Oracle® Application Server Web Services

Developer's Guide

10*g* (9.0.4)

Part No. B10447-01

September 2003

ORACLE®

Oracle Application Server Web Services Developer's Guide, 10*g* (9.0.4)

Part No. B10447-01

Primary Author: Thomas Van Raalte

Contributing Author: Rodney Ward

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

Contents

Send Us Your Comments

Oracle Application Server Web Services Developer’s Guide, 10*g* (9.0.4)

Part No. B10447-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- Electronic mail: appserverdocs_us@oracle.com
- FAX:650-506-7365 Attn: Oracle Application Server Documentation Manager
- Postal service:
  Oracle Corporation
  Oracle Application Server Web Services Developer’s Guide
  500 Oracle Parkway M/S 1op6
  Redwood Shores, CA 94065
  USA

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

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

Preface

This guide describes Oracle Application Server Web Services.

This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Organization
- Related Documentation
- Conventions

Intended Audience

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

- Configure software installed on the Oracle Application Server.
- Create programs that implement Web Services
- Create 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, "Oracle Application Server Web Services"

This chapter describes the Oracle Application Server 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 Oracle Application Server 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 Oracle Application Server 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 Oracle Application Server Web Services that are implemented as PL/SQL Stored Procedures or Functions.

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

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

Chapter 7, "Developing and Deploying JMS Web Services"

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

Chapter 8, "Building Clients that Use Web Services"

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

Chapter 9, "Web Services Tools"

This chapter describes the Oracle Application Server Web Services assembly tool, `WebServicesAssembler`, that assists in assembling Oracle Application Server Web Services.

Chapter 10, "Discovering and Publishing Web Services"

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

Chapter 11, "Consuming Web Services in J2EE Applications"

This chapter describes to consume Web Services in J2EE applications.

Chapter 12, "Advanced Topics for Web Services"

This chapter describes several advanced Oracle Application Server Web Services topics, including untyped request handling options and SOAP header support.

Appendix A, "Using Oracle SOAP"

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

Appendix B, "Web Services Security"

This appendix describes the architecture and configuration of security for Oracle Application Server Web Services, including the Oracle Application Server UDDI Registry.

Glossary

The glossary contains the Web Services glossary terms and descriptions.

Related Documentation

For more information, see these Oracle resources:

- Overview Guide in the Oracle Application Server 10*g* Documentation Library.
- Oracle Application Server Containers for J2EE User's Guide in the Oracle Application Server 10*g* Documentation Library.

Printed documentation is available for sale in the Oracle Store at

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

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

```
http://otn.oracle.com/membership/
```

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

```
http://otn.oracle.com/documentation/content.html
```

Conventions

The following conventions are used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
$	The dollar sign represents the Command Language prompt in Windows and the Bourne shell prompt in UNIX

1 Web Services Overview

This chapter provides an overview of Web Services. Chapter 2, "Oracle Application Server Web Services" describes the Oracle Application Server Web Services features, architecture, and implementation.

This chapter covers the following topics:

- What Are Web Services?
- Overview of Web Services Standards
- SOAP Message Exchange and SOAP Message Encoding

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 software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts. A Web Service supports direct interactions with other software applications using XML based messages and internet-based products.

A Web Service does the following:

- Exposes and describes itself – A Web Service defines its functionality and attributes so that other applications can understand it. By providing a WSDL file, a Web Service makes its functionality available to other applications.
- Allows other services to locate it on the web – A Web Service can be registered in a UDDI Registry so that applications can locate it.
- Can be invoked – Once a Web Service has been located and examined, the remote application can invoke the service using an Internet standard protocol.
- Web Services are of either request and response or one-way style, and they can use either synchronous or asynchronous communication. However, the fundamental unit of exchange between Web Services clients and Web Services, of either style or type of communication, is a message.

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. Oracle Application Server 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.

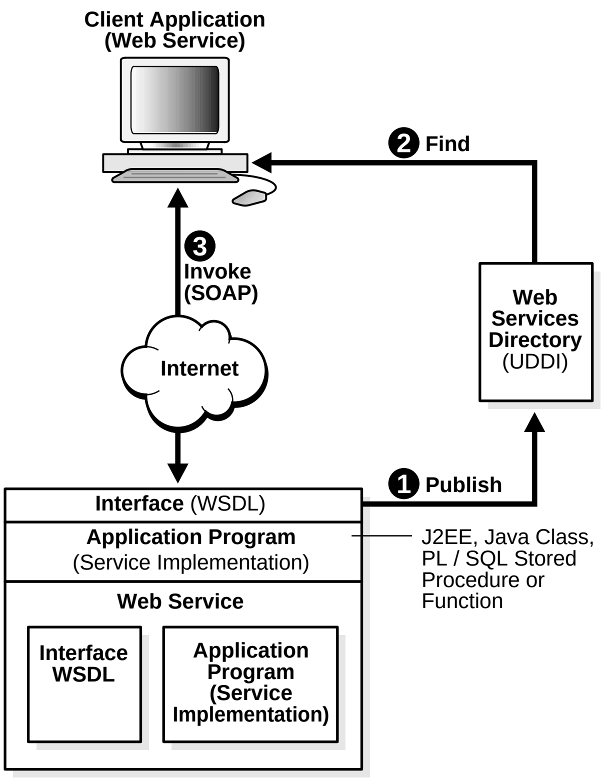
Overview of Web Services Standards

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

- SOAP Standard
- 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



SOAP Standard

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

SOAP has the following features:

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

> **See Also:** `http://www.w3.org/TR/SOAP/` for information on the 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.

SOAP Message Exchange and SOAP Message Encoding

The SOAP standard defines a lightweight, XML-based protocol for exchanging information in a decentralized, distributed environment. SOAP supports different styles of information exchange, including: Remote Procedure Call, **RPC Style**, and Message-oriented exchange, or **Document Style**. SOAP Messages, whether RPC Style or Document Style use a certain encoding, as specified with the `encodingStyle` attribute specified for SOAP message elements. This section describes these SOAP message features, in the following sections:

- SOAP Message Components
- Working With RPC Style SOAP Messages
- Working With Document Style SOAP Messages

SOAP Message Components

Each SOAP message is a transmission between a SOAP sender and a SOAP receiver. Each SOAP message consists of a SOAP envelope containing two sub-elements, a `Header` and a `Body`. The SOAP Header is optional. The children of the SOAP header are called `header blocks`; each header block represents a logical grouping of data. The SOAP Body is a mandatory element within a SOAP message. This is where the end-to-end information conveyed in a SOAP message is carried. The choice of what data is placed in a header block and what data goes in the SOAP Body element are decisions that are taken at the time that an application is designed.

Using Oracle Application Server Web Services, developers determine if an implementation supports RPC Style or Document Style messages. Developers write the appropriate application logic and the WebServicesAssembler configuration files for the implementation.

Working With RPC Style SOAP Messages

Oracle Application Server Web Services supports two types of SOAP message exchanges: RPC Style exchanges and Document-Style exchanges. RPC Style exchanges represent exchanges that can be modeled as remote procedure calls (RPC); these are used when there is a need to model a certain programatic behavior, with the exchanged messages conforming to a well-defined signature for the remote call and its return. Using RPC Style messages, SOAP specifies the form of the SOAP message body.

RPC style information exchange allows for request-response processing, where an endpoint receives a procedure oriented message and replies with a response message. Using the RPC style SOAP message exchange, the contents of the SOAP message body conform to a structure that specifies a procedure and includes set of parameters, or a response, with a result and any additional parameters. The SOAP message in the body is an XML document, but it is XML document that conforms the limitations specified in the SOAP specification.

Example 1–1 shows a SOAP RPC Style request that includes the `ChargeReservation` method with several parameters. Example 1–2 shows the SOAP RPC Style response message that includes the `ChargeReservationResponse`, with a "Response" string appended.

Example 1–1 SOAP RPC Style Request Message

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:helloWorld xmlns:ns1="urn:oracle-j2ee-ws_example-StatelessExample"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <param0 xsi:type="xsd:string">Wendy</param0>
      </ns1:helloWorld>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 1–2 SOAP RPC Style Response Message

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:helloWorldResponse
           xmlns:ns1="urn:oracle-j2ee-ws_example-StatelessExample"
           SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
             <return xsi:type="xsd:string">Hello World, Wendy</return>
      </ns1:helloWorldResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Working With Document Style SOAP Messages

Oracle Application Server Web Services supports two types of SOAP message exchanges: RPC Style exchanges and Document-Style exchanges. Document-style exchanges, also called message-oriented exchanges, model exchanges where XML documents are exchanged, where the exchange patterns are defined in the sending and the receiving applications. For Document Style messages, SOAP places no constraints on how the document sent in the SOAP message body is structured, the application, or an externally specified XML schema determines the structure of the XML document that is sent in the body of the SOAP message.

Message-oriented information exchange supports organizations and applications that need to exchange business or other types of documents where a message is sent but the sender may not expect or wait for an immediate response. Message-oriented information exchange is also called Document style SOAP message exchange. Document -style messages model exchanges where XML documents are exchanged, where the semantics of the exchange patterns are defined in the sending and the receiving applications.

Example 1–3 shows a sample Document Style SOAP message that is sent from a client to an Oracle Application Server Web Services document style service. The client sends an XML document that contains employee records with elements including name, emp_id, department, and contact information. A web service that processes this XML document to produce a phone listing may supply an XML document that contains only the name and phone number elements.

Example 1–3   Document Style SOAP Message

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<organisation>
```

```
  <employee>
   <name>Bob</name>
   <emp_id>1234</emp_id>
   <department>hr</department>
   <contact>
      <phone>827 644 5674</phone>
      <email>bob@organisation.com</email>
   </contact>
  </employee>
  <employee>
   <name>Susan</name>
   <emp_id>2434</emp_id>
   <department>it</department>
   <contact>
      <phone>827 644 5674</phone>
      <email>Susan@organisation.com</email>
   </contact>
  </employee>
</organisation>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 1–4 Document Style SOAP Message Processed by a Web Service

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
   <employee>
      <name>Bob</name>
        <phone>827 644 5674</phone>
      <name>Susan</name>
        <phone>827 644 5674</phone>
   </employee>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2 Oracle Application Server Web Services

This chapter describes the Oracle Application Server Web Services features, architecture, and implementation.

This chapter covers the following topics:

- Oracle Application Server OC4J (J2EE) and Oracle SOAP Based Web Services
- Oracle Application Server Web Services Standards
- Oracle Application Server Web Services Features
- Oracle Application Server Web Services Architecture
- Understanding WSDL and Client Proxy Stubs for Web Services
- Web Services Home Page
- About Universal Description, Discovery, and Integration Registry

Oracle Application Server OC4J (J2EE) and Oracle SOAP Based Web Services

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

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

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

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

Oracle Application Server Web Services Standards

Oracle Application Server Web Services supports the following Web Services standards:

- SOAP 1.1, including the following:
  - RPC/Encoded
  - Document/Literal
- WSDL 1.1
- UDDI 2.0

See Also: "Overview of Web Services Standards" on page 2-2

Oracle Application Server Web Services Features

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

- Developing End-to-End Web Services
- Deploying and Managing Web Services
- Using Oracle JDeveloper with Web Services
- Securing Web Services
- Aggregating Web Services

Developing End-to-End Web Services

Oracle Application Server Web Services provides comprehensive support for developing Web Services, including:

- Development Environment – Oracle Application Server 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 – Oracle Application Server Web Services Developers can use the same set of command line utilities to create, package, and deploy Web Services as other Oracle Application Server Containers for J2EE (OC4J) applications. In addition Oracle Application Server 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 – Oracle Application Server Web Services can generate WSDL and client-side proxy stubs. This generation occurs when the Web Service is assembled using the WebServices Assembly tool or alternatively, for a deployed Web Service, the first time the WSDL or the client-side proxy stubs are requested (after the first request, the previously generated WSDL or client-side proxy stubs are sent when requested).
- Registration, Publishing, and Discovery – Oracle Application Server 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 Oracle Application Server 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 or from a new Web Service client.
- Common Runtime Services – Oracle Application Server has a common runtime and brokering environment for J2EE Applications and Web Services. As a result, Web Services transparently inherit various services available with the J2EE Container including Transaction Management, Messaging, Naming, Logging, and Security Services.

Deploying and Managing Web Services

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

- Packaging and Assembly - The Web Services Assembly Tool assists with assembling Web Services and producing a J2EE .ear file.
- Deployment – Oracle Enterprise Manager provides a comprehensive set of facilities to deploy Web Services to Oracle Application Server. Oracle Enterprise Manager provides a single, consistent *Deploy Applications* wizard for deploying Web Services to Oracle Application Server. 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 Oracle Application Server instance.

Using Oracle JDeveloper with Web Services

The Oracle JDeveloper IDE supports Oracle Application Server Web Services. Oracle 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.

Oracle JDeveloper supports Oracle Application Server 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 Oracle Application Server 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

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

Oracle Application Server Web Services Architecture

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

Oracle Application Server Web Services supports the following RPC Web Services:

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

Oracle Application Server Web Services supports the following Document Style web services:

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

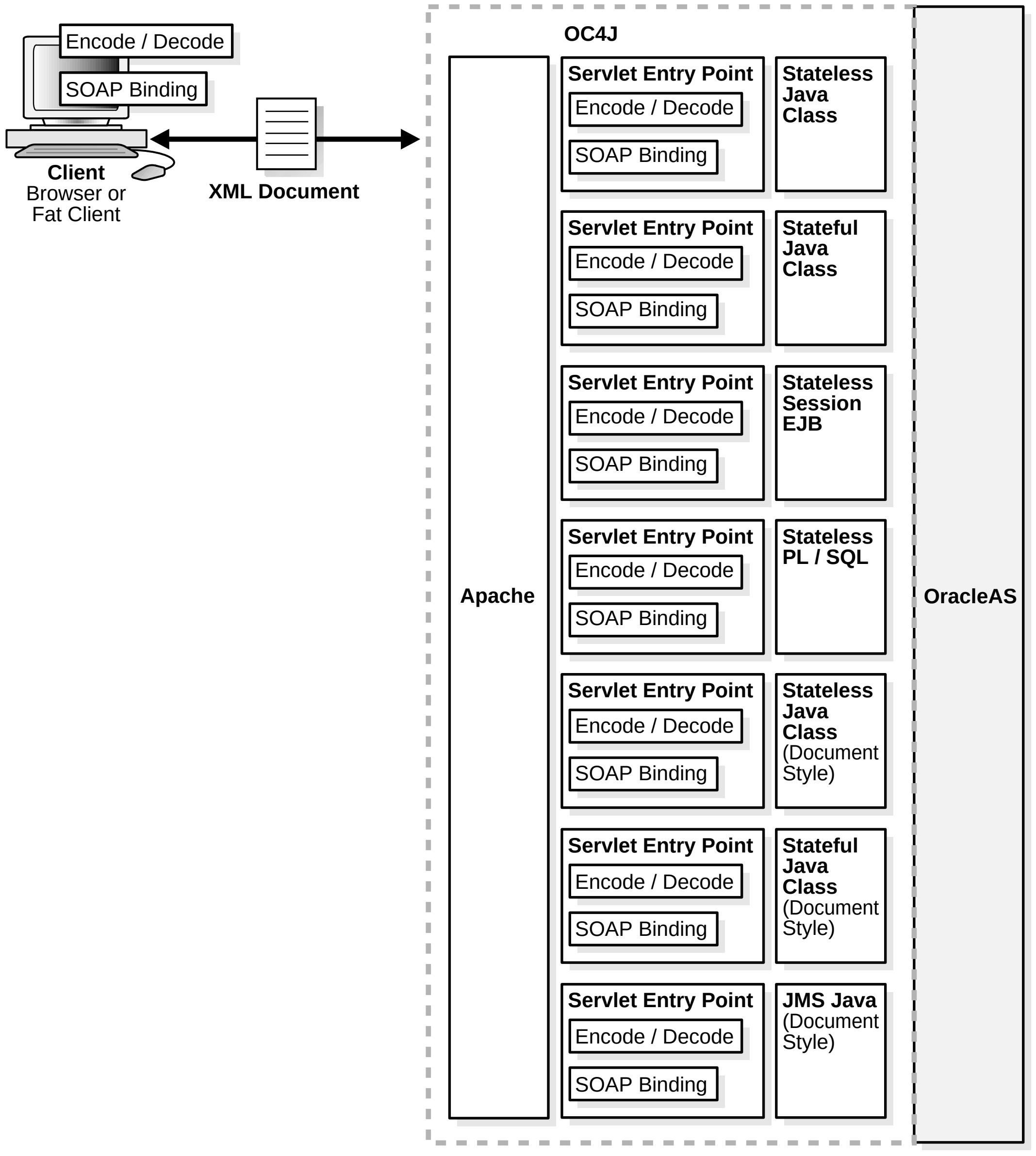
For each implementation type, Oracle Application Server Web Services uses a different Servlet that conforms to J2EE standards to provide an entry point to a Web Service implementation. Figure 2–1 shows the Web Services runtime architecture, including the Servlet entry points.

