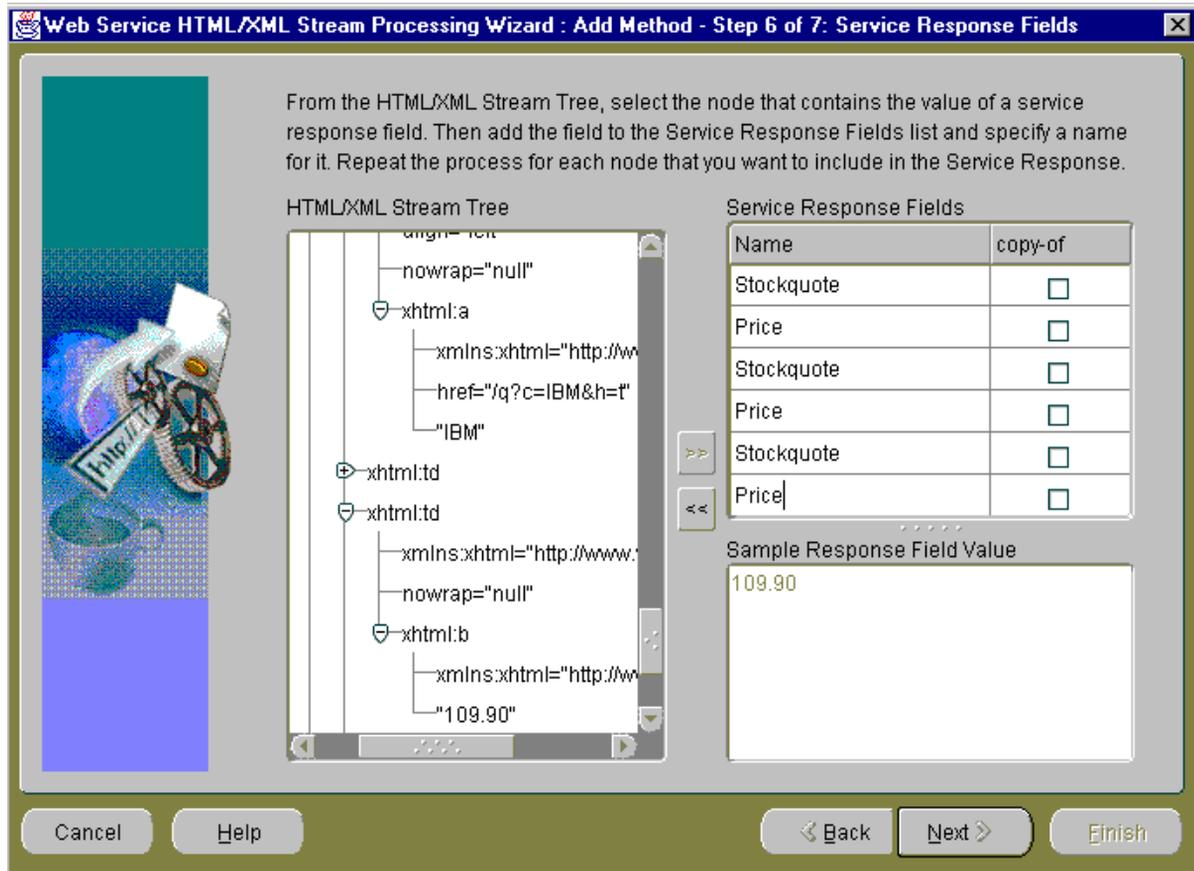
The following predicate filters out the first position: `position() != 1`. Enter this predicate expression in the **Predicate that filters the set of sibling nodes** field of the **Advanced Settings** pop-up window to filter the first sibling from the Web Service response.

For more information about predicates, filters, syntax, and composing a predicate expression, see the Xpath section of the following Web site:
<http://www.w3c.org/TR/xpath>.

Then, click **OK** to return to the **Result Node** window.

Click **Next** to continue to the next step.

8. In **Step 6 of 7: Service Response Fields**, you are finalizing the output extracting process. Based on the selected element from Step 5 of 7 Result Node, you can select elements of interest to be outputs and assign names to the output fields.



Service Response Field Names are mapped to XML Element names of the service response. By default, the value of each node selected in the HTML/XML stream is contained in an XML Element name as specified in the **Service Response Fields** table. For example, if the `<a>test` node from the HTML/XML stream tree is added to the **Service Response Fields** pane, the service response then contains an XML Element such as `<respA>test</respA>`, where `respA` is the corresponding service response field name. The value of the node is extracted using the XSLT *value-of* operation.

If the copy-of column is selected for a result field, the corresponding XML/HTML stream node is copied in the service response. For example, if the `<a>test` node from the HTML/XML stream tree is added to the **Service Response Fields** pane and the copy-of option is selected, the service response then contains an XML Element, such as `<respA><a>test</respA>`, where `respA` is the corresponding service response field name. The copy of a node is built using the XSLT *copy-of* operation as shown in the following code example taken from a generated XSL stylesheet. In this example, `<resp:Stockquote>` and `<resp:Price>` are the corresponding service response field names showing the copy of a node that was built using the XSLT *copy-of* operation where the Copy-of column option was selected.

```
- <resp:Stockquote>
  <xsl:copy-of select="./xhtml:td/xhtml:table/xhtml:tr[2]/xhtml:td[1]/xhtml:a/text()" />
</resp:Stockquote>
- <resp:Price>
  <xsl:copy-of select="./xhtml:td/xhtml:table/xhtml:tr[2]/xhtml:td[3]/xhtml:b/text()" />
</resp:Price>
- <resp:Stockquote>
  <xsl:copy-of select="./xhtml:td/xhtml:table/xhtml:tr[3]/xhtml:td[1]/xhtml:a/text()" />
</resp:Stockquote>
- <resp:Price>
  <xsl:copy-of select="./xhtml:td/xhtml:table/xhtml:tr[3]/xhtml:td[3]/xhtml:b/text()" />
</resp:Price>
- <resp:Stockquote>
  <xsl:copy-of select="./xhtml:td/xhtml:table/xhtml:tr[4]/xhtml:td[1]/xhtml:a/text()" />
</resp:Stockquote>
- <resp:Price>
  <xsl:copy-of select="./xhtml:td/xhtml:table/xhtml:tr[4]/xhtml:td[3]/xhtml:b/text()" />
</resp:Price>
```

In the **HTML/XML Response Tree** pane, navigate to the node that contains the value of the service response field of interest and select the value to highlight it. Then, click the double, right-arrow to the right of this **HTML/XML Response Tree** pane to move the value of the response field to the lower right **Sample Response Field Value** pane. This action also adds a row to the **Service**

Response Fields list in the upper right **Service Response Fields** pane. Select the empty field in the **Name** column of the **Service Response Fields** pane and enter a descriptive name for this field. Repeat this process for each element that you want to include in the service response. As you follow this process, you will be building a list of response fields of interest in the **Service Response Fields** list.

If you want to remove a service response field from the **Service Response Fields** list, select the value of the name in the **Service Response Fields** pane and click the double, left-arrow to the left side of this pane. This action removes this service response field from the **Service Response Fields** list.

When you have made all your selections, click **Next** to continue to the next step.

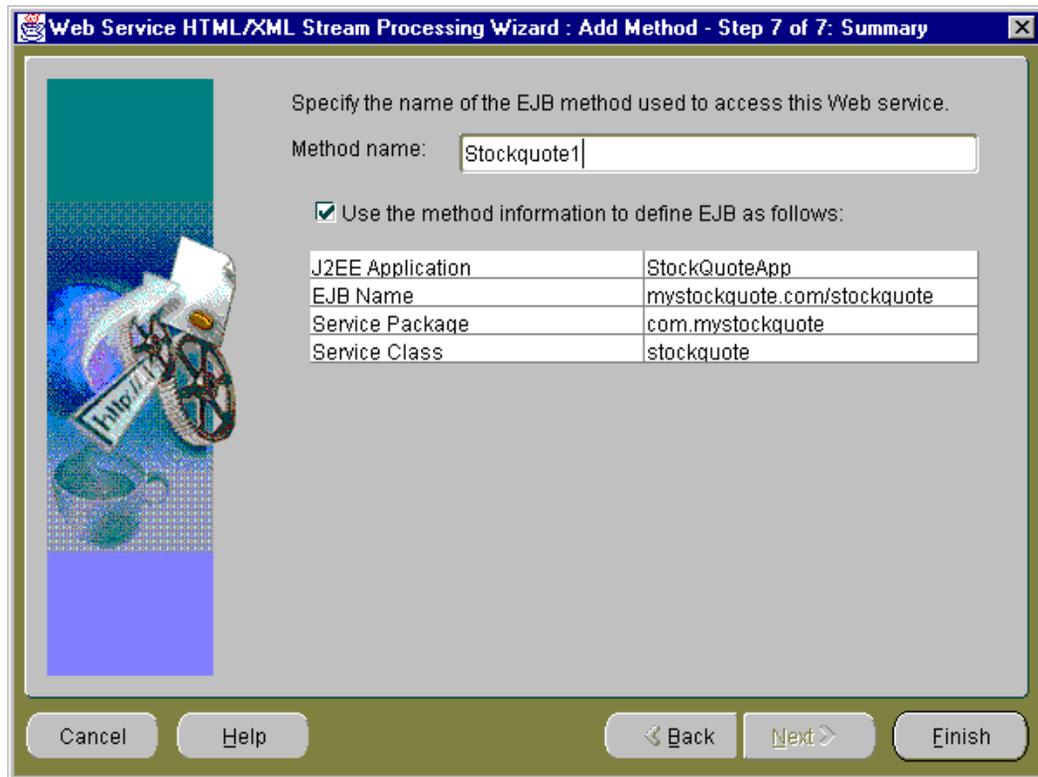
9. In **Step 7 of 7: Summary**, you must specify your EJB method name.

If this is the first HTML or XML stream you are processing in this session, then you will see only the EJB method name.

If this is the second or subsequent HTML or XML stream you are processing in this session, then the suggested EJB method information is displayed for your EJB method, describing the name for the J2EE Application, the EJB Name, the name of the service package, and the name of the service class. By default, the names are preselected based on the known information.

If you want to retain this suggested EJB method information and display it in the next step, the **Console** window, then leave the option **Use the method information to define EJB as follows** selected (with a check mark). If not, deselect this option and the EJB method information that appeared previously will be displayed in the **Console** window.

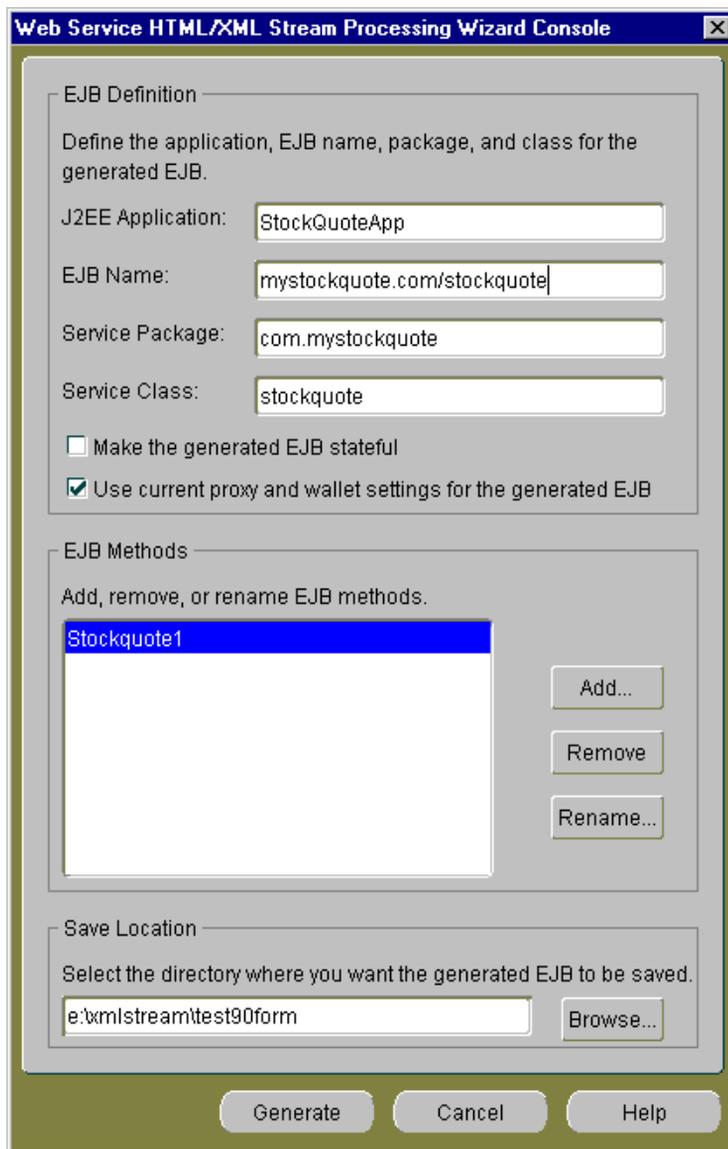
You cannot change the values for any of these EJB definition fields in this step; however, in the final step (**Console** window), you can change these values.



Enter an EJB method name, then click **Finish** to continue to the next step.

Note: Once you click **Finish** on the **Summary** window, you cannot return to a previous step. You really are finished with the process of creating an EJB method whose methods will access and process the specified XML or HTML stream.

10. In the final step, the **Console** window, you see the main window of the Web Service HTML/XML Stream Processing Wizard that remains in view once you reach this step of creating an EJB.



The **Console** window is divided into three sections: EJB Definition, EJB Methods, and Save Location.

EJB Definition Section

The **EJB Definition** section contains the EJB definition for your current EJB consisting of the J2EE application name, the EJB name, the service package name, and the service class name. You can change any of these definition names by placing the cursor in the field and editing the name.

You can make the generated EJB stateful by selecting the **Make the generated EJB stateful** option. By default this option is not selected.

You can choose to use the current proxy and wallet settings for the generated EJB by selecting the **Use current proxy and wallet settings for the generated EJB** option. By default this option is already selected.

EJB Methods Section

The **EJB Methods** section lets you add, remove, or rename EJB methods.

If you click **Add**, you return to Step 1 of this wizard, the **Step 1 of 7: HTML/XML Stream Type** window where you can begin again the process of adding another EJB method definition that accesses an HTML or XML stream through the HTTP/S protocol.

If you select an EJB method and click **Remove**, the highlighted EJB method is removed. Note that there is a confirmation window that pops up as part of this operation.

If you select an EJB method name and click **Rename**, a **Rename** pop-up window lets you rename the EJB method. You can click **OK** to complete the rename operation and return to the **Console** window, or you can click **Cancel** to cancel this rename operation and return to the **Console** window.

Save Location Section

The **Save Location** section lets you specify where you want the generated EJB method to be saved. You can either enter a drive and directory name or browse to the location by clicking **Browse**.

If you want, edit the EJB definition names in the **EJB Definition** section, then enter the directory name where you want to save your generated EJB. You can optionally browse to this desired directory location and select it, or browse to the desired directory and create a new directory name.

Select the **Make the generated EJB stateful** option if you are creating a multi-operational service. When you create a multi-operational service, which needs to maintain a conversational state with the client across method calls, you

must access other site content and perform the defined processing. In addition, keep the HTTP/S session information in its state so other method calls can share the same session information. The generated Java stub will then be modeled as a stateful session EJB.

For this sample use, leave the **Make the generated EJB stateful** box without a check mark because this is a single operational service.

Click **Generate** to save your generated EJB.

At this point, you can quit from the wizard by clicking **Cancel** and at the **Warning** confirmation pop-up window, click **OK**.

You can add another EJB method by clicking **Add** in the **EJB Methods** section, which starts you again at Step 1 of the wizard, Step 1 of 7: HTML/XML Stream Type.

The Web Service HTML/XML Stream Processing Wizard generates the following sets of files located within the parent directory name you specified in the last step, the **Console** window. The wizard will save the generated files using the following directory layout:

```
Root /
+ app.ear
+ src/
  + ... generated java sources ...
+ classes/
  + META-INF/
    + ejb-jar.xml
  + ... compiled classes and xml resources ....
+ deploy/
  + ejb.jar
  + META-INF/
    + application.xml
```

- An .ear file (which is a JAR containing the J2EE application that can be deployed in Oracle9iAS) is located within the parent directory you specified in the last step, the **Console** window. The .ear file contains the generated EJB, JAR, and XML files for your application, where the application.xml file located in the /deploy/META-INF directory for UNIX or the \deploy\META-INF directory for Windows serves as the EAR manifest file.
- A JAR file, containing your EJB application class files, is located within the /deploy directory for UNIX or the \deploy directory for Windows. The JAR file includes all EJB application class files and the deployment descriptor file.

- A standard J2EE EJB deployment descriptor (`ejb-jar.xml`), for all the beans in the module, is located within the `/classes/META-INF` directory for UNIX or the `\classes\META-INF` directory for Windows. The XML deployment descriptor describes the application components and provides additional information to enable the container to manage the application.
- The source code of a set of Java classes that you can use in your Java applications is located within the `/src` directory for UNIX or the `\src` directory for Windows. The generated JavaBean and EJB Java source code is contained in subdirectories according to their Java package names.
- The `/classes` directory for UNIX or the `\classes` directory for Windows contains the compiled generated classes and additional XML resources used by the generated code.

