Oracle9*i* Application Server

Web Services Developer’s Guide

Release 2 (v9.0.2)

January 2002

Part No. A95453-01

Oracle9*i* Application Server Web Services Developer’s Guide, Release 2 (v9.0.2)

Part No. A95453-01

Primary Authors: Thomas Van Raalte and Rodney Ward

Contributors: Marco Carrer, 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, Steve Muench, Giuseppe Panciera, Wei Qian, Venkata Ravipati, Susan Shepard, Alok Srivastava, Zhe (Alan) Wu, Joyce Yang, Chen Zhou

Contents

Send Us Your Comments

Oracle9*i* Application Server Web Services Developer’s Guide, Release 2 (v9.0.2)

Part No. A95453-01

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: Oracle9*i* Application Server Documentation Manager
- Postal service:
  Oracle Corporation
  Oracle9*i* 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 Oracle9*i*AS Web Services.

This preface contains these topics:

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 Oracle9*i* 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 Oracle9*i*AS 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 Oracle9*i*AS 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 Oracle9*i*AS 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 Oracle9*i*AS Web Services implemented as stateless PL/SQL Stored Procedures or Functions.

Chapter 6, "Building Clients that Use Web Services"

This chapter describes the steps required to build a client application that uses Oracle9*i*AS Web Services.

Chapter 7, "Web Services Assembly Tool"

This chapter describes the Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, that assists in assembling Oracle9*i*AS 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 Oracle9*i*AS Web Services.

Chapter 8, "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 9, "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 Oracle9*i* 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/admin/account/membership.html`

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**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
`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 `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* 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 (`*`digits`*` [ , `*`precision`*` ])`
{ }	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:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` 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;`

`Italics`	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		 `DB_NAME = `*`database_name`*

`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.	`SELECT last_name, employee_id FROM employees;`
		 `SELECT * FROM USER_TABLES;`
		 `DROP TABLE hr.employees;`

Convention	Meaning	Example

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		
		`sqlplus hr/hr`
		
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

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 - *HOME_NAME* > 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.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`
`C:\>`	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 *command prompt* in this manual.	`C:\oracle\oradata>`

Convention	Meaning	Example

	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.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"`
		 `C:\>imp SYSTEM/`*`password`* `FROMUSER=scott TABLES=(emp, dept)`

`HOME_NAME`	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.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_ BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ `C:\orant` for Windows NT ▪ `C:\orawin95` for Windows 95 ▪ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you install Oracle9*i* 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 *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Starting for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_ HOME*`\rdbms\admin` directory.

1 Web Services Overview

This chapter provides an overview of Web Services. Chapter 2, "Oracle9iAS Web Services" describes the Oracle9*i*AS 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. Oracle9*i*AS 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.

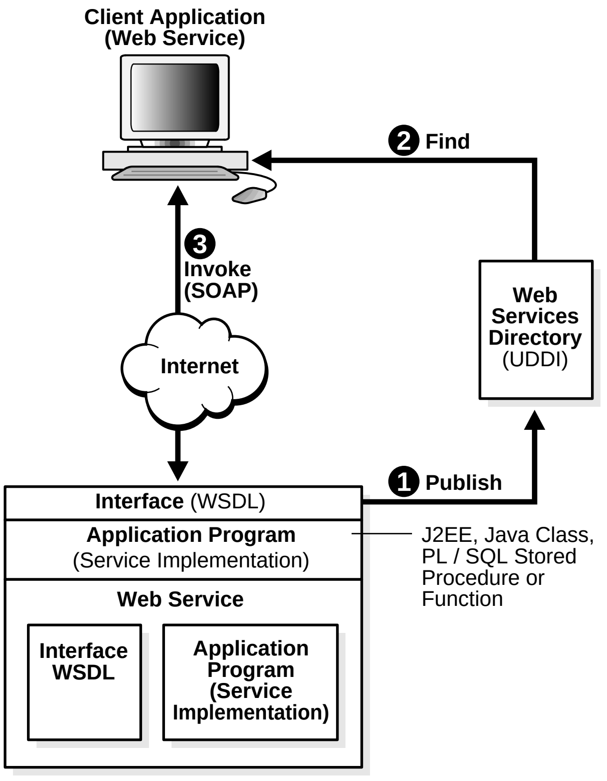
Overview of Web Services Standards

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.

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.

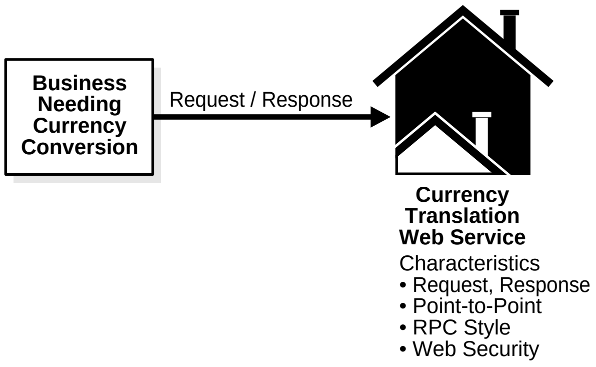
Scenario: Web Services with a Currency Converter Application

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.

2 Oracle9*i*AS Web Services

This chapter describes the Oracle9*i*AS Web Services features, architecture, and implementation.

This chapter covers the following topics:

Oracle9*i*AS OC4J (J2EE) and Oracle SOAP Based Web Services

Oracle9*i* Application Server (Oracle9*i*AS) supports two different Web Services options, a J2EE based Web Services environment built into Oracle9*i*AS OC4J, and an Apache SOAP (Oracle SOAP) based Web Services environment.

The chapters in this manual describe the Oracle9*i*AS 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 Oracle9*i*AS features.

Appendix A, "Using Oracle SOAP" describes the Oracle9*i*AS support for Apache SOAP (Oracle SOAP). Oracle9*i*AS includes support for Apache SOAP because this implementation was one of the earliest SOAP implementations and it supports many existing Web Services applications.

See Also: Appendix A, "Using Oracle SOAP"

Oracle9*i*AS Web Services Features

Oracle9*i*AS provides advanced runtime features and comprehensive support for developing and deploying Web Services. The Oracle9*i*AS 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

Oracle9*i*AS Web Services provides comprehensive support for developing Web Services, including:

- Development Environment – Oracle9*i*AS 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 – Oracle9*i*AS Web Services Developers can use the same set of command line utilities to create, package, and deploy Web Services as other Oracle9*i*AS Containers for J2EE (OC4J) Applications. In addition Oracle9*i*AS 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 – Oracle9*i*AS Web Services automatically generates WSDL and client-side proxy stubs transparently. The first time the WSDL or the client-side proxy stubs are requested they are generated. After the first request, the previously generated WSDL or client-side proxy stubs are sent when requested.
- Registration, Publishing, and Discovery – Oracle9*i*AS 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 Oracle9*i*AS 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 – Oracle9*i*AS 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 deploys and manages Oracle9*i*AS Web Services. Oracle Enterprise Manager provides the following support for Web Services:

- Deployment – Oracle Enterprise Manager provides a comprehensive set of facilities to deploy Web Services to Oracle9*i*AS. Oracle Enterprise Manager provides a single, consistent *Deploy Applications* wizard for deploying Web Services to Oracle9*i*AS. 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 Oracle9*i*AS instance.
- Packaging - The Web Services Assembly Tool assists with assembling the following types of Web Services: Stateless Java (including PL/SQL Stored Procedures and Functions), Stateful Java and Stateless Session EJBs.

Using Oracle9*i* JDeveloper with Web Services

The Oracle9*i* JDeveloper IDE supports Oracle9*i*AS Web Services. Oracle9*i* 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 Oracle9*i*AS 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 Oracle9*i*AS 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

Oracle9*i*AS Portal facility provides the ability to aggregate Oracle9*i*AS Web Services within an organization into a Portal. Additionally, portlets in the Oracle9*i*AS Portal framework can be published as Web Services.

Oracle9*i*AS Web Services Architecture

The Oracle9*i*AS Containers for J2EE (OC4J) provides the foundation for building applications as components and supports Oracle9*i*AS Web Services (for RPC style Web Services). Oracle9*i*AS Web Services are implemented as any of the following:

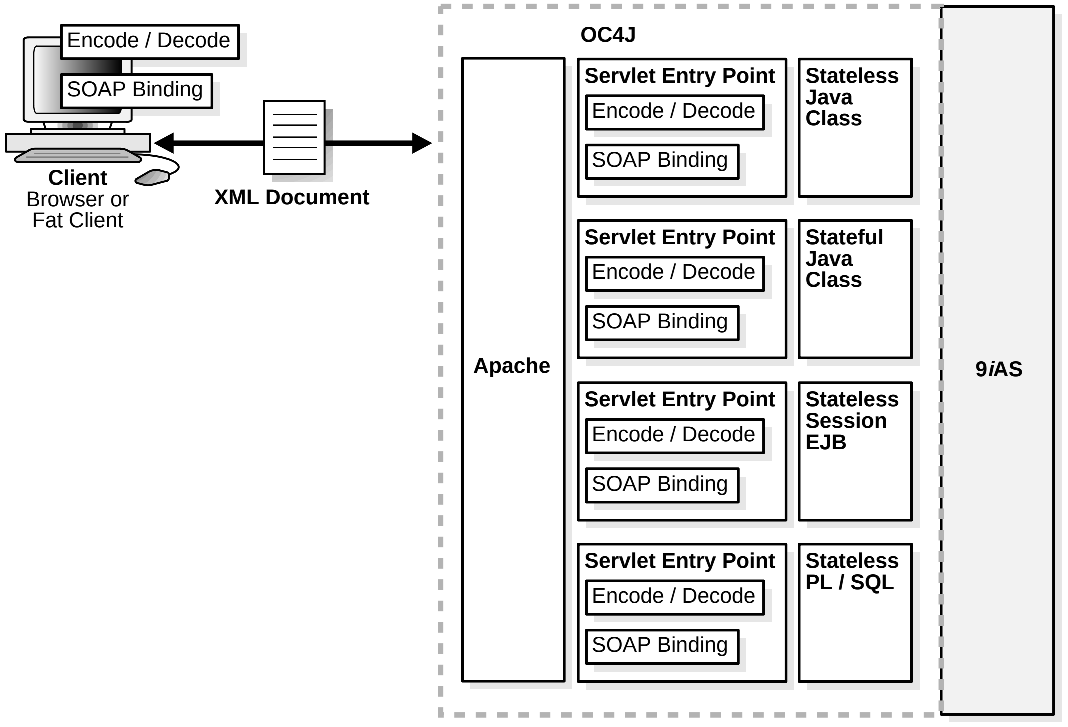
- Java Classes
- Stateless Session Enterprise Java Beans (EJBs)
- Stateless PL/SQL Stored Procedures or Functions

For each implementation type, Oracle9*i*AS 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 Oracle9*i*AS Web Services runtime architecture 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

Figure 2–1 Web Services Runtime Architecture (RPC with Servlet Entry Points)



About Servlet Entry Points for Web Services

To use Oracle9*i*AS Web Services, you need to deploy a J2EE .ear file to Oracle9*i*AS. The J2EE .ear file contains a Web Services Servlet configuration, and includes an implementation of the Web Service. Oracle9*i*AS Web Services supplies the Servlet classes, one for each supported implementation type. At runtime, Oracle9*i*AS uses the Servlet classes to access the user supplied Web Service implementation.

The Oracle9*i*AS 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.

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 Oracle9*i*AS Web Services runtime support. To make Web Services accessible, you deploy the Web Service implementation with the corresponding Web Services Servlet.

Note: Using Oracle9*i*AS 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 Oracle9*i*AS Web Services architecture differs; under Oracle9*i*AS Web Services, a unique Servlet instance supports each Web Service.

Web Services implementations under Oracle9*i*AS 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 Oracle9*i*AS Web Services.

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

Primitive Type	**Object Type**
`Boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`
`string`	`java.lang.String`
	`java.util.Date`
	`org.w3c.dom.Element`
	`org.w3c.dom.Document`
	`org.w3c.dom.DocumentFragment`
	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

Oracle9*i*AS Web Services are accessed as Servlets, thus, Web Services need to be assembled by configuring a `web.xml` file that is a component of a J2EE .war file. Web Services implementation classes must be in the J2EE .war, .ear, ejb.jar files, or available through the classpath using standard J2EE mechanisms.

To deploy Oracle9*i*AS Web Services, you have the following options, depending on the implementation of the service:

- For Java Classes - a .ear file can be deployed to Oracle9*i*AS. The .ear file is assembled to include a .war file containing the Java classes and the Servlet declaration in the `web.xml` configuration portion of the .war file, without requiring any other J2EE artifacts.

- For Stateless Session EJBs - the .war file that configures the Web Services Serlvet Entry Point must be added to the EJB's .ear file and the .ear file can be deployed to Oracle9*i*AS.
- For PL/SQL Stored Procedures and Functions - the treatment is the same as for a Java class, a .ear file is assembled with a .war file containing the class can be deployed to Oracle9*i*AS, without requiring any other J2EE artifacts.

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"

About Server Skeleton Code Generation for Web Services

The first time Oracle9*i*AS Web Services receives a request for a service, the Servlet entry point automatically does the following:

- 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 (using SOAP RPC), 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).

Understanding WSDL and Client Proxy Stubs for Web Services

Oracle9*i*AS Web Services provides automatic generation of a WSDL file and of client-side proxy stubs.

There are several elements to Oracle9*i*AS Web Services WSDL support. First, Web Services are based on interoperable XML data representations and arbitrary Java objects do not in general map to XML. Oracle9*i*AS 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 Oracle9*i*AS Web Services, an application developer does not necessarily need to statically generate the WSDL interfaces for a Web Service (although the developer can do so). The Oracle9*i*AS Web Services runtime can generate WSDL and client-side proxy stubs if they are not provided when a Web Service is deployed. These can be generated by the runtime on the server-side and delivered when they are requested by a Web Services client.

Optionally, Oracle9*i*AS also provides a set of tools to statically generate WSDL and client-side proxy stubs given a Java class or J2EE application.

See Also: "Using Oracle9iAS SOAP Management Utilities and Scripts" on page A-6

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 Oracle9*i*AS 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 6, "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, Oracle9*i*AS 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 6, "Building Clients that Use Web Services"

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 Oracle9*i*AS, 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 8, "Discovering and Publishing Web Services"

3 Developing and Deploying Java Class Web Services

This chapter describes the procedures you use to write and deploy Oracle9*i*AS Web Services that are implemented as Java classes.

This chapter covers the following topics:

Using Oracle9*i*AS Web Services With Java Classes

Oracle9*i*AS Web Services can be implemented as any of the following:

- Java Classes
- Stateless Session EJBs
- PL/SQL Stored Procedures or Functions

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.

Oracle9*i*AS supplies Servlets to access the Java classes which implement a Web Service. The Servlets handle requests generated by Web Services clients, run the Java methods that implement the Web Services and return results back to Web Services clients.

See Also:

- Chapter 4, "Developing and Deploying EJB Web Services"
- Chapter 5, "Developing and Deploying Stored Procedure Web Services"
- Chapter 6, "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 that a Web Services Servlet running under Oracle9*i*AS Web Services invokes when a Web Services client makes a 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 Oracle9*i*AS Web Services in the directory `$ORACLE_HOME/j2ee/home/demo/web_services/java_services` on UNIX or in `%ORACLE_HOME%\j2ee\home\demo\web_services\java_services` on Windows.

Writing Stateless and Stateful Java Web Services

Oracle9*i*AS Web Services supports stateful and stateless implementations for Java classes running as Web Services. For a stateful Java implementation, Oracle9*i*AS Web Services allows a single Java instance to serve the Web Service requests from an individual client.

For a stateless Java implementation, Oracle9*i*AS 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 3-8.

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 or Client-Side Proxy (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. If you expand the .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 `StatelessExampleImpl`.

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

```
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

```
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. For example, Example 3–1 shows the public constructor `StatelessExampleImpl()`.

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 for logging Servlet errors for the 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 4–1 lists the supported types for parameters and return values for Java methods that implement Web Services.

Note: See Table 4–1 for the list of supported types for parameters and return values.