The Oracle Application Server Web Services runtime architecture discussion includes the following:

- About Servlet Entry Points for Web Services
- What Are the Packaging and Deployment Options for Web Services
- About Server Skeleton Code Generation for Web Services

See Also: "SOAP Standard" on page 1-6 for information on RPC Style and Document Style Web Services.

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



About Servlet Entry Points for Web Services

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

The Oracle Application Server Web Services Servlet classes support the following Web Services implementation types:

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

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

Note: Using Oracle Application Server SOAP, based on Apache SOAP 2.3.1, there is only a single instance of a single Servlet entry point for all the Web Services in the entire system. The Oracle Application Server Web Services architecture differs; under Oracle Application Server Web Services, a unique Servlet instance supports each Web Service.

RPC Style Web Service implementations under Oracle Application Server 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 Oracle Application Server Web Services.

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

Note: The preceding restriction means that `org.w3c.dom.Element` types cannot be mixed as a parameter with other types in methods that implement a Web Service.

Table 2–1 Web Services Supported Data Types (for RPC 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`

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

Primitive Type	Object Type
	`java.util.Map`
	`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

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

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

See Also:

- Chapter 3, "Developing and Deploying Java Class Web Services"
- Chapter 4, "Developing and Deploying EJB Web Services"
- Chapter 5, "Developing and Deploying Stored Procedure Web Services"
- Chapter 6, "Developing and Deploying Document Style Web Services"

About Server Skeleton Code Generation for Web Services

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

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

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

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

Understanding WSDL and Client Proxy Stubs for Web Services

Oracle Application Server Web Services provides a tool to generate a WSDL file that can be packaged with a Web Service at assembly time, (if you do not package the WSDL file, it can be generated at runtime). This tool also supports generating client-side proxy stubs, given a WSDL file.

There are several elements to Oracle Application Server Web Services WSDL support. First, RPC style Web Services are based on interoperable XML data representations and arbitrary Java objects do not in general map to XML. Oracle Application Server 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 Oracle Application Server Web Services, an application developer can either statically generate the WSDL interfaces for a Web Service or the Oracle Application Server Web Services runtime can generate WSDL and client-side proxy stubs if they are not provided when a Web Service is deployed. These files can be generated by the runtime on the server-side and delivered when they are requested by a Web Services client.

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

See Also:

- "Generating Client-Side Proxies With WebServicesAssembler" on page 8-8
- "Generating WSDL Files and Client Side Proxies" on page 9-4

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 Oracle Application Server 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 Oracle JDeveloper with Web Services" on page 2-5
- Chapter 8, "Building Clients that Use Web Services"

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

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

See Also: Chapter 8, "Building Clients that Use Web Services"

Web Services Home Page

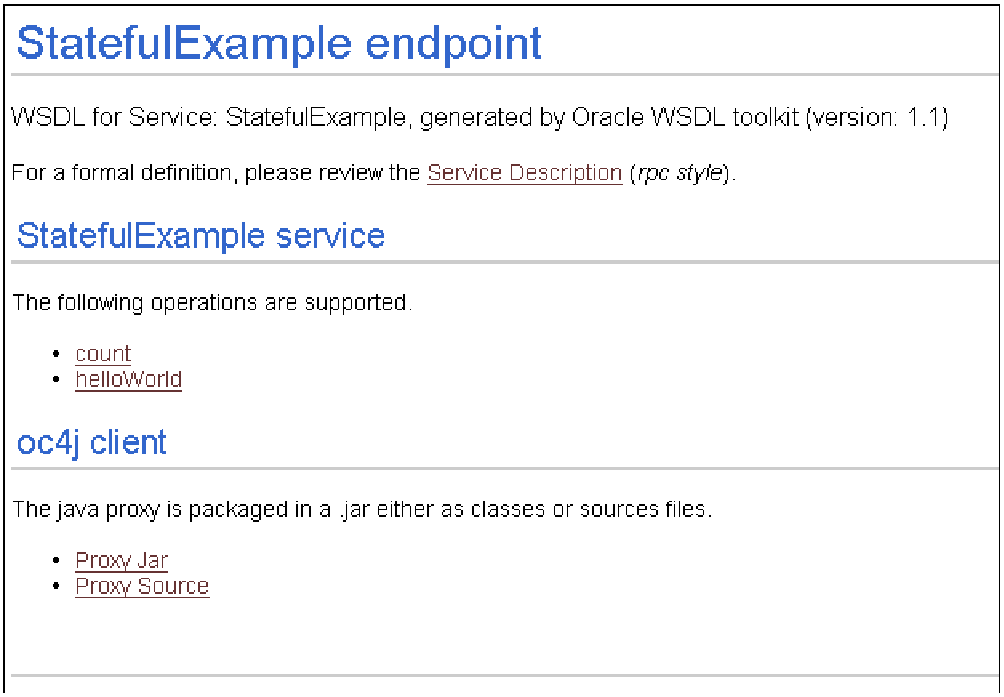
Oracle Application Server Web Services provides a Web Service Home Page for each deployed Web Service.

A Web Service Home Page provides the following:

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

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

Figure 2–2 Web Service Home Page



StatefulExample endpoint

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

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

StatefulExample service

The following operations are supported.

- count
- helloWorld

oc4j client

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

- Proxy Jar
- Proxy Source

About Universal Description, Discovery, and Integration Registry

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

- Business Entity - The top level XML element in a UDDI entry captures the starting set of information required by partners seeking to locate information about a business' services including its name, its industry or product category, its geographic location, and optional categorization and contact information. This includes support for *Yellow Pages* taxonomies to search for businesses by industry, product, or geography.
- Business Service - The businessService structure groups a series of related Web Services together so that they can be related to either a business process or a category of services. An example of a business process could be a

logistics / delivery process which could include several Web Services including shipping, routing, warehousing, and last-mile delivery services. By organizing Web Services into groups associated with categories or business processes, UDDI allows more efficient search and discovery of Web Services.

- Binding Information - Each businessService has one or more technical Web Service Descriptions captured in an XML element called a binding template. The binding template contains the information that is relevant for application programs that need to invoke or to bind to a specific Web Service. This information includes the Web Service URL address, and other information describing hosted services, routing and load balancing facilities.
- Compliance Information - While the bindingTemplate contains the information required to invoke a service, it is not always enough to simply know where to contact a particular Web Service. For instance, to send a business partner's Web Service a purchase order, the invoking service must not only know the location / URL for the service, but what format the purchase order should be sent in, what protocols are appropriate, what security required, and what form of a response will result after sending the purchase order. Before invoking a Web Service, it is useful to determine whether the specific service being invoked complies with a particular behavior or programming interface. Each bindingTemplate element, therefore, contains an element called a tModel that contains information which enables a client to determine whether a specific Web Service is a compliant implementation.

Oracle Enterprise Manager Features to Register Web Services

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

See Also: Chapter 10, "Discovering and Publishing Web Services"

3

Developing and Deploying Java Class Web Services

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

This chapter covers the following topics:

Using Oracle Application Server Web Services With Java Classes

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

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

See Also:

- Chapter 2, "Oracle Application Server Web Services"
- Chapter 4, "Developing and Deploying EJB Web Services"
- Chapter 5, "Developing and Deploying Stored Procedure Web Services"
- Chapter 8, "Building Clients that Use Web Services"

Writing Java Class Based Web Services

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

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

The sample code is supplied on the Oracle Technology Network Web site,

http://otn.oracle.com/sample_code/tech/java/web_services/content.html

After expanding the Web Services `demo.zip` file, the Java class based Web Service is in the directory under `webservices/demo/basic/java_services` on UNIX or in `\webservices\demo\basic\java_services` on Windows.

Writing Stateless and Stateful Java Web Services

Oracle Application Server Web Services supports stateful and stateless implementations for Java classes running as Web Services, as follows:

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

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

Building a Sample Java Class Implementation

Developing a Java Web Service consists of the following steps:

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

Defining a Java Class Containing Methods for the Web Service

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

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

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

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

Example 3–1 shows `StatelessExampleImpl`.

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

Example 3–2 shows `StatefulExampleImpl`.

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

```
package oracle.j2ee.ws_example;

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

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

```
package oracle.j2ee.ws_example;

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

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

When an error occurs while running a Web Service implemented as a Java class, the Java class should throw an exception. When an exception is thrown, the Web Services Servlet returns a Web Services (SOAP) fault. Use the standard J2EE and OC4J administration facilities to view the logs of Servlet errors for a Web Service that uses Java classes for its implementation.

When you create a Java class containing methods that implement a Web Service, the method's parameters and return values must use supported types, or you need to use an interface class to limit the methods exposed to those methods using only supported types. Table 3–1 lists the supported types for parameters and return values for Java methods that implement Web Services.

Note: See Table 3–1 for the list of supported types for parameters and return values.

There are several additional steps required to implement a Java Web Service if you need to handle or process SOAP request header entries.

See Also: "SOAP Header Support" on page 12-4

Defining an Interface for Explicit Method Exposure

Oracle Application Server Web Services allows you to limit the methods you expose as Web Services by supplying a public interface. To limit the methods exposed in a Web Service, include a public interface that lists the method signatures for the methods that you want to expose. Example 3–3 shows an interface to the method in the class `StatelessExampleImpl`. Example 3–4 shows an interface to the methods in the class `StatefulExampleImpl`.

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

```
package oracle.j2ee.ws_example;

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

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

```
package oracle.j2ee.ws_example;

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

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

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

Use a Web Services interface for the following purposes:

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

See Also: "Using Supported Data Types for Java Web Services" on page 3-7

Writing a WSDL File (Optional)

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

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

See Also: "Generating WSDL Files and Client Side Proxies" on page 9-4

Using Supported Data Types for Java Web Services

Table 3–1 lists the supported data types for parameters and return values for Oracle Application Server Web Services.

Table 3–1 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`
	`java.util.Map`
	`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.

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

Note: The preceding restriction means that `org.w3c.dom.Element` types cannot be mixed as a parameter with other types in methods that implement a Web Service.

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

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

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

Oracle Application Server Web Services allows Beans to be returned or passed in as arguments to J2EE Web Service methods, as long as the Bean only consists of property types that are listed in Table 3–1 or are another supported Java Bean.

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

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

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

See Also: Chapter 8, "Building Clients that Use Web Services"

Preparing and Deploying Java Class Based Web Services

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

This section contains the following topics.

- Creating a Configuration File to Assemble Java Class Web Services
- Running WebServicesAssembler To Prepare Java Class Web Services

Creating a Configuration File to Assemble Java Class Web Services

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

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

- Adding Web Service Top Level Tags
- Adding Java Stateless Service Tags
- Adding Java Stateful Service Tags
- Adding WSDL and Client-Side Proxy Generation Tags

Adding Web Service Top Level Tags

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

Example 3–5 shows a complete `config.xml` file, including the top level tags.

Table 3–2 Top Level `WebServicesAssembler` ***Configuration Tags***

Tag	Description
`<context>` *context* `</context>`	Specifies the context root of the Web Service. This tag is required.
`<datasource-JNDI-name>` `</datasource-JNDI-name>`	Specifies the datasource associated with the Web Service.
`<description>` *description* `</description>`	Provides a simple description of the Web Service. This tag is optional.
`<destination-path>` *dest_path* `</destination-path>`	Specifies the name of the generated J2EE .ear file output. The *dest_path* specifies the complete path for the output file. This tag is required.
`<display-name>` *disp_name* `</display-name>`	Specifies the Web Service display name. This tag is optional.
`<option name="source-path" [contextroot="`*path1*`"] >` *path2* `<option>`	Includes a specified file in the output .ear file. Use this option to specify java resources, or the name of an existing .war, .ear, or ejb-jar file that is used as a source file for the output J2EE .ear file. When a .war file is supplied as input, the optional contextroot specifies the root-context for the .war file. *path1* specifies the context-root for the .war. *path2* specifies the path to the file to include. For example: `<option name="source-path" contextroot="/test">/myTestArea/ws/src/statefull.war</option>`
`<stateless-java-service>` *sub-tags* `</stateless-java-service>`	Use this tag to add a Java Web Services that defines a stateless service. See Table 3–3 for a description of valid *sub-tags*.
`<stateful-java-service>` *sub-tags* `</stateful-java-service>`	Use this tag to add a Java Web Services that defines a stateful service. See Table 3–3 for a description of valid *sub-tags*.
`<temporary-directory>` *temp_dir* `</temporary-directory>`	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding Java Stateless Service Tags

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

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

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

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

Adding Java Stateful Service Tags

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

To support a clustered environment, for stateful Java Web Services with serializable java classes, the `WebServicesAssembler` adds a `<distributable>` tag in the `web.xml` of the Web Service's generated J2EE.ear file.