The following code is generated in the `<class-name>.java` file showing the remote interface (stockquote) of the generated EJB. In this case, a method (stockquote1) with parameters (Stockquote and h) for each non-hidden form parameter that returns an `org.wc3.dom.Element` is generated. This stockquote1 method is generated because the HTML stream was selected as being dynamically generated based on a submitted form defined in the HTML page.

```
public interface stockquote extends EJBObject
{
    public org.w3c.dom.Element stockquote1(java.lang.String Stockquote,
                                           java.lang.String Value)
        throws RemoteException;
}
```

Advanced Section -- Editing Changes You Can Make to Generated Files

The following sections describe some changes you can make by editing the content of specific generated files. These changes can adapt your XSLT stylesheet to an enhanced response definition or satisfy changing requirements for using your generated EJB with another Web proxy server.

Editing the Generated XSLT Stylesheet

The generated `<class-name>.jar` file, located in the last child `<class-name>` directory within the `/classes` directory on UNIX or `\classes` on Windows, contains three files:

- Sample output response XML file returned by the remote server
- Output response XSLT stylesheet file used for the scraping process
- XML response schema XSD file used for the returned response during the wizard session

During runtime operations, the XML response returned by the remote server upon access of the XML URL or the submission of a form, is filtered through the XSLT transformation defined in this stylesheet.

You can edit the filtering stylesheet XSLT file to add logic or to change the behavior of your application. You can make comparable edits to the output response XML XSD file to custom adapt your response file for your J2EE application. You must know how to modify stylesheets and response definition files to complete these changes successfully.

When you have completed your changes to the response stylesheet and response XML files and saved your changes, you must do the following:

- Rejar your `<class-name>.jar` file in the deploy directory.
- Rejar your EJB JAR file by jarring the content of the classes directory.
- Rejar the defined EAR file saved in the tool destination directory, by jarring the content of the deploy directory.

Modifying Environment Options in the Generated `ejb-jar.xml` File

The generated `ejb-jar.xml` file is located in the `/classes/META-INF` directory on UNIX or `\classes\META-INF` directory on Windows directly below the root directory where you saved your generated EJB. This file contains an environment section denoted by `<env-entry>` and `</env-entry>` tags where the Web proxy information is stored. Once you generate your EJB, you can later edit this `ejb-jar.xml` file to modify your Web proxy settings (host address name and port number) to satisfy any requirements you might have for using your generated EJB with other Web proxy servers. You must jar your `ejb.jar` and `ear` file again and redeploy them in your J2EE application server.

Consuming SOAP-Based Web Services Using WSDL

The `wsdl2ejb` utility can be used by J2EE developers to consume a Web Service described in Web Services Description Language (WSDL) document into their applications. This utility takes a WSDL document and some additional optional parameters and produces an EJB EAR file that can be deployed into Oracle9iAS OC4J. The EJB Remote Interface is generated based on the WSDL portType. Each WSDL operation is mapped to an EJB method. The EJB method parameters are derived from the WSDL operation input message parts, while the EJB method return value is mapped from the parts of the WSDL operation output message. The Oracle Simple Open Access Protocol (SOAP) Mapping Registry is used to map XML types to the corresponding Java types.

Additional references regarding WSDL and SOAP can be found in the following locations:

- The WSDL 1.1 specification is available at
<http://www.w3.org/TR/wsdl>
- The SOAP 1.1 specification is available at
<http://www.w3.org/TR/SOAP/>

The command-line options for running the `wsdl2ejb` utility are described in [Table 11-1](#).

Table 11-1 *wsdl2ejb Utility Command-Line Options*

Option	Description
<code>-conf <config file></code>	Allows the <code>wsdl2ejb</code> utility to load a configuration file.
<code>-d <destDir></code>	Allows a destination directory to be specified where the generated EJB EAR file is to be written.
<code>-Dhttp.proxyHost</code>	Allows the proxy host name to be specified when an HTTP URL is used to supply the location of the WSDL document and an HTTP proxy server is required to access it.
<code>-Dhttp.proxyPort</code>	Allows the proxy port number to be specified when an HTTP URL is used to supply the location of the WSDL document and an HTTP proxy server is required to access it.
<code>-jar</code>	Allows you to specify the <code>wsdl2ejb</code> utility as a JAR file.

To run the `wsdl2ejb` utility, enter the following command where `<destDir>` is the destination directory to where the generated EJB EAR file is to be written and the file `mydoc.wsdl` is the location of the WSDL document:

```
java -jar wsdl2ejb.jar -d <destDir> mydoc.wsdl
```

Note: The `wsdl2ejb.jar` file is located in your `$ORACLE_HOME/webservices/lib` Oracle9iAS installation directory for UNIX or `%ORACLE_HOME%\webservices\lib` Oracle9iAS installation directory for Windows.

If an HTTP URL is used to supply the location of the WSDL document and an HTTP proxy is required to access it, the following command and syntax must be used to run the utility:

```
java -Dhttp.ProxyHost=myProxyHost -Dhttp.proxyPort=80 -jar wsdl2ejb.jar -d  
<destDir> http://myhost/mydoc.wsdl
```

In this example, the utility uses the supplied WSDL to generate the EJB EAR file in the destination directory (`<destDir>`). The EJB class name, Java Naming and Directory Interface (JNDI) binding key, and Java package name are derived from the location of the SOAP service described in the WSDL.

In this command syntax, the `wsdl2ejb` utility maps the XML types, which are supported by default by the Oracle SOAP Mapping Registry.

The `wsdl2ejb` utility generates the following sets of files located within the destination directory name (`<destDir>`) that you specify in the command line. The utility saves the generated files using the following directory layout:

```
Root /  
+ app.ear  
+ src/  
  + ... generated java sources ...  
+ classes/  
  + META-INF/  
    + ejb-jar.xml  
  + ... compiled classes and xml resources ....  
+ deploy/  
  + ejb.jar  
  + META-INF/  
    + application.xml
```

- An .ear file (which is a JAR archive containing the J2EE application that can be deployed in Oracle9iAS) is located within the destination directory (<destDir>) you specified in the command line. The .ear file contains the generated EJB, JAR, and XML files for your application, where the application.xml file located in the /deploy/META-INF directory for UNIX or the \deploy\META-INF directory for Windows serves as the EAR manifest file.
- An archive JAR file containing your EJB application class files is located within the /deploy directory for UNIX or the \deploy directory for Windows. The JAR file includes all EJB application class files and the deployment descriptor file.
- A standard J2EE EJB deployment descriptor (ejb-jar.xml) for the generated bean in the module is located within the /classes/META-INF directory for UNIX or the \classes\META-INF directory for Windows. The XML deployment descriptor describes the application components and provides additional information to enable the container to manage the application.
- The source code of a set of Java classes that you can use in your Java applications is located within the /src directory for UNIX or the \src directory for Windows. The generated JavaBean and EJB Java source code is contained in subdirectories according to their Java package name. An EJB client stub is also generated.
- The /classes directory for UNIX or the \classes directory for Windows contains the compiled generated classes and additional XML resources used by the generated code.

Advanced Configuration

To have more controls on the EJB generated from a WSDL document, an XML configuration file can be supplied to the `wsdl2ejb` utility. Through the configuration file, developers can control several options on the WSDL source, as well as options on the generated EJB.

Developers can also use the configuration file to supply additional xml to Java type maps, so that WSDL documents using complex types can be supported.

The syntax of the `wsdl2ejb` configuration file is shown in its Document Type Definition (DTD) as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Specify the properties of the source WSDL document and of the target EJB. -->
<!ELEMENT wsdl2ejb (useProxy?, useWallet?, wsdl, ejb?, mapTypes?)>
```

```
<!-- Specify if the generated EJB should use the supplied HTTP proxy when accessing HTTP URLs -->
<!ELEMENT useProxy (#PCDATA)>
<!ATTLIST useProxy
    proxyHost CDATA    #REQUIRED
    proxyPort CDATA    #REQUIRED>

<!-- Specify the location of the wallet credential file used by the generated EJB for opening HTTPS
connection -->
<!ELEMENT useWallet (#PCDATA)>
<!ATTLIST useWallet
    location CDATA    #REQUIRED>

<!--
Specify how the wsdl2ejb tools should process the source WSDL document.
In addition to the mandatory location of the WSDL document, the name of the WSDL service and
its port can be specified. In this case, an EJB will be generated only for the supplied service and
port.
An alternative: the name of a WSDL service binding and the SOAP location to be used can be supplied.
In the latter case, an EJB using the specified binding and the supplied SOAP location will be used.
This is particularly useful when generating an EJB from a WSDL stored in a UDDI registry.
In fact, following a UDDI best practice, the WSDL SOAP location will be managed separately from the
WSDL document.
-->
<!ELEMENT wsdl (location, ((service-name, service-port) | (service-binding, soap-location)))?>

<!-- Specify the location of the source WSDL document (for example, "/home/mywsdl.wsdl",
"http://myhost/mywsdl.wsdl") -->
<!ELEMENT location (#PCDATA)>

<!-- Specify the name of the WSDL service to be used for the generation.
It is the name of one of the services defined in the source WSDL. -->
<!ELEMENT service-name (#PCDATA)>

<!-- Specify the service port of the WSDL service to be used for the generation.
It is the name of one ports of the service name defined above in the source WSDL. -->
<!ELEMENT service-port (#PCDATA)>

<!-- Specify the name of the WSDL binding to be used for the generation.
It is the name of one of the bindings defined in the source WSDL. -->
<!ELEMENT service-binding (#PCDATA)>

<!-- Specify the SOAP location service port of the WSDL service to be used for the generation.
It is the name of one ports of the service name defined above in the source WSDL. -->
<!ELEMENT soap-location (#PCDATA)>

<!-- Specify the properties related to the generated EJB. -->
```


Table 11–2 describes the elements, subelements, and attributes of the `wsdl12ejb` XML configuration file as defined in the DTD. Required elements and attributes are shown as **bold** text.

Table 11–2 Elements, Subelements, and Attributes of the `wsdl2ejb` XML Configuration File as Defined in the DTD

Element	Subelement	Attribute	Description
<code>useProxy</code>			Optional element. Specifies the proxy server attributes.
		proxyHost	Required attribute. Specifies the host name of the proxy server.
		proxyPort	Required attribute. Specifies the port number of the proxy server.
<code>useWallet</code>			Optional element. Specifies the Oracle Wallet attribute.
		location	Required attribute. Specifies the location of the Oracle Wallet credential file used by the EJB for opening the HTTPS connection.
<code>wsdl</code>			Required element. Specifies how the <code>wsdl12ejb</code> utility should process the source WSDL document. Requires the <code>location</code> element be specified and optionally, either the <code>service-name</code> and <code>service-port</code> pair of elements or the <code>service-binding</code> and <code>soap-location</code> pair of elements be specified.
	location		Required element. Specifies the location of the source WSDL document. Can be a file path or an URL.
	<code>service-name</code>		Optional element. Specifies the name of the WSDL service to be used for the generated EJB. If specified, must be specified with the <code>service-port</code> element as a pair of elements.
	<code>service-port</code>		Optional element. Specifies the service port of the WSDL service to be used for the generated EJB. If specified, must be specified with the <code>service-name</code> element as a pair of elements.
	<code>service-binding</code>		Optional element. Specifies the name of the WSDL binding to be used for the generated EJB. If specified, must be specified with the <code>soap-location</code> element as a pair of elements.
	<code>soap-location</code>		Optional element. Specifies the SOAP location service port of the WSDL service to be used for the generated EJB. If specified, must be specified with the <code>service-binding</code> element as a pair of elements.

Table 11–2 (Cont.) Elements, Subelements, and Attributes of the wsdl2ejb XML Configuration File as Defined in the DTD

Element	Subelement	Attribute	Description
ejb			Optional element. Specifies the properties related to the generated EJB.
	application-name		Optional element. Specifies the name of the J2EE application for the generated EAR file.
	ejb-name		Optional element. Specifies the JNDI binding key name for the generated EJB.
	package-name		Optional element. Specifies the name for the Java package under which the generated EJB belongs.
	remote-name		Optional element. Specifies the class name for the EJB Remote Interface.
	session-type		Optional element. Specifies whether the generated EJB should be stateless or stateful.
mapTypes			Optional element. Specifies the custom Java types and maps them to XML types.
	map		Optional element. Specifies the XML to JAR type map.
		encodingStyle	Required attribute. Specifies the name of the encoding style under which this map belongs.
		namespace-uri	Required attribute. Specifies the URI of the namespace for the XML type defined in this map.
		local-name	Required attribute. Specified the local name of the XML type defined in this map.
		java-type	Required attribute. Specifies the Java class name to which this type is mapped.
		java2xml-class-name	Required attribute. Specifies the Java class name of the type serializer.
		xml2java-class-name	Required attribute. Specifies the Java class name of the type deserializer.

Developers can run the `wsdl2ejb` utility with a configuration file using the following command:

```
java -jar wsdl2ejb.jar -conf wsdlconf.xml
```

Supported WSDL Documents

The `wsdl12e jb` utility supports most WSDL documents using SOAP binding. This support includes both Remote Procedure Call (RPC) and document style documents as well as types that are encoded or literal. [Table 11-3](#) shows how the supported XML Schema types are mapped to the corresponding Java type by default. Any other required type will have to be supported through the custom type mapping described previously.

Table 11-3 Supported XML Schema Types and Corresponding Java Type

Supported XML Schema Type	Corresponding Java Type
string	java.lang.String
int	int
decimal	BigDecimal
float	float
double	double
Boolean	Boolean
long	long
short	short
byte	byte
date	GregorianCalendar
timeInstant	java.util.Date

Note: Arrays of supported types, shown in [Table 11-3](#) are also supported.

Known Limitations of the `wsdl12e jb` Utility

The following information describes the known limitations of the `wsdl12e jb` utility:

- Supports only types defined by the W3C recommendation XML schema version whose namespace is: `http://www.w3.org/2001/XMLSchema`
- Supports only the One-way and Request-Response transmission primitives defined in the WSDL 1.1 specification.

- Does not support WSDL documents that use the `<import>` tag to include other WSDL documents.
- Does not support HTTP, MIME, or any other custom bindings.

Running the Demonstration

The `wsdl2ejb` demo directory contains examples on how to use the `wsdl2ejb` utility. All the commands are assumed to be executed from the `$ORACLE_HOME/webservices/demo/basic/wsdl2ejb` directory. The demonstration (demo) will use some sample WSDL documents as sources and generate EJB that can be used to invoke the Web Service operations.

The demos can be run using Jakarta ant. Review the `build.xml` file to make sure that the initial properties (`RMI_HOST`, `RMI_PORT`, `RMI_ADMIN`, `RMI_PWD`) are set correctly according to your configuration. The `build.xml` file will execute the `wsdl2ejb` utility on the demo WSDL documents, deploy the generated EJB, and execute the EJB clients.

Note: If you are executing the demos behind a firewall and need to set proxy information to access external HTTP sites, make sure this proxy information is specified in the `wsdl2ejb` configuration files (`rpc_doc_conf.xml`, `base_conf.xml`).

Note: The demos are based on WSDL/SOAP interoperability test suites. They access live SOAP services available on the Internet as SOAP interoperability test cases. The successful execution of these demos depends on the availability of these services.

The directory structure of the demos is as follows:

```
demo/web_services/wsdl2ejb:
- README.txt           : Readme file
- build.xml            : Jakarta ant build file to run all the demos
- rpc_doc              : directory for simple RPC and document style operations
  - rpc_doc_conf.xml   : wsdl2ejb configuration file for the rpc_doc demo
  - TestRpcDocClient.java : client for the rpc_doc demo
  - DocAndRpc.wsdl     : sample WSDL for the rpc_doc demo
  - (generated)        : directory where the EJB will be generated
- base
  - base_conf.xml      : wsdl2ejb configuration file for the base interoperability demo
```

- TestInteropBaseClient.java : client for the base interoperability demo
- InteropTest.wsdl : WSDL document for the base interoperability demo
- MySoapStructBean.java : bean utilized to map the custom type used in the example defined in the WSDL document
- MySoapStructBean.jar : packaged-compiled custom type bean
- (generated) : directory where the EJB will be generated

RPC and Document Style with Simple Types Example

This example uses a simple WSDL document that shows a couple of operations: Add and Multiply. Add is using the document-style operation using literal parts, while Multiply is RPC-style and uses encoded parts.

To generate the EJB stub, use the following command:

On UNIX

```
cd $ORACLE_HOME/webservices/demo/basic/wsdl2ejb
java -jar ../../../../lib/wsdl2ejb.jar -conf rpc_doc/rpc_doc_conf.xml
```

On Windows

```
cd %ORACLE_HOME%\webservices\demo\basic\wsdl2ejb
java -jar ..\..\..\lib\wsdl2ejb.jar -conf rpc_doc\rpc_doc_conf.xml
```

The utility generates the TestApp.ear file containing the definition of a stateless EJB, which can be used as a proxy for the Web Service. The EAR file can be deployed in Oracle9iAS OC4J as any standard EJB. Refer to *Oracle9iAS Containers for J2EE User's Guide* for information on how to deploy an EJB.

By looking at the generated EJB Remote Interface, you can see how the WSDL portType DocAndRpc.wsdl file has been mapped to Java.

WSDL PortType:

```
<types>
  <s:schema elementFormDefault="qualified" targetNamespace="http://soapinterop.org">
    <s:element name="Add">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="a" type="s:int" />
          <s:element minOccurs="1" maxOccurs="1" name="b" type="s:int" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="AddResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="AddResult" type="s:int" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</types>
```

```

        </s:sequence>
    </s:complexType>
</s:element>
</s:schema>
</types>
<message name="AddSoapIn">
    <part name="parameters" element="s0:Add" />
</message>
<message name="AddSoapOut">
    <part name="parameters" element="s0:AddResponse" />
</message>
<message name="MultiplySoapIn">
    <part name="a" type="xsd:int" />
    <part name="b" type="xsd:int" />
</message>
<message name="MultiplySoapOut">
    <part name="MultiplyResult" type="s:int" />
</message>
<portType name="TestSoap">
    <operation name="Add">
        <input message="s0:AddSoapIn" />
        <output message="s0:AddSoapOut" />
    </operation>
    <operation name="Multiply">
        <input message="s0:MultiplySoapIn" />
        <output message="s0:MultiplySoapOut" />
    </operation>
</portType>

```

From the `Test.java` file, the EJB Remote Interface is:

```

public org.w3c.dom.Element add(org.w3c.dom.Element parameters)
    throws RemoteException;

public int multiply(int a, int b)
    throws RemoteException;

```

When the WSDL operation is using RPC style and its parts are encoded, the parts XML schema type is mapped to a corresponding Java native type. In this example, `xsd:int` is mapped to Java `int`. In a document style using literal parts, each part is simply mapped to an `org.w3c.dom.Element`.

The following client code in the `TestRpcDocClient.java` file can be used to invoke the Add and Multiply Web Service operations. The code has been produced by modifying the client code stub generated by the `wsdl12ejb` utility.