Defining an Interface for Explicit Method Exposure

Oracle9*i*AS 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

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

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

```
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. 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 4–1 lists the supported types for parameters and return values for Java methods that implement Web Services.

See Also:

- "Modifying web.xml to Support Java Web Services" on page 3-8
- "Using Supported Data Types" on page 4-6

Writing a WSDL File or Client-Side Proxy (Optional)

When writing a Java class based Web Service, this step is optional. If you do not perform this step, the Oracle9*i*AS Web Services runtime generates a WSDL file and the client-side proxies for deployed Web Services. These files allow Oracle9*i*AS Web Services to supply a Web Service client with the WSDL or the client-side proxies that a client-side developer can use to build an application that uses a Web Service.

When you do not want to use the Oracle9*i*AS Web Services generated WSDL file or client-side proxies and you want to supply your own versions of these files, perform the following steps:

1. Manually create either the WSDL file or the client-side proxy Jar file, or both files for your service.
2. Name the supplied WSDL file or client-side proxy Jar file and place it in the appropriate location. The WSDL file must have a `.wsdl` extension. The client-side proxy Jar must have an `_proxy.jar` extension.The extension is placed after the service name.

   For example,

   ```
   simpleservice.wsdl
   simpleservice_proxy.jar
   ```

3. Add the manually created WSDL file or client-side Jar file to the .war file that contains the service implementation. There are several choices for adding the files to the .war file:

   - Place the files in the correct package directory under `WEB-INF/classes`. For example, if `simpleservice.wsdl` describes a service that is part of the demo package, place the file in the following directory:

     ```
     WEB-INF/classes/demo/simpleservice.wsdl
     ```

   - Place the Jar file in a Jar file under the `WEB-INF/lib` component of the .war file that is added to the .ear file and deployed as a Web Service.

     For example, the file `simpeservice_proxy.jar` could be added to `simpleservice.jar` and placed in `WEB-INF/lib`, or added to *`namex`*`.jar` in `WEB-INF/lib` (where *`namex`* is a file name).

Note: The Jar file containing the proxy stubs and optionally the proxy source must be a Jar component of a Jar file in `WEB-INF/lib`, it cannot simply be added to the `WEB-INF/lib` directory.

See Also:

- "Preparing and Deploying Java Class Based Web Services" on page 3-8
- "Getting WSDL Descriptions and Client-Side Proxy Jar for Web Services" on page 6-2

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 Oracle9*i*AS Web Services Servlet and the Java class that supplies the Java implementation. A Web Service implemented using a Java class includes a .war file that provides configuration information for the Web Services Servlet running under Oracle9*i*AS Containers for J2EE (OC4J). This section describes the procedures you use to assemble the .ear file that contains a Java class to run as a Web Service.

This section covers the following topics:

- Modifying web.xml to Support Java Web Services
- Preparing a .war File for Java Class Web Services
- Preparing the .ear File for Java Class Web Services
- Deploying Java Class Based Web Services

The Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9*i*AS Web Services. The Web Services assembly tool takes a configuration file which describes the location of the Java class and interface files and produces a J2EE .ear file that can be deployed under Oracle9*i*AS Web Services. This section describes how to assemble Oracle9*i*AS Web Services implemented as Java classes manually, without using `WebServicesAssembler`.

See Also: "Running the Web Services Assembly Tool" on page 7-2

Modifying web.xml to Support Java Web Services

To use a Java class as a Web Service, you need to add a `<servlet>` entry and a corresponding `<servlet-mapping>` entry in the `web.xml` file for each Java class that is deployed as a Web Service. The resulting `web.xml` file is assembled as part of a J2EE .war file that is included in the .ear file that defines the Web Service.

To modify `web.xml` to support Web Services implemented as Java classes, perform the following steps:

- Configure the servlet Tag in web.xml for Java Class Web Services
- Configure the servlet-mapping Tag in web.xml for Java Class Web Services

Configure the servlet Tag in web.xml for Java Class Web Services

To add Web Services based on Java classes you need to modify the `<servlet>` tag in the `web.xml` file. This supports using the Oracle9*i*AS Web Services Servlet to access the Java implementation for the Web Service. Table 3–1 describes the `<servlet>` tag and the values to include in the tag to add a Web Service based on a Java class.

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. The `<servlet>` entry in `web.xml` must contain a different value for the `<servlet-class>`, depending on whether the Web Services implantation is stateful or stateless.

Table 3–1 Servlet Tags Supporting Java Class Deployment

Servlet Tag	Description
`<init-param>`	The <init-param> tag contains a name value pair in, `<param-name>` `<param-value>`. **`class-name:`** The Web Services Servlet definition requires at least one `<param-name>` with the value **`class-name`**, and a corresponding `<param-value>` set to the fully qualified name of the Java class used for the Web Service implementation. **`session-timeout:`** An optional `<param-name>` with the value **`session-timeout`**, and a corresponding `<param-value>` set to an integer value in seconds specifies the timeout for the session. This optional parameter only applies for stateful Java Web Services. The default value for the session timeout for stateful Java sessions where no session timeout is specified is 60 seconds. **`interface-name:`** An optional `<param-name>` with the value **`interface-name`**, and a corresponding `<param-value>` set to the fully qualified name of the Java interface specifies the methods to include in the Web Service. This init parameter tells the Web Service Servlet generation code which methods should be exposed on the Web Service. If the parameter `interface-name` is not included in the `<servlet>` definition, then all public methods on the class are included in the Web Service. Note: when an interface is used the interface only supplies information to the Web Services support system. The Web Service implementation class specified for the `class-name` does not actually have to implement the interface.
`<servlet-class>`	This is always `oracle.j2ee.ws.StatelessJavaRpcWebService` for all stateless Java Web Services and `oracle.j2ee.ws.JavaRpcWebService` for all stateful Java Web Services.
`<servlet-name>`	Specifies the name for the Servlet that runs the Web Service.

Example 3–5 shows a sample `<servlet>` entry for Web Services implemented as a Java class running as a stateless Web Service. Example 3–6 shows a sample `<servlet>` entry for a Web Service implemented as a Java class running as a stateful Web Service.

Example 3–5 Sample Stateless <servlet> Entry for a Web Service

```
<servlet>
  <servlet-name>stateless java web service example</servlet-name>
  <servlet-class>oracle.j2ee.ws.StatelessJavaRpcWebService</servlet-class>
  <init-param>
    <param-name>class-name</param-name>
    <param-value>oracle.j2ee.ws_example.StatelessExampleImpl</param-value>
  </init-param>
  <init-param>
    <param-name>interface-name</param-name>
    <param-value>oracle.j2ee.ws_example.StatelessExample</param-value>
  </init-param>
</servlet>
```

Example 3–6 Sample Stateful <servlet> Entry for a Web Service

```
<servlet>
<servlet-name>stateful java web service example</servlet-name>
    <servlet-class>oracle.j2ee.ws.JavaRpcWebService</servlet-class>
    <init-param>
        <param-name>class-name</param-name>
        <param-value>oracle.j2ee.ws_example.StatefulExampleImpl</param-value>
    </init-param>
    <init-param>
        <param-name>interface-name</param-name>
        <param-value>oracle.j2ee.ws_example.StatefulExample</param-value>
    </init-param>
</servlet>
```

> **See Also:** Writing Stateless and Stateful Java Web Services on page 3-3

Configure the servlet-mapping Tag in web.xml for Java Class Web Services

To add Web Services based on Java classes, you need to modify the `<servlet-mapping>` tag in the `web.xml` file. This tag specifies the URL for the Servlet that implements a Web Service.

Example 3–7 shows sample `<servlet-mapping>` entries corresponding to the servlet entries shown in Example 3–5 and Example 3–6.

Example 3–7 Sample <servlet-mapping> Entries for Web Services

```
<servlet-mapping>
     <servlet-name>stateful java web service example</servlet-name>
     <url-pattern>/statefulTest</url-pattern>
</servlet-mapping>

<servlet-mapping>
     <servlet-name>stateless java web service example</servlet-name>
     <url-pattern>/statelessTest</url-pattern>
</servlet-mapping>
```

Preparing a .war File for Java Class Web Services

Web Services implemented with Java classes use a standard .war file to define J2EE Servlet configuration and deployment information. After modifying the `web.xml` file, add the implementation classes and any required support classes or Jar files either under `WEB-INF/classes`, or under `WEB-INF/lib` (or in a classpath location available to OC4J).

Note: All the classes that are required for a Web Service implementation must conform to the standard J2EE class loading norms. Thus, the implementation classes and support classes must either be in a .war or .ear file, or they must be available in the OC4J classpath.

Preparing the .ear File for Java Class Web Services

To add Web Services based on Java classes, you need to include an `application.xml` file and package the `application.xml` and .war file containing the Java classes into a J2EE .ear file.

Deploying Java Class Based 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.

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.

Oracle9*i*AS 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, Oracle9*i*AS Web Services supports the following encoding mechanisms:

- Standard SOAP v.1.1 encoding: Using standard SOAP v1.1 encoding, the server side Web Services Servlet that calls the Java class implementation handles serialization and encoding internally for the types supported by Oracle9*i*AS Web Services. Table 4–1 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 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 Oracle9*i*AS 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 6, "Building Clients that Use Web Services"

4 Developing and Deploying EJB Web Services

This chapter describes the procedures you use to write and deploy Oracle9*i*AS Web Services that are implemented as stateless session Enterprise Java Beans (EJBs).

This chapter covers the following topics:

Using Oracle9*i*AS Web Services With Stateless Session EJBs

Oracle9*i*AS Web Services can be implemented as any of the following:

- Java Classes
- Stateless Session EJBs
- PL/SQL Stored Procedures or Functions

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

Oracle9*i*AS supplies Servlets to access the EJBs which implement a Web Service. The Servlet handles 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 3, "Developing and Deploying Java Class Web Services"
- Chapter 5, "Developing and Deploying Stored Procedure Web Services"
- Chapter 6, "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 Oracle9*i*AS 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 Oracle9*i*AS Web Services installation in the following directory:
`$ORACLE_HOME/j2ee/home/demo/web_services\stateless_ejb` on UNIX
`%ORACLE_HOME%\j2ee\home\demo\web_services\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. Oracle9*i*AS Web Services runs EJBs that are deployed as Oracle9*i*AS 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
- Writing a WSDL File or Client-Side Proxy Stubs for EJB Web Services

> **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

```
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 Oracle9*i*AS 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 Services 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" 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 3-12

Using Supported Data Types

Table 4–1 lists the supported data types for parameters and return values for Oracle9*i*AS Web Services.

Table 4–1 Web Services Supported Data Types

Primitive Type	Object Type
`Boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`
`string`	`java.lang.String`
	`java.util.Date`
	`org.w3c.dom.Element`
	`org.w3c.dom.Document`
	`org.w3c.dom.DocumentFragment`
	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.

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 set*X* or get*X* methods, are in.

Oracle9*i*AS 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.

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 6, "Building Clients that Use Web Services"

Writing a WSDL File or Client-Side Proxy Stubs for EJB Web Services

When writing an EJB Web Service, this step is optional. If you do not perform this step, the Oracle9*i*AS Web Services runtime generates a WSDL file and the client-side proxy stubs for deployed Web Services. These files allow Oracle9*i*AS Web Services to supply a Web Service client with the WSDL or the client-side proxy stubs that a client-side developer can use to build an application that uses a Web Service.

When you do not want to use the Oracle9*i*AS Web Services generated WSDL file or client-side proxy stubs, and you want to supply your own versions of these files, perform the following steps:

1. Manually create either the WSDL file or the client-side proxy Jar file, or both files for your service.
2. Name the supplied WSDL file or client-side proxy Jar file and place it in the appropriate location. The WSDL file must have a `.wsdl` extension. The client-side proxy Jar must have an `_proxy.jar` extension.The extension is placed after the service name.

For example,

```
simpleservice.wsdl
simpleservice_proxy.jar
```

3. Add the manually created WSDL file or client-side Jar file to the .war file that is associated with the Web Service servlet for the service implementation. There are several choices for adding the files to the .war file:

   - Place the files in the correct package directory under `WEB-INF/classes`. For example, if `simpleservice.wsdl` describes a service that is part of the demo package, place the file in the following directory:

     ```
     WEB-INF/classes/demo/simpleservice.wsdl
     ```

   - Place the Jar file that you create in a Jar file under the `WEB-INF/lib` component of the .war file that is added to the .ear file and deployed as a Web Service. For example, the file `simpeservice_proxy.jar` could be added to `simpleservice.jar` and placed in `WEB-INF/lib`, or added to a Jar file *`namex`*`.jar` in `WEB-INF/lib` (where *`namex`* is a file name).

     Note: The Jar file containing the proxy stubs, and optionally the proxy source must be a Jar component of a Jar file in `WEB-INF/lib`, it cannot simply be added to the `WEB-INF/lib` directory.

     See Also:

     - "Preparing and Deploying Stateless Session EJB Based Web Services" on page 4-8
     - "Getting WSDL Descriptions and Client-Side Proxy Jar for Web Services" on page 6-2

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 Oracle9*i*AS Web Services Servlet and the ejb.jar file that supplies the EJB implementation. A Web Service implemented using a stateless session EJB includes a .war file that provides descriptive information for the Oracle9*i*AS Web Services Servlet running under Oracle9*i*AS Containers for J2EE (OC4J). This section describes the procedures you use to assemble the .ear file that contains a stateless session EJB and the Web

Services Servlet configuration, placed in a .war file, required to run a stateless session EJB as a Web Service.

This section covers the following topics:

- Preparing a JAR File With an EJB
- Modifying web.xml to Support EJB Web Services
- Preparing a WAR File for EJB Web Services
- Preparing an EAR File for EJB Web Services
- Deploying Web Services Implemented as EJBs

The Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9*i*AS Web Services. The Web Services assembly tool takes a configuration file which describes the location of the J2EE/EJB Jar files and produces a J2EE .ear file that can be deployed under Oracle9*i*AS Web Services. This section describes how to assemble Oracle9*i*AS Web Services implemented as stateless session EJBs manually, without using `WebServicesAssembler`.

See Also: "Running the Web Services Assembly Tool" on page 7-2

Preparing a JAR File With an EJB

To prepare a stateless session EJB to implement a Web Service, you need to prepare a JAR file as you would for any other standard J2EE stateless session EJB.

The JAR file should contain the following:

- The EJB classes and interfaces
- The EJB Deployment Descriptor: `ejb-jar.xml`

For the sample stateless session EJB, the EJB classes should include the classes developed for the `HelloServiceBean`, organized in the standard J2EE directory format.

Example 4–4 shows the stateless session EJB deployment descriptor for the `HelloServiceBean` that is included in the JAR file.

Example 4–4 EJB Deployment Configuration File: ejb-jar.xml

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.
//DTD Enterprise JavaBeans 1.1
//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
    <enterprise-beans>
      <session>
        <ejb-name>HelloService</ejb-name>
        <home>demo.HelloServiceHome</home>
        <remote>demo.HelloService</remote>
        <ejb-class>demo.HelloServiceBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      </session>
    </enterprise-beans>
</ejb-jar>
```

Modifying web.xml to Support EJB Web Services

This section provides instructions for assembling a Web Service based on a stateless session EJB. To use an EJB as a Web Service you need to add a `<servlet>` entry and corresponding `<ejb-ref>` and `<servlet-mapping>` entries in a `web.xml` file for each stateless session EJB that is deployed as a Web Service. The resulting `web.xml` file is packaged as part of a J2EE .war file that is included in the .ear file that is deployed to OC4J.

Note: All stateless session EJBs deployed as Oracle9*i*AS Web Services must conform to the restrictions limiting parameter and return types. See Table 4–1 for the list of supported types for parameters and return values.

To modify `web.xml` to support Web Services implemented as stateless session EJBs, perform the following steps:

- Configure the servlet Tag in web.xml for EJB Web Services
- Configure the ejb-ref Tag in web.xml for EJB Web Services
- Configure the servlet-mapping Tag in web.xml for EJB Web Services

Configure the servlet Tag in web.xml for EJB Web Services

To add Web Services based on stateless session EJBs, you need to modify the `<servlet>` tag in the `web.xml` file. This supports using the Oracle9*i*AS Web Services Servlet with a stateless session EJB. Table 4–2 describes the `<servlet>` tag and the values that it must include to add a Web Service based on a stateless session EJB.

Example 4–5 shows the sample `<servlet>` tag for the `HelloService` EJB Web Service, as extracted from the `web.xml` file in the .ear file, `statelessejb.ear`.

Table 4–2 Servlet Tags Supporting EJB Deployment

Servlet Tag	Description
`<init-param>`	The `<init-param>` tag contains a `<param-name>` and `<param-value>` pair. **`jndi-name:`** The Web Services Servlet definition requires a `<param-name>` with the value **`jndi-name`**, and a corresponding `<param-value>` set to the EJBs JNDI URL relative to the java:comp/env for the Servlet.
`<servlet-class>`	This is always `oracle.j2ee.ws.SessionBeanWebService` for all stateless session EJB Web Services.
`<servlet-name>`	Specifies the name for the Servlet that runs the Web Service.

Example 4–5 Sample Stateless <servlet> Entry for an EJB Based Web Service

```
<servlet>
  <servlet-name>stateless session bean web service - HelloService</servlet-name>
  <servlet-class>oracle.j2ee.ws.SessionBeanWebService</servlet-class>
  <init-param>
    <param-name>jndi-name</param-name>
    <param-value>ejb/HelloService</param-value>
  </init-param>
</servlet>
```

Configure the ejb-ref Tag in web.xml for EJB Web Services

To add Web Services based on stateless session EJBs, you need to modify the `<ejb-ref>` tag in the `web.xml` file. This tag defines the EJB reference that is used as the implementation of the Web Service.

Example 4–6 shows the `<ejb-ref>` tag for the `HelloServiceBean` for the `HelloService` EJB, as extracted from the `web.xml` file in the .ear file, `HelloService.ear`.

Example 4–6 Stateless Session EJB <ejb-ref> Entry for a Web Service

```
<ejb-ref>
  <ejb-ref-name>ejb/HelloService</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>demo.HelloServiceHome</home>
  <remote>demo.HelloService</remote>
  <ejb-link>HelloService</ejb-link>
</ejb-ref>
```

See Also: *Java 2 Platform Enterprise Edition Specification, v1.3* for more information on using the `<ejb-ref>` tag in `web.xml`, at

http://java.sun.com/j2ee/docs.html

Configure the servlet-mapping Tag in web.xml for EJB Web Services

To add Web Services based on stateless session EJBs, you need to modify the `<servlet-mapping>` tag in the `web.xml` file. This tag specifies the URL for the Servlet that implements a Web Service.

Example 4–7 shows a sample `<servlet-mapping>` entry corresponding to the servlet entry shown in Example 4–5.

Example 4–7 Sample <servlet-mapping> Entries for Web Services

```
<servlet-mapping>
  <servlet-name>stateless session bean web service - HelloService</servlet-name>
  <url-pattern>/HelloService</url-pattern>
</servlet-mapping>
```

Preparing a WAR File for EJB Web Services

To add Web Services based on stateless session EJBs, you need to use a .war file to package the J2EE Servlet configuration files. After modifying the `web.xml` file, add any required support classes under WEB-INF/classes or WEB-INF/lib and package all the files as a .war file.

Note: All the classes the are required for a Web Service implementation must conform to the standard J2EE class loading norms. Thus, the implementation classes and support classes must be packaged in the .war or .ear file, or they must be available in the OC4J classpath.

Preparing an EAR File for EJB Web Services

To add Web Services based on stateless session EJBs, you need to include an `application.xml` file and package the `application.xml`, the .war file and the ejb.jar file containing the stateless session EJB into a J2EE .ear file.

The Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9*i*AS Web Services.

See Also: "Running the Web Services Assembly Tool" on page 7-2

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 Oracle9*i*AS Documentation Library

5 Developing and Deploying Stored Procedure Web Services

This chapter describes the procedures you use to write and deploy Oracle9*i*AS 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 Oracle9*i*AS Web Services, PL/SQL procedures and functions that run on an Oracle database server.

This chapter covers the following topics:

Using Oracle9*i*AS Web Services with Stored Procedures

Oracle9*i*AS Web Services can be implemented as any of the following:

- Java Classes
- Stateless Session EJBs
- Stateless PL/SQL Stored Procedures or Functions

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.

Oracle9*i*AS 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, C/C++. These stored procedures can be exposed as Web Services using PL/SQL interfaces.

See Also:

- Chapter 3, "Developing and Deploying Java Class Web Services"
- Chapter 4, "Developing and Deploying EJB Web Services"

Writing Stored Procedure Web Services

Writing PL/SQL Stored Procedure based Web Services involves creating and installing a PL/SQL package on an Oracle database server that is available as a datasource to Oracle9*i*AS 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 with Oracle9*i*AS Web Services in the directory `$ORACLE_HOME/j2ee/home/demo/web_services/stored_procedure` on UNIX or in `%ORACLE_HOME%\j2ee\home\demo\web_services\stored_procedure` on Windows.

Developing a Stored Procedure Web Service consists of the following steps:

- Creating a PL/SQL Stored Procedure for a Web Service
- Generating a Stored Procedure Web Service Jar File
- Creating and Compiling Stored Procedure Web Service Interfaces
- Writing a WSDL File or Client Stubs for Stored Procedure Web Services (Optional)

See Also: "Preparing Stored Procedure Web Services" on page 5-8

Creating a PL/SQL Stored Procedure for a Web Service

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 Oracle9*i*AS Web Services.

For example, to use the sample `COMPANY` package, first create and load the supplied package 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 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 PL/SQL package:

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

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

Stored Procedure Web Services do not support OUT or IN/OUT parameters.

See Also:

- "Limitations for Stored Procedures Running as Web Services" on page 5-13
- *PL/SQL User's Guide and Reference* in the Oracle Database Documentation Library
- *Oracle Net Services Administrator's Guide* in the Oracle Documentation Library

Generating a Stored Procedure Web Service Jar File

After a PL/SQL stored procedure or function is available to support Stored Procedure Web Services, you need to generate the Java class to access the PL/SQL package. In this step you run Apache `ant` using the buildfile supplied in the directory `$ORACLE_HOME/j2ee/home/bin/spbuild.xml`, and setting Java properties to control the generation process. The Apache `ant` command calls the Oracle database server JPublisher tool and generates a Java class and methods to access the stored procedure (this step prepares a Jar file for use with Stored Procedure Web Services).

JPublisher generates a Java class that provides a mapping between a PL/SQL package name, including all PL/SQL types that the stored procedures in the package use, and Java classes with methods corresponding to the PL/SQL package and procedures or functions (and mapped Java types for each PL/SQL type).

To generate Stored Procedure Web Services using a PL/SQL package as the source, use the `ant` command as follows:

```
%ant -buildfile spbuild.xml Java_properties
```

Table 5–1 lists the Java *properties* that control the generation command. You can set the properties shown in Table 5–1 as system properties by using the `-D` flag at the Java command line.

For example, the following command generates the `company.jar` file using the `COMPANY` package as the PL/SQL stored procedure:

```
cd $ORACLE_HOME/j2ee/demo/web_services/stored_procedure
ant -buildfile spbuild.xml
-Dschema=scott/tiger
-Durl="jdbc:oracle:thin:@system1.us.oracle.com:1521:db817"
-Dpackage=Company
-DserviceJar=company.jar
-Dprefix=sp.company
-DOracleHome=/private2/oracleDBClient817/
```

This generates the Jar file `company.jar` that contains the Java source and class files to access the PL/SQL procedure (and supports Stored Procedure Web Services).

Table 5–1 Java Properties for Use with Ant and spbuild.xml

Option	Description
-DoracleHome=*directory*	Where *directory* is the directory path for the Oracle installation. Setting this property is required.
–Dpackage=*pkg_name*	Where *pkg_name* is the name of the PL/SQL package to export. Setting this property is required.
–Dprefix=*prefix*	Where *prefix* is the Java package prefix for generated classes. By default, the PL/SQL package is generated into a Java class in the default Java package. Setting this property is optional.
–Dschema=*user_name*/*pword*	Where *user_name* is the database user name and *pword* is the database password for the specified user name. Setting this property is required.
–DserviceJar=*jarfile*	Where *jarfile* is the name of the generated jar file for use with a Stored Procedure Web Service. Setting this property is required.
–Durl=*url_path*	Where *url_path* is the database connect string for the Oracle database server with the specified package to export. The user, password and url are the combination of database login username/password and a JDBC-style URL used to connect to the backend database which contains the stored procedures to be exported. For example: `-Durl=jdbc:oracle:thin:@system1.us.oracle.com:1521:tv1` Setting this property is required.

See Also:

- *Oracle9i JPublisher User's Guide* in the Oracle Database Documentation Library
- `http://jakarta.apache.org/ant/` for information on the Apache build tool

Creating and Compiling Stored Procedure Web Service Interfaces

The Java classes that are placed in the Jar file output from `ant` contain both support methods for accessing the PL/SQL stored procedure and JPublisher generated support methods that cannot be exposed as Oracle9*i*AS Web Services without generating errors. To limit the exposed methods to those that you want to expose from the PL/SQL package, you need to supply a public interface. The public interface limits the methods exposed as Stored Procedure Web Services.

To create the public interface expand the generated Jar file and examine the public methods in the generated Java source file (`Company.java`). Add the method signatures for the methods that you want to expose to the public interface that you create. You do not need to implement any of the interfaces that are added to the public interface, since the public interface just identifies the method names and signatures for Stored Procedure Web Services.

For example, if you are implementing a Stored Procedure Web Service using the `COMPANY` package and the generated Jar file from the supplied company sample, then create a public interface using the methods in the generated file `Company.java` as a guide for identifying method signatures. For the sample, the `Company.java` file is in the Jar file under `sp/company/Company.java`. The public interface `CompanyInterface.java` is shown in Example 5–1.

Example 5–1   Using a Public Interface to Expose Stored Procedure Web Services

```
import java.sql.*;
public interface CompanyInterface
{
public void addemp (
    java.math.BigDecimal id,
    String firstname,
    String lastname,
    String addr,
    double salary)
throws SQLException;

public void removeemp (
    java.math.BigDecimal id)
throws SQLException;

public void changesalary (
    java.math.BigDecimal id,
    double newsalary[])
      throws SQLException;

public double getempinfo (
    java.math.BigDecimal id,
    String firstname[],
    String lastname[])
    throws SQLException;
}
```

You need to compile the interface file using the Oracle9*i*AS Web Services standard Java compiler, as shown:

```
javac CompanyInterface.java
```

This generates `CompanyInterface.class`. This class that needs to be added to the J2EE .war file to support the deployment of the new Stored Procedure Web Service.

> **See Also:** "Preparing Stored Procedure Web Services" on page 5-8

Writing a WSDL File or Client Stubs for Stored Procedure Web Services (Optional)

This step is optional when writing a Stored Procedure based Web Service. If you do not perform this step, the Oracle9*i*AS Web Services runtime generates a WSDL file and the client-side proxies for deployed Web Services. These files allow Oracle9*i*AS Web Services to supply a Web Service client with the WSDL or the client-side proxies that a client-side developer can use to build an application that uses a Web Service.

When you do not want to use the Oracle9*i*AS Web Services generated WSDL file or client-side proxies, perform the following steps:

1. Manually create either the WSDL file or the client-side proxy Jar file, or both files for your service.
2. Name the supplied WSDL file or client-side proxy Jar file and place it in the appropriate location. The WSDL file must have a `.wsdl` extension. The client-side proxy Jar must have an `_proxy.jar` extension.The extension is placed after the service name.

   For example,

   ```
   simpleservice.wsdl
   simpleservice_proxy.jar
   ```

3. Add the manually created WSDL file or client-side Jar file to the J2EE .war file that contains the service implementation. There are several choices for adding the files to the .war file:

   - Place the files in the correct package directory under `WEB-INF/classes`. For example, if `simpleservice.wsdl` describes a service that is part of the demo package, place the file in the following directory:

     ```
     WEB-INF/classes/demo/simpleservice.wsdl
     ```

- Place the Jar file in a Jar file under the `WEB-INF/lib` component of the .war file that is added to the .ear file and deployed as a Web Service.

  For example, the file `simpeservice_proxy.jar` could be added to `simpleservice.jar` and placed in `WEB-INF/lib`, or added to *`namex`*`.jar` in `WEB-INF/lib` (where *`namex`* is a file name).

Note: The Jar file containing the proxy stubs, and optionally the proxy source must be a Jar component of a Jar file in `WEB-INF/lib`, it cannot simply be added to the `WEB-INF/lib` directory.

See Also:

- "Preparing Stored Procedure Web Services" on page 5-8
- "Getting WSDL Descriptions and Client-Side Proxy Jar for Web Services" on page 6-2

Preparing Stored Procedure Web Services

This section describes the procedures you use to prepare a PL/SQL procedure as a Stored Procedure Web Service. The Jar file generated using `ant` must be packaged with an appropriate interface class and `web.xml` file into a .war file.

After the PL/SQL package, the JPublisher generated Jar file and the interface file are available, you need to create a J2EE .war file that includes the configuration file, `web.xml`, and the generated Jar file. The Stored Procedure .war file is then assembled into an application .ear file to be deployed into Oracle9*i*AS Containers for J2EE (OC4J).

This section contains the following topics:

- Modifying web.xml To Support Stored Procedure Web Services
- Preparing a WAR File for Stored Procedure Web Services
- Preparing an EAR File for Stored Procedure Web Services
- Setting Up Datasources in Oracle9iAS Web Services (OC4J)

The Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9*i*AS Web Services. The Web Services assembly tool takes a configuration file which describes the location of the Java classes, Jar files, and interface files and produces a J2EE .ear file that can be deployed under Oracle9*i*AS Web Services. This section describes how to assemble Oracle9*i*AS Web Services implemented as PL/SQL Stored Procedures manually, without using the assembly tool, `WebServicesAssembler`.

See Also: "Running the Web Services Assembly Tool" on page 7-2

Modifying web.xml To Support Stored Procedure Web Services

To use Stored Procedure Web Services you need to add a `<servlet>` entry and a corresponding `<servlet-mapping>` entry in the `web.xml` file for each PL/SQL package that is deployed as a Web Service. The resulting `web.xml` file is assembled as part of a .war file that is included in the .ear file that defines the Web Service.

To modify `web.xml` to support Web Services implemented as Stored Procedures, perform the following steps:

- Configure the servlet Tag for Stored Procedure Web Services
- Configure the servlet-mapping Tag for Stored Procedure Web Services
- Configure the resource-ref Tag for Stored Procedure Web Services

Configure the servlet Tag for Stored Procedure Web Services

To add Web Services based on PL/SQL Stored Procedures you need to modify the `<servlet>` tag in the `web.xml` file. This tag supports using the Oracle9*i*AS Web Services Servlet to access the PL/SQL Stored Procedure implementation for the Web Service. Table 5–2 describes the `<servlet>` tag and the values to include in the tag to add a Web Service based on a PL/SQL Stored Procedure.

Example 5–2 shows the <servlet> tag for the sample Stored Procedure Web Service extracted from the `web.xml` file (to view this file, expand `spexample.ear`).

Table 5–2 Servlet Tags Supporting PL/SQL Stored Procedure Deployment

Servlet Tag	Description
`<init-param>`	The `<init-param>` tag contains a name value pair within `<param-name>` and `<param-value>` tags. **`class-name:`** The Stored Procedure Web Services Servlet definition requires at least one `<param-name>` with the value **`class-name`** and a corresponding `<param-value>`. The `<param-value>` specifies the fully qualified name of the Java class that accesses the PL/SQL Web Service implementation. You need to supply the class name for this parameter; you can find the class name in the Jar file generated using the `ant command`. See Also: "Generating a Stored Procedure Web Service Jar File" on page 5-4 **`interface-name:`** A `<param-name>` with the value **`interface-name`**, and a corresponding `<param-value>` set to the fully qualified name of the Java interface specifies the methods to include in the stored procedure Web Service. This init parameter tells the Web Service Servlet generation code which methods should be exposed as Web Services. **`datasource-JNDI-name:`** The Stored Procedure Web Services Servlet definition requires at least one `<param-name>` with the value **`datasource-JNDI-name`**, and a corresponding `<param-value>` set to the JNDI name of the backend database. The `data-sources.xml` OC4J config file describes the database server source.
`<servlet-class>`	This is always `oracle.j2ee.ws.StatelessStoredProcRpcWebService` for all stateless stored procedure Web Services.
`<servlet-name>`	This tag specifies the name for the Servlet that runs the Web Service.

Example 5–2 Sample Stored Procedure <servlet> Entry for a Web Service

```
<servlet>
  <servlet-name>stateless Stored Procedure web service example</servlet-name>
  <servlet-class>oracle.j2ee.ws.StatelessStoredProcRpcWebService</servlet-class>
  <init-param>
    <param-name>class-name</param-name>
    <param-value>sp.company.Company</param-value>
  </init-param>
  <init-param>
    <param-name>datasource-JNDI-name</param-name>
    <param-value>jdbc/OracleCoreDS</param-value>
  </init-param>
  <init-param>
    <param-name>interface-name</param-name>
    <param-value>CompanyInterface</param-value>
  </init-param>
</servlet>
```

Configure the servlet-mapping Tag for Stored Procedure Web Services

To add Web Services based on PL/SQL Stored Procedures you need to modify the `<servlet-mapping>` tag in the `web.xml` file. This tag specifies the URL for the Servlet that implements a Web Service.

Example 5–3 shows a sample `<servlet-mapping>` entry corresponding to the servlet entry shown in Example 5–2.

Example 5–3 Sample <servlet-mapping> Entry for Web Services

```
<servlet-mapping>
  <servlet-name>stateless Stored Procedure web service example</servlet-name>
  <url-pattern>/statelessSP</url-pattern>
</servlet-mapping>
```

Configure the resource-ref Tag for Stored Procedure Web Services

To add Web Services based on PL/SQL Stored Procedures you need to modify the `<resource-ref>` tag in the `web.xml` file. This tag specifies the data source for running the PL/SQL Stored Procedure.

Example 5–4 shows a sample `<resource-ref>` entry corresponding to the `<servlet>` tag's `datasource-JNDI-name` parameter shown in Example 5–2.

Example 5–4 Sample <resource-ref> Entry for Web Services

```
<resource-ref>
   <res-ref-name>jdbc/OracleCoreDS</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Application</res-auth>
</resource-ref>
```

Preparing a WAR File for Stored Procedure Web Services

Stored Procedure Web Services use a standard .war file to define J2EE Servlet configuration and deployment information. After modifying the `web.xml` file in the `WEB-INF` directory to support the Stored Procedure Web Service, add the implementation classes and any required support classes or Jar files either under `WEB-INF/classes`, or under `WEB-INF/lib` (or in a classpath location available to OC4J). Be sure to add the compiled interface classes and the generated Jar file to support the Stored Procedure Web Service.

Note: All the classes the are required for a Stored Procedure Web Service implementation must conform to the standard J2EE class loading norms. Thus, the implementation classes and support classes must either be in the .war or .ear file, or they must be available in the OC4J classpath.

See Also:

- "Generating a Stored Procedure Web Service Jar File" on page 5-4
- "Creating and Compiling Stored Procedure Web Service Interfaces" on page 5-5

Preparing an EAR File for Stored Procedure Web Services

To add Web Services based on PL/SQL Stored Procedures you need to include an `application.xml` file and package the `application.xml` and .war file containing the interface class and generated Jar file into a J2EE .ear file.

The Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9*i*AS Web Services.

See Also: "Running the Web Services Assembly Tool" on page 7-2

Setting Up Datasources in Oracle9*i*AS 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 Oracle9*i*AS 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 Oracle9*i*AS 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 .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 an .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

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)

6. User-defined Object Types.

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. 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.
4. NCHAR and related types are not supported.
5. JPublisher does not support internationalization.
6. JPublisher and Oracle9*i*AS 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.

7. 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.

8. 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.

6 Building Clients that Use Web Services

This chapter describes the Oracle9*i*AS Web Services features that allow you to easily create and run a client application that uses Oracle9*i*AS Web Services.

This chapter contains the following topics:

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 8, "Discovering and Publishing Web Services" provides detailed information on looking up Web Services in a UDDI registry.

Using a static client Oracle9*i*AS Web Services provides several options for locating Oracle9*i*AS Web Services, including:

- Using a known Web Service located at a known URL.
- Using Oracle9*i*AS 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 Oracle9*i*AS Web Services generates includes the URL to locate the associated Web Service.
- Using Oracle9*i*AS 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 Oracle9*i*AS 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 8, "Discovering and Publishing Web Services"

Getting WSDL Descriptions and Client-Side Proxy Jar for Web Services

Oracle9*i*AS Web Services supplies client-side programmers with the following files for deployed Web Services:

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

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 6–1 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 6–1 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 6–1 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 with a Specified 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 6–1 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 6–1 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`*`proxy_src.jar` is a

Jar file that contains the client-side source files for the client-side proxy that accesses the Web Service.

Table 6–1 URL for Accessing Client Side Proxy Stubs

URL Component	Description
context-root	The context-root is the value specified in the `<context-root>` tag for the web module associated with the Web Service. See the `META-INF/application.xml` in the Web Service's .ear file to determine this value.
host	This is the host of the Web Service's server running Oracle9*i*AS Web Services.
mypackage	This specifies the value that you want to use for the package name in the generated proxy Jar or proxy source.
port	This is the port of the Web Service's server running Oracle9*i*AS Web Services.
service	The service is the value specified in the `<url-pattern>` tag for the servlet associated with the Web Service. This is the service name. See the `WEB-INF/web.xml` in the Web Service's .war 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"

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 on a system with Oracle9*i*AS. When building client-side applications to access and use services under Oracle9*i*AS Web Services, the Oracle9*i*AS Web Services client-side proxy Jar class allows you to easily build the application.

The client side proxy Jar file that you get from a Web Service contains a Java class to serve as a proxy to the Web Service implementation residing on the Oracle9*i*AS Web Services server. 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 6–1 shows a source code sample client-side proxy extracted from a Web Service. For each method 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 6–2 shows client-side application code that uses the `helloWorld()` method from the supplied client-side proxy shown in Example 6–1.

Example 6–1   Sample Client-side Proxy Method for Web Services

```
/**
 * Web service proxy: StatefulExample
 *     generated by Oracle WSDL toolkit (Version: 1.0). */
public class StatefulExampleProxy {