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

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

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

Tag	Description
`<accept-untyped-request>` *value* `</accept-untyped-request>`	Setting *value* to `true` tells `WebServicesAssembler` to allow the Web Service to accept untyped requests. When the value is `false`, the Web Service does not accept untyped-request. Valid values: `true`, `false` (case is not significant; `TRUE` and `FALSE` are also valid) This tag is optional. Default value: `false`
`<class-name>` *class* `</class-name>`	Specifies the fully qualified class name for the class that supplies the Web Service implementation. This tag is required
`<interface-name>` *interface* `</interface-name>`	Specifies the fully qualified name of the interface that tells the Web Service Servlet generation code which methods should be exposed as Web Services. This tag is optional
`<ejb-resource>` *ejb-resource* `</ejb-resource>`	This is a backward compatibility tag. See Also: the top level <option name="source-path"> tag in Table 3–2. This tag is optional
`<java-resource>` *resource* `</java-resource>`	This is a backward compatibility tag. See Also: the top level <option name="source-path"> tag in Table 3–2. This tag is optional
`<message-style>` *rpc* `</message-style>`	Sets the message style. When defining a Java Web Service, if you include the `<message-style>` tag you must specify the value `rpc`. Valid Values: `doc`, `rpc` This tag is optional Default value: `rpc` (when the `<message-style>` tag is not supplied)
`<scope>` *scope* `</scope>`	Sets the scope of the session for stateful services. The `<scope>` tag only applies for stateful services. Use this tag only within the `<stateful-java-service>` tag. This tag is optional Valid Values: `application`, `session` Default Value: `session`

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

Tag	Description
`<session-timeout>` *value* `</session-timeout>`	Sets the session timeout for a stateful session. The `<session-timeout>` tag only applies for stateful services. Use this tag only within the `<stateful-java-service>` tag. Specify *value* with an integer that defines the timeout for the session in seconds. The default value for the session timeout for stateful Java sessions where no session timeout is specified is 60 seconds. This tag is optional
`<uri>` *URI* `</uri>`	Specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the *URI* is appended to the `<context>` to specify the Web Service location. This tag is required

Example 3–5 Sample WebServicesAssembler Configuration File

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

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

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

Adding WSDL and Client-Side Proxy Generation Tags

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

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

> **See Also:** "Generating WSDL Files and Client Side Proxies" on page 9-4

Running WebServicesAssembler To Prepare Java Class Web Services

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

Run the Oracle Application Server Web Services assembly tool, `WebServicesAssembler` as follows:

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

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

See Also:

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

Deploying Java Class Based Web Services

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

See Also: *Oracle Application Server Containers for J2EE User's Guide* in the Oracle Application Server 10*g* Documentation Library

Serializing and Encoding Parameters and Results for Web Services

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

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

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

Application Server Web Services. Table 3–1 lists the supported Web Services parameter and return value types when using standard SOAP v.1.1 encoding.

- Literal XML encoding. Using Literal XML encoding, a Web Service client can pass as a parameter, or a Java service can return as a result, a value that is encoded as a conforming W3C Document Object Model (DOM) `org.w3c.dom.Element`. When an `Element` passes as a parameter to a Web Service, the server side Java implementation processes the `org.w3c.dom.Element`. For return values sent from a Web Service, the Web Services client parses or processes the `org.w3c.dom.Element`.

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

See Also: Chapter 8, "Building Clients that Use Web Services"

4
Developing and Deploying EJB Web Services

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

This chapter covers the following topics:

- Using Oracle Application Server Web Services With Stateless Session EJBs
- Writing Stateless Session EJB Web Services
- Preparing and Deploying Stateless Session EJB Based Web Services

Using Oracle Application Server Web Services With Stateless Session EJBs

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

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

See Also:

- Chapter 2, "Oracle Application Server Web Services"
- Chapter 3, "Developing and Deploying Java Class Web Services"
- Chapter 5, "Developing and Deploying Stored Procedure Web Services"
- Chapter 8, "Building Clients that Use Web Services"

Writing Stateless Session EJB Web Services

Writing EJB based Web Services involves obtaining or building an EJB that implements a service. The EJB should contain one or more methods that a Web Services Servlet running under Oracle Application Server 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 is supplied on the Oracle Technology Network Web site,

http://otn.oracle.com/sample_code/tech/java/web_services/content.html

After expanding the Web Services `demo.zip` file, the EJB based Web Service is in the directory under `/webservices/demo/basic/stateless_ejb` on UNIX or in `\webservices\demo\basic\stateless_ejb` on Windows.

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

Developing a stateless session EJB consists of the following steps:

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

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

Defining a Stateless Session Remote Interface

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

Example 4–1 shows the `Remote` interface for the sample stateless session EJB.

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

```
package demo;

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

Defining a Stateless Session Home Interface

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

Example 4–2 shows the `EJBHome` interface for the sample stateless session EJB.

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

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

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

Defining a Stateless Session EJB Bean

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

Example 4–3 shows the source code for the `HelloService` Bean.

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

```
package demo;

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

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

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

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

Returning Results From EJB Web Services

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

Example 4–3 shows that the EJB Bean writer only needs to return values of supported types to create Web Services implemented as stateless session EJBs.

See Also: "Using Supported Data Types for Stateless Session EJB Web Services" on page 4-6

Error Handling for EJB Web Services

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

Serializing and Encoding Parameters and Results for EJB Web Services

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

> **See Also:** "Serializing and Encoding Parameters and Results for Web Services" on page 3-15

Using Supported Data Types for Stateless Session EJB Web Services

Table 4–1 lists the supported data types for parameters and return values for Oracle Application Server 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`
	`java.util.Map`
	`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.

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

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

Note: The preceding restriction means that `org.w3c.dom.Element` types cannot be mixed as a parameter with other types in methods that implement a Web Service.

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.

Oracle Application Server Web Services allows Beans to be returned or passed in as arguments to J2EE Web Service methods, as long as the Bean only consists of property types that are listed in Table 4–1 or are another supported Java Bean.

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

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

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

See Also: Chapter 8, "Building Clients that Use Web Services"

Writing a WSDL File for EJB Web Services (Optional)

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

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

See Also: "Generating WSDL Files and Client Side Proxies" on page 9-4

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 Oracle Application Server Web Services Servlet and includes the ejb.jar that supplies the Java implementation. This section describes how to use the Oracle Application Server Web Services tool, `WebServicesAssembler`. `WebServicesAssembler` takes an XML configuration file that describes the stateless session EJB Web Service and produces a J2EE .ear file that can be deployed under Oracle Application Server Web Services.

This section contains the following topics.

- Creating a Configuration File to Assemble Stateless Session EJB Web Services
- Running WebServicesAssembler To Prepare Stateless Session EJB Web Services
- Deploying Web Services Implemented as EJBs

Creating a Configuration File to Assemble Stateless Session EJB Web Services

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

Create `WebServicesAssembler` configuration file by adding the following:

- Adding Web Service Top Level Tags
- Adding Stateless Session EJB Service Tags
- Adding WSDL and Client-Side Proxy Generation Tags

Adding Web Service Top Level Tags

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

Example 4–4 shows a complete `config.xml` file, including the top level tags.

Table 4–2 Top Level `WebServicesAssembler` ***Configuration Tags***

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

Table 4–2 (Cont.) Top Level* `WebServicesAssembler` *Configuration Tags

Tag	Description

`<option name="source-path">`
path
`<option>`
	Includes a specified file in the output .ear file. Use this option to specify java resources, or the name of an existing .war, .ear, or ejb-jar file that is used as a source file for the output J2EE .ear file.
	When a .war file is supplied as input, the optional contextroot specifies the root-context for the .war file.
	path1 specifies the context-root for the .war.
	path2 specifies the path to the file to include.
	For example:
	`<option name="source-path"`
	`contextroot="/test">/myTestArea/ws/src/statefull.war`
	`</option>`
	This tag is optional.

`<stateless-session-ejb-service>`
sub-tags
`</stateless-session-ejb-service>`
	Use this tag to add a stateless session EJB Web Service. See Table 4–3 for a description of the valid *sub-tags*.

`<temporary-directory>`
temp_dir
`</temporary-directory>`
	Specifies a directory where the assembler can store temporary files.
	This tag is optional.

Adding Stateless Session EJB Service Tags

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

Table 4–3 shows the `<stateless-session-ejb-service>` sub-tags.

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

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

Tag	Description
`<accept-untyped-request>` *`value`* `</accept-untyped-request>`	Setting *value* to `true` tells `WebServicesAssembler` to allow the Web Service to accept untyped requests. When the value is `false`, the Web Service does not accept untyped-request. Valid values: `true`, `false` (case is not significant; `TRUE` and `FALSE` are also valid) This tag is optional. Default value: `false`
`<ejb-name>` *name* `</ejb-name>`	Specifies the *name* of the stateless session EJB. This tag is required
`<ejb-resource>` *resource* `</ejb-resource>`	This is a backward compatibility tag. See Also: the top level `<option name="source-path">` tag in Table 4–2. This tag is optional
`<path>` *path* `</path>`	This is a backward compatibility tag. See Also: the top level `<option name="source-path">` tag in Table 4–2. This tag is optional
`<uri>` *URI* `</uri>`	Specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the *URI* is appended to the `<context>` to specify the Web Service location. This tag is required.

Example 4–4 Sample Stateless Session EJB WebServicesAssembler Configuration File

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

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

Adding WSDL and Client-Side Proxy Generation Tags

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

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

> **See Also:** "Generating WSDL Files and Client Side Proxies" on page 9-4

Running WebServicesAssembler To Prepare Stateless Session EJB Web Services

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

Run the Oracle Application Server Web Services assembly tool, `WebServicesAssembler` as follows:

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

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

> **See Also:**
>
> - "Creating a Configuration File to Assemble Stateless Session EJB Web Services" on page 4-9
> - "Running the Web Services Assembly Tool" on page 9-2

Deploying Web Services Implemented as EJBs

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

See Also: *Oracle Application Server Containers for J2EE User's Guide* in the Oracle Application Server Documentation Library

5 Developing and Deploying Stored Procedure Web Services

This chapter describes how to write and deploy Oracle Application Server 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 Oracle Application Server Web Services, PL/SQL procedures and functions that run on an Oracle database server.

This chapter covers the following topics:

Using Oracle Application Server Web Services with Stored Procedures

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

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

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

See Also:

- Chapter 2, "Oracle Application Server Web Services"
- Chapter 3, "Developing and Deploying Java Class Web Services"
- Chapter 6, "Developing and Deploying Document Style Web Services"

Writing Stored Procedure Web Services

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

The sample code is supplied on the Oracle Technology Network Web site,

http://otn.oracle.com/sample_code/tech/java/web_services/content.html

After expanding the Web Services `demo.zip` file, the sample Stored Procedure Web Service is supplied in the directory under `webservices/demo/basic/stored_procedure` on UNIX or in `webservices\demo\basic\stored_procedure` on Windows.

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

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

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

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

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

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

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

See Also:

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

Preparing Stored Procedure Web Services

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

This section contains the following topics:

- Creating a Configuration File to Assemble Stored Procedure Web Services
- Running WebServicesAssembler With Stored Procedure Web Services
- Setting Up Datasources in Oracle Application Server Web Services (OC4J)

Creating a Configuration File to Assemble Stored Procedure Web Services

The Oracle Application Server Web Services assembly tool, `WebServicesAssembler`, assists in assembling Oracle Application Server Web

Services. This section describes how to create a configuration file to use to assemble a Stored Procedure Web Service. The Web Services assembly tool uses an XML configuration file that describes the Stored Procedure Web Service and produces a J2EE .ear file that can be deployed under Oracle Application Server Web Services.

Create `WebServicesAssembler` configuration file by adding the following:

- Adding Web Service Top Level Tags
- Adding Stateless Stored Procedure Java Service Tags
- Adding WSDL and Client-Side Proxy Generation Tags

Adding Web Service Top Level Tags

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

Example 5–1 shows a complete `config.xml` file, including the top level tags.

Table 5–1 Top Level `WebServicesAssembler` ***Configuration Tags***

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

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

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

Adding Stateless Stored Procedure Java Service Tags

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

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

---

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

Adding Stateless Stored Procedure Java Service Using Jar Generation

Using a configuration file that includes the `<jar-generation>` tag specifies Oracle Database Server connection information that allows the `WebServicesAssembler` to run Oracle JPublisher to generate the classes to support the Stored Procedure Web Service. The Oracle JPublisher generated classes support accessing the PL/SQL procedure or function and also includes classes for mapping Java types to PL/SQL types. The `WebServicesAssembler` packages the generated classes into a Jar file that is assembled with the Stored Procedure Web Service.

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

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

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

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

Tag	Description
`<database-JNDI-name>` *source_JNDI_name* `</database-JNDI-name>`	This tag specifies the JNDI name of the backend database. The data-sources.xml OC4J configuration file describes the database server source associated with the specified source_JNDI_name.

`<jar-generation>` *sub-tags* `</jar-generation>`	Table 5–3 describes the supported *sub-tags* for `<jar-generation>`.
	
	Example:
	
	`<jar-generation>`
	        `<schema>scott/tiger</schema>`
	        `<db-url>jdbc:oracle:thin:@system1:1521:orcl</db-url>`
	        `<prefix>sp.company</prefix>`
	        `<db-pkg-name>Company</db-pkg-name>`
	`</jar-generation>`

`<uri>` *URI* `</uri>`	This tag specifies servlet mapping pattern for the Servlet that implements the Web Service. The path specified as the *URI* is appended to the `<context>` to specify the Web Service location.