```

import java.io.*;
import java.util.*;

```

```
import javax.naming.*;

import org.w3c.dom.*;
import oracle.xml.parser.v2.*;

import org.mssoapinterop.asmx.Test;
import org.mssoapinterop.asmx.TestHome;

/**
 * This is a simple client template. To compile it,
 * please include the generated EJB jar file as well as
 * EJB and JNDI libraries in classpath.
 */
public class TestRpcDocClient
{
    // replace the values
    private static String RMI_HOST = "localhost";
    private static String RMI_PORT = "23791";
    private static String RMI_ADMIN = "admin";
    private static String RMI_PWD = "welcome";

    public TestRpcDocClient () {}

    public static void main(String args[] ) {

        TestRpcDocClient client = new TestRpcDocClient();

        try {

            RMI_HOST = args[0];
            RMI_PORT = args[1];
            RMI_ADMIN = args[2];
            RMI_PWD = args[3];

            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY, "com.evermind.server.rmi.RMIInitialContextFactory");
            env.put(Context.SECURITY_PRINCIPAL, RMI_ADMIN);
            env.put(Context.SECURITY_CREDENTIALS, RMI_PWD);
            env.put(Context.PROVIDER_URL, "ormi://" + RMI_HOST + ":" + RMI_PORT + "/Wsd12EjbTestApp1");
            Context ctx = new InitialContext(env);
            TestHome home = (TestHome) ctx.lookup("mssoapinterop.org/asmx/DocAndRpc.asmx");

            Test service = home.create();

            // call any of the Remote methods that follow to access the EJB

            //
            // Add test
            //

```

```

Document doc = new XMLDocument();
Element elAdd = doc.createElementNS("http://soapinterop.org", "s:Add");
Element elA = doc.createElementNS("http://soapinterop.org", "s:a");
Element elB = doc.createElementNS("http://soapinterop.org", "s:b");
elA.appendChild(doc.createTextNode("4"));
elB.appendChild(doc.createTextNode("3"));
elAdd.appendChild(elA);
elAdd.appendChild(elB);
doc.appendChild(elAdd);

Element elAddResponse = service.add(elAdd);
Node tNode = elAddResponse.getFirstChild().getFirstChild();
System.out.println("AddResponse: "+tNode.getNodeValue());

//
// Multiply Test
//
int a = 4;
int b = 3;
int iMultiplyResponse = service.multiply(a, b);
System.out.println("MultiplyResponse: "+iMultiplyResponse);

}
catch (Throwable ex) {
    ex.printStackTrace();
}
}
}
}

```

The result of the execution of the client is the following:

```

AddResponse: 7
MultiplyResponse: 12

```

Round 2 Interop Services: Base Test Suite Example

This example starts from a subset of the WSDL document defined by the base test suite of the second round of SOAP interoperability tests. The purpose of this demo example is to show the usage of built-in types in the SOAP Mapping Registry as well as how to add custom types mapping.

Start by looking at the WSDL portType in the `InteropTest.wsdl` file.

```

<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://soapinterop.org/xsd">

```

```
<complexType name="ArrayOfstring">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="string[]"/>
    </restriction>
  </complexContent>
</complexType>
<complexType name="ArrayOfint">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="int[]"/>
    </restriction>
  </complexContent>
</complexType>
<complexType name="ArrayOffloat">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="float[]"/>
    </restriction>
  </complexContent>
</complexType>
<complexType name="ArrayOfSOAPStruct">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
wsdl:arrayType="s:SOAPStruct[]"/>
    </restriction>
  </complexContent>
</complexType>
<complexType name="SOAPStruct">
  <all>
    <element name="varString" type="string"/>
    <element name="varInt" type="int"/>
    <element name="varFloat" type="float"/>
  </all>
</complexType>
</schema>
</types>

<message name="echoStringRequest">
  <part name="inputString" type="xsd:string"/>
</message>
<message name="echoStringResponse">
  <part name="return" type="xsd:string"/>
</message>
```

```
<message name="echoStringArrayRequest">
  <part name="inputStringArray" type="s:ArrayOfstring"/>
</message>
<message name="echoStringArrayResponse">
  <part name="return" type="s:ArrayOfstring"/>
</message>
<message name="echoIntegerRequest">
  <part name="inputInteger" type="xsd:int"/>
</message>
<message name="echoIntegerResponse">
  <part name="return" type="xsd:int"/>
</message>
<message name="echoIntegerArrayRequest">
  <part name="inputIntegerArray" type="s:ArrayOfint"/>
</message>
<message name="echoIntegerArrayResponse">
  <part name="return" type="s:ArrayOfint"/>
</message>
<message name="echoFloatRequest">
  <part name="inputFloat" type="xsd:float"/>
</message>
<message name="echoFloatResponse">
  <part name="return" type="xsd:float"/>
</message>
<message name="echoFloatArrayRequest">
  <part name="inputFloatArray" type="s:ArrayOffloat"/>
</message>
<message name="echoFloatArrayResponse">
  <part name="return" type="s:ArrayOffloat"/>
</message>
<message name="echoStructRequest">
  <part name="inputStruct" type="s:SOAPStruct"/>
</message>
<message name="echoStructResponse">
  <part name="return" type="s:SOAPStruct"/>
</message>
<message name="echoStructArrayRequest">
  <part name="inputStructArray" type="s:ArrayOfSOAPStruct"/>
</message>
<message name="echoStructArrayResponse">
  <part name="return" type="s:ArrayOfSOAPStruct"/>
</message>
<message name="echoVoidRequest"/>
<message name="echoVoidResponse"/>
<message name="echoBase64Request">
```

```
    <part name="inputBase64" type="xsd:base64Binary"/>
  </message>
  <message name="echoBase64Response">
    <part name="return" type="xsd:base64Binary"/>
  </message>
  <message name="echoDateRequest">
    <part name="inputDate" type="xsd:dateTime"/>
  </message>
  <message name="echoDateResponse">
    <part name="return" type="xsd:dateTime"/>
  </message>
  <message name="echoDecimalRequest">
    <part name="inputDecimal" type="xsd:decimal"/>
  </message>
  <message name="echoDecimalResponse">
    <part name="return" type="xsd:decimal"/>
  </message>
  <message name="echoBooleanRequest">
    <part name="inputBoolean" type="xsd:boolean"/>
  </message>
  <message name="echoBooleanResponse">
    <part name="return" type="xsd:boolean"/>
  </message>

  <portType name="InteropTestPortType">
    <operation name="echoString" parameterOrder="inputString">
      <input message="tns:echoStringRequest"/>
      <output message="tns:echoStringResponse"/>
    </operation>
    <operation name="echoStringArray" parameterOrder="inputStringArray">
      <input message="tns:echoStringArrayRequest"/>
      <output message="tns:echoStringArrayResponse"/>
    </operation>
    <operation name="echoInteger" parameterOrder="inputInteger">
      <input message="tns:echoIntegerRequest"/>
      <output message="tns:echoIntegerResponse"/>
    </operation>
    <operation name="echoIntegerArray" parameterOrder="inputIntegerArray">
      <input message="tns:echoIntegerArrayRequest"/>
      <output message="tns:echoIntegerArrayResponse"/>
    </operation>
    <operation name="echoFloat" parameterOrder="inputFloat">
      <input message="tns:echoFloatRequest"/>
      <output message="tns:echoFloatResponse"/>
    </operation>
  </portType>

```

```

<operation name="echoFloatArray" parameterOrder="inputFloatArray">
  <input message="tns:echoFloatArrayRequest" />
  <output message="tns:echoFloatArrayResponse" />
</operation>
<operation name="echoStruct" parameterOrder="inputStruct">
  <input message="tns:echoStructRequest" />
  <output message="tns:echoStructResponse" />
</operation>
<operation name="echoStructArray" parameterOrder="inputStructArray">
  <input message="tns:echoStructArrayRequest" />
  <output message="tns:echoStructArrayResponse" />
</operation>
<operation name="echoVoid">
  <input message="tns:echoVoidRequest" />
  <output message="tns:echoVoidResponse" />
</operation>
<operation name="echoBase64" parameterOrder="inputBase64">
  <input message="tns:echoBase64Request" />
  <output message="tns:echoBase64Response" />
</operation>
<operation name="echoDate" parameterOrder="inputDate">
  <input message="tns:echoDateRequest" />
  <output message="tns:echoDateResponse" />
</operation>
<operation name="echoDecimal" parameterOrder="inputDecimal">
  <input message="tns:echoDecimalRequest" />
  <output message="tns:echoDecimalResponse" />
</operation>
<operation name="echoBoolean" parameterOrder="inputBoolean">
  <input message="tns:echoBooleanRequest" />
  <output message="tns:echoBooleanResponse" />
</operation>
</portType>

```

Notice that the WSDL document contains more complex types than the previous demo. Array of primitives types are now used as well as the struct primitive types. With the exception of the SOAPStruct complex type, every other type is supported as built-in type in the SOAP Mapping Registry. You then need to add a new complex type definition to the SOAP Mapping Registry to handle the SOAPStruct complex type.

The SOAPStruct schema definition is the following:

```

<complexType name="SOAPStruct">
  <all>

```

```
<element name="varString" type="string"/>
<element name="varInt" type="int"/>
<element name="varFloat" type="float"/>
</all>
</complexType>
```

In the `MySoapStructBean.java` file, this `SOAPStruct` complex type can be mapped to a simple JavaBean class such as the following, and have the marshalling and unmarshalling actions handled by the `BeanSerializer`.

```
public class MySoapStructBean implements java.io.Serializable
{
    private String m_varString = null;
    private int m_varInt = 0;
    private float m_varFloat = 0;

    public MySoapStructBean() {}
    public MySoapStructBean(String s, int i, float f) {
        m_varString = s;
        m_varInt = i;
        m_varFloat = f;
    }

    public String getVarString () { return m_varString; }
    public int getVarInt() { return m_varInt; }
    public float getVarFloat() { return m_varFloat; }

    public void setVarString (String s) { m_varString = s; }
    public void setVarInt(int i) { m_varInt = i; }
    public void setVarFloat(float f) { m_varFloat = f; }
}
```

With the mapping JavaBean class ready, and having identified what serializer and deserializer to use, you can now configure the `wsdl2ejb` utility so that a new schema to Java map is added. This can be achieved by adding the following to the `wsdl2ejb` configuration file, `base_conf.xml`:

```
<mapTypes jar="base/MySoapStructBean.jar" >
  <map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    local-name="SOAPStruct"
    namespace-uri="http://soapinterop.org/xsd"
    java-type="MySoapStructBean"
    java2xml-class-name="org.apache.soap.encoding.soapenc.BeanSerializer"
    xml2java-class-name="org.apache.soap.encoding.soapenc.BeanSerializer" />
</mapTypes>
```

The `MySoapStructBean.jar` file contains the definition of the `MySoapStructBean` class. With this map, the `SOAPStruct` complex type, belonging to the `http://soapinterop.org/xsd` namespace, will be mapped to the `MySoapStructBean` JavaBean class and the converse is true as well. For more information about SOAP serializers and deserializers, see the Oracle SOAP documentation.

With this additional configuration, you can now run the `wsdl2ejb` utility with the following command:

```
On UNIX
cd $ORACLE_HOME/webservices/demo/basic/wsdl2ejb
java -jar ../../../../lib/wsdl2ejb.jar -conf base/base_conf.xml
On Windows
cd %ORACLE_HOME%\webservices/demo/basic/wsdl2ejb
java -jar ../../../../lib/wsdl2ejb.jar -conf base\base_conf.xml
```

The `wsdl2ejb` utility generates the `InteropLabApp.ear` file that contains the definition of a stateless EJB, which can be used as a proxy for the Web Service. The EAR file can be deployed in Oracle9iAS OC4J as any standard EJB. See *Oracle9iAS Containers for J2EE User's Guide* for information on how to deploy an EJB.

The `TestInteropBaseClient.java` class file, saved in the `base` directory, can be used to test the generated EJB after it has been deployed. The result of the execution of the client is the following:

```
echoString: Hello World!
echoStringArray[0]: Hello World!
echoStringArray[1]: Seems to work!
echoStringArray[2]: Fine!
echoStringArray[3]: WOW
echoInteger: 7
echoIntegerArray[0]: 1
echoIntegerArray[1]: 2
echoIntegerArray[2]: 3
echoIntegerArray[3]: 4
echoFloat: 1.7777
echoFloatArray[0]: 1.1
echoFloatArray[1]: 1.2
echoFloatArray[2]: 1.3
echoFloatArray[3]: 1.4
echoStruct: varString=Hello World , varInt=1 , varFloat=1.777
echoStructArray: varString[0]=Hello World , varInt[0]=0 , varFloat=[0]=1.7771
echoStructArray: varString[1]=Hello World 1 , varInt[1]=1 , varFloat=[1]=1.7772
```