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

Example 6–2   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());
  }
}
```

Using Web Services Security Features

When you run a client-side application that uses Oracle9*i*AS Web Services, you can access secure Web Services by setting properties in the client application. Table 6–2 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 6–2 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.setProperties()` to add properties). In addition, the client side stubs include the `_setTranportProperties` 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 6–2 Web Services HTTP Transport Security Properties

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

Table 6–2 (Cont.) Web Services HTTP Transport Security Properties

Property	Description
`http.username`	Specifies the HTTP authentication username.
`java.protocol.handler.pkgs`	Specifies a list of package prefixes for `java.net.URLStreamHandlerFactory` The prefixes should be separated by `"|"` vertical bar characters. This value should contain: `HTTPClient` This value is required by the Java protocol handler framework; it is not defined by Oracle9*i* Application Server. This property must be set when using HTTPS. If this property is not set using HTTPS, a `java.net.MalformedURLException` is thrown. **Note:** This property must be set as a system property. For example, set this property as shown in either of the following: ■ `java.protocol.handler.pkgs=HTTPClient` ■ `java.protocol.handler.pkgs=sun.net.www.protocol|HTTPClient`
`oracle.soap.transport.allowUserInteraction`	Specifies the allows user interaction parameter. The case of the value specified is ignored. When this property is set to `true` and either of the following are true, the user is prompted for a username and password: 1. If any of properties `http.authType`, `http.username`, or `http.password` is not set, and a `401` HTTP status is returned by the HTTP server. 2. If either of properties `http.proxyAuthType`, `http.proxyUsername`, or `http.proxyPassword` is not set and a `407` HTTP response is returned by the HTTP proxy. Valid values: `true`, `false` Specifying any value other than `true` is considered as `false`.
`oracle.ssl.ciphers`	Specifies a list of `:` separated cipher suites that are enabled. Default value: The list of all cipher suites supported with Oracle SSL.
`oracle.wallet.location`	Specifies the location of an exported Oracle wallet or exported trustpoints. Note: The value used is not a URL but a file location, for example: `/etc/ORACLE/Wallets/system1/exported_wallet` (on UNIX) `d:\oracle\system1\exported_wallet` (on Windows) This property must be set when HTTPS is used with SSL authentication, server or mutual, as the transport.
`oracle.wallet.password`	Specifies the password of an exported wallet. Setting this property is required when HTTPS is used with client, mutual authentication as the transport.

Working with WSDL and JDeveloper to Use Web Services

The Web Services WSDL allows you to manually, or using Oracle9*i* JDeveloper or another IDE, build client applications that use Web Services.

The Oracle9*i* JDeveloper IDE supports Oracle9*i*AS Web Services with WSDL features and provides unparalleled productivity for building end-to-end J2EE and integrated Web Services applications.

JDeveloper supports Oracle9*i*AS 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 Oracle9*i*AS Web Services. Currently, many IDEs have the capability to create SOAP requests, given a WSDL description for the service.

7 Web Services Assembly Tool

The Oracle9*i*AS Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle9*i*AS 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 Oracle9*i*AS Web Services. The assembly tool assists with assembling the following types of Web Services:

- Stateless Java (including PL/SQL Stored Procedures and Functions)
- Stateful Java
- Stateless Session EJB

This chapter contains the following topics:

Running the Web Services Assembly Tool

Run the Web Services assembly tool as follows:

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

Where *file* is a Web Services assembly tool configuration file.

See Also:

- Web Services Assembly Tool Configuration File Specification on page 7-2
- Web Services Assembly Tool Configuration File Sample on page 7-4

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 7–1 shows the Web Services Assembly Tool Configuration file DTD.

Example 7–1 Assembly Tool Input File DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Specify the properties of the web services to be assembled. -->
<!ELEMENT web-service (display-name?, description?, destination-path,
temporary-directory, context, stateful-java-service*, stateless-java-service*,
stateless-session-ejb-service*)>
<!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 direcotry where the assembly tool can create temporary
directories and files. -->
<!ELEMENT temporary-directory (#PCDATA)>
<!-- Specify the context root of the web services. For example, "/webservies".
-->
<!ELEMENT context (#PCDATA)>
<!-- Specify the properties of a stateful Java service -->
<!ELEMENT stateful-java-service (interface-name?, class-name, uri,
java-resource*, ejb-resource*)>
<!-- Specify the properties of a stateless Java service -->
<!ELEMENT stateless-java-service (interface-name?, class-name, uri,
database-jndi-name?, java-resource*, ejb-resource*)>
<!-- Specify the properties of a stateless session ejb service -->
```

```
<!ELEMENT stateless-session-ejb-service (path, uri, ejb-name, ejb-resoruce*)>
<!-- 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 your 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)>
```

Web Services Assembly Tool Configuration File Sample

The sample configuration file shown in Example 7–2 defines two services to be wrapped in an Enterprise ARchive file (EAR). Table 7–1 describes the sample configuration file tags.

Example 7–2 Sample Web Services Assembly Tool Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE application-server PUBLIC "Orion Application Server Config"
"http://xmlns.oracle.com/ias/dtds/webservices-assembler.dtd">
<web-service>
    <display-name>Web Services Demo</display-name>
    <description>This is a demo.</description>
    <destination-path>/tmp/work/ws_example.ear</destination-path>
    <temporary-directory>/tmp</temporary-directory>
    <context>/webservices</context>
    <stateless-java-service>
        <interface-name>StatelessExample</interface-name>
        <class-name>StatelessExampleImpl</class-name>
        <uri>/statelessTest</uri>
        <java-resource>/tmp/work/java/</java-resource>
        <java-resource>/home/oracle/mylib.jar</java-resource>
    </stateless-java-service>

    <stateless-session-ejb-service>
        <path>/tmp/work/ejb/MyCart.jar</path>
        <uri>/statelessEjbTest</uri>
        <ejb-name>Cart</ejb-name>
        <ejb-resource-path>/tmp/work/ejb/ </ejb-resource-path>
    </stateless-session-ejb-service>
</web-service>
```

Table 7–1 Sample Web Services Configuration File Tag Components and Descriptions

Component	SubComponent	Description
web-services		Specifies the properties used to assemble the Web Service.
	display-name	Specifies the display name of the Web Services.
	description	A simple description of the Web Services.
	destination-path	Specifies that the resulting EAR will be stored as /tmp/work/ws_example.ear
	temporary-directory	Specifies a temporary directory that ws_assembler can store temporary files.
	context	Specifies the context root of the Web Service is called "/webservices".
stateful-java-service		
stateless-java-service		Specifies the properties of the stateless Java Web Service.
	interface-name	(optional) Specifies the interface class that defines methods to be exposed on the Web Service.
	class-name	Specifies the Java class to be exposed on the Web Service.
	uri	Specifies the URI of the service is /statelessTest. This URI is used in the URL to access the WSDL and client jar, and invoke the Web Service.
	web-resource-path	Specifies "/tmp/work/java/" as the directory that contains all the Java classes and jar libraries that the Java class depends on.
stateless-session-ejb-service		Specifies the properties of the stateless session ejb service.
	path	Specifies the location of the ejb jar file to be exposed as a Web Service.
	uri	Specifies the URI of the service is /statelessEjbTest. This URI is used in the URL to access the WSDL and client jar, and invoke the Web Service.
	ejb-name	Specifies the name of the stateless session EJB.
	ejb-resource-path	(optional) Specifies "/tmp/work/ejb/" as the directory that contains all the EJB jars that the stateless session ejb depends on

Web Services Assembly Tool Configuration File Sample Output

After running the Web Services Assembly tool with the sample input file shown in Example 7–2, the generated output is an EAR file (`/tmp/work/ws_example.ear`) The generated EAR file, `ws_example.ear`, has the structure shown in Example 7–3.

Example 7–3 Structure of Web Services Assembly Tool Sample Ear File

```
+------ MyCart.jar
+------ META-INF/
|       `------ application.xml
`------ ws_example.war
        +------ index.html
        `------ WEB-INF/
                +------- web.xml
                +------- lib/
                |        +------ mylib.jar
                `------- classes/
                         +------ StatelessExample.class
                         `------ StatelessExampleImpl.class
```

Example 7–4 shows the sample `application.xml` file.

Example 7–4 Web Services Assembly Tool Sample application.xml File

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_2.dtd">
<application>
    <display-name>Web Services - ws_example</display-name>
    <module>
        <web>
            <web-uri>ws_example</web-uri>
            <context-root>/webservices</context-root>
        </web>
    </module>
    <module>
        <ejb>MyCart.jar</ejb>
    </module>
</application>
```

```
The web.xml looks as follows --
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<servlet>
    <servlet-name>stateless java web service - statelessTest</servlet-name>
    <servlet-class>oracle.j2ee.ws.StatelessJavaRpcWebService</servlet-class>
    <init-param>
        <param-name>class-name</param-name>
        <param-value>StatelessExampleImpl</param-value>
    </init-param>
    <init-param>
        <param-name>interface-name</param-name>
        <param-value>StatelessExample</param-value>
    </init-param>
</servlet>
<servlet>
    <servlet-name>stateless session bean web service - Cart</servlet-name>
    <servlet-class>oracle.j2ee.ws.SessionBeanWebService</servlet-class>
    <init-param>
        <param-name>jndi-name</param-name>
        <param-value>ejb/Cart</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>stateless java web service - statelessTest</servlet-name>
    <url-pattern>/statelessTest</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>stateless session bean web service - Cart</servlet-name>
    <url-pattern>/statelessEjbTest</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
<ejb-ref>
    <ejb-ref-name>ejb/Cart</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>CartHome</home>
    <remote>Cart</remote>
    <ejb-link>Cart</ejb-link>
</ejb-ref>
</web-app>
```

Web Services Assembly Tool Limitations

The `WebServicesAssembler` tool has the following limitations:

- Upload / download: this tool is designed for use as a server side tool. 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, or welcome files or perform other administrative tasks.
- Does not modify an existing EAR file: the Web Services Assembly tool cannot take a EAR file as input, parse the contents and then return a new EAR file or modify the existing EAR file.

8 Discovering and Publishing Web Services

Oracle9*i*AS Containers for Java2 Enterprise Edition (J2EE), or OC4J, provides a Universal Discovery Description and Integration (UDDI) Web Services registry in which Web Service provider administrators in an enterprise environment can publish their Web Services for use by Web Service consumers (programmers). Web Service 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 Service provider administrator working with programmers who have completed a Web Service implementation using the J2EE stack (either EJBs, javabeans, JSPs, or servlets) and exposing the implementation as a Simple Open Access Protocol (SOAP)-based Web Service, can publish the Web Service by providing all the metadata and pointers to the interface specification in the UDDI registry. In this way, the Web Service provider administrator publishes the availability of these Web Services for the Web Service consumer or programmer 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 taxonomies, such as the NAICS, ISO3166, and UNSPSC classification systems.

- 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 Service description format and Web Service discovery protocol; a UDDI registry can contain metadata for any type of service, with best practices already defined for those described by Web Service 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 that contains descriptive business information about the business that publishes information about a Web Service, such as information about its 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, it contains descriptive business service information about a particular family of technical services including its name, brief description, technical service description information, and category information that is useful for performing searches
- bindingTemplate -- the logical child of a single businessService data structure, it contains technical information about a Web Service entry point and references to interface specifications.
- tModel -- descriptions of specifications for Web Services or taxonomies that form the basis for technical fingerprints; its role is to represent the technical specification of the Web Service, in order to facilitate Web Service consumers 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 Service consumers can easily identify other compatible Web Services.

Figure 8–1 UDDI Information Model Showing the Relationship Among the Four Main Data Structure Types

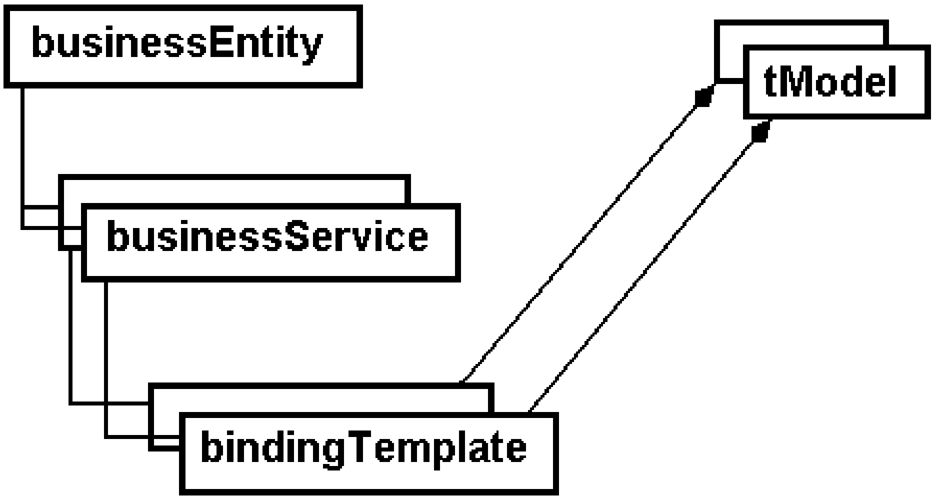


Figure 8–1 shows the UDDI information model and the relationships among its four data structure types.

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 the *UDDI Data Structure Reference V1.0 Open Draft Specification 30 September 2000* and the *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 Service Registry

This section describes a subset of features that provide UDDI support for Web Services deployed in OC4J as the Oracle 9*i*AS 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 comprised of two parts:

- Web Service discovery -- programmers can use the Inquiry API available for Java programmers to implement their own Web Service 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 Service registry. See Using the Inquiry API for more information about using the Inquiry API and locating the Javadoc documentation that describes the Inquiry API.
- Web Service publishing -- Web Service provider administrators can publish OC4J Web Services into the enterprise-wide Oracle UDDI Web Service registry using the **Application Server: iAS: OC4J home: Deployed Applications: Deploy Application Wizard** provided through Oracle Enterprise Manager (OEM). 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 Service servlets contained in the EAR file) to the Oracle UDDI registry.

  Web Service 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 OEM.

Web Service Discovery

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

Using Tools

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

Using the Inquiry API

The Inquiry API lets programmers search for the available registered Web Services by providing find (browse and drill down) calls and a get calls for locating and getting information in each of the four data structures shown in Figure 8–1.

The inquiry API allows programmers to discover and use Web Services using the Java language. Programs can be written in any language and 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 the *`<ias-http-server-host-name>`* is where the Oracle HTTP Server powered by Apache is installed and the *`<ias-port-number>`* is the port number for the Oracle HTTP Server.

The Inquiry API is located in the Oracle9*i*AS 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 *`<ORACLE_HOME>`*`/ds/uddi/demo/demo.zip` for UNIX and *`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, which 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 alphabetic 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 taxonomy 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 free up resources.

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

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

Web Service Publishing

Using OEM, Web Service 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** steps 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 or the **Publish Web Services** step of the **Deploy Applications Wizard** lets Web Service provider administrators publish Web Services (servlets) that are found in your EAR file. Any Web Service 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 classification taxonomies provided. Any unpublished Web Service in an application appears with the status of `Not Published` and when the Web Service 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 you publish J2EE applications to the UDDI registry after entering all required Service Details and tModel Details information.

Web Service provider administrators can update the published Web Services they discover through the OEM 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 classification taxonomies:

- 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 Taxonomy (ISO 3166)

  This a list of all country names and each corresponding two character code element. For more information about ISO3166, see the Web site at: `http://www.din.de/gremien/nas/nabd/iso3166ma/`.

When you publish a Web Service, you can select the classification taxonomy and the category to which you want to register the Web Service. You have the option of publishing your Web Service to any or all three of these classification taxonomies and to as many categories and subcategories as you wish within each classification taxonomy.

> **See Also:** "Database Character Set and Built-in ISO-3166 Taxonomy" on page 8-13

Publishing a Web Service Using the OEM Deploy Applications Wizard

To publish a Web Service using the OEM **Deploy Applications Wizard**, 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 **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 Service you want to register from the list of Web Services known to your 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 Service or the tModel sections, click **Browse UDDI Registry** and browse to the desired classification taxonomy 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 taxonomy, 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 taxonomy, 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 you want to add all the categories to which you want to register this Web Service.
   g. After you have entered all the necessary information on the **Web Services Details** window and are ready to publish the Web Service 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 Service to publish and repeat this entire process again as described in Steps 3 and 4.
6. After you have published all the Web Services you want for this application, click **Next** to continue to the **Summary** window where you can review all the application deployment information.
7. If there are no further changes, click **Deploy** to deploy your J2EE application on the OC4J container and you will return to the OEM OC4J Home page where you can repeat the process of deploying another J2EE application on the OC4J container by clicking **Deploy Application**.

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

Publishing a Web Service Using the OEM Web Services Details Window

To publish a Web Service using the OEM **Web Services Details** window, 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 you select one of the three standard classification taxonomies: 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, you must 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 Service or the tModel sections, click **Browse UDDI Registry** and browse to the desired classification taxonomy 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 taxonomy, 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 taxonomy, 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 you want to add all the categories to which you want to register this Web Service.

   i. After you have entered all the required information on the **Web Services Details** window and are ready to publish the Web Service to the Oracle UDDI registry, click **Apply**. You return 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 a Published Web Service in the UDDI Registry

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

To update a published Web Service using OEM to discover the Web Service, 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 you select one of the three standard classification taxonomies: NAICS, UNSPSC, or ISO 3166 by clicking its link. The **UDDI Registry** window lets you browse any of the three classification taxonomies and discover published Web Services associated with any category or sub-category.

   Click the desired classification taxonomy link.

3. The **UDDI:** <*Classification Taxonomy Name*> window lets you drill down from category to sub-category to discover published Web Services associated with any category or sub-category. Each taxonomy is organized in a hierarchical tree of which you can navigate down a particular branch by clicking the category name to determine all its sub-category 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 sub-category by successively clicking the desired category links.

4. The **Web Services:** <*Category Name*> window lets you 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 Service listed for the selected category, its corresponding service name, service key, and business key are also listed. If no Web Services are published for a selected category or sub-category, none are listed.

   To view the complete details of a particular published Web Service 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 Service published in the Oracle UDDI Registry. This information includes:

   - Service Details

     Service details include information such as the Web Service name, Web Service description, and the URL of the Web Service access point.

       Category

       Category information includes the classification taxonomy and the corresponding code value and its category name.

   - tModel Details

     tModel details include information that describe the interface that the Web Service implements, such as the tModel name, tModel description, and URL to the interface specification, typically a WSDL document.

       Category

       Category information includes the classification taxonomy 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 Service. You can add categories (click **Add Category**) to which both this Web Service and tModel are to be registered. You can remove categories (click **Delete**) from which this Web Service 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 you have made all your selections or completed all changes for this Web Service, 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 just began your current session again.

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 Taxonomy Name*> link to return to the desired **Web Services:**<*Classification Taxonomy Name*> window. At this window, you can select another Web service 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 taxonomy to discover Web Services for other categories. At each desired category, select the desired Web Service to view its details, make any necessary changes, and finally click **Apply** to save your changes.

Database Character Set and Built-in ISO-3166 Taxonomy

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 intended to be 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 taxonomy ISO-3166 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 an UTF-8 database, you can upgrade the built-in ISO-3166 taxonomy to the one with accurate descriptions using the following instructions:

- Drop the existing ISO-3166 taxonomy by running the SQL script, `clrISO.sql`, for example:

  ```
  cd <ORACLE_HOME>/ds/uddi/admin
  sqlplus system/manager @clrISO.sql
  ```

- Load the ISO-3166 taxonomy with accurate descriptions by using SQL loader control file `iso3166-99.ctl`, for example:

  ```
  cd <ORACLE_HOME>/ds/uddi/admin
  sqlldr userid=system/manager control=iso3166-99.ctl
  ```

9 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

Oracle9*i*AS 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 Oracle9*i*AS.

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.