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

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

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

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

    <stateless-stored-procedure-java-service>
       <jar-generation>
         <schema>scott/tiger</schema>
         <db-url>jdbc:oracle:thin:@system1:1521:orcl</db-url>
         <prefix>sp.company</prefix>
         <db-pkg-name>Company</db-pkg-name>
       </jar-generation>
      <!-- Specifies the web service will be accessed in the uri named
             "statelessSP" within the servlet context. -->
       <uri>/statelessSP</uri>
       <database-JNDI-name>/jdbc/OracleDataSource</database-JNDI-name>
    </stateless-stored-procedure-java-service>
  <wsdl-gen>
    <wsdl-dir>wsdl</wsdl-dir>
    <!--force 'true'  will write over existing wsdl -->
    <option name="force">true</option>
    <!-- change this to point to your soap servers http listener -->
    <option name="httpServerURL">http://localhost:8888</option>
  </wsdl-gen>
  <proxy-gen>
    <proxy-dir>proxy</proxy-dir>
    <!-- include-source 'true'  will create an additional jar with only the proxy source-->
    <option name="include-source">true</option>
  </proxy-gen>
</web-service>
```

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

Using a configuration file that specifies the stored procedure `<class-name>` and `<interface-name>` assembly options when a pre-generated Jar file that includes the required classes to support the Web Service is available. The `<class-name>` and `<interface-name>` tags specified in a configuration file support using a previously generated Jar file that contains the Java classes that provide a mapping between the PL/SQL procedure or function and the Web Service.

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

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

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

Tag	Description

`<class-name>`
class
`</class-name>`
	The Stored Procedure Web Services Servlet definition requires a `<param-name>` with the value class-name and a corresponding `<param-value>` set to the fully qualified name of the Java class that accesses the PL/SQL Web Service implementation.
	
	You need to use the configuration file `<class-name>` tag to supply the class name for this parameter; you can find the class name in the Jar file you provide that is specified in the top level `<option name="source-path">` tag.

`<database-JNDI-name>`
source_JNDI_name
`</database-JNDI-name>`
	This tag specifies the JNDI name of the backend database.
	
	The `data-sources.xml` OC4J configuration file describes the database server source associated with the specified source_JNDI_name.

`<interface-name>`
interface
`</interface-name>`
	A Stored Procedure Web Services Servlet definition requires a `<param-name>` with the value interface-name, and a corresponding `<param-value>` set to the fully qualified name of the Java interface that specifies the methods to include in the stored procedure Web Service.
	
	The `<interface-name>` tag provides the name of the interface that tells the Web Service Servlet generation code which methods should be exposed as Web Services. You can find the interface name in the Jar file you provide that is specified in the top level `<option name="source-path">` tag.

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

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

See Also:

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

Adding WSDL and Client-Side Proxy Generation Tags

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

See Also: "Generating WSDL Files and Client Side Proxies" on page 9-4

Running WebServicesAssembler With Stored Procedure Web Services

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

Run the Oracle Application Server Web Services assembly tool, `WebServicesAssembler` as follows:

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

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

See Also:

- "Creating a Configuration File to Assemble Stored Procedure Web Services" on page 5-3
- "Running the Web Services Assembly Tool" on page 9-2

Setting Up Datasources in Oracle Application Server 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 Oracle Application Server 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 Oracle Application Server Web Services does not provide explicit support or tools to do this).

When using an emulated data source with CLOB or BLOB types in the stored procedure, the emulated data source must use the `location` attribute to specify the JNDI name. The name cannot be specified using the `ejb-location`.

> **See Also:** *Oracle Application Server Containers for J2EE User's Guide* in the Oracle Application Server 10*g* Documentation Library

Deploying Stored Procedure Web Services

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

> **See Also:** *Oracle Application Server Containers for J2EE User's Guide* in the Oracle Application Server 10*g* Documentation Library

Limitations for Stored Procedures Running as Web Services

This section covers the following topics:

- Supported Stored Procedure Features for Web Services
- Unsupported Stored Procedure Features for Web Services
- Database Server Release Limitation for Boolean Use in Oracle PL/SQL Web Services
- TIMESTAMP and DATE Granularity Limitation
- LOB (CLOB/BLOB) Emulated Data Source Limitation

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, OUT, IN, INOUT parameter modes. When a stored procedure contains OUT or INOUT parameters, the INOUT and OUT data are passed back to the client as attributes of the returned objects. The declared stored procedure return value, if the stored procedure is a function, will also be included as an attribute of the returned objects INOUT 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. Oracle 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. In this case the Java method names will be renamed to avoid compilation errors for the generated code. However, at runtime, the PL/SQL engine may report `PLS-00307` error (too many declarations of <*method name*> match this call). The error is due to PL/SQL limitation on overloading resolution.

5. Simple PL/SQL types

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

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

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

6. TIMESTAMP is supported, along with variations TIMESTAMP WITH LOCAL TIME ZONE and TIMESTAMP WITH TIME ZONE.

7. DATE is supported.

8. User-defined Object Types.

9. Oracle JPublisher and Oracle Application Server Web Services provide support for the following LOB types: BLOB, CLOB, and BFILE.

   If your PL/SQL procedures use LOB types as input/output types, then the WebServices Assembler will not publish those stored procedures that will cause runtime errors. For instance, the WebServices Assembler will not publish a method containing BFILE as an IN parameter.

10. SYS.XMLTYPE is supported. SYS.XMLTYPE is mapped into the type, org.w3c.dom.DocumentFragment in Web Services.

    See Also: *Oracle9i JPublisher User's Guide* in the Oracle Database Documentation Library

Unsupported Stored Procedure Features for Web Services

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

1. Only procedures and functions within a PL/SQL package are exported as Web Services. Top-level stored procedures must be wrapped inside a package. Methods must be wrapped into package-level methods with a default "this" reference.
2. NCHAR and related types are not supported.
3. Oracle JPublisher translates almost all PL/SQL types to Java types. The deployment tools for Stored Procedure Web Services generate "jdbc" style for builtin and number types and "oracle" style for user types and lob types. The lob types are converted to java types that can be serialized/deserialized by Web Services. The user types that conform to java beans are also serialized/deserialized by Web Services. Check the Oracle JPublisher documentation for full details of these styles, and for the caveats associated with them.
4. Fractional seconds in a TIMESTAMP value are not preserved when using Stored Procedure Web Services.
5. TIMESTAMP as a field in a user defined ADT is not supported. However, DATE as a field in a user defined ADT is supported.

See Also: *Oracle9i JPublisher User's Guide* in the Oracle Database Documentation Library

Database Server Release Limitation for Boolean Use in Oracle PL/SQL Web Services

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

The PL/SQL script that defines this package is located at the following location on UNIX:

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

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

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

TIMESTAMP and DATE Granularity Limitation

Fractional seconds in a TIMESTAMP value are not preserved when using Stored Procedure Web Services.

LOB (CLOB/BLOB) Emulated Data Source Limitation

When using an emulated data source with CLOB or BLOB types, the emulated data source must use the `location` attribute to specify the JNDI name. The name cannot be specified using the `ejb-location`.

6 Developing and Deploying Document Style Web Services

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

This chapter covers the following topics:

Using Document Style Web Services

This chapter describes Document Style Web Services that are implemented with Java classes and describes the difference between writing stateful and stateless Document Style Java Web Services.

The sample code is supplied on the Oracle Technology Network Web site,

http://otn.oracle.com/sample_code/tech/java/web_services/content.html

After expanding the Web Services `demo.zip` file, the Document Style Web Services samples are in the `stateless` and `stateful` directories under `webservices/demo/basic/java_doc__services` on UNIX or in `webservices\demo\basic\java_doc_services` on Windows.

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

See Also:

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

Writing Document Style Web Services

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

This section covers the following topics:

- Supported Method Signatures for Document Style Web Services
- Writing Stateless and Stateful Document Style Web Services

- Writing Classes and Interfaces for Document Style Web Services

Supported Method Signatures for Document Style Web Services

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

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

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

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

Passing Null Values for Document Style Web Services

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

Arrays of Elements

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

See Also:

- "Handling Messages for Document Style Web Services" on page 6-8
- `http://www.w3.org/DOM/` for information on the W3C Document Object Model (DOM)

Writing Stateless and Stateful Document Style Web Services

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

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

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

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

Writing Classes and Interfaces for Document Style Web Services

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

- Defining Methods in a Document Style Web Service
- Defining an Interface for Explicit Method Exposure
- Handling Messages for Document Style Web Services

Defining Methods in a Document Style Web Service

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

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

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

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

The `StatelessDoc` Web Service sample is implemented with `StatelessDocImpl`, a public class and the interface `StatelessDoc`. The `StatelessDocImpl` class defines two public methods: `displayElement()`, that displays the incoming message on the server where the web service runs, and `processElement()`, that takes an incoming message and returns a transformed message to the client. The private method `applyXSLtoXML()` is a helper method that transforms the incoming message, as specified in the `converter.xsl` file.

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

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

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

public class StatelessDocImpl implements StatelessDoc
```

```
{
    public StatelessDocImpl()
    {  }

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

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

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

The `StatfulDoc` Web Service sample is implemented with `StatefulDocImpl`, a public class and the interface `StatefulDoc`. The `StatefulDocImpl` class defines two public methods: `startShopping()` that initializes the state of the customer information and `makePurchase()`, that modifies the state of the customer information and returns the updated information to the client. The private method `processElement()` is a helper method that processes the customer's XML element representing a purchase and returns the updated XML element.

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

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

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

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

Defining an Interface for Explicit Method Exposure

Oracle Application Server Web Services allows you to limit the methods you expose as Document Style Web Services by supplying a public interface. To limit the methods exposed in a Web Service, include a public interface that lists the method signatures for the methods that you want to expose. Example 6–3 shows an interface for the methods in the class `StatelessDocImpl`. Example 6–4 shows an interface for the methods in the class `StatelefulDocImpl`.

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

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

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

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

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

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

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

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

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

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

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

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

Handling Messages for Document Style Web Services

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

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

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

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

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

Preparing Document Style Web Services

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

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

This section contains the following topics:

- Creating a Configuration File to Assemble Document Style Web Services
- Running WebServicesAssembler With Document Style Web Services

Creating a Configuration File to Assemble Document Style Web Services

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

Create `WebServicesAssembler` configuration file by adding the following:

- Adding Web Service Top Level Tags
- Adding Java Service Tags with Document Message Style Specified
- Adding WSDL and Client-Side Proxy Generation Tags

Adding Web Service Top Level Tags

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

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

Table 6–2 Top Level `WebServicesAssembler` ***Configuration Tags***

Tag	Description

`<context>`
context
`</context>`
	Specifies the context root of the Web Service.
	
	This tag is required.

`<datasource-JNDI-name>`
name
`</datasource-JNDI-name>`
	Specifies the datasource associated with the Web Service.

`<description>`
description
`</description>`
	Provides a simple description of the Web Service.
	
	This tag is optional.

`<destination-path>`
dest_path
`</destination-path>`
	Specifies the name of the generated J2EE .ear file output. The *dest_path* specifies the complete path for the output file.
	
	This tag is required.

`<display-name>`
disp_name
`</display-name>`
	Specifies the Web Service display name.
	
	This tag is optional.

`<option name=source-path">`
path
`<option>`
	Includes a specified file in the output .ear file. Use this option to specify java resources, or the name of an existing .war, .ear, or ejb-jar file that is used as a source file for the output J2EE .ear file.
	
	When a .war file is supplied as input, the optional contextroot specifies the root-context for the .war file.
	
	path1 specifies the context-root for the .war.
	
	path2 specifies the path to the file to include.
	
	For example:
	
	`<option name="source-path" contextroot="/test">/myTestArea/ws/src/statefull.war</option>`
	
	This tag is optional.

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

Tag	Description
`<stateless-java-service>` *sub-tags* `</stateless-java-service>`	Use this tag to add a Document Style Web Services that defines a stateless service. See Table 6–3 for a description of valid *sub-tags*.
`<stateful-java-service>` *sub-tags* `</stateful-java-service>`	Use this tag to add a Document Style Web Services that defines a stateful service. See Table 6–3 for a description of valid *sub-tags*.
`<temporary-directory>` *temp_dir* `</temporary-directory>`	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding Java Service Tags with Document Message Style Specified

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

- Adding Stateful Document Style Java Service Tags
- Adding Stateless Document Style Java Service Tags

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

Tag	Description
`<class-name>` *value* `</class-name>`	The Document Style Web Service definition requires at least one `<class-name>` tag. The *value* specifies the name of the Java class that provides the Document Style Web Service implementation. This tag is required.
`<interface-name>` *interface* `</interface-name>`	A Document Style Web Service configuration file supports the optional `<interface-name>` tag. The corresponding *interface* value supplied specifies the name of the Java interface that lists the methods to include in the Document Style Web Service. This tag is optional.
`<java-resource>` *resource* `</java-resource>`	This tag supports adding a Java *resource*. This specifies the location of the java resources to include in the Document Style Web Service. Include multiple `<java-resource>` tags to include multiple Java resources. This tag is optional

Table 6–3 (Cont.) Java Service WebServicesAssembler Configuration Tags - Document Style

Tag	Description

`<message-style>`
`doc`
`</message-style>`
	When defining a Document Style Web Service, you must include the `<message-style>` tag and specify the value `doc`.
	
	Valid Values: `doc`, `rpc`
	
	This tag is required for Document Style Web Services.
	
	Default value: `rpc` (when the `<message-style>` tag is not supplied)

`<scope>`
value
`</scope>`
	The `<scope>` tag only applies for stateful services. Use this tag only within the `<stateful-java-service>` tag.
	
	This tag is optional.
	
	Valid Values: `application`, `session`
	
	Default Value: `session`

`<session-timeout>`
value
`</session-timeout>`
	This optional parameter only applies for stateful services. Use this tag only within the `<stateful-java-service>` tag.
	
	Specify *value* with an integer that defines the timeout for the session timeout. session. The default value for the session timeout for stateful Java sessions where no session timeout is specified is 60 seconds.
	
	This tag is optional.

`<uri>`
URI
`</uri>`
	This tag specifies servlet mapping pattern for the Servlet that implements the Document Style Web Service. The path specified as the *URI* is appended to the `<context>` to specify the Document Style Web Service location.
	
	This tag is optional.

Adding Stateful Document Style Java Service Tags

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

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

Adding Stateless Document Style Java Service Tags

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

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

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

Adding WSDL and Client-Side Proxy Generation Tags

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

See Also: "Generating WSDL Files and Client Side Proxies" on page 9-4

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

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

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

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

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

    <!-- generate the proxy -->

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

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

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

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

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

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

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

</web-service>
```

Running WebServicesAssembler With Document Style Web Services

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

Run the Oracle Application Server Web Services assembly tool, `WebServicesAssembler` as follows:

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

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

See Also:

- "Creating a Configuration File to Assemble Document Style Web Services" on page 6-9
- "Running the Web Services Assembly Tool" on page 9-2

Deploying Document Style Web Services

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

See Also: *Oracle Application Server Containers for J2EE User's Guide* in the Oracle Application Server 10*g* Documentation Library

7

Developing and Deploying JMS Web Services

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

Oracle Application Server Web Services supports asynchronous message facilities with JMS Web Services.

This chapter covers the following topics:

JMS Web Services Overview

This section covers the following topics:

- Using JMS Web Services
- JMS Web Services Backend Message Processing

Using JMS Web Services

The sample code for JMS Web Services is supplied on the Oracle Technology Network Web site,

http://otn.oracle.com/sample_code/tech/java/web_services/content.html

After expanding the Web Services `demo.zip` file, the samples are in the `demo1` and `demo2` directories under `webservices/demo/basic/jms_service` on UNIX and `webservices\demo\basic\jms_service`.

JMS Web Services examples show both OC4J/JMS and Oracle JMS. In the samples, `demo1` uses OC4J/JMS and `demo2` uses Oracle JMS.