```
echoStructArray: varString[2]=Hello World 2 , varInt[2]=2 , varFloat=[2]=1.7773
echoStructArray: varString[3]=Hello World 3 , varInt[3]=3 , varFloat=[3]=1.7774
echoVoid.
echoDecimal: 1.77709999999999990194510246510617434978485107421875
echoBoolean: true
echoBase64[0]: 1
echoBase64[1]: 2
echoBase64[2]: 3
echoBase64[3]: 4
echoDate: Sat Nov 10 12:30:00 EST 2001
```

Dynamic Invocation of Web Services

When a Java2 Enterprise Edition (J2EE) application acquires a WSDL document at runtime, the dynamic invocation API is used to invoke any SOAP operation described in the WSDL document. The dynamic invocation API describes a `WebServiceProxyFactory` factory class that can be used to build instances of a `WebServiceProxy`. Each created `WebServiceProxy` instance is based on the location of the WSDL document, (and optionally on additional qualifiers), that identify which service and port should be used. The `WebServiceProxy` class exposes methods to determine the WSDL portType, including the syntax and signatures of all operations exposed by the WSDL service and to invoke the defined operations.

This section briefly describes the dynamic invocation API and how to use it.

For Java samples, refer to the code supplied with Oracle9iAS Web Services in `$ORACLE_HOME/webservices/demo/basic/java_services/dynamicproxy` on UNIX or in `%ORACLE_HOME%\webservices\demo\basic\java_services\dynamicproxy` on Windows. For EJB samples, refer to the code supplied in the directory `$ORACLE_HOME/webservices/demo/basic/stateless_ejb` on UNIX or `%ORACLE_HOME%\webservices\demo\basic\stateless_ejb` on Windows.

Dynamic Invocation API

The dynamic invocation API contains two packages, `oracle.j2ee.ws.client` and `oracle.j2ee.ws.client.wsdl`, which contain additional classes grouped by interface, class, and exception, as shown in [Table 11-4](#) and [Table 11-5](#).

Table 11–4 The oracle.j2ee.ws.client Package

Classes	Description
Classes	
WebServiceProxyFactory	This class creates a WebServiceProxy class given a WSDL document.
Interfaces	
WebServiceProxy	This interface represents a service defined in a WSDL document.
WebServiceMethod	This interface invokes a Web Service method.
Exceptions	
WebServiceProxyException	This class describes exceptions raised by the WebServiceProxy API.

Table 11–5 The oracle.j2ee.ws.client.wsdl Package

Classes	Description
Interfaces	
PortType	This interface represents a port type.
Operation	This interface represents a WSDL operation.
Input	This interface represents an input message, and contains the name of the input and the message itself.
Output	This interface represents an output message, and contains the name of the output and the message itself.
Fault	This interface represents a fault message, and contains the name of the fault and the message itself.
Message	This interface describes a message used for communication with an operation.
Part	This interface represents a message part and contains the part's name, elementName, and typeName.
Classes	
OperationType	This class represents an operation type which can be one of request-response, solicit response, one way, or notification.

The `oracle.j2ee.ws.client` package is described in more detail in this section. The API documentation describes to use this proxy API can be found in the Oracle9iAS Documentation Library as Proxy API Reference (Javadoc) under Oracle9iAS Web Services, which is located under the J2EE and Internet Applications tab.

The `WebServiceProxyFactory` class contains methods that can instantiate a `WebServiceProxy` class given either the URL or the Java input stream of the WSDL document. Four methods let you use either the first service and its first port in the supplied WSDL document or use the name of one of services and the name of one of the ports of the service to create a `WebServiceProxy` instance. Two methods also let you create a `WebServiceProxy` instance for a WSDL document, which has been authored following the UDDI best practices for WSDL. A method lets you supply additional optional initialization parameters to the `WebServiceProxy` instance.

[Table 11–6](#) briefly describes the `WebServiceProxyFactory` factory class methods and the required parameters for each method. See the JavaDoc for more detailed information about this factory class and its methods.

Table 11–6 *WebServiceProxyFactory Factory Methods and Parameters*

Methods	Parameters
<code>createWebServiceProxy()</code>	<code>java.io.InputStream isWsdL</code> <code>java.net.URL baseUrl</code>
<code>createWebServiceProxy()</code>	<code>java.net.URL wsdlURL</code>
<code>createWebServiceProxyFromBinding()</code>	<code>java.io.InputStream wsdlis</code> <code>java.net.URL baseUrl</code> <code>java.lang.String szBindingName</code> <code>java.lang.String szSoapLocation</code>
<code>createWebServiceProxyFromService()</code>	<code>java.io.InputStream wsdlis</code> <code>java.net.URL baseUrl</code> <code>java.lang.String szServiceName</code> <code>java.lang.String szServicePort</code>
<code>createWebServiceProxyFromBinding()</code>	<code>java.net.URL wsdlUrl</code> <code>java.lang.String szBindingName</code> <code>java.lang.String szSoapLocation</code>
<code>createWebServiceProxyFromService()</code>	<code>java.net.URL wsdlUrl</code> <code>java.lang.String szServiceName</code> <code>java.lang.String szServicePort</code>
<code>setProperties()</code>	<code>java.util.Hashtable ht</code>

Table 11–7 describes the `WebServiceProxy` interface. The `WebServiceProxyFactory` factory methods optionally take additional parameters that are provided in the `WebServiceProxy` interface that can be used to dynamically invoke an operation in a WSDL document.

Table 11–7 *WebServiceProxy Interface Methods and Parameters*

Methods	Parameters	Description
<code>getXMLMappingRegistry()</code>	None	Returns the SOAP mapping registry used by the <code>WebServiceProxy</code> and contains information that lets clients use this registry to query for XML to or from Java type mapping as well as extend the mapping registry with new map definitions.
<code>getPortType()</code>	None	Returns a structure describing the WSDL portType used by this proxy and contains information about operations associated with this port type.
<code>getMethod()</code>	<code>szOperationName</code> <code>szInputName</code> <code>szOutputName</code>	Returns a <code>WebServiceMethod</code> method, which can be used to invoke Web Service methods. Name of the WSDL operation to be executed. Name of the <code>wsdl:input</code> tag for the operation to be executed. Name of the <code>wsdl:output</code> tag for the operation to be executed.
<code>getMethod()</code>	<code>szOperationName</code>	Returns a <code>WebServiceMethod</code> method, which can be used to invoke Web service methods and provides a signature that can be used for non-overloaded WSDL operations. Name of the WSDL operation to be executed.

Table 11–8 describes the `WebServiceMethod` interface, which is used to invoke a Web Service method.

Table 11–8 *WebServiceMethod Interface Methods and Parameters*

Methods	Parameters	Description
<code>getInputEncodingStyle()</code>	None	Returns the encoding style to be used by the input message parts, null if none has been specified in the source WSDL.
<code>getOutputEncodingStyle()</code>	None	Returns the encoding style to be used by the output message parts, null if none has been specified in the source WSDL.
<code>invoke()</code>	<code>inMsgPartNames</code> <code>inMsgPartValues</code>	Executes one of the service operations with the set of supplied input parts and returns the object, if the response message contains only one part, return the response part, otherwise an array of the output message parts. If the invoked WSDL operation has no output parts, null will be returned. Name of the parts supplied in the input message. Corresponding value of the parts whose name is supplied in the <code>inMsgPartNames</code> parameter. If the invoked WSDL operation has no input parts, null or empty arrays parameters can be supplied

The `oracle.j2ee.ws.client.wsdl` package exposes methods to determine the WSDL portType, including the syntax and signatures of all operations exposed by the WSDL service.

WebServiceProxy Client

The following client code shows the use of the dynamic invocation API followed by the output of the client execution. The client code shows the following:

- Initializes proxy parameters in the `WebServiceProxyFactory`.
- Creates an instance of the proxy given a URL of a WSDL document.
- Performs WSDL introspection.
- Shows the input message parts.
- Executes a Web Service operation with a set of supplied input parts and returns the result.

The WSDL document is described as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
- <definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://soapinterop.org"
xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" targetNamespace="http://soapinterop.org"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types />
  - <message name="AddSoapIn">
    <part name="a" type="s:int" />
    <part name="b" type="s:int" />
  </message>
  - <message name="AddSoapOut">
    <part name="AddResult" type="s:int" />
  </message>
  - <portType name="TestSoap">
    - <operation name="Add">
      <input message="tns:AddSoapIn" />
      <output message="tns:AddSoapOut" />
    </operation>
  </portType>
  - <binding name="TestSoap" type="tns:TestSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    - <operation name="Add">
      <soap:operation soapAction="http://soapinterop.org/Add" style="rpc" />
      - <input>
        <soap:body use="encoded" namespace="http://soapinterop.org"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
      - <output>
        <soap:body use="encoded" namespace="http://soapinterop.org"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>
  - <service name="Test">
    - <port name="TestSoap" binding="tns:TestSoap">
      <soap:address location="http://msssoapinterop.org/asmx/Rpc.asmx" />
    </port>
  </service>
</definitions>
```

```
package oracle.j2ee.ws.client.impl;
```

```
import java.util.*;
import java.io.*;
import java.net.*;
import oracle.j2ee.ws.client.*;
```

```
import oracle.j2ee.ws.client.wsdl.*;
import org.apache.soap.util.xml.QName;
import org.apache.soap.util.xml.XMLJavaMappingRegistry;

public class Client {

    public static void main(String[] args) throws Exception {

        String szWsdUrl = "http://mssoapinterop.org/asmx/Rpc.asmx?WSDL";

        URL urlWsdl = new URL(szWsdUrl);
        System.err.println("Wsd url = " + urlWsdl);

        WebServiceProxyFactory wsfact= new WebServiceProxyFactory();

        //
        // Set some initial parameters
        //
        Hashtable ht = new Hashtable();
        ht.put("http.proxyHost", "www-proxy.us.oracle.com");
        ht.put("http.proxyPort", "80");
        wsfact.setProperties(ht);

        //
        // Create an instance of the proxy
        //
        WebServiceProxy wsp = wsfact.createWebServiceProxy(urlWsdl);

        //
        // Optional: Wsd Introspection
        //
        PortType pt = wsp.getPortType();
        List opList = pt.getOperations();
        for (int i = 0; i < opList.size(); i++) {

            Operation op = (Operation) opList.get(i);
            String szOpName = op.getName();
            String szInput = op.getInput().getName();
            String szOutput = op.getOutput().getName();

            System.err.println("operation["+i+"] = [" + szOpName +
                ", " + szInput + ", " + szOutput + "]);

            //
            // show input message parts
```

```

//
Message msgIn = op.getInput().getMessage();
Map    mapParts = msgIn.getParts();
Collection colParts = mapParts.values();
Iterator itParts = colParts.iterator();

WebServiceMethod wsm = wsp.getMethod(szOpName);
String szInEncStyle = wsm.getInputEncodingStyle();
XMLJavaMappingRegistry xmr = wsp.getXMLMappingRegistry();

while (itParts.hasNext()) {
    Part part = (Part) itParts.next();
    String szPartName = part.getName();
    QName qname = part.getTypeName();
    String szJavaType = xmr.queryJavaType(qname,
szInEncStyle).getName();
    System.err.println("part name = " + szPartName +
        ", type = " + qname +
        ", java type = " + szJavaType);
}
}

//
// invoke operation/method Add(2,10)
//
String[] inMsgPartNames = new String[2];
inMsgPartNames[0] = "a";
inMsgPartNames[1] = "b";
Object[] inMsgPartValues = new Object[2];
inMsgPartValues[0] = new Integer(2);
inMsgPartValues[1] = new Integer(10);

WebServiceMethod wsm = wsp.getMethod("Add");
Object objRet = wsm.invoke(inMsgPartNames,
    inMsgPartValues);

System.err.println("Calling method Add(" + inMsgPartValues[0] + ", "
+
    inMsgPartValues[1] + ")");
System.err.println("return = " + objRet);
}
}

```

The output of the client execution is as follows:

```
WSDL url = http://mssoapinterop.org/asmx/Rpc.asmx?WSDL
operation[0] = [Add,,]
part name = b, type = http://www.w3.org/2001/XMLSchema:int, java type = int
part name = a, type = http://www.w3.org/2001/XMLSchema:int, java type = int
Calling method Add(2,10)
return = 12
```

Known Limitations

The following information describes the known limitations of the dynamic invocation API:

- Supports invoking operations defined in the WSDL document defined by the W3C recommendation XML schema version whose namespace is:
`http://www.w3.org/2001/XMLSchema`
- Does not support WSDL documents that use the `<import>` tag to include other WSDL documents.
- Does not support HTTP, MIME, or any other custom bindings.

Using Oracle SOAP

This appendix covers the following topics:

- [Understanding Oracle9iAS SOAP](#)
- [Apache SOAP Documentation](#)
- [Configuring the SOAP Request Handler Servlet](#)
- [Using Oracle9iAS SOAP Management Utilities and Scripts](#)
- [Deploying Oracle9iAS SOAP Services](#)
- [Using Oracle9iAS SOAP Handlers](#)
- [Using Oracle9iAS SOAP Audit Logging](#)
- [Using Oracle9iAS SOAP Pluggable Configuration Managers](#)
- [Working With Oracle9iAS SOAP Transport Security](#)
- [Using Oracle9iAS SOAP Sample Services](#)
- [Using the Oracle9iAS SOAP EJB Provider](#)
- [Using PL/SQL Stored Procedures With the SP Provider](#)
- [SOAP Troubleshooting and Limitations](#)
- [Oracle9iAS SOAP Differences From Apache SOAP](#)

Understanding Oracle9iAS SOAP

In addition to the Oracle9iAS Web Services previously described in this chapter, that use a unique Servlet interface and J2EE deployment for Web Services, Oracle9iAS also provides Oracle9iAS SOAP that is derived from Apache 2.2 SOAP and includes a number of enhancements.

The SOAP Message Processor (Oracle9iAS SOAP), provides the following facilities:

- SOAP Protocol Handling - It provides an implementation of the interoperable SOAP specification. This includes support for Cookies and Sessions which is particularly useful to pass state information for stateful Web Services request/response.
- Support for SOAP requests with Attachments (XML Payloads).
- Parsing - Oracle9iAS SOAP Processor integrates the Oracle XML Parser. For RPC-style requests, the Oracle9iAS SOAP Processor can efficiently parse the incoming XML document, ensure the request is well-formed, and possibly validate the request. Similarly, it can also encode/serialize a Java response into a SOAP message.
- Invoking Web Service Using Customized Web Services Servlet - The SOAP Processor un-marshals the message contents and depending on the Servlet, calls the Web Services implementation. Web Services can be implemented as Java Classes, EJBs, or PL/SQL Stored Procedures.
- Engaging a security manager to possibly authenticate the sender - Before invoking the Web Services implementation, the Oracle9iAS SOAP Processor (Servlet) authenticates the user using a standard JAAS-based User Manager plug-in. Oracle9iAS SOAP Processor also supports Oracle's Single Sign-On Server and third-party authentication services to provide single-sign on for Web Services.
- Exception Handling - When exceptions occur during processing, the Java Exception is transformed to a SOAP fault and delivered to the service client.

Apache SOAP Documentation

Oracle9iAS SOAP is a modified version of Apache SOAP 2.2. Most of the documentation that applies to Apache SOAP 2.2 also applies to Oracle9iAS SOAP. The Apache SOAP 2.2 documentation can be found at the following site:

<http://xml.apache.org/soap/docs/index.html>

Configuring the SOAP Request Handler Servlet

The Oracle9iAS SOAP Request Handler uses an XML configuration file to set required servlet parameters. By default, this file is named `soap.xml` and is placed in the directory `$SOAP_HOME/webapps/soap/WEB-INF` on UNIX or `%SOAP_HOME%\webapps\soap\WEB-INF` on Windows. The XML namespace for this file is:

`http://xmlns.oracle.com/soap/2001/04/config`

To use a different configuration file for SOAP installation, modify the path name specified for the `SoapConfig` parameter in the `soap.properties` file. For example, to change the configuration file from the default, `soap.xml`, to `newConfig.xml`, modify the value set for `soapConfig` in `soap.properties`.

```
servlet.soaprouter.initArgs=soapConfig=soap_home/soap/webapps/soap/WEB-INF/newConfig.xml
```

Where `soap_home` is the full path to the SOAP installation on your system.

The `pathAuth` boolean attribute, if set to `true`, enforces that clients must specify the unique service URL in order to post a message to the deployed service. The service URL is the SOAP servlet URL with the service URI appended on at the end. The default value of this attribute (if unspecified) is `false`.

[Table A-1](#) lists the SOAP Request Handler Servlet XML configuration file elements.

Table A-1 SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
<code>errorHandlers</code>	Specifies a list of handlers for the error handler chain.
<code>faultListeners</code>	This is an optional element that defines a list of <code>faultListener</code> elements. The <code>faultListener</code> element specifies a class that is invoked when a fault occurs. To cause a stack trace to be added to the SOAP fault that is returned to the user, specify a <code>faultListener</code> of <code>org.apache.soap.server.DOMFaultListener</code> .

Table A-1 (Cont.) SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
handler	<p>The handlers element is an optional element that defines a list of handler elements. The handler element defines a global handler that can be configured to be invoked on every SOAP request in one of three contexts: request, response, error. You can define any number of handlers. The handler's name attribute specifies the name of the handler; each handler must have a unique name. The handler's class attribute specifies the Java class that implements the handler, and this class must implement the interface <code>oracle.soap.server.Handler</code>. Each handler may have any number of options, which are name-value pairs. The contexts are configured in the elements: <code>requestHandlers</code>, <code>responseHandlers</code>, and <code>errorHandlers</code>. Each of these elements defines an ordered list of handler names, or a chain of handlers.</p> <p>Note that SOAP creates one instance of each uniquely identified handler. Every appearance of a specific handler name in any chain refers to the same instance of the handler. Handlers are destroyed when the SOAP servlet is destroyed.</p>
logger	<p>Error and informational messages are logged using the class defined in the logger element. The logger class must extend <code>oracle.soap.server.Logger</code>.</p> <p>Oracle9iAS SOAP includes the class <code>oracle.soap.server.impl.ServletLogger</code> that collects the servlet log methods so that SOAP messages are logged to the servlet log file. <code>ServletLogger</code> is the default logger. For the default logger, the severity option can be to any of the following values: <code>status</code>, <code>error</code>, <code>debug</code>.</p> <p>If you specify <code>error</code>, you will get both <code>status</code> and <code>error</code> messages. Similarly, if you specify <code>debug</code>, you will get all three types of messages.</p> <p>Oracle9iAS SOAP includes two logger implementations. To log to the servlet log, use <code>oracle.soap.server.impl.ServletLogger</code>. To log to stdout, use <code>oracle.soap.server.impl.StdoutLogger</code>.</p> <p>You may implement your own logger by implementing the <code>oracle.soap.server.Logger</code> interface.</p>

Table A-1 (Cont.) SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
<code>providerManager</code>	<p>The <code>providerManager</code> is an optional element that allows a configuration manager to be defined. This defines how the server accesses provider deployment information.</p> <p>The <code>providerManager</code> class attribute specifies a Java class that implements <code>oracle.soap.server.ProviderManager</code>. The default configuration manager, <code>oracle.soap.server.impl.XMLProviderConfigManager</code>, persists the deployed providers to a file in XML format. It accepts a <code>filename</code> option. The filename is the path to the registry filename which may be a simple file name, relative path or an absolute path. If it is not an absolute path, then the path is determined from the filename and the servlet context. The default filename is <code>WEB-INF/providers.xml</code>.</p> <p>An alternative provider configuration manager, <code>oracle.soap.server.impl.BinaryProviderConfigManager</code>, persists the deployed providers in a file as a serialized object. The default file is <code>WEB-INF/providers.dd</code>.</p> <p>To specify a different configuration manager add a class attribute to the <code>configManager</code> element. For example:</p> <pre><osc:configManager class="fully.qualified.classname">.</pre>
<code>requestHandlers</code>	Specifies a list of handlers for the request handler chain
<code>responseHandlers</code>	Specifies a list of handlers for the response handler chain
<code>serviceManager</code>	<p>The <code>serviceManager</code> is an optional element that allows a configuration manager to be defined and <code>ServiceManager</code> options to be set. This defines how the server accesses service deployment information. The <code>serviceManager</code> class attribute specifies a Java class that implements <code>oracle.soap.server.ServiceManager</code>.</p> <p>The default Oracle9iAS SOAP configuration manager class is <code>oracle.soap.server.impl.XMLServiceConfigManager</code> which stores the service deployment information in an XML file. Using <code>XMLServiceConfigManager</code>, the file name is specified with the <code>filename</code> option. The filename is the path to the registry filename which may be a simple file name, relative path or an absolute path. If it is not an absolute path, then the path is determined from the filename and the servlet context. The default filename is <code>WEB-INF/services.xml</code>.</p> <p>To specify a different configuration manager add a class attribute to the <code>configManager</code> element.</p> <p>For example:</p> <pre><osc:configManager class="fully.qualified.classname">.</pre> <p>An alternative service configuration manager, <code>oracle.soap.server.impl.BinaryServiceConfigManager</code>, persists the deployed services in a file as a serialized object. The default file is <code>WEB-INF/services.dd</code>.</p> <p>The service manager can automatically deploy the provider manager and the service manager as SOAP services. To allow these managers to be exposed as services, set the <code>autoDeploy</code> option to <code>true</code>. By default <code>autoDeploy</code> value is <code>false</code>.</p>

Using Oracle9iAS SOAP Management Utilities and Scripts

To use the Oracle9iAS SOAP management utilities, you need to set up the execution environment for executing SOAP management utilities using one of the supplied client side scripts. The `clientenv` scripts set the `CLASSPATH` and add the `$SOAP_HOME/bin` directory to the path.