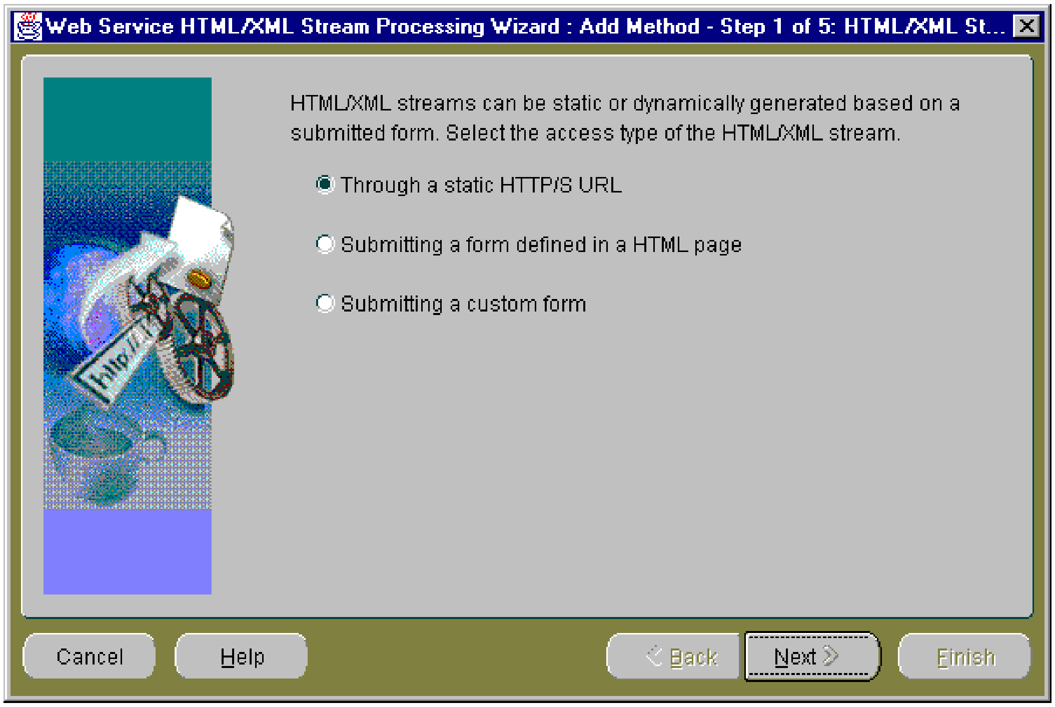
1. Invoke the Web Service HTML / XML Stream Processing Wizard using the following command:

   ```
   java -jar WebServicesHtmlXmlWizard.jar
   ```

Note: The WebServicesHtmlXmlWizard.jar file is located in your `<ORACLE_HOME>/j2ee/home` Oracle9*i*AS installation directory.

---

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 1 of 5: HTML/XML St...

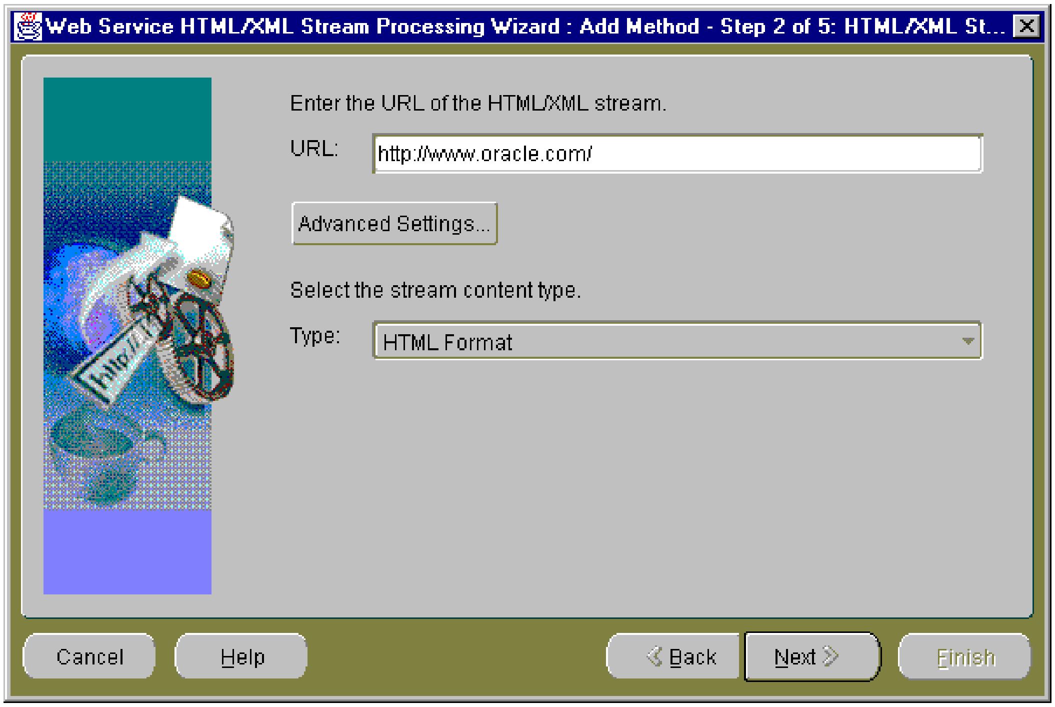
HTML/XML streams can be static or dynamically generated based on a submitted form. Select the access type of the HTML/XML stream.

[✓ Through a static HTTP/S URL]

Submitting a form defined in a HTML page

Submitting a custom form

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 2 of 5: HTML/XML St...

Enter the URL of the HTML/XML stream.

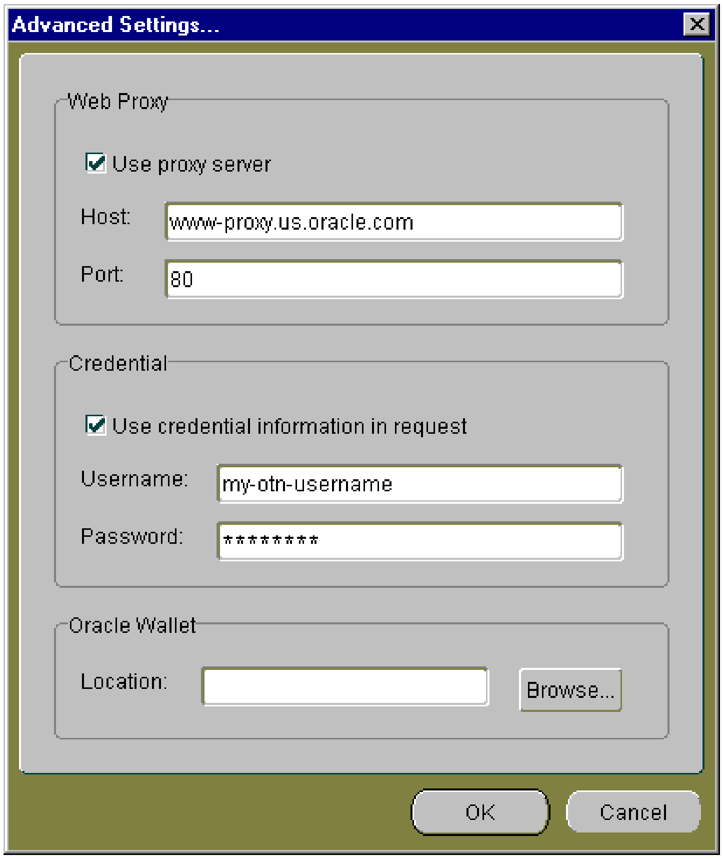
URL: http://www.oracle.com/

Select the stream content type.

Type: HTML Format

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.


Advanced Settings...

Web Proxy

✓ Use proxy server

Host: www-proxy.us.oracle.com

Port: 80

Credential

✓ Use credential information in request

Username: my-otn-username

Password: ********

Oracle Wallet

Location:

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.

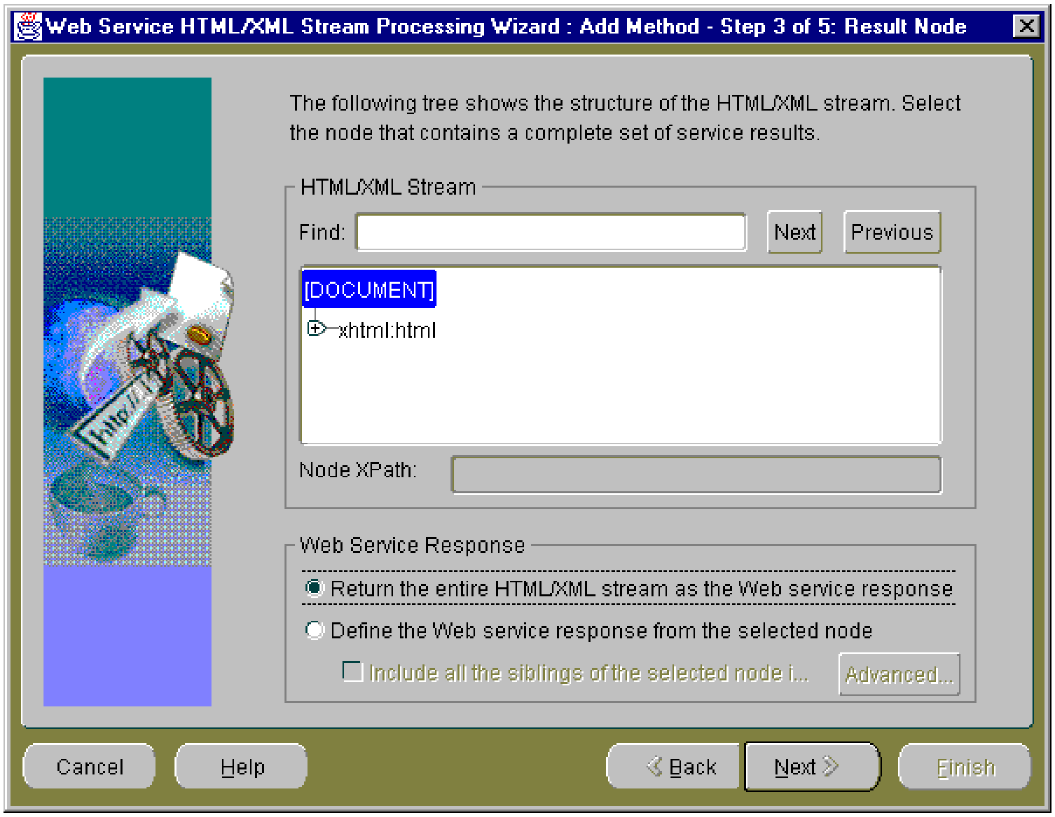
If the URL you are accessing requires basic HTTPS authentication, use the **Oracle Wallet** section of the **Advanced Settings** pop-up window to enter the Wallet location.

---

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 3 of 5: Result Node

The following tree shows the structure of the HTML/XML stream. Select the node that contains a complete set of service results.

HTML/XML Stream

Find:

[DOCUMENT]

xhtml:html

Node XPath:

Web Service Response

[✓ Return the entire HTML/XML stream as the Web service response]

Define the Web service response from the selected node

Include all the siblings of the selected node i...

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 9-24 for more information about the **Service Response Fields** window.

Click **Next** to continue to the next step.

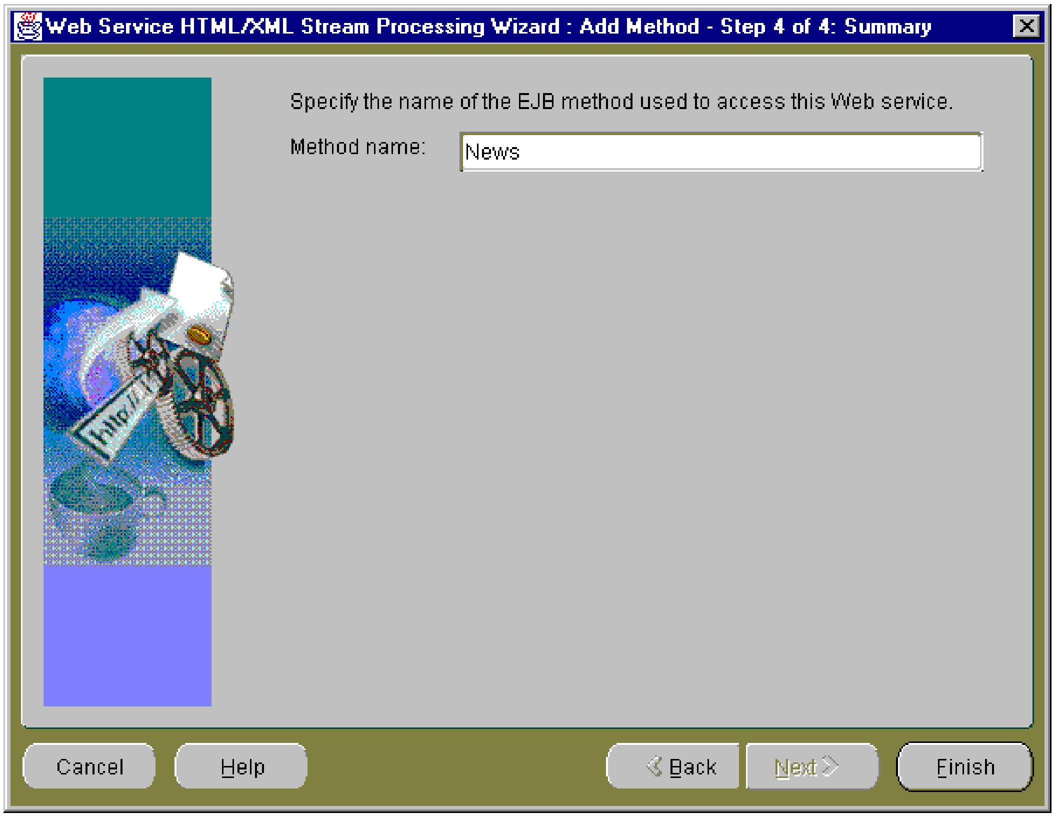
6. In **Step 4 of 4: Summary**, you must specify your EJB method name.

   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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 4 of 4: Summary

Specify the name of the EJB method used to access this Web service.

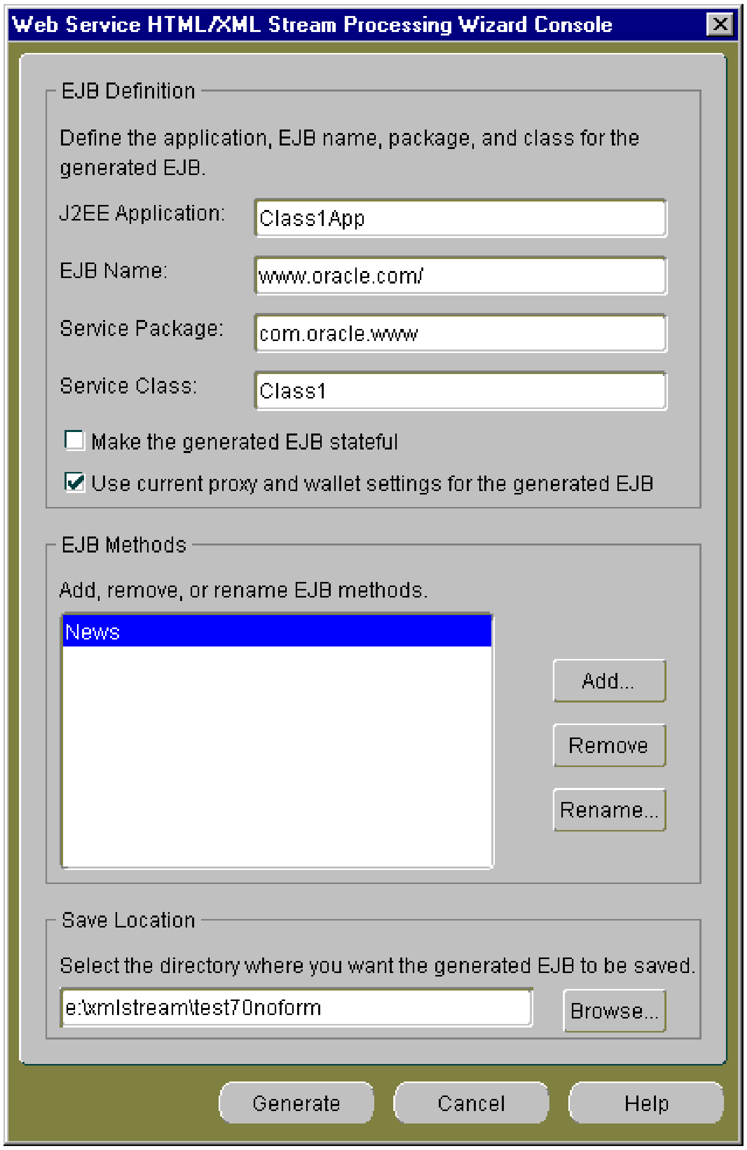
Method name: News

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.



EJB Definition

Define the application, EJB name, package, and class for the generated EJB.

J2EE Application: Class1App

EJB Name: www.oracle.com/

Service Package: com.oracle.www

Service Class: Class1

☐ Make the generated EJB stateful

☑ Use current proxy and wallet settings for the generated EJB

EJB Methods

Add, remove, or rename EJB methods.

News

Save Location

Select the directory where you want the generated EJB to be saved.

e:\xmlstream\test70noform

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 Oracle9*i*AS) 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

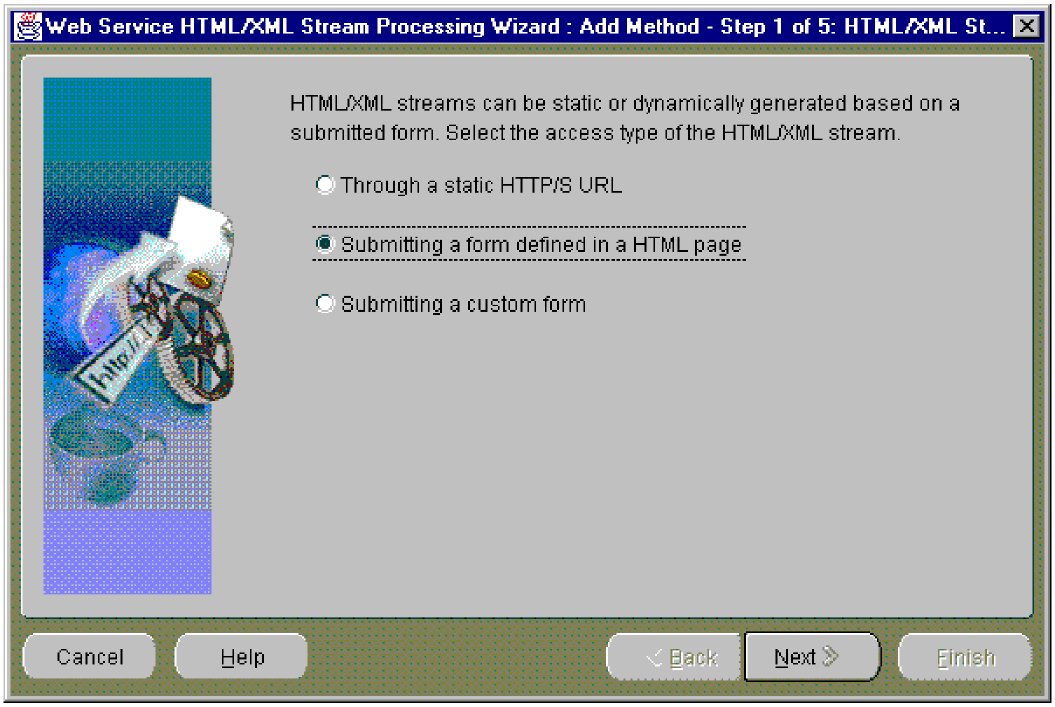
The following steps generate the Java stubs that consume a dynamic XML or HTML stream requiring a form to be submitted.