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

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

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

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

JMS Web Services Backend Message Processing

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

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

- Using an MDB for Message Processing
- Using a JMS Client for Message Processing

Using an MDB for Message Processing

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

Figure 7–1 MDB Based JMS Web Service

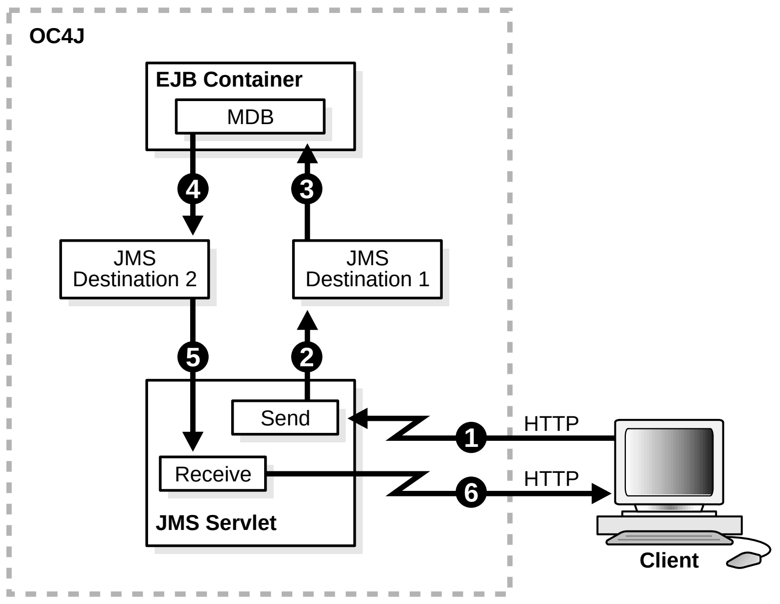


Figure 7–1 includes an MDB that is configured to listen to a JMS destination. The MDB based JMS Web Service works with the following steps:

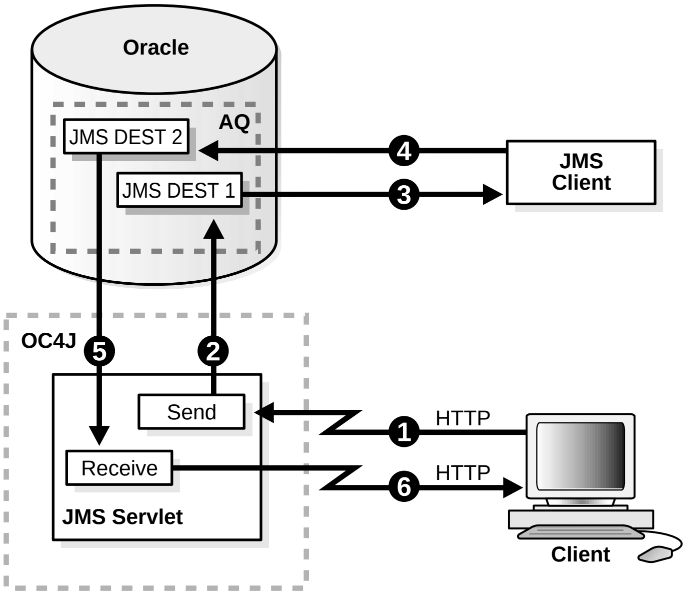
1. A JMS Web Service client performs a send operation on the JMS Web Service to send a message.
2. The JMS Web Service processes the incoming message and directs it to a JMS destination, JMS Destination 1.
3. The EJB container invokes the MDB listening on JMS Destination 1.
4. After processing the message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

Using a JMS Client for Message Processing

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

For example, Figure 7–2 shows a JMS Web Service that use a server-side JMS client for message processing.

Figure 7–2 JMS Client Based JMS Web Service



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

1. A JMS Web Service client performs a send operation on the JMS Web Service to send a message.
2. The JMS Web Service then processes the incoming message and directs it to JMS DEST 1.

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

Writing JMS Web Services and Handling Messages

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

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

This section covers the following:

- Using an MDB for Backend Message Processing
- Using a JMS Standalone Program for Backend Message Processing
- Message Processing and Reply Messages

See Also:

- "Preparing and Configuring JMS Web Services" on page 7-11
- Chapter 4, "Developing and Deploying EJB Web Services"

Using an MDB for Backend Message Processing

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

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

- Developing the MDB that Processes Incoming Messages
- Developing the MDB that Generates Outgoing Messages
- Compiling and Preparing the MDB EJB.jar File

- Assembling the JMS Web Service With the MDB
- Defining the Server-Side Resource References

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

Developing the MDB that Processes Incoming Messages

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

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

Example 7–1 shows an MDB method that handles an incoming JMS Message. Also see `MessageBean.java` in the `demo1` directory for the complete code.

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

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

```
      }
    } catch (JMSException ex) {
      ex.printStackTrace();
      mdc.setRollbackOnly();
    } catch (Throwable te) {
      te.printStackTrace();
    }
  }
```

Developing the MDB that Generates Outgoing Messages

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

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

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

Example 7–2 Sample Outgoing Message for JMS Web Service

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

Compiling and Preparing the MDB EJB.jar File

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

Assembling the JMS Web Service With the MDB

Assemble the MDB's EJB.jar file with the JMS Web Service .ear file using the `WebServicesAssembler` tool and a configuration file containing the top-level tag

`<option name=source-path">` that specifies the EJB .jar, and the `<jms-doc-service>` that defines the JMS Web Service configuration.

See Also:

- "Preparing and Configuring JMS Web Services" on page 7-11
- "Deploying JMS Web Services" on page 7-18

Defining the Server-Side Resource References

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

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

See Also: Chapter 3, "AQ Programmatic Environments" in the *Application Developer's Guide - Advanced Queuing* in the Oracle9*i* Database Documentation library

Using a JMS Standalone Program for Backend Message Processing

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

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

See Also:

- "Using an MDB for Backend Message Processing" on page 7-6
- "Deploying JMS Web Services" on page 7-18

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

Message Processing and Reply Messages

The JMS Web Service processes an incoming message, a JMS Web Service `send` operation message, and places the message on a JMS destination. This section covers details that a developer needs to know to consume and process the JMS messages that originate from a JMS Web Service.

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

```
JMSType
JMSReplyTo
JMSExpiration
JMSPriority
JMSDeliveryMode
```

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

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

Example 7–3

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

See Also:

- "Developing the MDB that Processes Incoming Messages" on page 7-7
- "Adding JMS Doc Service Tags" on page 7-13

Preparing and Configuring JMS Web Services

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

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

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

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

This section contains the following topics:

- Creating a Configuration File to Assemble JMS Web Services
- Running WebServicesAssembler With JMS Web Services

Creating a Configuration File to Assemble JMS Web Services

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

Create `WebServicesAssembler` configuration file by adding the following:

- Adding Web Service Top Level Tags
- Adding JMS Doc Service Tags
- Adding WSDL and Client-Side Proxy Generation Tags

Adding Web Service Top Level Tags

Table 7–1 describes the top level `WebServicesAssembler` configuration file tags. Add these tags to provide top level information describing the JMS Web Service.

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

Table 7–1 Top Level `WebServicesAssembler` ***Configuration Tags***

Tag	Description
`<context>` *context* `</context>`	Specifies the context root of the Web Service. This tag is required.
`<datasource-JNDI-name>` *name* `</datasource-JNDI-name>`	Specifies the datasource associated with the Web Service.
`<description>` *description* `</description>`	Provides a simple description of the Web Service. This tag is optional.

Table 7–1 (Cont.) Top Level `WebServicesAssembler` ***Configuration Tags***

Tag	Description
`<destination-path>` *dest_path* `</destination-path>`	Specifies the name of the generated J2EE .ear file output. The *dest_path* specifies the complete path for the output file. This tag is required.
`<display-name>` *disp_name* `</display-name>`	Specifies the Web Service display name. This tag is optional.
`<option name="source-path">` *path* `<option>`	Includes a specified file in the output .ear file. Use this option to specify java resources, or the name of an existing .war, .ear, or ejb-jar file that is used as a source file for the output J2EE .ear file. When a .war file is supplied as input, the optional contextroot specifies the root-context for the .war file. *path1* specifies the context-root for the .war. *path2* specifies the path to the file to include. For example: `<option name="source-path" contextroot="/test">/myTestArea/ws/src/statefull.war</option>` This tag is optional.
`<jms-doc-service>` *sub-tags* `</jms-doc-service>`	Use this tag to add a JMS Web Service. See Table 7–2 for a description of the valid *sub-tags*.
`<temporary-directory>` *temp_dir* `</temporary-directory>`	Specifies a directory where the assembler can store temporary files. This tag is optional.

Adding JMS Doc Service Tags

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

Table 7–2 JMS Service `WebServicesAssembler` ***Configuration Tags***

Tag	Description
`<connection-factory-resource-ref>` *resource-ref* `</connection-factory-resource-ref>`	Specifies the Topic Connection Factory or Queue Connection Factory resource reference *resource-ref* for the JMS destination associated with the JMS Web Service. This tag is required.
`<jms-delivery-mode>` *delivery-mode* `</jms-delivery-mode>`	Sets the `JMSDeliveryMode` message header to the specified *delivery-mode* value for the JMS message that is created with a `send` operation. This tag is valid when the `<operation>` value is: `send` or `both` This tag is optional.
`<jms-expiration>` *expiration* `</jms-expiration>`	Sets the `JMSExpiration` message header to the specified *expiration* value for the JMS message that is created with a `send` operation. This tag is valid when the `<operation>` value is: `send` or `both` This tag is optional.
`<jms-message-type>` *message-type* `</jms-message-type>`	Sets the `JMSType` for the message to the specified *message-type* for the JMS message that is created with a `send` operation This tag is valid when the `<operation>` value is: `send` or `both` This tag is optional.
`<jms-priority>` *priority* `</jms-priority>`	Sets the `JMSPriority` message header to the specified *priority* value for the JMS message that is created with a `send` operation. This tag is valid when the `<operation>` value is: `send` or `both` This tag is optional.
`<receive-timeout>` *priority* `</receive-timeout>`	Provides a configurable timeout value to specify the receive timeout in milliseconds. This specifies the time in milliseconds that a receive operation waits for a new message. This tag is valid when the `<operation>` value is: `receive` or `both` When this tag is not specified or when the value is set to 0, a JMS receive operation blocks indefinitely. Valid values are 0 and positive integers. Default value: 0 This tag is optional.

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

Tag	Description
`<operation>` *op* `</operation>`	Specifies the operation *op* that the JMS Web Service supports. Using the `send` and `receive` operation: ▪ If the destination is a JMS Queue, `send` means enqueue, and `receive` means dequeue. ▪ If the destination is a topic, `send` means publish and `receive` means subscribe. The send operation uses the `<connection-factory-resource-ref>` and the corresponding JMS destination `<queue-resource-ref>` or `<topic-resource-ref>` to determine the JMS destination for a send operation on the service. With the receive operation, when the `<reply-to-connection-factory-resource-ref>` tag is not set, then the `receive` operation uses the `<connection-factory-resource-ref>` and the corresponding JMS destination `<queue-resource-ref>` or `<topic-resource-ref>`. When the `<reply-to-connection-factory-resource-ref>` tag is set, then the `<reply-to-*>` tags specify the JMS destination for `receive` operations. Valid values: `send`, `receive`, `both` Default value: `both` This tag is optional.
`<queue-resource-ref>` *queue-ref* `</queue-resource-ref>`	Specifies the resource reference *queue-ref* of the destination JMS queue. Either a `<topic-resource-ref>` or a `<queue-resource-ref>` must be specified, but not both. When a `<queue-resource-ref>` is specified, the `<connection-factory-resource-ref>` must refer to a corresponding Queue connection factory.
`<reply-to-connection-factory-resource-ref>` *reply-to-conn-factory-res-ref* `</reply-to-connection-factory-resource-ref>`	If the `<operation>` specified is `both`, then `receive` operations use the `<reply-to-connection-factory-resource-ref>`. The specified *reply-to-conn-factory-res-ref* value specifies the JMS destination connection factory for `receive` operations. Also, if the MDB, or any JMS consumer, expects to send results back then the name of the destination connection factory to which the reply message will be sent has to be specified in this parameter. See Also: "Message Processing and Reply Messages" on page 7-10. This tag is optional.

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

Tag	Description
`<reply-to-queue-resource-ref>` *reply-to-queue-res-ref* `</reply-to-queue-resource-ref>`	Specifies the resource reference *reply-to-queue-res-ref* of the destination JMS queue. When a `<reply-to-queue-resource-ref>` is specified, the `<reply-to-connection-factory-resource-ref>` must refer to a corresponding Queue connection factory. If the `<reply-to-connection-factory-resource-ref>` tag is set, then either a `<reply-to-topic-resource-ref>` or a `<reply-to-queue-resource-ref>` must be specified, but not both. This tag is optional.
`<reply-to-topic-resource-ref>` *reply-to-topic-res-ref* `</reply-to-topic-resource-ref>`	Specifies the resource reference *reply-to-topic-res-ref* of the destination JMS Topic. When a `<reply-to-topic-resource-ref>` is specified, the `<reply-to-connection-factory-resource-ref>` must refer to a corresponding Topic connection factory. If the `<reply-to-connection-factory-resource-ref>` tag is set, then either a `<reply-to-topic-resource-ref>` or a `<reply-to-queue-resource-ref>` must be specified, but not both. This tag is optional.
`<topic-resource-ref>` *topic-ref* `</topic-resource-ref>`	Specifies the resource reference *topic-ref* of the destination JMS Topic. Either a `<topic-resource-ref>` or a `<queue-resource-ref>` must be specified, but not both. When a `<topic-resource-ref>` is specified, the `<connection-factory-resource-ref>` must refer to a corresponding Topic connection factory.
`<topic-subscription-name>` *topic-name* `</topic-subscription-name>`	When a JMS provider supports durable JMS topics, the JMS Doc service supports using the durable topics. To specify a durable topic, use this tag to specify the *topic-name*. This tag is only valid when a `<topic-resource-ref>` is supplied. This tag is optional.
`<uri>` *URI* `</uri>`	This tag specifies servlet mapping pattern for the Servlet that implements the JMS Web Service. The path specified as the *URI* is appended to the `<context>` to specify the JMS Web Service location. This tag is optional.

Adding WSDL and Client-Side Proxy Generation Tags

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

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

> **See Also:** "Generating WSDL Files and Client Side Proxies" on page 9-4

Example 7–4 Sample WebServicesAssembler Configuration File for JMS Web Service

```
<web-service>
  <display-name>JMS Web Service Example</display-name>
  <description>JMS Web Service Example</description>
   <!-- Name of the destination -->
  <destination-path>./jmsws1.ear</destination-path>
  <temporary-directory>./tmp</temporary-directory>
  <!-- Context root of the application -->
  <context>/jmsws1</context>
  <!-- Path of the jar file with MDBs definied/implemented in it -->
  <option name="source-path">MDB/mdb_service1.jar</option>

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

  <jms-doc-service>
    <uri>JmsReceive</uri>
    <connection-factory-resource-ref>jms/logQueueConnectionFactory</connection-factory-resource-ref>
    <queue-resource-ref>jms/logQueue</queue-resource-ref>
    <operation>receive</operation>
   </jms-doc-service>
    <!-- generate the wsdl -->
    <wsdl-gen>
           <wsdl-dir>wsdl</wsdl-dir>
           <!-- over-write a pregenerated wsdl , turn it 'false' to use the pregenerated wsdl-->
           <option name="force">true</option>
           <option name="httpServerURL">http://localhost:8888</option>
           <!-- do not package the wsdl -generate it again on the server-->
           <option name="packageIt">false</option>
    </wsdl-gen>
    <!-- generate the proxy -->
    <proxy-gen>
```

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

Running WebServicesAssembler With JMS Web Services

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

Run the Oracle Application Server Web Services assembly tool, `WebServicesAssembler` as follows:

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

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

See Also:

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

Deploying JMS Web Services

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

See Also: *Oracle Application Server Containers for J2EE User's Guide* in the Oracle Application Server 10*g* Documentation Library

Limitations for JMS Web Services

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

8 Building Clients that Use Web Services

This chapter describes the Oracle Application Server Web Services features that allow you to easily create and run a client application that uses Oracle Application Server 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 10, "Discovering and Publishing Web Services" provides detailed information on looking up Web Services in a UDDI registry.

Using a static client Oracle Application Server Web Services provides several options for locating Oracle Application Server Web Services, including:

- Using a known Web Service located at a known URL.
- Using Oracle Application Server 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 Oracle Application Server Web Services generates includes the URL to locate the associated Web Service.
- Using Oracle Application Server 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 Oracle Application Server Web Services generates includes the URL to locate the associated Web Service.