To set the client environment, on UNIX, use the following commands:

```
cd $SOAP_HOME/bin
source clientenv.csh
```

On Windows, use the following commands:

```
cd %SOAP_HOME%\bin
clientenv.bat
```

The `clientenv` scripts sets environment variables that are used by the other scripts and the samples. You can override these by setting the environment variables yourself. The variable `SOAP_URL` is the URL of the SOAP server and `JAXP` is set to use the `DocumentBuilderFactory` for the Oracle XML parser.

Managing Providers

The `providerMgr` script runs the SOAP client that manages providers. Run the script without any parameters for usage information.

On UNIX, use the following command:

```
providerMgr.sh options
```

On Windows, use the following command:

```
providerMgr.bat options
```

Where the *options* for `providerMgr` are:

```
deploy ProviderDescriptorFile
```

This deploys the provider described in the *ProviderDescriptorFile* and makes the provider available.

```
undeploy ProviderID
```

This removes the provider with the supplied *ProviderID*. The *ProviderID* is the `id` attribute specified in the provider descriptor file.

The Java provider is deployed once at installation time with `id=java-provider`, but any provider you create must be explicitly deployed. For example, on UNIX, to deploy a provider using the provider deployment descriptor `provider.xml`, use the following command:

```
providerMgr.sh deploy provider.xml
```

Using the Service Manager to Deploy and Undeploy Java Services

The `ServiceMgr` is an administrative utility that deploys and undeploys SOAP services. To deploy a service, first set the SOAP environment, then use the `deploy` command. On UNIX, the command is:

```
source clientenv.csh
ServiceMgr.sh deploy ServiceDescriptorFile
```

For Windows, the command is:

```
clientenv.bat
ServiceManager.bat deploy ServiceDescriptorFile
```

The `deploy` option makes the service specified in *ServiceDescriptorFile* available.

When you are ready to undeploy a service, use the `undeploy` command with the registered service name as an argument. On UNIX, the command is:

```
ServiceManager.sh undeploy ServiceID
```

For Windows, the command is:

```
ServiceManager.bat undeploy ServiceID
```

This makes the service with the given `id` unavailable. The *ServiceID* is the service `id` attribute specified in the service descriptor file.

The `ServiceMgr` supports listing and querying SOAP services. To list the available services, first set the SOAP environment, then use the `list` command. On UNIX, the command is:

```
source clientenv.csh
ServiceMgr.sh list
```

On Windows, the command is:

```
clientenv.bat  
ServiceMgr.bat list
```

To query a service and obtain the descriptor parameters set in the service deployment descriptor file, use the `query` command. On UNIX, the command is:

```
ServiceMgr.sh query ServiceID
```

On Windows, the command is:

```
ServiceMgr.bat query ServiceID
```

Where *ServiceID* is the service id attribute set in the service descriptor file.

Generating Client Proxies from WSDL Documents

The `wsdl2java` script takes as input a WSDL document and returns a Java class which can be used to call the service. The Java class contains methods with the same names as those described in the WSDL document. The generated code make calls to the Apache client side libraries.

On UNIX, use the following command:

```
wsdl2java.sh options
```

On Windows, use the following command:

```
wsdl2java.bat options
```

Where the *options* for `wsdl2java` are:

```
wsdl2java . sh WsdldocumentURL OutputDir [-k PackageName] [-s ServiceName]  
[-p PortName]
```

Where:

WsdldocumentURL is the URL of the WSDL document.

OutputDir is the output directory for generated proxy Java code.

`-k PackageName` is the package name for generated proxy Java code.

`-s ServiceName` is the service name for which proxy will be generated.

-p *PortName* the port name of the service. The proxy is generated for the specified port of the service.

The output directory structure is:

output root dir / service name / port name / package name / java proxy source code

By default, the *PackageName* will be the same as the WSDL service name.

If neither of -s and -p options is specified, proxies for all ports of all services are generated. Without -p option specified, proxies for all ports of the specified service are generated.

Generating WSDL Documents from Java Service Implementations

The `java2wsdl` script takes as input a Java class and creates as output a WSDL document describing the class as an RPC service. When the Java class is used as a Web Service, the associated WSDL document can be transmitted to developers who might wish to call the service.

On UNIX, use the following command:

```
java2wsdl.sh options
```

On Windows, use the following command:

```
java2wsdl.bat options
```

Where the *options* for `wsdl2java` are:

```
java2wsdl.sh ClassName OutputFile SoapURL ClassURL1 ClassURL2 ...
```

Where:

ClassName is the fully qualified path name of a Java `.class` file that is to be a Web Service.

OutputFile is the output WSDL document name.

SoapURL is the SOAP endpoint.

ClassURL list serves as a class path for searching referenced classes

Deploying Oracle9iAS SOAP Services

This section covers the following topics related to deploying and undeploying Oracle9iAS SOAP Services:

- [Creating Deployment Descriptors](#)
- [Installing a SOAP Web Service in OC4J](#)
- [Disabling an Installed SOAP Web Service](#)
- [Installing a SOAP Web Service in an OC4J Cluster](#)

Creating Deployment Descriptors

Deployment descriptors include service deployment descriptors and provider deployment descriptors. A provider deployment descriptor file is an XML file that describes, to the SOAP servlet, the configuration information for a provider. A service deployment descriptor file is an XML file that describes, to the SOAP servlet, the configuration information for a service.

Services written in Java only require a service descriptor. All Java service descriptors may point to the same Java provider descriptor supplied with the Oracle9iAS SOAP installation.

Each service written as a PL/SQL stored procedure requires one service descriptor and one provider descriptor for each database user. The advantage of this is that when a password or user is changed, only one descriptor needs to be updated, not every service descriptor.

See the Stored Procedure section for more information.

Services written as an EJB require one service descriptor and one provider descriptor for each EJB container user.

See the EJB section of this document for more information.

Note: For developers who wish to write their own providers, the Apache style provider interface and descriptors are also supported. Apache descriptors contain both service and provider properties in a single file, so common provider information must be duplicated for every service.

A service deployment descriptor file defines the following information:

- The service ID
- The service provider type (for example, Java)
- The available methods

The best way to write a descriptor is to start with a copy of an existing descriptor from one of the sample directories.

[Example A-1](#) shows the Java SimpleClock service descriptor file `SimpleClockDescriptor.xml`. This descriptor file is included in the `samples/simpleclock` directory. The service descriptor file must conform to the service descriptor schema (the schema, `service.xsd`, is located in the directory `$SOAP_HOME/schemas` on UNIX or in `%SOAP_HOME%\schemas` on Windows).

The service descriptor file identifies methods associated with the service in the `isd:provider` element that uses the `methods` attribute. The `isd:java class` element identifies the Java class that implements the SOAP service, and provides an indication of whether the class is static.

Example A-1 Java Service Descriptor File for Sample Simple Clock Service

```
<isd:service xmlns:isd="http://xmlns.oracle.com/soap/2001/04/deploy/service"
  id="urn:jurassic-clock"
  type="rpc" >
  <isd:provider
    id="java-provider"
    methods="getDate"
    scope="Application" >
    <isd:java class="samples.simpleclock.SimpleClockService"/>
  </isd:provider>
  <!-- includes stack trace in fault -->
  <isd:faultListener class="org.apache.soap.server.DOMFaultListener"/>
</isd:service>
```

Note: The service descriptor file does not define the method signature for service methods. SOAP uses reflection to determine method signatures.

Installing a SOAP Web Service in OC4J

Install an Oracle9iAS SOAP Web Service in Oracle9iAS Containers for J2EE (OC4J) by performing the following steps:

1. Create service and provider deployment descriptors.
2. Copy Java classes and Jars implementing the service to the correct locations.

Copy Java .class files to \$SOAP_HOME/WEB-INF/classes. The new classes will automatically be found by the OC4J servlet container.

Copy Java .jar files to \$SOAP_HOME/WEB-INF/libs.

In order for the new files to be found by the OC4J servlet container, you must either restart the OC4J servlet engine or update the SOAP application configuration file.

3. Deploy the provider descriptor by executing the command:

```
providerMgr.sh deploy FileName
```

where *FileName* is the name of the provider descriptor xml file.

4. Deploy the service by executing the command:

```
serviceMgr.sh deploy FileName
```

Where *FileName* is the name of the service descriptor xml file.

Disabling an Installed SOAP Web Service

To disable an installed service, run the command:

```
serviceMgr.sh undeploy ServiceID
```

where *ServiceID* is the id attribute of the service element in the service descriptor.

Installing a SOAP Web Service in an OC4J Cluster

An OC4J Cluster consists of two or more machines of similar configuration front ended with a hardware or software dispatcher. OC4J has facilities for insuring that files on the machines remain in synch. For instance, a servlet installed once will automatically be installed on the local file systems of the other machines in the system.

However, Oracle9iAS SOAP services are not handled by the OC4J clustering software. It is necessary to install an Oracle9iAS SOAP service on every machine in

the cluster. If the service is not installed on all machines in the cluster, 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.

Using Oracle9iAS SOAP Handlers

A handler is a class that implements the `oracle.soap.server.Handler` interface. A handler can be configured as part of a chain in one of three contexts: request, response, or error. Note that handlers in a chain are invoked in the order they are specified in the configuration file.

Request Handlers

Handlers in the request chain are invoked on every request that arrives, immediately after the SOAP Request Handler Servlet reads the SOAP Envelope. If any handler in the request chain throws an exception, the processing of the chain is immediately terminated and the service is not invoked.

The error chain is invoked if any exception occurs during request chain invocation.

Response Handlers

Handlers in the response chain are invoked on every request immediately after the service completes. If any handler in the response chain throws an exception, processing of the chain is immediately terminated. The error chain is invoked if any exception occurs during response chain invocation.

Error Handlers

When an exception occurs during either request-chain invocation, service invocation, or response-chain invocation, the SOAP Request Handler Servlet invokes the handlers in the error chain. In contrast to the request and response chains, an exception from an error handler is logged and processing of the error chain continues. All handlers in the error chain are invoked, regardless of whether one of the error handlers throws an exception.

Configuring Handlers

Configure handlers and handler chains in the SOAP configuration file. Handlers can be invoked for each service request or response, or when an error occurs.

Handlers are global in the sense that they apply to every SOAP request and cannot be configured on a subset of requests, such as all requests for a particular service.

Configure a handler by setting parameters in the SOAP configuration file, `soap.xml`. [Example A-2](#) shows a sample segment from a SOAP configuration file showing the configuration for a handler.

Example A-2 Handler Configuration

```
<osc:handlers>
  <osc:handler name="auditor"
    class="oracle.soap.handlers.audit.AuditLogger">
    <osc:option name="auditLogDirectory"
      value="/private1/oracle/app/product/tv02/soap/webapps/soap/WEB-INF"/>
    <osc:option name="filter" value="(!(host=localhost))"/>
  </osc:handler>
</osc:handlers>

<osc:requestHandlers names="auditor"/>
<osc:responseHandlers names="auditor"/>
<osc:errorHandlers names="auditor"/>
```

Using Oracle9iAS SOAP Audit Logging

The Oracle SOAP audit logging feature monitors and records SOAP usage. Audit logging maintains records for postmortem analysis and accountability. The SOAP audit logging feature complements the audit logging capabilities available with the OC4J server which hosts the SOAP Request Handler Servlet (SOAP server).

Oracle SOAP stores audit trails as XML documents. Using XML documents, Oracle SOAP creates portable audit trails and enables the transformation of complete audit trails or individual audit records to different formats.

By default, Oracle SOAP audit logging uses an audit logger class that implements the `Handler` interface (part of the `oracle.soap.server` package). The audit logger class is invoked conditionally to monitor events including service requests, service responses, and errors.

This section covers the following topics:

- [Audit Logging Information](#)
- [Auditable Events](#)
- [Configuring the Audit Logger](#)

Audit Logging Information

Table A-2 lists the audit logging elements available for each audit log record. Individual audit log records may not contain all these elements. In the log file, each audit log record is stored as a `SoapAuditRecord` element.

Table A-2 Auditable Audit Record Elements

Audit Record Element	Description
HostName	Specifies the hostname of the client that sent the request.
IpAddress	Specifies the IP address of the client that sent the request.
Method	Specifies the method name for the SOAP request.
Request Envelope	Provides the complete SOAP request message.
Request Envelope Method	Name of the Method in the SOAP request envelope
Request Envelope URI	Specifies the URI of the service in the SOAP request envelope.
Response Envelope	Provides the complete SOAP response message.
ServiceURI	Specifies the service URI for the SOAP request.
SoapAuditRecord	Contains an individual record. The <code>chainType</code> attribute indicates if the record is generated as part of a request or a response.
TimeStamp	Specifies the system time when the SOAP audit record was generated.
User	Specifies the username associated with the request. Note, this element is only provided when a user context is associated with the service request or service response.

Audit Logging Output

The XML schema for the generated audit log is provided in the file `SoapAuditTrail.xsd` in the directory `$SOAP_HOME/schema` on UNIX or `%SOAP_HOME%\schema` on Windows. Refer to the schema file for complete details on the format of a generated audit log record.

Auditable Events

The audit logger class is invoked when an auditable event occurs and the SOAP Request Handler Servlet is configured to enable auditing for the event. Auditable events include a service request or a service response.

Audit Logging Filters

An audit logging filter can be specified in the SOAP configuration file to limit the set of auditable events that are recorded to the audit log. The SOAP server applies event filters to request and response events. [Table A-4](#) shows the filter attributes available to select with an event filter specification. When applied, filters limit the number of records generated in the audit log. For example, when a filter is specified for a particular host, only the auditable events generated for the specified host are saved to the audit log.

The syntax for defining auditable events with a filter is derived from RFC 2254. [Table A-3](#) shows the filter syntax, and [Example A-3](#) provides several examples.

See Also:

- ["Configuring the Audit Logger"](#) on page A-18
- <ftp://ftp.isi.edu/in-notes/rfc2254.txt> on RFC 2254

Table A-3 *Audit Trail Events Filter Attributes*

Audit Event Filter Attributes	Description
Host	<p>Specifies the hostname of the host for the service request or response. If this attribute is not specified in a filter, the hostname of the client is not used in filtering audit log records.</p> <p>Fully specify the hostname of the client or use wildcards ("*"). Wildcards embedded within the specified hostname are not supported the examples show valid and invalid uses of wildcards. If a wildcard is used then the wildcard must be the first character in the filter. Case is ignored for hostnames. Care should be used in setting this attribute. Depending on the DNS setup, the hostname returned could be fully qualified or nonqualified; for example, <code>explosives.acme.com</code> or <code>explosives</code>. For some IP addresses, the DNS may not be able to resolve the hostname.</p> <p>Legal values for a Host filter attribute include the following examples:</p> <p style="padding-left: 2em;"><code>explosives.acme.com</code>, <code>*.acme.com</code>, <code>*.com</code></p> <p>Illegal values for a Host filter attribute include the following examples:</p> <p style="padding-left: 2em;"><code>*</code>, <code>explosives.acme.*</code>, <code>explosives.*</code>, <code>ex*s.acme.com</code>, <code>*ives.acme.com</code></p>

Table A-3 (Cont.) Audit Trail Events Filter Attributes

Audit Event Filter Attributes	Description
<i>ip</i>	<p>Specifies the IP address of the client for the service request or response.</p> <p>The IP address of the client has to be either fully specified, using all four bytes, in the dot separate decimal form, or specified using wildcards ("*"). Embedded wildcards are not supported. If a wildcard is used then the wildcard must be the last character in the filter.</p> <p>If this attribute is not used in a filter then the IP address of the client is not used in filtering.</p> <p>Legal values for an <i>ip</i> filter attribute include the following examples:</p> <p style="padding-left: 40px;">138.2.142.154, 138.2.142.*, 138.2.*, 138.*</p> <p>Illegal values for an <i>ip</i> filter attribute include the following examples:</p> <p style="padding-left: 40px;">*, 138.2.*.154, *.2, 138.*.152, 138.2.142, 138.2, 138</p>
<i>urn</i>	Specifies the service URN. Wildcards are not supported for this attribute.
<i>username</i>	<p>Specifies the transport level username associated with the client.</p> <p>Wildcards are not supported in a username filter attribute.</p>

Table A-4 Audit Log Filter Syntax

Filter Value	Description
<i>attr</i>	1*(any US-ASCII char except "*", "(", ")", "&", " ", "!", "*", "=")
<i>equal</i>	"="
<i>filter</i>	"("filtercomp")"
	Whitespaces between "("filtercomp and ")" are not allowed.
<i>filtercomp</i>	<i>and</i> <i>or</i> <i>not</i> <i>item</i> <i>and</i> = "&" filterlist <i>or</i> = " " filterlist <i>not</i> = "!" filter
<i>filterlist</i>	2*2 filter
<i>filtertype</i>	<i>equal</i>
<i>item</i>	<i>attr filtertype value</i> Whitespaces between <i>attr</i> , <i>filtertype</i> and <i>value</i> are not allowed.