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>*`/j2ee/home` Oracle9*i*AS installation directory.

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 1 of 5: HTML/XML St...
http://

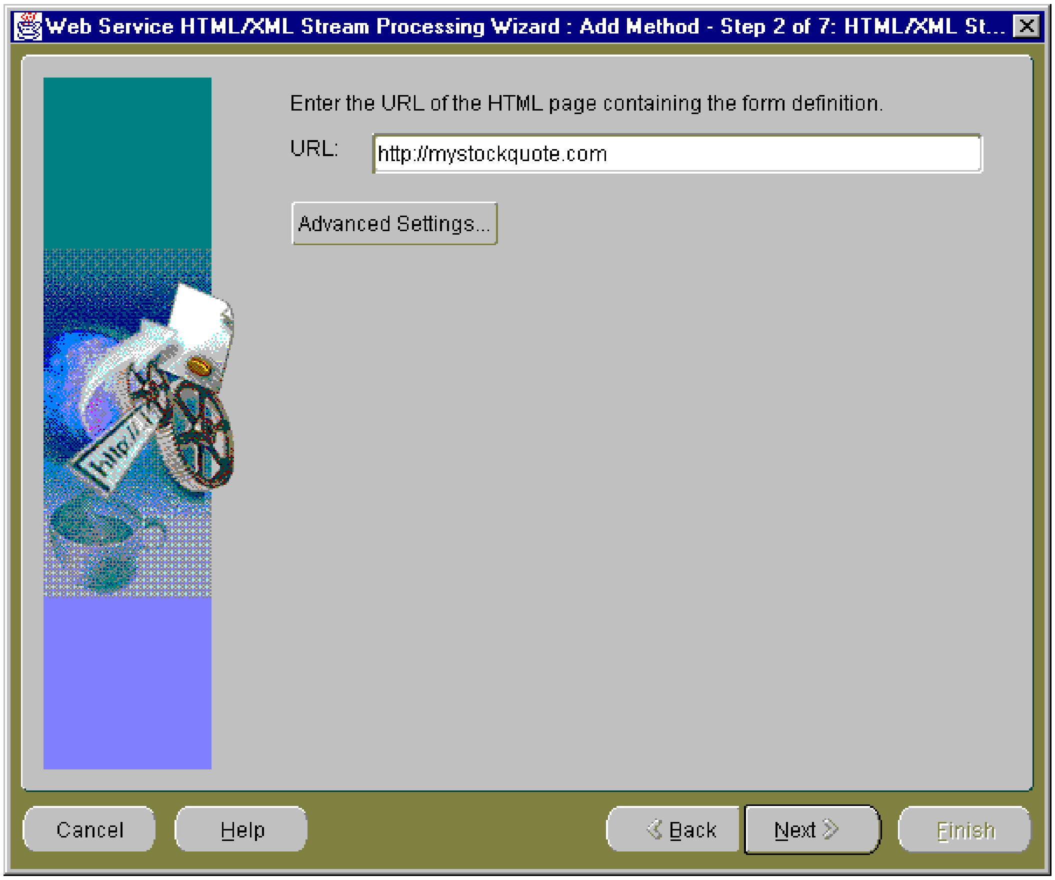
HTML/XML streams can be static or dynamically generated based on a submitted form. Select the access type of the HTML/XML stream.

( ) Through a static HTTP/S URL
[✓ Submitting a form defined in a HTML page]
( ) Submitting a custom form

Cancel
Help
< Back
Next >
Finish

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.

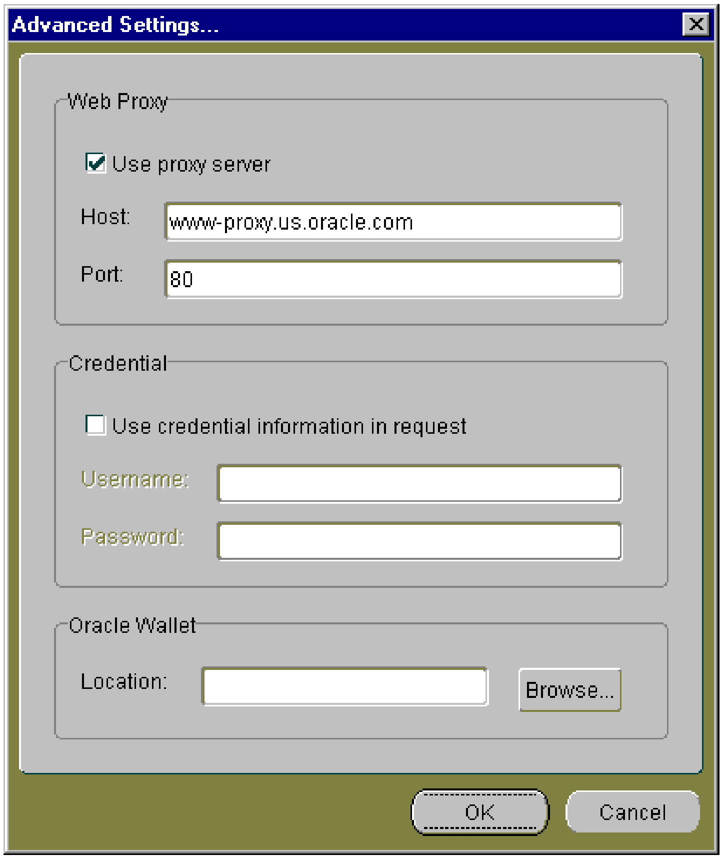

Web Service HTML/XML Stream Processing Wizard : Add Method - Step 2 of 7: HTML/XML St...

Enter the URL of the HTML page containing the form definition.

URL: http://mystockquote.com

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.


Advanced Settings...

Web Proxy

[✓] Use proxy server

Host: www-proxy.us.oracle.com

Port: 80

Credential

[ ] Use credential information in request

Username:

Password:

Oracle Wallet

Location:

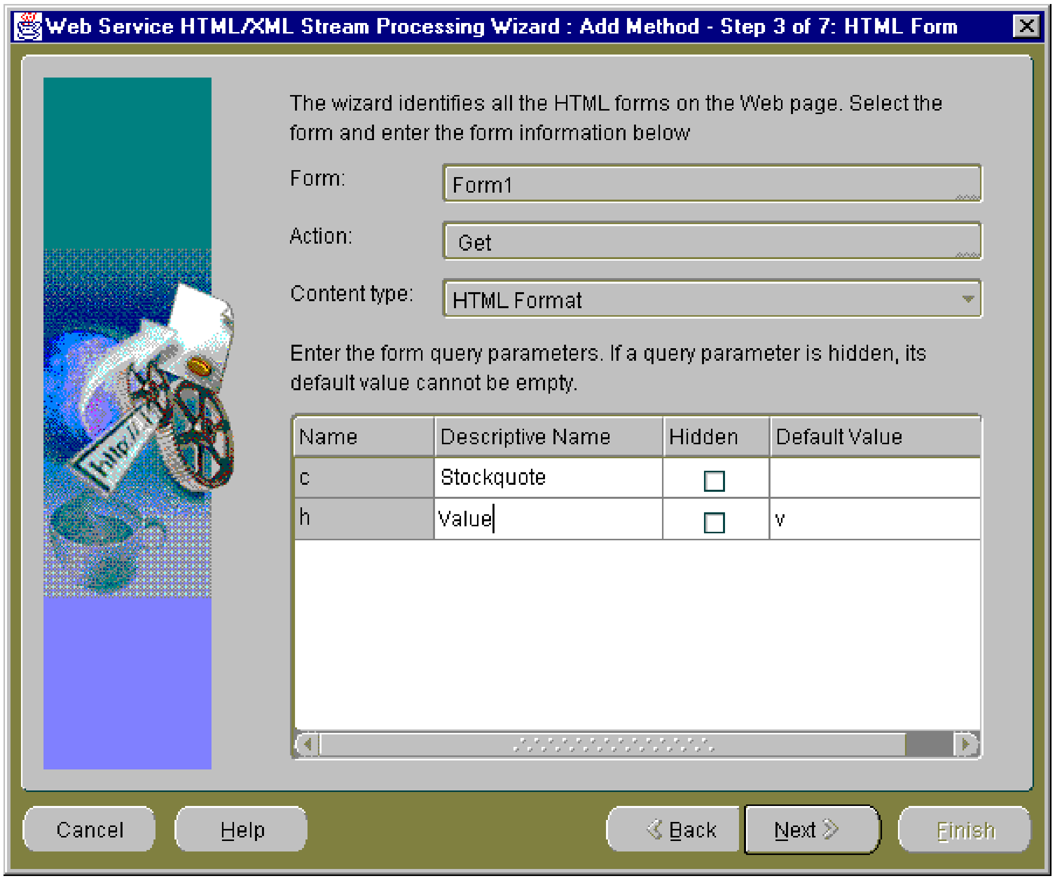
Note: If the URL you are accessing requires basic HTTP authentication, select **Use credential information in request**, then enter the user name and password in the **Credential** section of the **Advanced Settings** pop-up window.

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.



Web Service HTML/XML Stream Processing Wizard : Add Method - Step 3 of 7: HTML Form

The wizard identifies all the HTML forms on the Web page. Select the form and enter the form information below

Form:	Form1
Action:	Get
Content type:	HTML Format

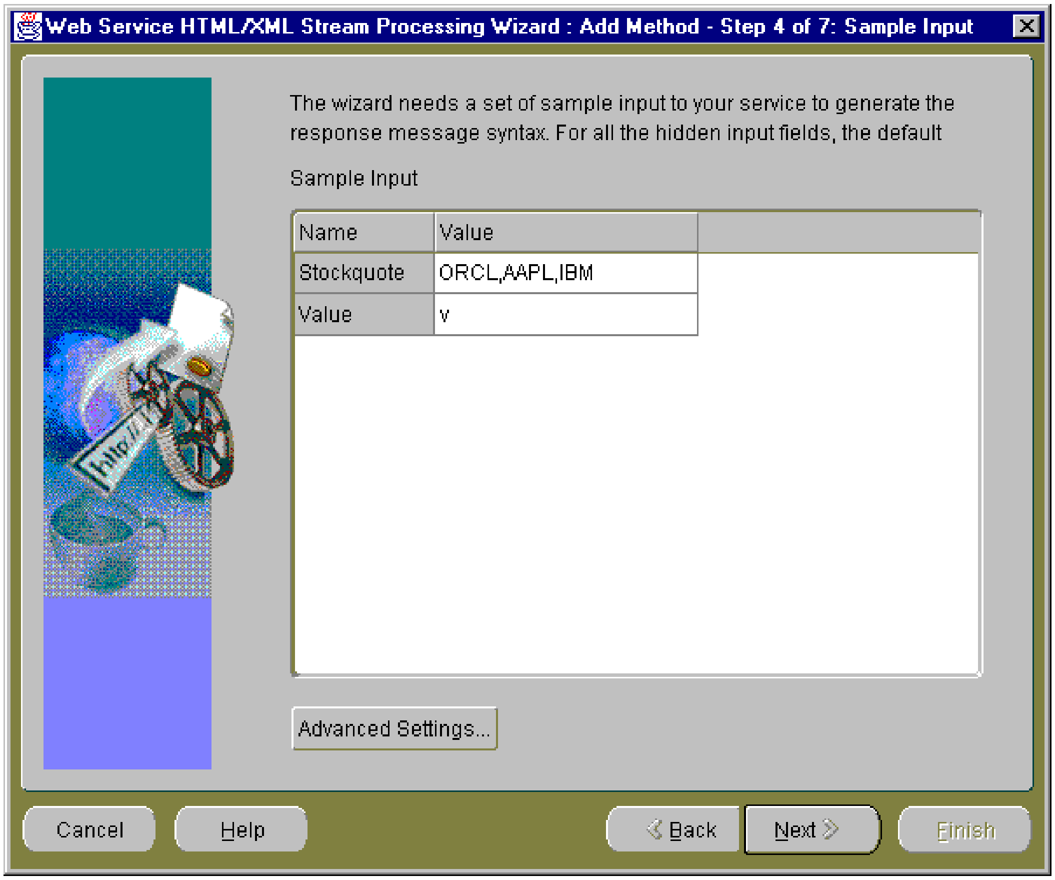
Enter the form query parameters. If a query parameter is hidden, its default value cannot be empty.

Name	Descriptive Name	Hidden	Default Value
c	Stockquote	☐	
h	Value	☐	v

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 4 of 7: Sample Input
Advanced Settings...
Cancel
Help
Back
Next
Finish

The wizard needs a set of sample input to your service to generate the response message syntax. For all the hidden input fields, the default

Sample Input

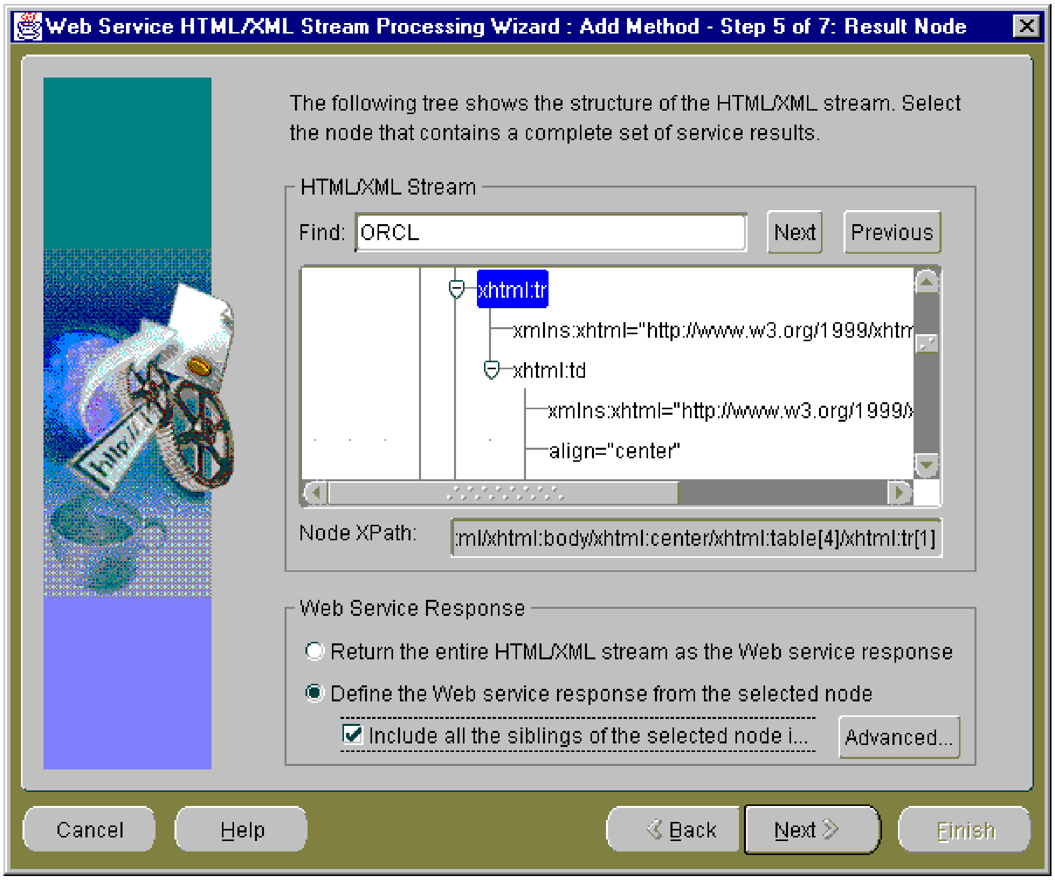
Name	Value
Stockquote	ORCL,AAPL,IBM
Value	v

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 5 of 7: Result Node

The following tree shows the structure of the HTML/XML stream. Select the node that contains a complete set of service results.

HTML/XML Stream

Find: ORCL

```
xhtml:tr
  xmlns:xhtml="http://www.w3.org/1999/xhtm
  xhtml:td
    xmlns:xhtml="http://www.w3.org/1999/x
    align="center"
```

Node XPath: :ml/xhtml:body/xhtml:center/xhtml:table[4]/xhtml:tr[1]

Web Service Response

- Return the entire HTML/XML stream as the Web service response
- [✓] Define the Web service response from the selected node
- [✓] Include all the siblings of the selected node i...

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.

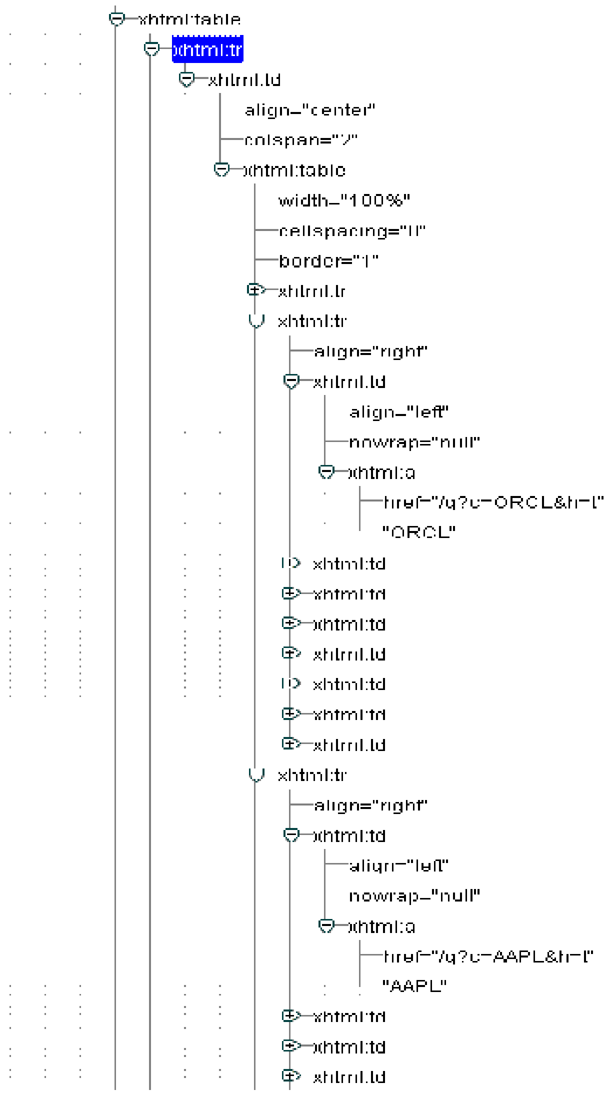
If you want to select just the node that contains the complete set of information you are interested in, select the second option **Define the Web service response from the selected node**. Then, navigate to the node you want by moving down the HTML / XML stream tree.