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

See Also: Chapter 10, "Discovering and Publishing Web Services"

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

This section covers the following:

- Using the Web Service Home Page to Save WSDL and Client Side Proxies
- Getting Web Service WSDL and Client-Side Proxies Directly
- Generating Client-Side Proxies With WebServicesAssembler

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

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

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

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

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

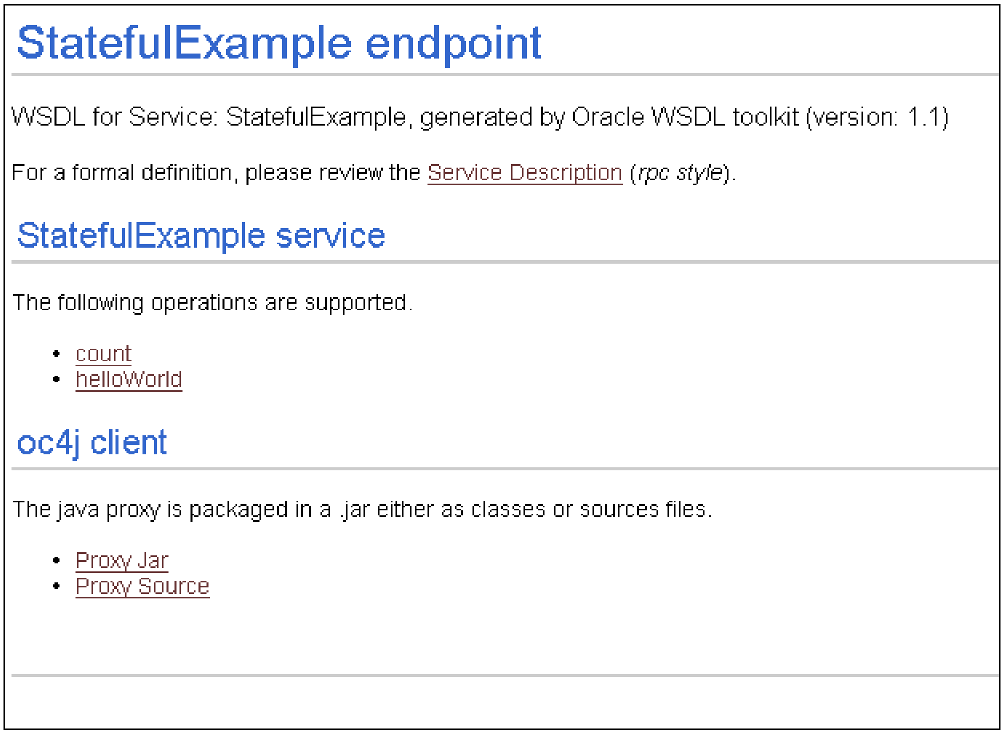
Figure 8–1 shows the Web Service Home Page for StatefulExample, at the following endpoint,

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

A Web Service Home Page provides the following:

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

Figure 8–1 Web Service Home Page


StatefulExample endpoint

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

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

StatefulExample service

The following operations are supported.

- count
- helloWorld

oc4j client

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

- Proxy Jar
- Proxy Source

Limitations for Web Service Test Pages

Web Service Test Pages have the following limitations:

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

Getting Web Service WSDL and Client-Side Proxies Directly

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

This section covers the following:

- Getting WSDL Service Descriptions
- Getting Client-Side Proxy Jar and Client-Side Proxy Source Jar
- Getting Client-Side Proxy Jar and Client-Side Proxy Source by Package

Getting WSDL Service Descriptions

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

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

or

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

This command returns a WSDL description in the form service`.wsdl`. The service`.wsdl` description contains the WSDL for the Web Service named service, located at the specified URL. Using the WSDL that you obtain, you can build a client application to access the Web Service.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or

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

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

Table 8–1 URL for Accessing Client Side Proxy Stubs

URL Component	Description
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 Oracle Application Server 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 Oracle Application Server 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"

Generating Client-Side Proxies With WebServicesAssembler

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

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

Note: When you are generating client-side proxies and you are accessing an external WSDL file from behind a firewall, make sure to set the appropriate security properties shown in Table 8–4, such as `http.proxyHost` and `http.proxyPort`.

Example 8–1 shows a sample `WebServicesAssembler` that includes a `<proxy-gen>` tag.

Example 8–1 WebServicesAssembler Proxy Gen Configuration File

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

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

Tag	Description
`<proxy-dir>` *directory* `</proxy-dir>`	Specifies the directory for the generated client-side proxy stubs Jar file that is included in the generated Web Service `.ear` file. This tag is required.
`<option name="include-source">` *value* `</option>`	Setting *value* to `true` tells `WebServicesAssembler` to include the classes and the source in the generated client-side proxy. When the value is false, the source is not included in the generated Jar. This tag is optional. Valid values: `true`, `false` Default value: `false`

`<option name="wsdl-location">` *URL* `</option>`
 	
or
 	
`<option name="wsdl-location" package-name="`*package*`">` *URL* `</option>`
	This tag sets the *URL* to use for the source WSDL to use to generate the client-side proxy.
	
	This option also supports the optional attribute `package-name`. The `package-name` can specify the name *package* for the generated client-side proxy.
	
	This tag is optional.
	
	Examples:
	
	`<option name="wsdl-location">` `http://system1:8888/webservice3/TestService?WSDL` `</option>`
	
	`<option name="wsdl-location" package-name="myPackage.proxy">` `http://system1:8888/webservice3/TestService?WSDL` `</option>`

See Also: Chapter 9, "Web Services Tools"

Working with Client-Side Proxy Jar to Use Web Services

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

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

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

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

Note: When you are accessing an external Web Service from behind a firewall, make sure to set the appropriate security properties shown in Table 8–4, such as `http.proxyHost` and `http.proxyPort`.

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

```
public class StatefulExampleProxy {

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

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

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

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

Setting the Web Services Proxy Client CLASSPATH

When you build a Web Services clients using a proxy, you need to use the correct `CLASSPATH` to run the client. Table 8–3 lists jars that you need to include in the `CLASSPATH`.

Table 8–3 Web Services CLASSPATH Components for a Client Using a Client-side Proxy

Component Jar	Description
proxy`.jar`	The *proxy* jar file that provides access to the Web Service.
`$ORACLE_HOME/lib/xmlparserv2.jar`	The Oracle XML parser jar.
`$ORACLE_HOME/j2ee/home/lib/http_client.jar`	The Oracle HTTP client jar.
`$ORACLE_HOME/soap/lib/soap.jar`	The Oracle SOAP jar.
`$ORACLE_HOME/j2ee/home/lib/mail.jar`	Generally, this is available in the JRE. If this is not available in the JRE, then include it in the CLASSPATH.
`$ORACLE_HOME/j2ee/home/lib/activation.jar`	Generally, this is available in the JRE. If this is not available in the JRE, then include it in the CLASSPATH
`$ORACLE_HOME/jlib/javax-ssl-1_1.jar`	Used when the client uses SSL to connect to a Web Service that uses SSL. In this case, do not include `$ORACLE_HOME/lib/jsee.jar` in the `CLASSPATH`.
`$ORACLE_HOME/jlib/jssl-1_1.jar`	Required when the client is using SSL to connect to a Web Service that uses SSL. In this case, either `$ORACLE_HOME/jlib/javax-ssl-1_1.jar` or `$ORACLE_HOME/lib/jsse.jar` must be specified.
`$ORACLE_HOME/lib/jsse.jar`	Used when the client uses SSL to connect to a Web Service that uses SSL. In this case, do not include `$ORACLE_HOME/jlib/javax-ssl-1_1.jar` in the `CLASSPATH.`
`$ORACLE_HOME/webservices/lib/wsdl.jar`	Required when the client is using a Dynamic Proxy.
`$ORACLE_HOME/webservices/lib/dsv2.jar`	Required when the client is using a Dynamic Proxy.

Using Java Beans as Parameters for Web Services

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

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

Using Web Services Security Features

When you run a client-side application that uses Oracle Application Server Web Services, you can access secure Web Services by setting properties in the client application. Table 8–4 shows the available properties that provide credentials and other security information for Web Services clients. Table 8–3 lists jar file that need to be included in the CLASSPATH, including those required to support SSL.

In a Web Services client application, you can set the security properties shown in Table 8–4 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 8–4 Web Services HTTP Transport Security Properties

Property	Description
`http.authRealm`	Specifies the realm for which the HTTP authentication username/password is specified. This property is mandatory when using basic authentication.
`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.proxyAuthRealm`	Specifies the realm for which the proxy authentication username/password is specified.
`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.proxyUsername`	Specifies the HTTP proxy authentication username.
`http.username`	Specifies the HTTP authentication username.

Table 8–4 (Cont.) Web Services HTTP Transport Security Properties

Property	Description
`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 Oracle 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.

Table 8–4 (Cont.) Web Services 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.

Working with WSDL Files and Oracle JDeveloper to Use Web Services

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

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

Oracle JDeveloper supports Oracle Application Server 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 Oracle Application Server Web Services. Currently, many IDEs have the capability to create SOAP requests, given a WSDL description for the service.

9 Web Services Tools

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

This chapter contains the following topics:

Running the Web Services Assembly Tool

Run the Web Services assembly tool as follows:

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

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

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

Note: When running `WebServicesAssembler.jar` from the command line, the `PATH` environment variable should include the `JDK/bin` directory (the directory with the `javac` compiler).

Web Services Assembly Tool Configuration File Sample

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

See Also:

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

Example 9–1 Sample Web Services Assembly Tool Configuration File

```
<web-service>

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

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

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

  </web-service>
```

Web Services Assembly Tool Configuration File Sample Output

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

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

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

Generating WSDL Files and Client Side Proxies

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

This section covers the following topics:

- Generating and Assembling WSDL Files
- Generating Client-Side Proxies with WSDL

Generating and Assembling WSDL Files

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

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

   Example 9–3 shows a configuration file that includes the `<wsdl-gen>` tag.

2. Allowing the Oracle Application Server Web Services runtime to generate the WSDL file when the WSDL is requested by a Web Service client (after the WEB Service is deployed). In this case, you do not specify the `<wsdl-gen>` tag in the configuration file.

3. Creating a WSDL file manually. In this case, use the `<wsdl-gen>` tag during assembly of the J2EE .ear file to specify the path to the WSDL file. At assembly time when the Web Service is prepared, the `WebServicesAssembler` packages the WSDL file with the Web Service.

Table 9–1 describes the `<wsdl-gen>` `WebServicesAssembler` configuration file sub-tags.

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

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

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

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

```
<web-service>

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

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

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

</web-service>
```

Manually Producing a WSDL File

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

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

Generating Client-Side Proxies with WSDL

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

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

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

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

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

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

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

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

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

  </web-service>
```

Web Services Assembly Tool Configuration File Specification

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

Example 9–5 shows the Web Services Assembly Tool Configuration file DTD.

Example 9–5   Assembly Tool Input File DTD

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

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

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

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

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

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

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

Web Services Assembly Tool Limitations

The `WebServicesAssembler` tool has the following limitations:

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

10 Discovering and Publishing Web Services

Oracle Application Server Containers for J2EE (OC4J), provides a Universal Discovery Description and Integration (UDDI) Web Services registry known as the Oracle Application Server UDDI Registry, in which Web Services provider administrators in an enterprise environment can publish their Web Services for use by Web Services consumers (programmers). Web Services consumers can use the UDDI inquiry interface to discover these published Web Services by browsing, searching, and drilling down in the OracleAS UDDI Registry to select one or more Web Services from among those registered, and use those services in their applications for a particular enterprise process.

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

This chapter is organized into the following main sections:

As part of the OC4J OracleAS UDDI Registry, a SOAP API as defined by the UDDI v2 specification is provided to be used primarily by Web Services application developers (see the OracleAS SOAP API Reference Javadoc on the Oracle Application Server 10*g* (9.0.4) Documentation CD-ROM). This API provides the inquiry and publishing functions by implementing the inquiry and publishing API defined by the UDDI v2 specification. The use of this API is described in Web Services Discovery on page 10-7 and Web Services Publishing on page 10-11.

In addition, a set of management facilities and tools are provided for all management and operational requirements of the registry as described in OracleAS UDDI Registry Administration on page 10-25. Some of these tools are provided through Oracle Enterprise Manager as described in Web Services Publishing on page 10-11.

A Java-based client library is also provided to facilitate additional tool development and application development (see the Oracle Application Server UDDI Client API Reference Javadoc on the Oracle Application Server 10*g* (9.0.4) Documentation CD-ROM).

UDDI open database support is provided for Microsoft SQL Server, IBM DB2, and Oracle (non-infrastructure) databases as described in UDDI Open Database Support on page 10-71.

Finally, OracleAS UDDI Registry leveraging OracleAS Syndication Services provides a subscription service allowing publishers in the registry to monitor or obtain changes in the registry (see UDDI Subscription Service on page 10-79). See Subscribing to an Offer on page 10-83 for information about using the UDDI Content Subscription Manager that allows publishers and administrators to subscribe to offers from content providers through specialized content connectors managed by OracleAS Syndication Services.

UDDI Registration on page 10-2 describes the types of searches that can be performed in a UDDI registration, describes an overview of the data structure of a UDDI registry as specified by the UDDI v2 specification, and finally describes a subset of the Oracle implementation of the UDDI registry as support for Web Services discovery and Web Services publishing.

UDDI Registration

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

1. 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.
2. Yellow pages topical search -- containing industrial categories based on standard classifications, such as NAICS, ISO-3166, and UNSPSC.
3. Green pages service search -- containing technical information about Web Services that are exposed by a business, including references to specifications of interfaces for Web Services, as well as support for pointers to various file and URL-based discovery mechanisms.