Table A-4 (Cont.) Audit Log Filter Syntax

Filter Value	Description
<i>value</i>	1*(any octet except ASCII representation of ") - 0x29). The character "*" has a special meaning. The "*" character is referred to as a wildcard and matches anything.

Example A-3 Sample Audit Log Filters

```
(ip=138.2.142.154)
(! (host=localhost))
(! (host=* .acme.com))
(&(host=* .acme.com)(username=daffyy))
(&(ip=138.2.142.*) (| (urn=urn:www-oracle-com:AddressBook)(username=daffyy))
```

Configuring the Audit Logger

Configure the default SOAP audit logger supplied with Oracle9i Application Server by setting parameters in the SOAP configuration file, `soap.xml`. To enable the default audit logger and turn on audit logging, do the following in the configuration file.

- Define the name and options for the audit log handler. The default SOAP audit logger is defined in the class `oracle.soap.handlers.audit.AuditLogger`. The default audit logger supports several options that you specify in the configuration file. [Table A-5](#) shows the available audit logger options.
- Add the name for the audit logger handler to the `requestHandler`, `responseHandler`, or `errorHandler` chain (or to all of the handler chains).

[Example A-4](#) shows a sample segment from a SOAP configuration file including the audit logging configuration options. [Example A-4](#) shows configuration options set to use all options. However, this configuration would produce an extremely large audit log, and is not recommended.

Note: When you audit errors using the audit logger, depending on when the error occurs in the request-chain or the response-chain, it is possible that the request or response message may not be included in the audit log record, even with `includeRequest` or `includeResponse` enabled.

Example A-4 Audit Logging Configuration

```
<osc:handlers>
  <osc:handler name="auditor"
    class="oracle.soap.handlers.audit.AuditLogger">
    <osc:option name="auditLogDirectory"
      value="/private1/oracle/app/product/tv02/soap/webapps/soap/WEB-INF"/>
    <osc:option name="filter" value="(!(host=localhost))"/>
    <osc:option name="includeRequest" value="true"/>
    <osc:option name="includeResponse" value="true"/>
  </osc:handler>
</osc:handlers>
<osc:requestHandlers names="auditor"/>
<osc:responseHandlers names="auditor"/>
<osc:errorHandlers names="auditor"/>
```

Table A-5 Audit Logger Configuration Options

Option	Description
<code>auditLogDirectory</code>	Specifies the directory where the audit log file is saved. The <code>auditLogDirectory</code> option is required. The name of the generated audit log file is <code>OracleSoapAuditLog.timestamp</code> , where <i>timestamp</i> is the date and time the file is first generated. Valid values: any string that is a valid directory
<code>filter</code>	Specifies the audit event filter. This option is optional. If a <code>filter</code> is not specified SOAP server logs every event. Valid values: any valid filter.
<code>includeRequest</code>	Specifies that the audit record include the request message for the event that generated the audit log record. Valid values: <code>true</code> , <code>false</code> Any value other than <code>true</code> or <code>false</code> is treated as an error. Default Value: <code>false</code>

Table A-5 (Cont.) Audit Logger Configuration Options

Option	Description
<code>includeResponse</code>	<p>Specifies that the audit record include the response message for the event that generated the audit log record.</p> <p>Valid values: <code>true</code>, <code>false</code></p> <p>Any value other than <code>true</code> or <code>false</code> is treated as an error.</p> <p>Default Value: <code>false</code></p>

See Also: ["Using Oracle9iAS SOAP Handlers"](#) on page A-13

Using Oracle9iAS SOAP Pluggable Configuration Managers

Oracle9iAS SOAP supports pluggable configuration managers similar to those supported in Apache SOAP 2.2. Since Oracle9iAS SOAP supports provider deployment descriptors separate from service deployment descriptors, the interface details using Oracle9iAS SOAP are slightly different from Apache SOAP 2.2. In Oracle9iAS SOAP, configuration managers are configured separately for the provider manager and the service manager. All configuration managers must implement the `oracle.soap.server.ConfigManager` interface.

To simplify development, when you write a configuration manager implementation, you may use the abstract class that is provided with Oracle9iAS SOAP (`oracle.soap.server.impl.BaseConfigManager`). This abstract class provides a standard implementation for most of the `ConfigManager` interface with two abstract methods that read and write the persistent store.

[Example A-5](#) shows a sample implementation of a provider configuration manager.

Example A-5 Sample Provider Configuration Manager Implementation.

```

public class MyProviderConfigManager extends BaseConfigManager
{
    public void setOptions(Properties options)
        throws SOAPException
    {
        // handle implementation specific options
    }

    public void readRegistry()
        throws SOAPException
    {
        // read the deployed providers from persistent store
    }

    public void writeRegistry()
        throws SOAPException
    {
        // write the deployed providers to persistent store
    }
}

```

The `setOptions` method is passed the options specified in any `<option>` elements specified in the `<configManager>` element. Synchronization of reading/writing the registry is the responsibility of the specific configuration manager implementation.

Working With Oracle9iAS SOAP Transport Security

Oracle9i Application Server uses the security capabilities of the underlying transport that sends SOAP messages. Oracle9i Application Server supports the HTTP and HTTPS protocols for sending SOAP messages. HTTP and HTTPS support the following security features:

- HTTP proxies
- HTTP authentication (basic RFC 2617)
- Proxy authentication (basic RFC 2617)

Oracle9iAS SOAP Client transport uses the modified, to support Oracle Wallet Manager, `HTTPClient` package. Oracle9iAS SOAP transport defines several properties to support these features. [Table A-6](#) lists the client-side security properties that Oracle9i Application Server supports.

In an Oracle9iAS SOAP Client application, you can set the security properties shown in [Table A-6](#) as system properties by using the `-D` flag at the Java command line. You can also set security properties in the Java program by adding these properties to the system properties (use `System.setProperty()` to add properties).

[Example A-6](#) shows how Oracle9i Application Server supports overriding the values specified for system properties using Oracle9i Application Server transport specific APIs. The `setProperty()` method in the class `OracleSOAPHTTPConnection` contains set properties specifically for the HTTP connection (this class is in the package `oracle.soap.transport.http`).

Example A-6 Setting Security Properties for OracleSOAPHTTPConnection

```
org.apache.soap.rpc.Call call = new org.apache.soap.rpc.Call();
oracle.soap.transport.http.OracleSOAPHTTPConnection conn =
(oracle.soap.transport.http.OracleSOAPHTTPConnection) call.getSOAPTransport();
java.util.Properties prop = new java.util.Properties();
// Use client code to set name-value pairs of properties in prop
.
.
.
conn.setProperty(prop);
```

Note: The property `java.protocol.handler.pkgs` must be set as a system property.

Table A-6 SOAP HTTP Transport Security Properties

Property	Description
<code>http.authType</code>	<p>Specifies the HTTP authentication type. The case of the value specified is ignored.</p> <p>Valid values: <code>basic</code>, <code>digest</code></p> <p>The value <code>basic</code> specifies HTTP basic authentication.</p> <p>Specifying any value other than <code>basic</code> or <code>digest</code> is the same as not setting the property.</p>
<code>http.password</code>	Specifies the HTTP authentication password.
<code>http.proxyAuthType</code>	<p>Specifies the proxy authentication type. The case of the value specified is ignored.</p> <p>Valid values: <code>basic</code>, <code>digest</code></p> <p>Specifying any value other than <code>basic</code> or <code>digest</code> is the same as not setting the property.</p>
<code>http.proxyHost</code>	Specifies the hostname or IP address of the proxy host.
<code>http.proxyPassword</code>	Specifies the HTTP proxy authentication password.
<code>http.proxyPort</code>	<p>Specifies the proxy port. The specified value must be an integer. This property is only used when <code>http.proxyHost</code> is defined; otherwise this value is ignored.</p> <p>Default value: 80</p>
<code>http.proxyRealm</code>	Specifies the realm for which the proxy authentication username/password is specified.
<code>http.proxyUsername</code>	Specifies the HTTP proxy authentication username.
<code>http.realm</code>	Specifies the realm for which the HTTP authentication username/password is specified.
<code>http.username</code>	Specifies the HTTP authentication username.
<code>java.protocol.handler.pkgs</code>	<p>Specifies a list of package prefixes for <code>java.net.URLStreamHandlerFactory</code>. The prefixes should be separated by " " vertical bar characters.</p> <p>This value should contain: <code>HTTPClient</code></p> <p>This value is required by the Java protocol handler framework; it is not defined by Oracle9i Application Server. This property must be set when using HTTPS. If this property is not set using HTTPS, a <code>java.net.MalformedURLException</code> is thrown.</p> <p>Note: This property must be set as a system property.</p> <p>For example, set this property as shown in either of the following:</p> <ul style="list-style-type: none"> ■ <code>java.protocol.handler.pkgs=HTTPClient</code> ■ <code>java.protocol.handler.pkgs=sun.net.www.protocol HTTPClient</code>

Table A-6 (Cont.) SOAP HTTP Transport Security Properties

Property	Description
<code>oracle.soap.transport.allowUserInteraction</code>	<p>Specifies the allows user interaction parameter. The case of the value specified is ignored. When this property is set to <code>true</code> and either of the following are true, the user is prompted for a username and password:</p> <ol style="list-style-type: none"> 1. If any of properties <code>http.authType</code>, <code>http.username</code>, or <code>http.password</code> is not set, and a 401 HTTP status is returned by the HTTP server. 2. If either of properties <code>http.proxyAuthType</code>, <code>http.proxyUsername</code>, or <code>http.proxyPassword</code> is not set and a 407 HTTP response is returned by the HTTP proxy. <p>Valid values: <code>true</code>, <code>false</code></p> <p>Specifying any value other than <code>true</code> is considered as <code>false</code>.</p>
<code>oracle.soap.transport.1022ContentType</code>	<p>Specifies the value for the Oracle9iAS Content-Type HTTP header. The value for this property supports Oracle SOAP servers running either Oracle 9iAS Release 1.0.2.2 or Release 9.0.x. This property provides interoperability between Oracle9iAS Release 9.0.2 Oracle SOAP clients and older server versions (as distributed with Oracle9iAS Release 1.0.2.2).</p> <p>Valid values: <code>true</code>, <code>false</code> (case is ignored)</p> <p>Setting the value to <code>true</code> specifies to use the Oracle9 iAS Release 1.0.2.2 content-type HTTP header values when the SOAP message is sent. In this case, the value is set to: <code>content-type: text/xml</code></p> <p>Setting the value to <code>false</code> specifies to use the iAS version 9.0.2 content-type header value when the SOAP message is sent. In this case, the value is set to: <code>content-type: text/xml; charset=utf-8</code></p> <p>The value <code>false</code> is the default value.</p> <p>Note: for SOAP messages with attachments, the content-type HTTP header is always set to the value: <code>multipart/related</code>.</p>
<code>oracle.ssl.ciphers</code>	<p>Specifies a list of: separated cipher suites that are enabled.</p> <p>Default value: The list of all cipher suites supported by Oracle SSL are supported.</p>

Table A-6 (Cont.) SOAP HTTP Transport Security Properties

Property	Description
<code>oracle.wallet.location</code>	<p>Specifies the location of an exported Oracle wallet or exported trustpoints.</p> <p>Note: The value used is not a URL but a file location, for example:</p> <p><code>/etc/ORACLE/Wallets/system1/exported_wallet</code> (on UNIX)</p> <p><code>d:\oracle\system1\exported_wallet</code> (on Windows)</p> <p>This property must be set when HTTPS is used with SSL authentication, server or mutual, as the transport.</p>
<code>oracle.wallet.password</code>	<p>Specifies the password of an exported wallet. Setting this property is required when HTTPS is used with client, mutual authentication as the transport.</p>

Apache Listener and Servlet Engine Configuration for SSL

When using Apache listener and `mod_ssl` (or `mod_oss`), the following directives must be set for the `soap servletlocation/directory`:

```
SSLOption +StdEnvVars +ExportCertData
```

This directive can be set conditionally, refer to `mod_ssl/mod_oss` documentation for details. By default this directive is disabled for performance reasons. If this directive is not set then the servlet engine does not have a way to access the SSL related data (such as the cipher suite, client cert etc).

Using JSSE with Oracle9iAS SOAP Client

This section describes how to use SSL with the Oracle9iAS SOAP Client side when the Oracle security infrastructure is not available. Availability of Oracle security infrastructure means the availability of Oracle client side libraries (including `$ORACLE_HOME/lib/*`, `$ORACLE_HOME/jlib/javax-ssl-1_2.jar`, and `$ORACLE_HOME/jlib/jssl-1_2.jar`).

Oracle9iAS SOAP uses the following class as the default transport class:

```
oracle.soap.transport.http.OracleSOAPHTTPConnection
```

This class uses a modified version of `HTTPClient` package . For information on `HTTPClient`, see the following site:

<http://www.innovation.ch/java/HTTPClient/>

This version of `HTTPClient` package is integrated with Oracle Java SSL and supports Oracle Wallet for HTTPS transport. If a SOAP client side does not have Oracle client side available, it is still possible to use HTTPS as a transport with Oracle9iAS SOAP Client side libraries.

To do this, follow these steps:

1. Use the following transport class:

```
class org.apache.soap.transport.http.SOAPHTTPConnection
```

If using RPC then call the following method by passing an instance of `org.apache.soap.transport.http.SOAPHTTPConnection` as an argument:

```
method org.apache.soap.rpc.Call#setSOAPTransport  
(org.apache.soap.transport.SOAPTransport)
```

For example:

```
org.apache.soap.rpc.Call myCallObj = new  
org.apache.soap.rpc.Call();  
myCallObj.setSOAPTransport(new  
org.apache.soap.transport.http.SOAPHTTPConnection());
```

If using messaging, then call the following method by passing an instance of `org.apache.soap.transport.http.SOAPHTTPConnection` as an argument:

```
org.apache.soap.messaging.Message#setSOAPTransport  
(org.apache.soap.transport.SOAPTransport)
```

For example:

```
org.apache.soap.messaging.Message myMsgObj = new  
org.apache.soap.messaging.Message();  
myMsgObj.setSOAPTransport(new  
org.apache.soap.transport.http.SOAPHTTPConnection());
```

2. Download Java Secure Socket Extension (JSSE) and configure JSSE according to the supplied instructions. JSSE is available at the following site:

`http://java.sun.com/products/jsse/`

- Make sure the files `jnet.jar`, `jcrt.jar` and `jsse.jar` are in the classpath or in the installed extensions directory (`$JRE_HOME/lib/ext`).
- Make sure that SunJSSE provider is correctly configured. This can be done either statically by editing the `$JRE_HOME/lib/security/java.security` file and adding the line:

```
security.provider.num=com.sun.net.ssl.internal.ssl.Provider
```

Where *num* is 1-based preference order or by dynamically by adding the provider at run time by adding the following line of code:

```
Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
```

Dynamic addition of security providers requires that appropriate permissions are set.

- Make sure the system property `java.protocol.handler.pkgs` is set to `com.sun.net.ssl.internal.www.protocol`
- If using proxy server, make sure that the following system properties are set is set to the correct proxy hostname and proxy port, respectively:

```
https.proxyHost
https.proxyPort
```

- If using SSL with server side authentication and the default `TrustManager`, ensure that the certificate signer of the server is one of the following files:

```
$JRE_HOME/lib/security/jssecacerts
```

or if `jssecacerts` does not exist:

```
$JRE_HOME/lib/security/cacerts
```

- To override the `KeyManager/TrustManager` keystore default locations, use the system properties:

```
javax.net.ssl.keystore
```

```
javax.net.ssl.keyStoreType  
javax.net.ssl.keyStorePassword  
javax.net.ssl.trustStore  
javax.net.ssl.trustStoreType  
javax.net.ssl.trustStorePassword
```

Please consult JSSE documentation for details. If using a specific third party JSSE implementation, please consult the appropriate documentation.

See Also: `HTTPClient` information at the site:

<http://www.innovation.ch/java/HTTPClient/>

Using Oracle9iAS SOAP Sample Services

The section lists the samples included with Oracle9iAS SOAP. The class files for all of the samples are in `samples.jar`.

To run any sample, you need to ensure that `samples.jar` is available on your servlet's CLASSPATH. Please refer to the README included with each sample for more information.

The Xmethods Sample

The clients in the `xmethods` sample represent the easiest way to get started with SOAP because they are clients that access existing services that are hosted on systems on the internet. Information on these services can be found at the site:

<http://www.xmethods.org>

This sample is in `SOAP_HOME/samples/xmethods`.

The AddressBook Sample

This sample has a service implemented in Java and several clients. This sample illustrates literal XML encoding. See `SOAP_HOME/samples/addressbook` for the sample source code. This directory also contains a script that illustrates how to run the sample addressbook clients using HTTPS as transport.

The StockQuote Sample

This sample has a service implemented in Java and one client. It is located in `SOAP_HOME/samples/stockquote`

The Company Sample

This sample has a service that is comprised of PL/SQL stored procedures and several clients. It is located in `$$SOAP_HOME/samples/sp/company`. Check the README file in this directory for details on how to setup, compile, and test this sample service.