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.

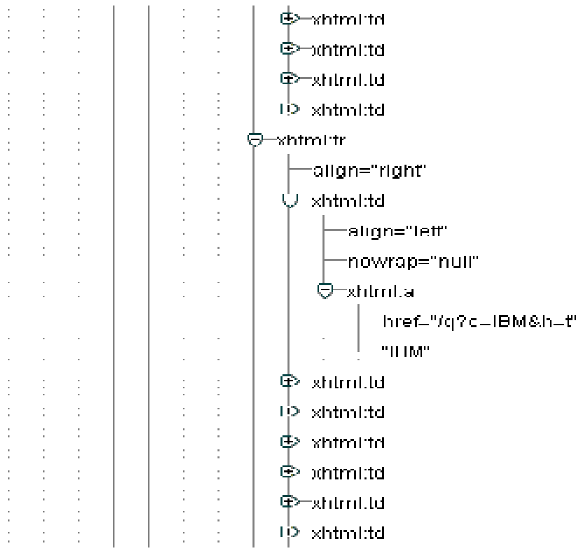
Figure 9–1 and Figure 9–2 together show an excerpt of what the xhtml tree would appear like when expanded. The selected node `xhtml:tr` is located in the next lower table node, which is one level lower and than the `xhtml:tr` nodes for ORCL and its two siblings AAPL and IBM.

Figure 9–1 Expanded xhtml Tree Showing the Selected Node of Interest Relative to the Nodes for ORCL and Sibling Nodes AAPL and IBM (Part1)



```
xhtml:table
  xhtml:tr
    xhtml:td
      align="center"
      colspan="2"
      xhtml:table
        width="100%"
        cellspacing="0"
        border="1"
        xhtml:tr
        xhtml:tr
          align="right"
          xhtml:td
            align="left"
            nowrap="null"
            xhtml:a
              href="/q?s=ORCL&d=t"
              "ORCL"
          xhtml:td
          xhtml:td
          xhtml:td
          xhtml:td
          xhtml:td
          xhtml:td
          xhtml:td
        xhtml:tr
          align="right"
          xhtml:td
            align="left"
            nowrap="null"
            xhtml:a
              href="/q?s=AAPL&d=t"
              "AAPL"
          xhtml:td
          xhtml:td
          xhtml:td
```

Figure 9–2 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.

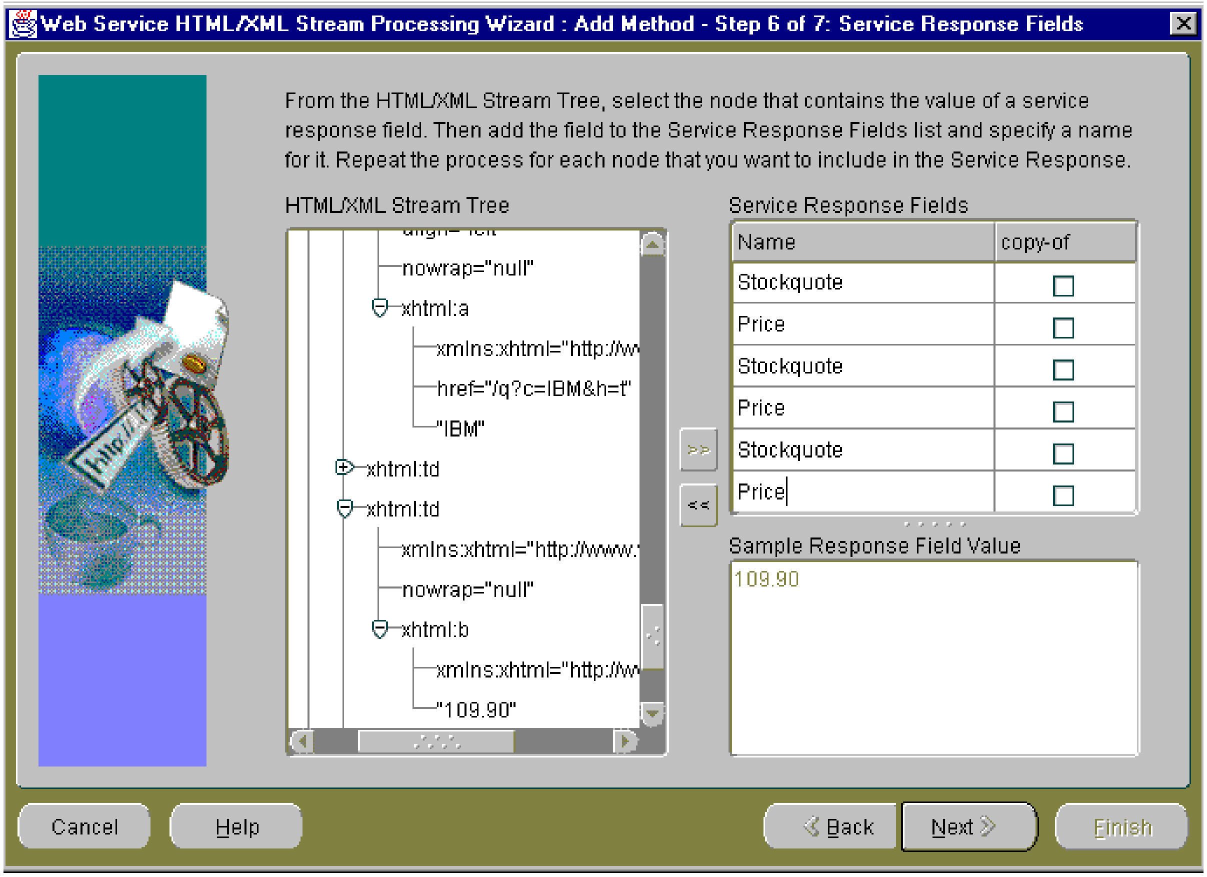
The following predicate filters out the first position: position() != 1. Enter this predicate expression in the **Predicate that filters the set of sibling nodes** field of the **Advanced Settings** pop-up window to filter the first sibling from the Web Service response.

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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 6 of 7: Service Response Fields

From the HTML/XML Stream Tree, select the node that contains the value of a service response field. Then add the field to the Service Response Fields list and specify a name for it. Repeat the process for each node that you want to include in the Service Response.

HTML/XML Stream Tree

```
nowrap="null"
xhtml:a
  xmlns:xhtml="http://w
  href="/q?c=IBM&h=t"
  "IBM"
xhtml:td
xhtml:td
  xmlns:xhtml="http://www.
  nowrap="null"
  xhtml:b
    xmlns:xhtml="http://w
    "109.90"
```

Service Response Fields

Name	copy-of
Stockquote	☐
Price	☐
Stockquote	☐
Price	☐
Stockquote	☐
Price	☐

Sample Response Field Value

109.90

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</a> 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</a> 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</a></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.

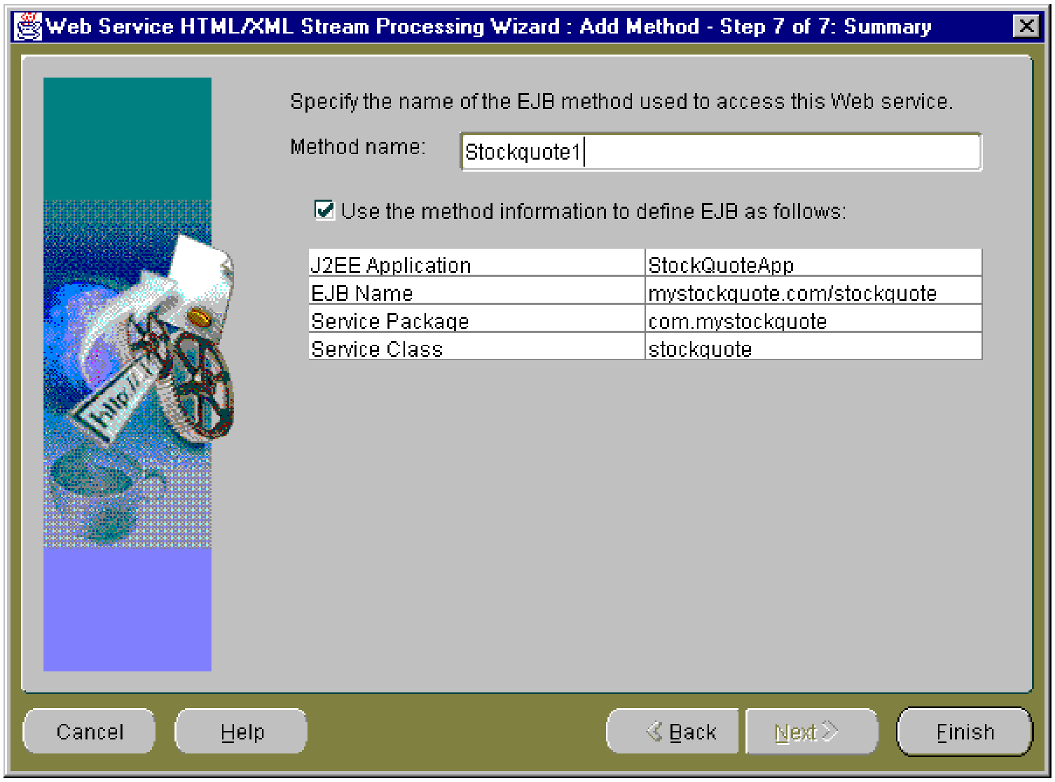
9. In **Step 7 of 7: Summary**, you must specify your EJB method name.

   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.


Web Service HTML/XML Stream Processing Wizard : Add Method - Step 7 of 7: Summary
Cancel
Help
Back
Next
Finish

Specify the name of the EJB method used to access this Web service.

Method name: Stockquote1

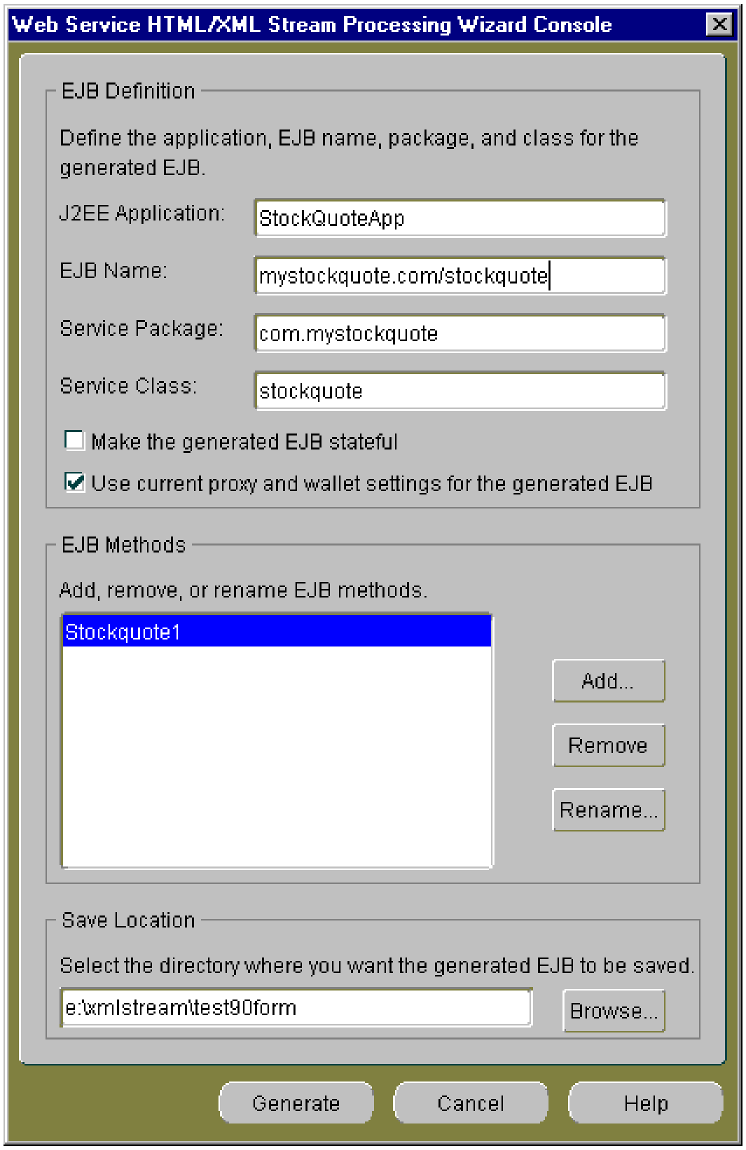
✓ Use the method information to define EJB as follows:

J2EE Application	StockQuoteApp
EJB Name	mystockquote.com/stockquote
Service Package	com.mystockquote
Service Class	stockquote

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.


Web Service HTML/XML Stream Processing Wizard Console

EJB Definition

Define the application, EJB name, package, and class for the generated EJB.

J2EE Application: StockQuoteApp

EJB Name: mystockquote.com/stockquote

Service Package: com.mystockquote

Service Class: stockquote

☐ Make the generated EJB stateful

☑ Use current proxy and wallet settings for the generated EJB

EJB Methods

Add, remove, or rename EJB methods.

Stockquote1

Save Location

Select the directory where you want the generated EJB to be saved.

e:\xmlstream\test90form

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 Oracle9*i*AS) 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 Oracle9*i*AS 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 9–1.

Table 9–1 wsdl2ejb Utility Command-Line Options

Option	Description
-conf *`<config file>`*	Allows the `wsdl2ejb` utility to load a configuration file.
-d *`<destDir>`*	Allows a destination directory to be specified where the generated EJB EAR file is to be written.
-Dhttp.proxyHost	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.
-Dhttp.proxyPort	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.
-jar	Allows you to specify the `wsdl2ejb` 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>`*`/j2ee/home` Oracle9*i*AS installation directory.

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 Oracle9*i*AS) 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 additional 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 WDSL 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. -->
<!ELEMENT ejb (application-name?, ejb-name?, package-name?, remote-name?, session-type?)>

<!-- Specify the name of the J2EE application for the generated EAR.  -->
<!ELEMENT application-name (#PCDATA)>

<!-- Specify the JNDI binding key name for the generated EJB.  -->
<!ELEMENT ejb-name (#PCDATA)>

<!-- Specify the name for Java package under which the generated EJB will belong. (for example, com.oracle)
-->
<!ELEMENT package-name (#PCDATA)>

<!-- Specify the class name for the EJB Remote Interface (for example, MyWsdlEjb)  -->
<!ELEMENT remote-name (#PCDATA)>
```

```
<!-- Specify the if the generated EJB should be stateless or stateful (for example, Stateless | Stateful)
-->
<!ELEMENT session-type (#PCDATA)>

<!--
    Specify the custom Java types and map them to XML types.
     The JAR attribute value will point to a JAR file containing the defintion of the custom
     types or the serializer/deserializer to be used for the custom type.
-->
<!ELEMENT mapTypes (map*)>
<!ATTLIST mapTypes
          jar           CDATA   #IMPLED>