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

UDDI Registry Data Structure

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

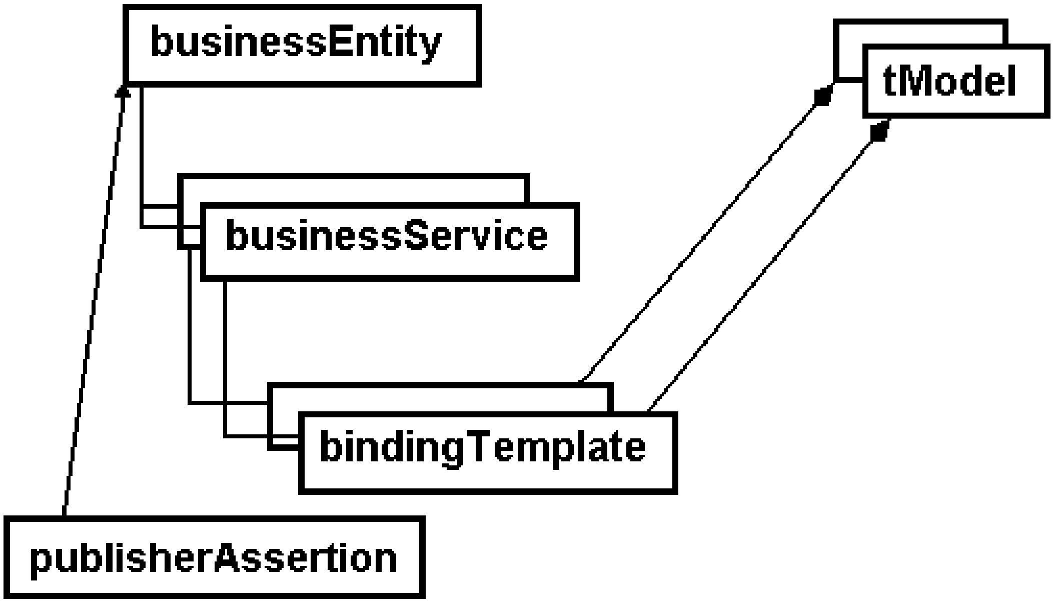
1. businessEntity -- the top-level, logical parent data structure; contains descriptive information about the business that publishes information about Web Services, such as business services, categories, contacts, discovery URLs, and identifier and category information that is useful for performing searches.
2. businessService -- the logical child of a single businessEntity data structure as well as the logical parent of a bindingTemplate structure; contains descriptive business service information about a particular family of technical services including its name, brief description, technical service description, and category information that is useful for performing searches.
3. bindingTemplate -- the logical child of a single businessService data structure; contains technical information about a Web Services entry point and references to interface specifications.
4. tModel -- description of specifications for Web Services, or a classification that forms the basis for technical identification; represents the technical specification of Web Services, in order to facilitate Web Services consumer searching for registered Web Services that are compatible with a particular technical specification. That is, based on the descriptions of the specifications for Web

Services in the tModel structure, Web Services consumers can easily identify other compatible Web Services.

5. publisherAssertion -- information about a relationship between two parties, asserted by one or both.

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

Figure 10–1 UDDI Information Model Showing the Relationship Among the Five Main Data Structure Types



Because UDDI makes use of XML and SOAP, each of these data structure types contains a number of elements and attributes that further serve to describe a business or to have a technical purpose. See *UDDI Version 2.03, Data Structure Reference Published Specification, Dated 19 July 2002* and *UDDI Version 2.04 API, Published Specification Dated 19 July 2002* for a complete description of the UDDI service description framework, http://www.uddi.org/specification.html. This description includes the 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.

See Standard Classification Support on page 10-45 for more information about the standard classifications that are supported in the OracleAS UDDI Registry.

OracleAS UDDI Registry for Enterprise Web Services

This section describes a subset of features that provide UDDI support for Web Services.

The OracleAS UDDI Registry support for Web Services deployed in OC4J is composed of the following parts:

- Web Services discovery -- consumers can use the Inquiry API available for Java programmers to implement their own Web Services discovery tool to search, locate, and drill down to discover OC4J Web Services in the OracleAS UDDI Registry, as well as in any other accessible UDDI v1.0 Web Services registry. See Using the Inquiry API on page 10-7 for more information about using the Inquiry API and locating the Javadoc documentation that describes the Inquiry API.
- Web Services publishing -- Web services provider administrators can deploy OC4J Web Services using Oracle Enterprise Manager. As part of the deployment process, the administrator can also publish the OC4J container, and in this process, there is a step where you can publish Web Services to the Oracle UDDI Registry.

  Web Services provider administrators can also update published Web Services by searching, locating, and drilling down to OC4J Web Services using the **Application Server: <Instance-name>: OC4J home: Administration: Related Links: UDDI Registry** link provided through Oracle Enterprise Manager.
- Replication management -- allows administrators to create a logical registry that comprises one or more Oracle UDDI implementations and UDDI implementations from other vendors that also implement the UDDI v2.0 Replication Specification.
- Subscription service -- allows publishers in the registry to monitor or obtain changes in the registry through subscriptions created using OracleAS Syndication Services.

Installation and First Use

The OracleAS UDDI Registry is preinstalled with Oracle Application Server and available through the following URLs:

- Getting started information:
  `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddi/`
- UDDI inquiry SOAP endpoint:
  `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddi/inquiry`
- UDDI publishing SOAP endpoint:
  `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddi/publishing`
- UDDI administration endpoint:
  `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddi/admin`
- UDDI replication SOAP endpoint:
  `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddirepl/replication`
- UDDI replication HTTPS Wallet Password Administration endpoint:
  `http://`*`<OracleAS-host>:<OracleAS_port>`*`/uddirepl/admin/wallet`
- Subscription management application:
  `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddisub/subscription/ui`

See User Management on page 10-26 for the set of UDDI users and groups available to help you get started.

Automatic Postinstallation Configuration

A postinstallation configuration step is necessary to set up the following:

- UDDI core tModels
- A node businessEntity representing the registry node
- The businessEntity discoveryURL prefix and the operatorName

Postinstallation configuration is done automatically when you try to access (either through the browser or SOAP invocation programmatically) the UDDI inquiry or publishing SOAP endpoints.

As a result, if you have not accessed the inquiry or publishing SOAP endpoints and try to access other UDDI features, such as subscription management, Oracle Enterprise Manager integrated Web Services deployment and publishing, and so forth, those features will not function.

Web Services Discovery

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

Using Tools

Consumers can use Oracle9*i* JDeveloper release 9.0.3, create their own inquiry browse tool, or use third-party tools to browse and drill down for information about Web Services from theOracleAS UDDI Registry, as well as from any other accessible UDDI v1.0 Web Services registry. Consumers can use the Inquiry API available for Java programmers to implement their own Web Services discovery interface.

Using the Inquiry API

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

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

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

A set of sample demonstration (`uddidemo.zip`) files are located on the Oracle technology Network (OTN) Web site
http://otn.oracle.com/tech/java/oc4j/demos/ .

Within the `uddidemo.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 Application Server UDDI client library.

The program example does the following:

- Gets an instance of SoapTransportLiaison. This is an implementation that handles the details of communication between the UDDI client and server using SOAP 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 OracleAS UDDI Registry on the command line, using parameters, such as `-Dhttp.proxyHost=<`*`hostname`*`> -Dhttp.proxyPort=<`*`portnum`*`>`.

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

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

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

  Note: The UddiClient instance, by default, operates as a UDDI v2.0 client (the latest version supported). If a specific version is needed, the version can be specified either through another constructor, or the JVM property `oracle.uddi.client.defaultVersion`.

  For example:

  ```
  -Doracle.uddi.client.defaultVersion=1
  ```

- Uses the UddiClient instance to perform a find business request. Specifically, it finds all business entities that start with the letter *T* and prints out the response.

Note that input parameters and return values are objects that precisely mimic the XML elements defined in the UDDI specification.

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

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

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

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

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

- Finds tModel operations with multiple arguments. This is a new UDDI v2.0 feature. A find_xx request now allows multiple arguments. For example, find tModel operations that have a name pattern, such as "uddi%inquiry%" and are classified as wsdlSpec or xmlSpec in uddi-org:types taxonomy.

```
System.out.println("\nListing tModels with the name pattern
\"uddi%inquiry%\" ");
System.out.println("and classified as \"wsdlSpec\" or \"xmlSpec\" ");
System.out.println("under uddi-org:types taxonomy.");
// Use the UddiElement factory to create UDDI-specific objects
// that are needed in inquiries.
CategoryBag cbTM = (CategoryBag)uddiEltFactory.createCategoryBag();
KeyedReference krTM1 =
 (KeyedReference)uddiEltFactory.createKeyedReference();
krTM1.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UDDI_TYPE);
krTM1.setKeyValue(CoreTModelConstants.UDDI_TYPE_VALUE_WSDL_SPEC);
cbTM.addUddiElement(krTM1);

KeyedReference krTM2 =
 (KeyedReference)uddiEltFactory.createKeyedReference();
krTM2.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UDDI_TYPE);
krTM2.setKeyValue(CoreTModelConstants.UDDI_TYPE_VALUE_XML_SPEC);
cbTM.addUddiElement(krTM2);

FindQualifiers fqTM =
 (FindQualifiers)uddiEltFactory.createFindQualifiers();
List listFQTM = uddiEltFactory.createList();
listFQTM.add(FindQualifiers.OR_ALL_KEYS);
```

```
fqTM.setFindQualifierStringList(listFQTM);

// Actual find tModel operation:
// Integer(10) means a maximum of 10 tModel operations are
// to be returned.
//
TModelList tModelList =
  uddiClient.findTModel("uddi%inquiry%",
                        null,
                        cbTM,
                        fqTM,
                        new Integer(10));

// Print some response information.
System.out.println("The response is: ");
List listTModelInfo =
 tModelList.getTModelInfos().getUddiElementList();
for (int i = 0; i < listTModelInfo.size(); i++) {
  TModelInfo tModelInfo = (TModelInfo)listTModelInfo.get(i);
  System.out.println(tModelInfo.getTModelKey());
  System.out.println("name=" + tModelInfo.getName());
  }
```

- Closes the UddiClient instance when finished to release resources.

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

- Provides URLs (in comments) to the OracleAS UDDI Registry and four public UDDI registries.

Web Services Publishing

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

Using Oracle Enterprise Manager

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

- Navigate to the **Application Server: <Instance-name>: OC4J home: Deployed Applications: Deploy Application Wizard**. The **Deploy Application** wizard takes you through the process of deploying a J2EE application on the OC4J container. In order to publish a J2EE Web Service, you must first assemble it as a J2EE Enterprise Archive (EAR) file. See the chapter on using the Web Services assembly tool for more information. See *Oracle Application Server Containers for J2EE User's Guide* for information about EAR file-based deployment of J2EE Web applications.

The second-to-last step, the **Publish Web Services** step, of the **Deploy Application** wizard lets Web Services provider administrators publish (OC4J) Web Services (servlets) that are found in the EAR file. Any Web Services servlet in an application that you want to access must be published to the OracleAS UDDI Registry to one or more desired categories within one or more of the classifications provided. Any unpublished Web Services servlet in an application appears with the status of `Not Published` and when the Web Services servlet is published, the status changes to `Published`.

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

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

Publishing Web Services Using Deploy Applications Wizard

Web Services provider administrators can publish J2EE Web Services, which are produced by the OracleAS Web Services assembly tool, using the Oracle Enterprise Manager Deploy Applications wizard. They can do this as follows:

1. Invoke Oracle Enterprise Manager and navigate to the **Application Server: <Instance-name>** 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. Perform the steps in each window of the **Deploy Application** wizard and provide the essential information for each step.
3. At the **Publish Web Services** window, select the desired Web Services to register from the list of Web Services known to the application whose status is `Not Published`. Do this by clicking its corresponding radio button in the

Select column. Then 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 **Services Details** section and in the tModel Details section.
   a. To add categories for either the **Services Details** or the **tModel Details** sections, click **Browse UDDI Registry**, browse to the desired classification, and drill down as needed through each desired category, noting all desired category names and values.
   b. Click **Add Category** to add an empty row of category information.
   c. Select the desired classification, then enter the value code and its corresponding category name for the desired category.
   d. Repeat this process (Steps b and c) as many times as it takes to add all the categories to which to register this Web Services.
   e. After entering all the required information on the **Web Services Details** window, publish the Web Services to the OracleAS UDDI Registry by clicking **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 publishing all Web Services for this application, click **Next** to continue to the **Summary** window where all the application deployment information can be reviewed.
7. If there are no further changes, click **Deploy** to deploy the J2EE application on the OC4J container. Doing this returns you to the Oracle Enterprise Manager OC4J Home page. Then, to repeat the process of deploying another J2EE application on the OC4J container, click **Deploy Application**.

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

Publishing Web Services Using Web Services Details Window

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

1. Invoke Oracle Enterprise Manager and navigate to the **Application Server: <Instance-name>** window, and then, to the **OC4J: home** window. Locate the

UDDI Registry link in the **Related Links** column within the **Administration** section of the **OC4J: home** window.

Click the **UDDI Registry** link.

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

   Click the **Administration** link.

3. At the **Web Services Details** window, enter the required information in each of the fields in the **Services Details** section and in the **tModel Details** section.

   a. Enter the service name, service description, and service URL to the servlet in the **Services 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 **Services Details** or the **tModel Details** sections, click **Browse UDDI Registry**, browse to the desired classification, and drill down as needed through each desired category, noting all desired category names and values.

   d. Click **Add Category** to add an empty row of category information.

   e. Select the desired classification, then enter the value code and its corresponding category name for the desired category.

   f. Repeat this process (Steps d and e) as many times as needed to add all the categories to which to register this Web Services.

   g. After entering all required information on the **Web Services Details** window, publish the Web Services to the OracleAS UDDI Registry by clicking **Apply**. This returns you to the **UDDI Registry** window where you can choose to publish another J2EE application to the OracleAS UDDI Registry by following the same steps again, beginning at Step 2.

Updating Published Web Services in the OracleAS UDDI Registry

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

To update published Web Services using Oracle Enterprise Manager to discover Web Services, do the following:

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

   Click the **UDDI Registry** link.

2. The **UDDI Registry** window lets the administrator select one of the three standard classification: NAICS, UNSPSC, or ISO-3166 by clicking its link. The **UDDI Registry** window lets the administrator browse any of the three classifications and discover published Web Services associated with any category or subcategory.

   Click the desired classification link.

3. The **UDDI Registry: <*Classification Name*>** window lets the administrator drill down from category to subcategory to discover published Web Services associated with any category or subcategory. Each classification is organized in a hierarchical tree. Navigate down a particular branch by clicking the category name to determine all its subcategory names, and so forth. As you navigate down a branch, also note the change in the category code value.

   Navigate to the desired category or subcategory by successively clicking the desired category links.

4. The **Web Services: <*Category Name*>** window lets the administrator continue to drill down through the categories, or you can view all Web Services published in a particular category by selecting the corresponding radio button in the Select column for that category, and clicking **View Services**.

   Select the corresponding radio button in the Select column for the desired category and click **View Services**.

5. The **Web Services** window lists all Web Services published for that category name. For Web Services listed for the selected category, the corresponding service name, service key, and business key are also listed. If the selected category or subcategory has no published Web services, none is listed.

   To view the complete details of a particular published Web Services listed for a category, either click its service name link or select its corresponding radio button in the **Select** column and click **View Details**.

Click the desired service name link.

6. The **Web Services Details** window displays detailed information for the selected Web Services published in the OracleAS UDDI Registry. This information includes:

   - Service Details

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

     Category

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

   - tModel Details

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

     Category

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

Category information can be added or deleted for both the **Service Details** and **tModel Details** sections. You can browse the OracleAS UDDI Registry (click **Browse UDDI Registry**) looking for categories in which to register Web Services. You can add categories (click **Add Category**) to which both the Web Services and tModel are to be registered. You can remove categories (click **Delete**) to which the Web Services and tModel are registered.

Service and tModel detail information can be modified by moving the cursor to the appropriate field and making the necessary changes.

After making all selections or completing all changes for this Web Services, click **Apply** to save your changes.

If you have made changes to any field and you decide you want to return to the original set of values for all selections, click **Revert**. The window refreshes with the original set of values for all selections as if you had just begun your current session.

Make your modifications and click **Apply** to save your changes.