The Provider Sample

This includes a template provider that can be used as a starting point for creating your own provider.

The AddressBook2 Sample

This sample demonstrates use of the Addressbook service with session scope. It shows how to maintain the same HTTP session across SOAP Calls. It contains an example of a SOAP client proxy generated from a WSDL service description file. It is located in `$$SOAP_HOME/samples/addressbook2`

The Messaging Sample

This sample is an example of a message-based SOAP service. It is located in `$$SOAP_HOME/samples/messaging`

The Mime Sample

This sample does SOAP with attachments using both RPC and message based services. It is located in `$$SOAP_HOME/samples/mime`.

Using the Oracle9iAS SOAP EJB Provider

This section compares the Oracle9iAS SOAP EJB providers with the Apache-SOAP 2.2 EJB providers.

Stateless Session EJB Provider

In Apache SOAP, the Stateless EJB provider, on receiving the SOAP request, performs a JNDI lookup on the home interface of the EJB. The Stateless EJB provider then invokes a create on the EJB's Home Interface in order to get a

reference to a stateless EJB. Then it uses this EJB reference to invoke the requested method.

Oracle9iAS SOAP uses the same mechanism to support Stateless Session EJBs as Apache SOAP.

Stateful Session EJB Provider in Apache SOAP

On receiving a first time SOAP request, the Apache SOAP Stateful Session EJB provider first locates the Home Interface through a JNDI lookup and using a subsequent create obtains an object reference to a Stateful Session EJB. The provider then invokes the requested method on the object reference.

In the next step the provider serializes the `EJBHandle` of the specified EJB reference and appends it to the targetURI with an "@" delimiter. The Stateful Session EJB provider then sends this modified target URI back to the requesting SOAP client. If the client wants to reuse the same EJB instance, it must retrieve this "modified" target URI for the service from the Response and set it in the next SOAP Call.

Upon receiving this request, the Stateful EJB provider extracts the stringified EJB reference and deserializes it into an `EJBHandle` from which it can obtain the EJB reference. It can then invoke the method on the specified EJB.

The drawback of the Apache SOAP implementation is that the client must be EJB aware and that it could not operate with other SOAP servers.

Oracle9iAS SOAP offers an alternative solution for Stateful Session EJBs that allows for client interoperability.

Stateful Session EJB Provider in Oracle9iAS SOAP

The Oracle9iAS SOAP Stateful Session EJB provider binds the EJB reference to the current session, if none is bound, otherwise, it merely retrieves the EJB reference from the session. In order for the client to access the same Stateful Session EJB, the client has to simply maintain its current session between successive Calls.

If at any point in a session, the SOAP client invokes a create on the EJB's Home Interface, the provider binds the EJB reference from the create to the session, to be used for other call requests within the session.

Entity EJB Provider in Oracle9iAS SOAP

In order for a SOAP client to execute a business method on an entity EJB, it first needs to either "create" a new EJB upon which to run the method or *find* an already

existing EJB which suit some criteria. Access to an entity EJB occurs within a session. At the start of the session the SOAP client must invoke a "create" or "find" (in order to specify the bean object interest). While maintaining the same session, all other business methods are directed to that EJB. A subsequent "find" or "create" within the same or different session directs business method execution requests to the newly "created" (or "found") EJB.

Another issue is that EJB specification provides that some "find" methods can return either a Collection of EJB refs or single EJB ref.

The Oracle solution for Entity EJBs embraces the following solution for this problem:

It disallows find methods that return "Collections". This allows for the provider to uniquely specify an Entity EJB to target subsequent business method requests.

Deployment and Use of the Oracle9iAS SOAP EJB Provider

To install an EJB provider and deploy Web Services to the provider under OC4J, where the application server hosts both the SOAP servlet and the deployed EJB's, follow these steps:

1. Deploy an EJB provider to SOAP using a provider descriptor.

The provider descriptor specifies the following:

- EJB access credentials by the middle tier
- JNDI context factory class
- JNDI context factory URL
- Provider class name
- Provider id

2. Create the EJB Web Service:

- Define the associated EJB classes and package the EJB into an EAR file as defined by J2EE spec.
- Define the service descriptor which specifies following details of the EJB Web Service:
 - * JNDI Location
 - * Home interface class name

- * Application Deployment Name of this EJB Web Service in OC4J
 - * The provider id to which this service is to be associated
3. Deploy ear file in OC4J. Modify the OC4J specific EJB descriptor to correct the JNDI location for the EJB (as described in sample README).

Current Known EJB Provider Limitations

All service methods can only take primitive Java types as arguments to the methods. User-defined Java types are currently not supported.

Using PL/SQL Stored Procedures With the SP Provider

The Oracle9iAS SOAP Stored Procedure (SP) Provider supports exposing PL/SQL stored procedures or functions as SOAP services. The Oracle9i Database Server allows procedures implemented in other languages, including Java and C/C++, to be exposed using PL/SQL; these stored procedures are exposed as SOAP services through PL/SQL interfaces.

The SP Provider framework works by translating PL/SQL procedures into Java wrapper classes, and then exporting the generating Java classes as SOAP Java services.

SP Provider Supported Functionality

The SP Provider supports the following:

- PL/SQL stored procedures. both procedures and functions (this document uses procedure to refer to both)
- IN parameter modes
- Packaged procedures only (top-level procedures must be wrapped in a package before they can be exported)
- Overloaded procedures (however, if two different PL/SQL types map to the same Java type during translating, there may be errors during the export of the PL/SQL package; these errors may be fixed by avoiding the overloading, or else by writing a new dummy package which does not contain the offending overloaded procedures)
- Simple types
- (user-defined) object types

SP Provider Unsupported Functionality

The SP provider does not support the following:

- The SP Provider framework uses JPublisher to translate from PL/SQL to Java; hence, it inherits all of the restrictions of JPublisher.
- **BOOLEAN** Due to a restriction in the OCI layer, the JDBC drivers do not support the passing of **BOOLEAN** parameters to PL/SQL stored procedures. Please refer to the *JDBC Developer's Guide and Reference* for a workaround.
- **NCHAR** and related types
- JPublisher does not support internationalization.
- **LOB** types. JPublisher does not provide comprehensive support for **LOB** types; if your PL/SQL procedures use **LOB** types as input/output types, the translation may not work in all cases. If you see an error, the offending procedures will have to be rewritten before the package can be exported as a SOAP service.

SP Provider Supported Simple PL/SQL Types

The SOAP SP provider supports the following simple types. **NULL** values are supported for all of the simple types listed, except **NATURALN** and **POSITIVEN**.

The JPublisher documentation provides full details on the mappings of these types.

- **VARCHAR2** (**STRING**, **VARCHAR**)
- **LONG**
- **CHAR** (**CHARACTER**)
- **NUMBER** (**DEC**, **DECIMAL**, **DOUBLE PRECISION**, **FLOAT**, **INTEGER**, **INT**, **NUMERIC**, **REAL**,
- **SMALLINT**)
- **PLS_INTEGER**
- **BINARY_INTEGER** (**NATURAL**, **NATURALN**, **POSITIVE**, **POSITIVEN**)

Due to a bug in JPublisher, many integer numeric types are translated into `java.math.BigDecimal` instead of the Java scalar types---the workaround for this bug is to temporarily use `java.math.BigDecimal` as the argument and return types.

The sample SP service has examples of the use of `BigDecimal`.

Using Object Types

JPublisher supports the use of user-defined object types. The SP Provider framework generates `oracle.sql.CustomDatum` style classes since these allow automatic serialization using the default `BeanSerializer` in SOAP.

Refer to the company sample for an example of using object types.

Deploying a Stored Procedure Provider

[Example A-7](#) shows a sample provider deployment descriptor for a stored procedure. You may use any unique id for the provider name (the example uses "company-provider").

The attributes `user`, `password`, and `url` are used to create the URL to connect to the database, and they are all required. The number of connections for a service, handled by this provider, is set using `connections_per_service`; this is optional and defaults to 10.

Deploy the sample provider descriptor shown in [Example A-7](#), appropriately edited for the local configuration, using the provider manager.

Example A-7 Sample SP Provider Deployment Descriptor

```
<isd:provider xmlns:isd="http://xmlns.oracle.com/soap/2001/04/deploy/provider"
  id="company-provider"
  class="oracle.soap.providers.sp.SpProvider">
  <!-- edit the following option "values" as appropriate -->
  <isd:option key="user" value="YOUR-USER-NAME" />
  <isd:option key="password" value="YOUR-PASSWORD" />
  <isd:option key="url" value="jdbc:oracle:thin:@YOUR-HOST:YOUR-PORT:YOUR-SID"
/>
  <isd:option key="connections_per_service" value="3" />
</isd:provider>
```

Translating PL/SQL Stored Procedures into Java

The shell script `$SOAP_HOME/bin/sp2jar.sh` translates a PL/SQL package and all its contained procedures/functions into a Java class with equivalent methods. If the package uses any user-defined types, these types are also translated into equivalent Java classes.

The `README` file in the samples directory has an example of the usage of the `sp2jar.sh` command to translate the company example into a jar file of compiled

Java classes. The README also describes how to load the PL/SQL packages into the database.

Let us assume for the rest of the document that a PL/SQL package company has been installed on a database, and it has been exported into a set of compiled Java classes available in the jar file `company.jar`.

The generated `company.jar` should be made available in the CLASSPATH of the SOAP servlet, just as for other Java services.

Deploying a Stored Procedure Service

[Example A-8](#) shows a sample service deployment descriptor for a stored procedure. Notice that the `id` attribute in the provider element identifies the provider under which this service is deployed.

The service descriptor looks exactly like that for a Java service, since the SP Provider framework translated PL/SQL procedures into Java class methods. All of the information specific to PL/SQL are part of the provider descriptor---the service itself looks like a Java service.

If the procedures use object types, it is necessary to define a type mapping for each object type. The XML type name must be identical to the SQL type name and must be in UPPER CASE (see EMPLOYEE and ADDRESS below). The `javaType` attribute identifies the `oracle.sql.CustomDatum` type that was generated by JPublisher.

The default BeanSerializer can be used to serialize/deserialize the types.

The generated method names are in lower-case since this is the default setting of JPublisher.

Deploy the sample service descriptor shown in [Example A-8](#) using the service manager.

Example A-8 Sample Stored Procedure Service Deployment Descriptor

```
<isd:service xmlns:isd="http://xmlns.oracle.com/soap/2001/04/deploy/service"
  id="urn:www-oracle-com:company"
  type="rpc" >

  <isd:provider
    id="company-provider"
    methods="addemp getemp getaddress getempinfo changesalary removeemp"
    scope="Application" >
    <isd:java class="samples.sp.company.Company"/>
  </isd:provider>
```

```
<isd:mappings>
  <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:x="urn:company-sample" qname="x:EMPLOYEE"
    javaType="samples.sp.company.Employee"
    java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
    xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:x="urn:company-sample" qname="x:ADDRESS"
    javaType="samples.sp.company.Address"
    java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
    xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
</isd:mappings>

<isd:faultListener class="org.apache.soap.server.DOMFaultListener"/>

</isd:service>
```

Invoking a SOAP Service that is a Stored Procedure

SOAP services that are PL/SQL stored procedures are invoked in exactly the same manner as any other SOAP service. The `company.jar` file created during the translating/deployment of a PL/SQL package is also needed on the client-side to compile application programs that invoke the SOAP service (this jar file is needed only if the stored procedures have input/output types that are user-defined types; if the procedures use only builtin-types, the generated jar file is not needed on the client).

The `README` file in the `company` samples directory has instructions on how to compile and test the sample client.

SOAP Troubleshooting and Limitations

This section lists several techniques for troubleshooting Oracle9iAS Web Services, including:

- [Tunneling Using the `TcpTunnelGui` Command](#)
- [Setting Configuration Options for Debugging](#)
- [Using DMS to Display Runtime Information](#)
- [SOAP Limitations for Java Type Precedence with Overloaded Methods](#)

Tunneling Using the TcpTunnelGui Command

SOAP provides the `TcpTunnelGui` command to display messages sent between a SOAP client and a SOAP server. `TcpTunnelGui` listens on a TCP port, which is different than the SOAP server, and then forwards requests to the SOAP server.

Invoke `TcpTunnelGui` as follows:

```
java org.apache.soap.util.net.TcpTunnelGui TUNNEL-PORT SOAP-HOST SOAP-PORT
```

[Table A-7](#) lists the command line options for `TcpTunnelGui`.

Table A-7 *TcpTunnelGui* Command Arguments

Argument	Description
<i>TUNNEL-PORT</i>	The port that <code>TcpTunnelGui</code> listens to on the same host as the client
<i>SOAP-HOST</i>	The host of the SOAP server
<i>SOAP-PORT</i>	The port of the SOAP server

For example, suppose the SOAP server is running as follows,

```
http://system1:8080/soap/servlet/soaprouter
```

You would then invoke `TcpTunnelGui` on port 8082 with this command:

```
java org.apache.soap.util.net.TcpTunnelGui 8082 system1 8080
```

To test a client and view the SOAP traffic, you would use the following SOAP URL in the client program:

```
http://system1:8082/soap/servlet/soaprouter
```

Setting Configuration Options for Debugging

To add debugging information to the SOAP Request Handler Servlet log files, change the value of the severity option for the value `debug` in the file `soap.xml`. This file is placed in the directory `$SOAP_HOME/webapps/soap/WEB-INF` on UNIX or in `%SOAP_HOME%\webapps\soap\WEB-INF` on Windows.

For example, the following `soap.xml` segment shows the value to set for `severity` to enable debugging:

```
<!-- severity can be: error, status, or debug -->
<osc:logger class="oracle.soap.server.impl.ServletLogger">
  <osc:option name="severity" value="debug" />
</osc:logger>
```

After stopping and restarting the SOAP Request Handler Servlet, you can view debug information in the file `x.log`. The file is in the directory `$ORACLE_HOME/Apache/logs` on UNIX or in `%ORACLE_HOME%\Apache\x\logs` on Windows.

Using DMS to Display Runtime Information

Oracle9iAS Web Services is instrumented with DMS to gather information on the execution of the SOAP Request Handler Servlet, the Java Provider, and on individual services.

DMS information includes execution intervals from start to stop for the following:

- Total time spent in SOAP request and response (includes time in providers and services)
- Total time spent in the Java Provider (includes time in services)
- Total time executing services (`soap/java-provider/service-URI`)

To view the DMS information, go to the following site:

`http://hostname:port/soap/servlet/Spy`

SOAP Limitations for Java Type Precedence with Overloaded Methods

Oracle9iAS SOAP supports Java inbuilt (primitive) types, wrapper types, one dimensional arrays of inbuilt types, and one dimensional arrays of wrapper types as parameters for SOAP RPC.

An inbuilt type parameter always takes precedence to a wrapper type parameter when the Java provider searches for an overloaded method. When there isn't a clear winner, for an overloaded method, a fault with appropriate message is returned.

For example:

A java class containing `aMethod(int)` hides `aMethod(Integer)` in the same class.

A java class containing `aMethod(int[])` hides `aMethod(Integer[])` in the same class.

A java class, when deployed as a SOAP RPC service returns a fault when a client invokes `aMethod()` containing the signatures, `aMethod(int, Float)` and `aMethod(Integer, float)`. In this case, there is no clear winner for resolving the precedence of the overloaded `aMethod()`.

Oracle9iAS SOAP Differences From Apache SOAP

This section covers differences between Apache Soap and Oracle9iAS SOAP.

Service Installation Differences

Additional instructions are provided for installing services when Oracle9iAS SOAP is used in conjunction with OC4J.

Optional Provider Enhancements

Oracle9iAS SOAP supports both the Apache Provider interface, defined in `org.apache.soap.util.Provider`, and an enhanced provider interface, defined in `oracle.soap.server.Provider`.

The native Apache provider includes only two methods, `locate()` and `invoke()`. The Oracle Provider interface combines the `locate` and `invoke` methods, so that the provider does not have to store input parameters between the `locate()` and `invoke()` calls. Additionally, the Oracle Provider interface has `init()` and `destroy()` methods, which the SOAP servlet calls only once when the provider is instantiated. This allows providers to perform one time initialization such as opening a database or network connection, and to perform one time clean up activities.

When using the Apache provider interface, a single deployment descriptor supplies both service and provider properties. When using the Oracle Provider interface, these properties are separated between a service descriptor file and a provider descriptor file. This allows common provider properties to be shared among services. When a provider property changes, only a single descriptor file must be changed. Please see the Deployment section of this document for more information.

Oracle Transport libraries

Oracle transport libraries are included for use with SOAP clients. Use of these libraries enables use of the Oracle Wallet Manager for keeping certificates securely, and use of the HttpClient libraries for HTTP connection management. The HttpClient libraries fix a security problem in the native Apache code which incorrectly returns cookies to servers other than the originating server.

Modifications to Apache EJB Provider

The Apache EJB provider has been modified to work with the OC4J EJB container. In addition, the client interface to services provided by stateful and entity EJB's has been improved. The EJB handle is contained in the HttpSession association with the connection rather than being concatenated to the returned URL. Since the HttpSession cookie is handled transparently by the SOAP client, no special coding is required in the client.

Stored Procedure Provider

A special provider has been added which allows services to be written using PL/SQL Stored Procedures or Functions.