!--
     Specify a new XML to JAR type map.
     EncodingStyle: name of the encodingStyle under which this map will belong
                    (for example, http://schemas.xmlsoap.org/soap/encoding/)
     namespace-uri      : uri of the namespace for the XML type defined in this map
     local-name         : localname of the XML type defined in this map
      java-type          : Java class name to which this type is mapped to (for example, com.org.MyBean)
     java2xml-class-name: Java class name of the type serializer
                          (for example, org.apache.soap.encoding.soapenc.BeanSerializer)
     xml2java-class-name: Java class name of the type deserializer
                          (for example, org.apache.soap.encoding.soapenc.BeanSerializer)
-->
<!ELEMENT map (#PCDATA)>
<!ATTLIST map
          encodingStyle       CDATA   #REQUIRED
          namespace-uri       CDATA   #REQUIRED
          local-name          CDATA   #REQUIRED
          java-type           CDATA   #REQUIRED
          java2xml-class-name CDATA   #REQUIRED
          xml2java-class-name CDATA   #REQUIRED>
```

Table 9–2 describes the elements, subelements, and attributes of the `wsdl2ejb` XML configuration file as defined in the DTD. Required elements and attributes are shown as **bold** text.

Table 9–2 Elements, Subelements, and Attributes of the wsdl2ejb XML Configuration File as Defined in the DTD

Element	Subelement	Attribute	Description
useProxy			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.
useWallet			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.
wsdl			Required element. Specifies how the `wsdl2ejb` utility should process the source WSDL document. Requires the location element be specified and optionally, either the service-name and service-port pair of elements or the service-binding and soap-location pair of elements be specified.
	location		Required element. Specifies the location of the source WSDL document. Can be a file path or an URL.
	service-name		Optional element. Specifies the name of the WSDL service to be used for the generated EJB. If specified, must be specified with the service-port element as a pair of elements.
	service-port		Optional element. Specifies the service port of the WSDL service to be used for the generated EJB. If specified, must be specified with the service-name element as a pair of elements.
	service-binding		Optional element. Specifies the name of the WSDL binding to be used for the generated EJB. If specified, must be specified with the soap-location element as a pair of elements.
	soap-location		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 service-binding element as a pair of elements.
ejb			Optional element. Specifies the properties related to the generated EJB.

Table 9–2 Elements, Subelements, and Attributes of the wsdl2ejb XML Configuration File as Defined in the DTD (Cont.)

Element	Subelement	Attribute	Description
	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 `wsdl2ejb` 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 9–3 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 though the custom type mapping described previously.

Table 9–3 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 9–3 are also supported.

Known Limitations of the `wsdl2ejb` Utility

The following information describes the known limitations of the `wsdl2ejb` 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 `demo/web_services/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.

Both 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: Both 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>/j2ee/home/demo/web_services/wsdl2ejb
java -jar ../../../wsdl2ejb.jar -conf rpc_doc/rpc_doc_conf.xml

On Windows
cd $<ORACLE_HOME>\j2ee\home\demo\web_services\wsdl2ejb
java -jar ..\..\..\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 Oracle9*i*AS 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>
<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 `wsdl2ejb` 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 + "/Wsdl2EjbTestApp1");
      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 folllow 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>
```

```
    <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>/j2ee/home/demo/web_services/wsdl2ejb
java -jar ../../../wsdl2ejb.jar -conf base/base_conf.xml
On Windows
cd $<ORACLE_HOME>\j2ee\home\demo\web_services\wsdl2ejb
java -jar ..\..\..\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 Oracle9*i*AS 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.

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 9–4 and Table 9–5.

Table 9–4 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.

Table 9–4 The oracle.j2ee.ws.client Package (Cont.)

Classes	Description
WebServiceMethod	This interface invokes a Web Service method.
Exceptions	
WebServiceProxyException	This class describes exceptions raised by the WebServiceProxy API.

Table 9–5 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 9–6 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 9–6 WebServiceProxyFactory Factory Methods and Parameters

Methods	Parameters

`createWebServiceProxy()`	`java.io.InputStream isWsdl`
	`java.net.URL baseURL`

`createWebServiceProxy()`	`java.net.URL wsdlURL`

`createWebServiceProxyFromBinding()`	`java.io.InputStream wsdlis`
	`java.net.URL baseUrl`
	`java.lang.String szBindingName`
	`java.lang.String szSoapLocation`

`createWebServiceProxyFromService()`	`java.io.InputStream wsdlis`
	`java.net.URL baseUrl`
	`java.lang.String szServiceName`
	`java.lang.String szServicePort`

`createWebServiceProxyFromBinding()`	`java.net.URL wsdlUrl`
	`java.lang.String szBindingName`
	`java.lang.String szSoapLocation`

`createWebServiceProxyFromService()`	`java.net.URL wsdlUrl`
	`java.lang.String szServiceName`
	`java.lang.String szServicePort`

`setProperties()`	`java.util.Hashtable ht`

Table 9–7 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 9–7 WebServiceProxy Interface Methods and Parameters

Methods	Parameters	Description
`getXMLMapping Registry()`	None	Returns the SOAP mapping registry used by the WebServiceProxy 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.
`getPortType()`	None	Returns a structure describing the WSDL portType used by this proxy and contains information about operations associated with this port type.
`getMethod()`		Returns a WebServiceMethod method, which can be used to invoke Web Service methods.
	szOperationName szInputName szOutputName	Name of the WSDL operation to be executed. Name of the wsdl:input tag for the operation to be executed. Name of the wsdl:output tag for the operation to be executed.
`getMethod()`		Returns a WebServiceMethod method, which can be used to invoke Web service methods and provides a signature that can be used for non-overloaded WSDL operations.
	szOperationName	Name of the WSDL operation to be executed.

Table 9–8 describes the WebServiceMethod interface, which is used to invoke a Web Service method.

Table 9–8 WebServiceMethod Interface Methods and Parameters

Methods	Parameters	Description
`getInputEncodingStyle()`	None	Returns the encoding style to be used by the input message parts, null if none has been specified in the source WSDL.
`getOutputEncodingStyle()`	None	Returns the encoding style to be used by the output message parts, null if none has been specified in the source WSDL.
`invoke()`		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.
	inMsgPartNames	Name of the parts supplied in the input message.
	inMsgPartValues	Corresponding value of the parts whose name is supplied in the inMsgPartNames 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://mssoapinterop.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 szWsdlUrl = "http://mssoapinterop.org/asmx/Rpc.asmx?WSDL";

    URL urlWsdl = new URL(szWsdlUrl);
    System.err.println("Wsdl 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: Wsdl 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.

A
Using Oracle SOAP

This appendix covers the following topics:

Understanding Oracle9*i*AS SOAP

In addition to the Oracle9*i*AS Web Services previously described in this chapter, that use a unique Servlet interface and J2EE deployment for Web Services, Oracle9*i*AS also provides Oracle9*i*AS SOAP that is derived from Apache 2.2 SOAP and includes a number of enhancements.

The SOAP Message Processor (Oracle9*i*AS 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 - Oracle9*i*AS SOAP Processor integrates the Oracle XML Parser. For RPC-style requests, the Oracle9*i*AS 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 Oracle9*i*AS SOAP Processor (Servlet) authenticates the user using a standard JAAS-based User Manager plug-in. Oracle9*i*AS 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

Oracle9*i*AS SOAP is a modified version of Apache SOAP 2.2. Most of the documentation that applies to Apache SOAP 2.2 also applies to Oracle9*i*AS 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 Oracle9*i*AS 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
`errorHandlers`	Specifies a list of handlers for the error handler chain.
`faultListeners`	This is an optional element that defines a list of faultListener elements. The faultListener 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 faultListener of org.apache.soap.server.DOMFaultListener.

Table A–1 (Cont.) SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
`handler`	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 oracle.soap.server.Handler. Each handler may have any number of options, which are name-value pairs. The contexts are configured in the elements: requestHandlers, responseHandlers, and errorHandlers. Each of these elements defines an ordered list of handler names, or a chain of handlers. 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.
`logger`	Error and informational messages are logged using the class defined in the logger element. The logger class must extend `oracle.soap.server.Logger`. Oracle9*i*AS SOAP includes the class `oracle.soap.server.impl.ServletLogger` that collects the servlet log methods so that SOAP messages are logged to the servlet log file. `ServletLogger` is the default logger. For the default logger, the severity option can be to any of the following values: `status`, `error`, `debug`. If you specify `error`, you will get both `status` and `error` messages. Similarly, if you specify `debug`, you will get all three types of messages. Oracle9*i*AS SOAP includes two logger implementations. To log to the servlet log, use oracle.soap.server.impl.ServletLogger. To log to stdout, use oracle.soap.server.impl.StdOutLogger. You may implement your own logger by implementing the oracle.soap.server.Logger interface.

Table A–1 (Cont.) SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
`providerManager`	The providerManager is an optional element that allows a configuration manager to be defined. This defines how the server accesses provider deployment information. The `providerManager` class attribute specifies a Java class that implements `oracle.soap.server.ProviderManager`. The default configuration manager, oracle.soap.server.impl.XMLProviderConfigManager, persists the deployed providers to a file in XML format. It accepts a filename 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 WEB-INF/providers.xml. An alternative provider configuration manager, oracle.soap.server.impl.BinaryProviderConfigManager, persists the deployed providers in a file as a serialized object. The default file is WEB-INF/providers.dd. To specify a different configuration manager add a class attribute to the configManager element. For example: <osc:configManager class="*fully.qualified.classname*">.
`requestHandlers`	Specifies a list of handlers for the request handler chain
`responseHandlers`	Specifies a list of handlers for the response handler chain
`serviceManager`	The serviceManager is an optional element that allows a configuration manager to be defined and ServiceManager options to be set. This defines how the server accesses service deployment information. The `serviceManager` class attribute specifies a Java class that implements `oracle.soap.server.ServiceManager`. The default Oracle9*i*AS SOAP configuration manager class is `oracle.soap.server.impl.XMLServiceConfigManager` which stores the service deployment information in an XML file. Using `XMLServiceConfigManager`, the file name is specified with the `filename` 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 WEB-INF/services.xml. To specify a different configuration manager add a class attribute to the `configManager` element. For example: <osc:configManager class="*fully.qualified.classname*">. An alternative service configuration manager, oracle.soap.server.impl.BinaryServiceConfigManager, persists the deployed services in a file as a serialized object. The default file is WEB-INF/services.dd. 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 autoDeploy option to true. By default autoDeploy value is false.

Using Oracle9*i*AS SOAP Management Utilities and Scripts

To use the Oracle9*i*AS 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:

```
providerrMgr.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 Oracle9*i*AS SOAP Services

This section covers the following topics related to deploying and undeploying Oracle9*i*AS 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 Oracle9*i*AS 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 Oracle9*i*AS SOAP Web Service in Oracle9*i*AS 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, Oracle9*i*AS SOAP services are not handled by the OC4J clustering software. It is necessary to install an Oracle9*i*AS 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 Oracle9*i*AS 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 Oracle9*i*AS 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 `chainType` 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`	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.
	
	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, `explosives.acme.com` or `explosives`. For some IP addresses, the DNS may not be able to resolve the hostname.
	
	Legal values for a `Host` filter attribute include the following examples:
	
	`explosives.acme.com, *.acme.com, *.com`
	
	Illegal values for a `Host` filter attribute include the following examples:
	
	`*, explosives.acme.*, explosives.*, ex*s.acme.com, *ives.acme.com`

Table A–3 (Cont.) Audit Trail Events Filter Attributes

Audit Event Filter Attributes	Description

`ip`	Specifies the IP address of the client for the service request or response.
	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.
	If this attribute is not used in a filter then the IP address of the client is not used in filtering.
	Legal values for an `ip` filter attribute include the following examples:
	`138.2.142.154, 138.2.142.*, 138.2.*, 138.*`
	Illegal values for an `ip` filter attribute include the following examples:
	`*, 138.2.*.154, *.2, 138.*.152, 138.2.142, 138.2, 138`

`urn`	Specifies the service URN. Wildcards are not supported for this attribute.
`username`	Specifies the transport level username associated with the client. Wildcards are not supported in a username filter attribute.

Table A–4 Audit Log Filter Syntax

Filter Value	Description
attr	1*(any US-ASCII char except "*", "(", ")", "&", "|", "!", "*", "=")
equal	"="
filter	"("*filtercomp*")" Whitespaces between "("*filtercomp* and ")" are not allowed.
filtercomp	*and* | *or* | *not* | *item* *and* = "&" *filterlist* *or* = "|" *filterlist* *not* = "!" *filter*
filterlist	2*2 *filter*
filtertype	*equal*
item	*attr filtertype value* Whitespaces between *attr*, *filtertype* and *value* are not allowed.

Table A–4 (Cont.) Audit Log Filter Syntax

Filter Value	Description
value	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=daffy))
(&(ip=138.2.142.*)(|(urn=urn:www-oracle-com:AddressBook)(username=daffy)))
```

Configuring the Audit Logger

Configure the default SOAP audit logger supplied with Oracle9*i* 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
`auditLogDirectory`	Specifies the directory where the audit log file is saved. The `auditLogDirectory` option is required. The name of the generated audit log file is `OracleSoapAuditLog.`*`timestamp`*, where *`timestamp`* is the date and time the file is first generated. **Valid values:** any string that is a valid directory
`filter`	Specifies the audit event filter. This option is optional. If a `filter` is not specified SOAP server logs every event. **Valid values:** any valid filter.
`includeRequest`	Specifies that the audit record include the request message for the event that generated the audit log record. **Valid values:** `true`, `false` Any value other than `true` or `false` is treated as an error. **Default Value:** `false`

Table A–5 (Cont.) Audit Logger Configuration Options

Option	Description
`includeResponse`	Specifies that the audit record include the response message for the event that generated the audit log record. **Valid values:** `true`, `false` Any value other than `true` or `false` is treated as an error. **Default Value:** `false`

See Also: "Using Oracle9iAS SOAP Handlers" on page A-13

Using Oracle9*i*AS SOAP Pluggable Configuration Managers

Oracle9*i*AS SOAP supports pluggable configuration managers similar to those supported in Apache SOAP 2.2. Since Oracle9*i*AS SOAP supports provider deployment descriptors separate from service deployment descriptors, the interface details using Oracle9*i*AS SOAP are slightly different from Apache SOAP 2.2. In Oracle9*i*AS 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 the abstract class that is provided with Oracle9*i*AS 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 Oracle9*i*AS SOAP Transport Security

Oracle9*i* Application Server uses the security capabilities of the underlying transport that sends SOAP messages. Oracle9*i* 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)

Oracle9*i*AS SOAP Client transport uses the modified, to support Oracle Wallet Manager, `HTTPClient` package. Oracle9*i*AS SOAP transport defines several properties to support these features. Table A–6 lists the client-side security properties that Oracle9*i* Application Server supports.

In an Oracle9*i*AS 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.setProperties()` to add properties).

Example A–6 shows how Oracle9*i* Application Server supports overriding the values specified for system properties using Oracle9*i* Application Server transport specific APIs. The `setProperties()` 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 OracleSOAPHHTTPConnection

```
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.setProperties(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
`http.authType`	Specifies the HTTP authentication type. The case of the value specified is ignored. Valid values: `basic`, `digest` The value basic specifies HTTP basic authentication. Specifying any value other than `basic` or `digest` is the same as not setting the property.
`http.password`	Specifies the HTTP authentication password.
`http.proxyAuthType`	Specifies the proxy authentication type. The case of the value specified is ignored. Valid values: `basic`, `digest` Specifying any value other than `basic` or `digest` is the same as not setting the property.
`http.proxyHost`	Specifies the hostname or IP address of the proxy host.
`http.proxyPassword`	Specifies the HTTP proxy authentication password.
`http.proxyPort`	Specifies the proxy port. The specified value must be an integer. This property is only used when `http.proxyHost` is defined; otherwise this value is ignored. Default value: `80`
`http.proxyRealm`	Specifies the realm for which the proxy authentication username/password is specified.
`http.proxyUsername`	Specifies the HTTP proxy authentication username.
`http.realm`	Specifies the realm for which the HTTP authentication username/password is specified.
`http.username`	Specifies the HTTP authentication username.
`java.protocol.handler.pkgs`	Specifies a list of package prefixes for `java.net.URLStreamHandlerFactory` The prefixes should be separated by `"|"` vertical bar characters. This value should contain: `HTTPClient` This value is required by the Java protocol handler framework; it is not defined by Oracle9*i* Application Server. This property must be set when using HTTPS. If this property is not set using HTTPS, a `java.net.MalformedURLException` is thrown. **Note:** This property must be set as a system property. For example, set this property as shown in either of the following: ■ `java.protocol.handler.pkgs=HTTPClient` ■ `java.protocol.handler.pkgs=sun.net.www.protocol|HTTPClient`

Table A–6 (Cont.) SOAP HTTP Transport Security Properties

Property	Description
`oracle.soap.transport.allowUserInteraction`	Specifies the allows user interaction parameter. The case of the value specified is ignored. When this property is set to `true` and either of the following are true, the user is prompted for a username and password: **1.** If any of properties `http.authType,` `http.username`, or `http.password` is not set, and a `401` HTTP status is returned by the HTTP server. **2.** If either of properties `http.proxyAuthType`, `http.proxyUsername`, or `http.proxyPassword` is not set and a `407` HTTP response is returned by the HTTP proxy. Valid values: `true`, `false` Specifying any value other than `true` is considered as `false`.

`oracle.soap.transport.1022ContentType`	Specifies the value for the Oracle9iAS Content-Type HTTP header. The value for this property supports Oracle SOAP servers running either Oracle 9*i*AS Release 1.0.2.2 or Release 9.0.x. This property provides interoperablity between Oracle9iAS Release 9.0.2 Oracle SOAP clients and older server versions (as distributed with Oracle9*i*AS Release 1.0.2.2).
	Valid values: `true`, `false` (case is ignored)
	Setting the value to `true` specifies to use the Oracle9 *i*AS Release 1.0.2.2 content-type HTTP header values when the SOAP message is sent. In this case, the value is set to:
	`content-type: text/xml`
	Setting the value to `false` 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:
	`content-type: text/xml; charset=utf-8`
	The value `false` is the default value.
	Note: for SOAP messages with attachments, the content-type HTTP header is always set to the value: `multipart/related`.

`oracle.ssl.ciphers`	Specifies a list of `:` separated cipher suites that are enabled. Default value: The list of all cipher suites supported by Oracle SSL are supported.

Table A–6 (Cont.) SOAP HTTP Transport Security Properties

Property	Description
`oracle.wallet.location`	Specifies the location of an exported Oracle wallet or exported trustpoints. Note: The value used is not a URL but a file location, for example: `/etc/ORACLE/Wallets/system1/exported_wallet` (on UNIX) `d:\oracle\system1\exported_wallet` (on Windows) This property must be set when HTTPS is used with SSL authentication, server or mutual, as the transport.
`oracle.wallet.password`	Specifies the password of an exported wallet. Setting this property is required when HTTPS is used with client, mutual authentication as the transport.

Apache Listener and Servlet Engine Configuration for SSL

When using Apache listener and mod_ssl (or mod_ossl), 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_ossl 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 Oracle9*i*AS SOAP Client

This section describes how to use SSL with the Oracle9*i*AS 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`).

Oracle9*i*AS 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 Oracle9*i*AS 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`, `jcert.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 Oracle9*i*AS SOAP Sample Services

The section lists the samples included with Oracle9*i*AS 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 Oracle9*i*AS SOAP EJB Provider

This section compares the Oracle9*i*AS 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.

Oracle9*i*AS 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 retrive 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 deserilaizes 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.

Oracle9*i*AS SOAP offers an alternative solution for Stateful Session EJBs that allows for client interoperablity.

Stateful Session EJB Provider in Oracle9*i*AS SOAP

The Oracle9*i*AS SOAP Stateful Session EJB provider binds the EJB reference to the current session, if none is bound, otherwise, it merely retrives the EJB reference from the session. In order for the client to access the same Stateful Session EJB, the client has to simply maintain it's 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 Oracle9*i*AS SOAP

In order for a SOAP client to exceute 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 exceution 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 Oracle9*i*AS 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 specfic EJB descriptor to correct the JNDI locationfor 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 Oracle9*i*AS SOAP Stored Procedure (SP) Provider supports exposing PL/SQL stored procedures or functions as SOAP services. The Oracle9*i* 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 proceudres 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 Oracle9*i*AS Web Services, including:

- Tunneling Using the TcpTunnelGui Command
- Setting Configuration Options for Debugging
- Using DMS to Display Runtime Information
- SOAP Limitations for Java Type Prcedence 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
TUNNEL-PORT	The port that `TcpTunnelGui` listens to on the same host as the client
SOAP-HOST	The host of the SOAP server
SOAP-PORT	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

Oracle9*i*AS 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 Prcedence with Overloaded Methods

Oracle9*i*AS 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()`.

Oracle9*i*AS SOAP Differences From Apache SOAP

This section covers differences between Apache Soap and Oracle9*i*AS SOAP.

Service Installation Differences

Additional instructions are provided for installing services when Oracle9*i*AS SOAP is used in conjunction with OC4J.

Optional Provider Enhancements

Oracle9*i*AS 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 Oracle9*i*AS SOAP and OC4J. The `com`, `calculator`, `weblogic_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 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 `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, digsig, 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

Oracle9*i*AS 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 Oracle9*i*AS 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