Utility Enhancements

The `wsdl2java` and `java2wsdl` scripts simplify building client side code from WSDL descriptions and for generating WSDL descriptions of Java services.

Modifications to Sample Code

The Apache samples have been modified to work with Oracle9iAS SOAP and OC4J. The `com`, `calculator`, `webllogic_ejb` samples have been omitted. New samples illustrating use of Oracle Stored Procedures and OC4J EJB's as Web Services have been added.

Handling the mustUnderstand Attribute in the SOAP Header

This section describes the check that is performed for the `mustUnderstand` attribute within the header blocks of the SOAP envelope, and describes the difference between the Apache SOAP and the Oracle SOAP processing of this attribute.

Setting the mustUnderstand Check

The check for the `mustUnderstand` attribute is enabled in the deployment descriptor of the service by setting the `checkMustUnderstands` flag. If this flag is set to `true`, the check for the `mustUnderstand` attribute within each header block is performed. If the `checkMustUnderstands` flag is set to `false`, the check for the `mustUnderstand` attribute is not performed. The default value of the `checkMustUnderstands` flag is `true`.

How the mustUnderstand Check Works

If the `checkMustUnderstands` flag is set to `true`, then a check is made on all header entries of the envelope after the global request handlers have finished processing and before handing the envelope to the appropriate service. At this point, if any header entries contain a `mustUnderstand` attribute that is set to `true` or to "1", then an exception is thrown. Note, the global handler(s) can be used to process one or more header blocks that have the `mustUnderstand` attribute set to `true`.

If the `checkMustUnderstands` flag is set to `false`, then header entries of the envelope are not checked to see if any entries contain a `mustUnderstand` attribute that is set to `true` or to "1". It is then understood that it is up to the service implementation to make sure that this check is done before processing the body of the envelope.

Differences Between Apache SOAP and Oracle SOAP for mustUnderstand

The differences between Apache SOAP and Oracle SOAP with respect to the handling of the `mustUnderstand` attribute are the following:

1. In the Apache service deployment descriptor and the Oracle Service deployment descriptor, you may include the `checkMustUnderstands` attribute. In Apache, the default value of the `checkMustUnderstands` attribute is `false`, in Oracle SOAP the default value of this attribute is `true`.
2. In Apache SOAP, if the service deployment descriptor contains `checkMustUnderstands='true'` and a message with `mustUnderstand='1'` or `mustUnderstand="true"` arrives at the server then a fault is sent back with the fault code value of:

```
mustUnderstand
```

This fault code is not namespace qualified and is incorrect.

In Oracle SOAP the fault code that is sent back is namespace qualified and is defined by SOAP 1.1:

SOAP-ENV:MustUnderstand

3. In Apache SOAP, the `mustUnderstand` attribute has to be handled by the service implementation. In Oracle SOAP, the `mustUnderstand` attribute can be either handled in the SOAP handlers or in the service implementation. This is very useful for processing headers (with `mustUnderstand` set to '1') which have a 'global' use. Examples of such headers/functionality are encryption, `digest`, authentication, logging etc.

Glossary

Dynamic Web Service Client

When you want to use Web Services, you can develop a **dynamic Web Service client**. With A dynamic client the client performs a lookup to find the Web Service's location in a UDDI registry before accessing the service.

SOAP

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.

See Also: <http://www.w3.org/TR/SOAP/> for information on Simple Object Access Protocol (SOAP) 1.1 specification

Static Web Service Client

When you want to use Web Services, you can develop a **static client**. A static client knows where a Web Service is located without looking up the service in a UDDI registry.

Stored Procedure Web Service

Oracle9iAS Web Services implemented as stateless PL/SQL Stored Procedures or Functions are called **Stored Procedure Web Services**. Stored Procedure Web Services enable you to export, as services running under Oracle9iAS Web Services, PL/SQL procedures and functions that run on an Oracle database server.

UDDI

Universal Description, Discovery, and Integration (UDDI) is a specification for an online electronic registry that serves as electronic *Yellow Pages*, providing an information structure where various business entities register themselves and the services they offer through their WSDL definitions.

See Also: <http://www.uddi.org> for information on Universal Description, Discovery and Integration specifications.

Web Service

A Web Service is a discrete business process that does the following:

- Exposes and describes itself – A Web Service defines its functionality and attributes so that other applications can understand it. A Web Service makes this functionality available to other applications.
- Allows other services to locate it on the web – A Web Service can be registered in an electronic *Yellow Pages*, so that applications can easily 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.
- Returns a response – When a Web Service is invoked, the results are passed back to the requesting application over the same Internet standard protocol that is used to invoke the service.

Web Services Description Language (WSDL)

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 the description available over the Internet.

See Also: <http://www.w3.org/TR/wsdl> for information on the Web Services Description Language (WSDL) format.

Index

C

- class-name configuration tag, 3-11, 6-12
- client-side proxies
 - using, 8-9
- client-side proxies for Web Services
 - generating, 8-8
- Client-side proxy
 - getting directly, 8-5
- connection-factory-resource-ref configuration tag, 7-13
- consuming Web Services
 - advanced section
 - editing generated XSLT stylesheet, 11-31
 - modifying environment options in generated ejb-jar.xml file, 11-32
 - sample uses
 - handling an XML or HTML stream accessed through a custom form, 11-14
 - handling an XML or HTML stream accessed through a form, 11-13
 - handling an XML or HTML stream accessed through static URL, 11-3
- SOAP-based Web services using WSDL
 - document, 11-33
 - running the demonstration, 11-41
 - using configuration file, 11-35
 - using wsdl2ejb utility command-line options, 11-33
- using Web Service HTML/XML Stream Processing Wizard, 11-2
- XML or HTML streams, 11-2
- context configuration tag, 3-10

D

- database-JNDI-name, 5-6
- database-JNDI-name configuration tag, 5-6
- data-sources.xml configuration file, 5-11
- db-pkg-name configuration tag, 5-7
- db-url configuration tag, 5-7
- description configuration tag, 3-10
- destination-path configuration tag, 3-10
- discovering Web Services, 10-5
- display-name configuration tag, 3-10
- document style interface, 6-7
- dynamic invocation of Web Services, 11-52
 - dynamic invocation API, 11-52
 - WebServiceProxy client, 11-56

E

- ejb-name configuration tag, 4-10
- ejb-resource configuration tag, 3-11, 4-10
- EJBs sample code, 4-2
- Element
 - arrays of, 3-7, 4-6, 6-4
 - null values, 6-3

G

- generating client proxies, A-8
- generating WSDL documents, A-9
- getting client-side proxies for Web Services, 8-3
- getting WSDL descriptions for Web Services, 8-3

H

http.authType property, 8-12
http.password property, 8-12
http.proxyAuthType property, 8-12
http.proxyHost property, 8-12
http.proxyPassword property, 8-12
http.proxyPort property, 8-12
http.proxyRealm property, 8-12
http.proxyUsername property, 8-12
http.realm property, 8-12
http.username property, 8-12

I

interface-name configuration tag, 3-11, 6-12

J

jar-generation configuration tag, 5-6
Java Beans, 3-7, 4-6
java class interface, 3-5
java2wsdl script, A-9
java.protocol.handler.pkgs property, 8-13
java-resource configuration tag, 3-12, 6-12
jms-delivery-mode configuration tag, 7-13
jms-doc-service configuration tag, 7-13
jms-expiration configuration tag, 7-13
jms-message-type configuration tag, 7-14
jms-priority configuration tag, 7-14

L

locating Web Services, 8-2

M

message-style configuration tag, 3-12, 6-12
method-name configuration tag, 5-7

O

operation configuration tag, 7-14
option name="force" configuration tag, 9-6
option name="httpServerURL" configuration tag, 9-6

option name="include-source" configuration tag, 8-9
option name="packageIt" configuration tag, 9-6
option name="source-path" configuration tag, 3-10
option name="wsdl-location" configuration tag, 8-9
option package-name configuration tag, 8-9
Oracle SOAP, A-23
audit logger
 configuring, A-18
 filter, A-15
 HostName, A-15
 IpAddress, A-15
 Method element, A-15
 schema, A-15
 ServiceURI element, A-15
 TimeStamp element, A-15
 User element, A-15
auditLogDirectory option, A-19
client API
 security features, A-21
configuration
 handlers, A-13
 soap.xml, A-3
debugging
 setting values in soap.xml, A-37
deploying services, A-7
deployment descriptor, A-10
error handlers, A-13
errorHandlers deployment parameter, A-3
faultListeners deployment parameter, A-3
filter option, A-19
handlers
 deployment parameter, A-4
 error, A-13
 request, A-13
 response, A-13
HostName element, A-15
HTTP transport properties
 http.authType property, A-23
 http.password property, A-23
 http.proxyAuthType property, A-23
 http.proxyHost property, A-23
 http.proxyPassword property, A-23
 http.proxyPort property, A-23

- http.proxyRealm property, A-23
- http.proxyUsername property, A-23
- http.realm property, A-23
- http.username property, A-23
- java.protocol.handler.pkgs property, A-23
- oracle.wallet.location property, A-25
- oracle.soap.transport.allowUserInteraction property, A-24
- oracle.wallet.password property, A-25
- http.authType property, A-23
- http.password property, A-23
- http.proxyAuthType property, A-23
- http.proxyHost property, A-23
- http.proxyPassword property, A-23
- http.proxyPort property, A-23
- http.proxyRealm property, A-23
- http.proxyUsername property, A-23
- http.realm property, A-23
- http.username property, A-23
- includeRequest option, A-19
- includeResponse option, A-20
- IpAddress element, A-15
- java.protocol.handler.pkgs property, A-23
- listing services, A-7
- logger
 - setting values in soap.xml, A-37
- logger deployment parameter, A-4
- Method element, A-15
- oracle.wallet.location property, A-25
- oracle.soap.transport.allowUserInteraction property, A-24
- oracle.soap.transport.1022ContentType property, A-24
- oracle.ssl.ciphers property, A-24
- oracle.wallet.password property, A-25
- providerManager deployment parameter, A-5
- querying services, A-7
- request handlers, A-13
- requestHandlers deployment parameter, A-5
- response handlers, A-13
- responseHandlers deployment parameter, A-5
- security features, A-21
- service manager
 - deploying services, A-7
 - listing services, A-7

- querying services, A-7
- undeploying services, A-7
- verifying services, A-7
- serviceManager deployment parameter, A-5
- ServiceURI element, A-15
- servlet.soaprouter.initArgs parameter, A-3
- soap.properties
 - soapConfig, A-3
- soap.xml, A-3
- TcpTunnelGui command, A-37
- TimeStamp element, A-15
- troubleshooting, A-36
- undeploying services, A-7
- User element, A-15
- oracle.wallet.location property, 8-13
- oracle.soap.transport.allowUserInteraction property, 8-13
- oracle.ssl.ciphers property, 8-13
- oracle.wallet.password property, 8-13

P

- packageName request parameter, 8-6
- packaging WSDL, 9-5
- path configuration tag, 4-10
- PL/SQL stored procedures
 - setting up datasources, 5-11
 - writing, 5-2
- prefix configuration tag, 5-7
- publishing API, 10-14
- publishing Web services, 10-8

Q

- queue-resource-ref configuration tag, 7-14

R

- reply-to-connection-factory-resource-ref configuration tag, 7-15
- reply-to-queue-resource-ref configuration tag, 7-15
- reply-to-topic-resource-ref configuration tag, 7-15

S

- schema configuration tag, 5-7
- scope configuration tag, 3-12, 6-12
- session-timeout configuration tag, 3-12, 6-12
- simple object access protocol
 - what is SOAP, 1-6
- SOAP
 - features, 1-6
 - web services, 1-6
 - what is SOAP, 1-6
- SOAP client-side
 - HTTP transport properties
 - http.authType property, 8-12
 - http.password property, 8-12
 - http.proxyAuthType property, 8-12
 - http.proxyHost property, 8-12
 - http.proxyPassword property, 8-12
 - http.proxyPort property, 8-12
 - http.proxyRealm property, 8-12
 - http.proxyUsername property, 8-12
 - http.realm property, 8-12
 - http.username property, 8-12
 - java.protocol.handler.pkgs property, 8-13
 - oracle.wallet.location property, 8-13
 - oracle.soap.transport.allowUserInteraction property, 8-13
 - oracle.ssl.ciphers property, 8-13
 - oracle.wallet.password property, 8-13
- stateful document style, 6-4
- stateful java class, 3-3
- stateful-java-service configuration tag, 3-10, 6-11
- stateless document style, 6-4
- stateless java class, 3-3
- stateless session EJBs
 - helloStatelessSession sample code, 4-2
 - writing, 4-2
- stateless-java-service configuration tag, 3-10, 6-11
- stateless-session-ejb-service configuration tag, 4-9

T

- temporary-directory configuration tag, 3-10
- topic-resource-ref configuration tag, 7-15

U

- UDDI
 - Oracle UDDI registry, 10-9
 - production environment configuration, 10-30
 - publishing Web services
 - using OEM Deploy Applications Wizard, 10-9
 - using OEM Web Services Details window, 10-11
 - registration, 10-2
 - registry, 10-2
 - registry administration
 - administrative entity management, 10-21
 - built-in validated category management, 10-23
 - command-line tool uddiadmin.jar, 10-20
 - database configuration, 10-22
 - import operation, 10-22
 - performance monitoring and tuning, 10-20
 - server configuration, 10-20
 - server configuration parameters reference information, 10-26
 - transport security, 10-25
 - user account management, 10-21
 - standard taxonomies
 - ISO3166, 10-2
 - NAICS, 10-2
 - UNSPSC, 10-2
 - updating published Web services
 - using OEM Web Services Details window, 10-12
 - Web service discovery, 10-4
 - using inquiry API, 10-5
 - using tools, 10-5
 - Web service publishing, 10-4
 - using publishing API, 10-14
 - using tools, 10-8
- UDDI registry administration
 - user management, 10-19
- uddiadmin.jar
 - registry administration command-line tool, 10-20
- uri configuration tag, 3-12, 4-10, 5-6, 6-12, 7-15

W

Web Services

- Bean support, 3-7, 4-6
- client-side proxies, 8-3, 8-9
 - packageName request parameter, 8-6
- discovering, 10-5
- document style
 - deploying, 6-16
 - interface, 6-7
 - null value for Element, 6-3
 - stateful, 6-4
 - stateless, 6-4
- encoding parameters, 3-15
- encoding results, 3-15
- generating client proxies, A-8
- generating client-side proxies, 8-8
- generating WSDL documents, A-9
- home page, 8-3
- Java Bean support, 3-7, 4-6
- java class
 - deploying, 3-9, 3-14
 - interface, 3-5
 - preparing, 3-9
 - stateful, 3-3
 - stateless, 3-3
 - supported parameter types, 3-7
 - supported return value types, 3-7
 - supported types, 3-5
- JMS
 - deploying, 7-17
 - preparing an EAR file, 7-17
- locating, 8-2
- PL/SQL stored procedures, 5-2
 - deploying, 5-12
 - preparing an EAR file, 5-10
 - setting up datasources, 5-11
- proxies, 8-9
- publishing, 10-8
- serializing parameters, 3-15
- serializing results, 3-15
- stateless session EJBs, 4-2
 - bean code, 4-4
 - deploying, 4-8, 4-11
 - developing web services, 4-3

- error handling, 4-5
- home interface, 4-3, 4-4
- preparing, 4-8
- remote interface, 4-3
- returning results, 4-5
- sample code, 4-2
- supported parameter types, 4-6
- supported return value types, 4-6
- test page, 8-3
- WSDL descriptions, 8-3, 8-8
- WebServicesAssembler, 9-5
- DTD, 9-9
- generating WSDL, 9-5
- running, 9-2
- sample input file, 9-2
- sample output, 9-4
- tag
 - class-name, 3-11, 6-12
 - connection-factory-resource-ref, 7-13
 - context, 3-10
 - db-pkg-name, 5-7
 - db-url, 5-7
 - description, 3-10
 - destination-path, 3-10
 - display-name, 3-10
 - ejb-name, 4-10
 - ejb-resource, 3-11, 4-10
 - interface-name, 3-11, 6-12
 - jar-generation, 5-6
 - java-resource, 3-12, 6-12
 - jms-delivery-mode, 7-13
 - jms-doc-service, 7-13
 - jms-expiration, 7-13
 - jms-message-type, 7-14
 - jms-priority, 7-14
 - message-style, 3-12, 6-12
 - method-name, 5-7
 - operation, 7-14
 - option name="force", 9-6
 - option name="httpServerURL", 9-6
 - option name="include-source", 8-9
 - option name="packageIt", 9-6
 - option name="source-path", 3-10
 - option name="wsdl-location", 8-9
 - option package-name, 8-9

- path, 4-10
- prefix, 5-7
- proxy-dir, 8-9
- queue-resource-ref, 7-14
- reply-to-connection-factory-resource-ref, 7-1
5
- reply-to-queue-resource-ref, 7-15
- reply-to-topic-resource-ref, 7-15
- schema, 5-7
- scope, 3-12, 6-12
- session-timeout, 3-12, 6-12
- stateful-java-service, 3-10, 6-11
- stateless-java-service, 3-10, 6-11
- stateless-session-ejb-service, 4-9
- stateless-stored-procedure-java-service, 5-5
- temporary-directory, 3-10
- topic-resource-ref, 7-15
- uri, 3-12, 4-10, 5-6, 6-12, 7-15
- wSDL dir, 9-6
- WSDL file, 9-7
- WSDL file
 - getting directly, 8-5
- wSDL2java script, A-8