To discover and update other published Web Services for the same category, at the top of the **Web Services Details** window, select the desired **Web Services:<*Classification Name*>** link to return to the desired **Web**

Services:<*Classification Name*> window. At this window, select another Web Services to view in more detail, make any necessary changes, and finally click **Apply** to save your changes.

Alternatively, you can select the **UDDI Registry** link at the top of the **Web Services Details** window to return to the **UDDI Registry** window where you can navigate to another classification to discover Web Services for other categories. At each desired category, select the desired Web Services to view its details, make any necessary changes, and finally click **Apply** to save your changes.

Using the Publishing API

The UDDI publishing API lets programmers, following authentication, publish Web Services by providing save and delete calls for each of the five key UDDI data structures (businessEntity, businessService, bindingTemplate, tModel, and publisherAssertion).

The publishing API allows programmers to publish Web Services using the Java language. Programs can be written in any language and use SOAP to publish Web Services. The Java API is provided as a convenience for Java programmers.

The publishing API is located in the Oracle Application Server installation directory, `<ORACLE_ HOME>/uddi/` for UNIX and `<ORACLE_HOME>\uddi\` for Windows. The API documentation that describes how to use this publishing API can be found on the Oracle Application Server Documentation Library CD-ROM as UDDI Client API Reference (Javadoc) under OracleAS Web Services, which is located under the J2EE and Internet Applications tab.

A set of sample demonstration (`uddidemo.zip`) files are located on the Oracle technology Network (OTN) Web site http://otn.oracle.com/tech/java/oc4j/demos.

The UddiPublishingExample.java Example

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

The program example does the following:

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

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

- Sets the proxy information for the transport if the system properties http.proxyHost and http.proxyPort are set. These properties can be set on the command line. If these properties are not set, this command has no effect.

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

- Uses SoapTransportLiaison and the URL of a UDDI publishing registry to initialize an instance of UddiClient, which connects to the specified OracleAS UDDI Registry. The UddiClient instance is the primary interface by which clients send requests to the OracleAS UDDI Registry Authentication is done using the UDDI get_authToken message in this example.

  ```
  SimpleAuthenticationLiaison auth =
    new SimpleAuthenticationLiaison(szUserName, szPassword);
  UddiClient uddiClient = new UddiClient(null, szPublishingUrl, transport,
  auth);
  ```

  Note: The UddiClient instance, by default, operates as a UDDI v2.0 client (the latest release supported). If a specific release is needed, the release can be specified, either through another constructor, or by the JVM property oracle.uddi.client.defaultVersion.

- Performs authentication. You should make this call before doing any publishing.

  ```
  UddiClient.authenticate();
  ```

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

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

- Performs various publishing operations that include creating and saving a tModel, a businessEntity, a businessService, and a bindingTemplate data structure for the purpose of creating a business that provides a Google-interface-compatible service.

- Creates a tModel data structure that represents a Google-compatible service by using the UddiElementFactory instance.

  ```
  TModel tModel = (TModel)uddiEltFactory.createTModel();
  tModel.setName("urn:google.com:search-interface");
  ```

  – Creates and includes the OverviewDoc data structure in the tModel data structure by using the UddiElementFactory instance.

  ```
  OverviewDoc overviewDocTm =
   (OverviewDoc)uddiEltFactory.createOverviewDoc();
       tModel.setOverviewDoc(overviewDocTm);
  overviewDocTm.setOverviewURL("http://api.google.com/GoogleSearch.wsdl");
  ```

  – In the tModel data structure, uses the UddiElementFactory instance to create a CategoryBag data structure and its KeyedReference, which will be used for searching. Classify the tModel data structure as a SOAP/WSDL-based interface and put it under the "applicable service providers" category.

  ```
  CategoryBag catBagTm =
   (CategoryBag)uddiEltFactory.createCategoryBag();
  tModel.setCategoryBag(catBagTm);

  KeyedReference krTm1 =
  (KeyedReference)uddiEltFactory.createKeyedReference();

  catBagTm.addUddiElement(krTm1);
  krTm1.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UDDI_TYPE);
  krTm1.setKeyName("wsdlSpec");
  krTm1.setKeyValue("wsdlSpec");

  KeyedReference krTm2 =
   (KeyedReference)uddiEltFactory.createKeyedReference();
  catBagTm.addUddiElement(krTm2);
  krTm2.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UDDI_TYPE);
  krTm2.setKeyName("wsdlSpec");
  krTm2.setKeyValue("wsdlSpec");

  KeyedReference krTm3 =
   (KeyedReference)uddiEltFactory.createKeyedReference();
  catBagTm.addUddiElement(krTm3);
  krTm3.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UNSPSC_7_3);
  krTm3.setKeyName("application service providers");
  krTm3.setKeyValue("81.11.21.06.00");
  ```

- Publishes the Google search interface tModel business operation.

  ```
  System.out.println("\nPublish the google search interface tModel.");
  TModel tMSaved = uddiClient.saveTModel(tModel);
  String szGoogleTModelKey = tMSaved.getTModelKey();
  System.out.println("The tModel is saved with tModelKey assigned to be " +
                        szGoogleTModelKey);
  ```

- Creates a businessEntity data structure that represents a Google-compatible service by using the UddiElementFactory instance.

  ```
  BusinessEntity businessEntity =
    (BusinessEntity)uddiEltFactory.createBusinessEntity();
  businessEntity.setName("ACME search Inc.", "en");
  ```

  In the businessEntity data structure, uses the UddiElementFactory instance to create a CategoryBag data structure and its KeyedReference data structure, which will be used for searching. Classify the businessEntity data structure under the "information services and data processing services" category.

  ```
  KeyedReference krBe1 =
   (KeyedReference)uddiEltFactory.createKeyedReference();
  catBagBe.addUddiElement(krBe1);
  krBe1.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_NAICS_1997);
  krBe1.setKeyName("Information Services and Data Processing Services");
  krBe1.setKeyValue("514");
  ```

- Creates a businessService data structure that represents a Google-compatible service by using the UddiElementFactory instance.

  ```
  BusinessServices businessServices =
  (BusinessServices)uddiEltFactory.createBusinessServices();
  businessEntity.setBusinessServices(businessServices);
  BusinessService businessService =
  (BusinessService)uddiEltFactory.createBusinessService();
  businessServices.addUddiElement(businessService);
  businessService.setName("ACME Web Search service", "en");
  ```

  In the businessService data structure, uses the UddiElementFactory instance to create a CategoryBag data structure and its KeyedReference data structure, which will be used for searching. Classify the businessService data structure under the "application service providers" category.

  ```
  CategoryBag catBagBs =
   (CategoryBag)uddiEltFactory.createCategoryBag();
  ```

```
businessService.setCategoryBag(catBagBs);
KeyedReference krBs1 =
 (KeyedReference)uddiEltFactory.createKeyedReference();
catBagBs.addUddiElement(krBs1);
krBs1.setTModelKey(CoreTModelConstants.TAXONOMY_KEY_UNSPSC_7_3);
krBs1.setKeyName("application service
providers");krBs1.setKeyValue("81.11.21.06.00");
```

- Creates the bindingTemplates data structure that represents a Google-compatible service by using the UddiElementFactory instance.

  ```
  BindingTemplates bindingTemplates =
   (BindingTemplates)uddiEltFactory.createBindingTemplates();
  businessService.setBindingTemplates(bindingTemplates);
  BindingTemplate bindingTemplate =
   (BindingTemplate)uddiEltFactory.createBindingTemplate();
  bindingTemplates.addUddiElement(bindingTemplate);
  ```

  – Creates and includes the access point in the bindingTemplates data structure by using the UddiElementFactory instance.

  ```
  AccessPoint accessPoint =
  (AccessPoint)uddiEltFactory.createAccessPoint();
  bindingTemplate.setAccessPoint(accessPoint);
  accessPoint.setUrlType("http");
  accessPoint.setContent("http://foobar.net/search-g");
  ```

  – Creates and includes the tModel instance details in the bindingTemplates data structure by using the UddiElementFactory instance.

  ```
  TModelInstanceDetails tModelInstanceDetails =
  (TModelInstanceDetails)uddiEltFactory.createTModelInstanceDetails();
  bindingTemplate.setTModelInstanceDetails(tModelInstanceDetails);
  ```

  – Declares that the bindingTemplate data structure implements the Google search interface.

  ```
  TModelInstanceInfo tModelInstanceInfo =
   (TModelInstanceInfo)uddiEltFactory.createTModelInstanceInfo();
  tModelInstanceDetails.addUddiElement(tModelInstanceInfo);
  tModelInstanceInfo.setTModelKey(szGoogleTModelKey);
  ```

- Publishes the businessEntity data structure and its containing businessService and bindingTemplate data structures.

  ```
  System.out.println("Publish the ACME Search Inc. businessEntity...");
  ```

```
BusinessEntity bESaved = uddiClient.saveBusiness(businessEntity);
System.out.println("The saved businessEntity (in XML) is:");

bESaved.setName("The ACME search Inc.", "en");
BusinessEntity bEUpdated = uddiClient.saveBusiness(bESaved);
```

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

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

- Closes the UddiClient instance when finished to release resources and to log out from the registry.

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

The UddiPublisherAssertionExample.java Example

Within the `uddidemo.zip` file is a Java program file, `UddiPublisherAssertionExample.java`. This file provides Java programmers with a starting point that demonstrates the key constructs and the sequence in using the Oracle UDDI client library for publisher assertion-related operations.

A publisher assertion, which is a UDDI v2.0 feature, is an assertion made by a publisher who is expressing a particular fact about a business registration and its relationships to other business data within the OracleAS UDDI Registry. Publisher assertions are used to establish visible relationships between registered data. Once completed, a set of assertions can be seen by the general inquiry message named findRelatedBusinesses.

The program example does the following:

- Initializes instances of two UddiClients.

```
UddiClient uddiClient1 =
  createUddiClient(szInquiryUrl, szPublishingUrl, szUserName1, szPassword1);
UddiClient uddiClient2 =
  createUddiClient(szInquiryUrl, szPublishingUrl, szUserName2, szPassword2);
DispositionReport dispositionReport = null;
```

- Creates the business entities to be used.

```
String bEKey1 =
  createBusinessEntity(uddiClient1,
                       "bE1 - UddiPublisherAssertionExample");
String bEKey2 =
  createBusinessEntity(uddiClient2,
                       "bE2 - UddiPublisherAssertionExample");
```

- Creates for uddiClient1 a publisher assertion that represents a peer-to-peer relationship from bE1 to bE2.

```
System.out.println("");
System.out.println("uddiClient1 attempts to create a peer-to-peer
relationship ");
System.out.println("from bE1 to bE2...");
dispositionReport =
  uddiClient1.addPublisherAssertion
  (createPeerToPeerPublisherAssertion(uddiClient1, bEKey1, bEKey2));
System.out.println("Done.");
```

- Makes a query for uddiClient1 for relationships yet to be established; that is, looking for those relationships that the toKey side has not yet acknowledged.

```
AssertionStatusReport assertionStatusReport1 =
  uddiClient1.getAssertionStatusReport
  (AssertionStatusItem.COMPLETION_STATUS_TOKEY_INCOMPLETE);
printOutXml("pending relationships for uddiClient1: case toKey incomplete",
   assertionStatusReport1);
```

- Makes a query for uddiClient2 for relationships yet to be established; that is, looking for those relationships that the toKey side has not yet acknowledged.

```
AssertionStatusReport assertionStatusReport2 =
  uddiClient2.getAssertionStatusReport
  (AssertionStatusItem.COMPLETION_STATUS_TOKEY_INCOMPLETE);
printOutXml("pending relationships for uddiClient2: case toKey incomplete",
   assertionStatusReport2);
```

- Shows uddiClient2 agreeing to the peer-to-peer relationship requested by creating a publisher assertion.

```
System.out.println("");
System.out.println("uddiClient2 agrees to the peer-to-peer relationship ");
System.out.println("from bE1 to bE2");
dispositionReport =
  uddiClient2.addPublisherAssertion
  (createPeerToPeerPublisherAssertion(uddiClient2, bEKey1, bEKey2));
```

```
System.out.println("Done.");
```

- Makes another query for uddiClient2 for relationships yet to be established to see if there are other peer-to-peer relationships to be established. There are no more pending relationships to be established.

```
AssertionStatusReport assertionStatusReport2After =
  uddiClient2.getAssertionStatusReport
  (AssertionStatusItem.COMPLETION_STATUS_TOKEY_INCOMPLETE);
printOutXml("pending relationships for client2: toKey incomplete (should be
  none)", assertionStatusReport2After);
```

- Finds related businesses that have established peer-to-peer relationships (that have published assertions) by calling the general inquiry message findRelatedBusinesses.

```
RelatedBusinessesList rbList =
  uddiClient1.findRelatedBusinesses
  (bEKey1,
   createPeerToPeerKeyedReference(uddiClient1),
   null);
printOutXml("find all businesses that are peers to " + bEKey1, rbList);
```

- Deletes a publisher assertion relationship between bE1 and bE2, owned by uddiClient1.

```
System.out.println("");
System.out.println("Delete a publisherAssertion...");
dispositionReport =  uddiClient1.deletePublisherAssertion
  (createIdentityPublisherAssertion(uddiClient1, bEKey1, bEKey2));
System.out.println("Done");
```

- Shows another way of deleting all publisher assertion relationships owned by uddiClient1 by using the setPublisherAssertions call.

```
System.out.println("");
System.out.println("Delete all publisherAssertions of uddiClient1 ");
System.out.println("by using setPublisherAssertions...");
publisherAssertions =
  uddiClient1.setPublisherAssertions(null);
printOutXml("Done. The current list:", publisherAssertions);
```

OracleAS UDDI Registry Administration

The following sections describe new OracleAS UDDI Registry administration features.

Using the Command-Line Tool uddiadmin.jar

Many administrative operations are done using the command-line tool `uddiadmin.jar` described in the sections that follow.

The command-line tool `uddiadmin.jar` is located in the `uddi/lib/uddiadmin.jar` file for UNIX and in the `uddi\lib\uddiadmin.jar` file for Windows. Administrators can use this tool for various administrative activities. In general, the command-line tool takes the command-line parameters of the following form:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] <action to perform and additional parameters>

where the <username> belongs to the uddiadmin group
```

The default user name is `ias_admin` and the default password is `ias_admin123`.

Note that the `-verbose` option will cause stack trace information to be printed out when an exception is encountered.

Server Configuration

The following parameters are used for server configuration operations. See Server Configuration Properties Reference Information on page 10-48 for more information about these configuration parameters.

getProperties

Parameters: `<registry admin URL> <username> <password> [-verbose] -getProperties`

Description: Lists the current registry configuration parameters.

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>  [-verbose]
-getProperties
```

setProperty

Parameters: `<registry admin URL> <username> <password> [-verbose] -setProperty <name>=<value>`

Description: Changes the value of the named configuration parameter. The OracleAS UDDI Registry J2EE application needs to be restarted for the parameters to take effect.

Warning: Be very careful when using the -setProperty option to change server configuration property values. Making an incorrect property setting could cause severe damage to the integrity of the registry.

User Management

OracleAS UDDI Registry for 10*g* (9.0.4) uses the Oracle Internet Directory (OID) of the Oracle Application Server infrastructure as the default user repository. This is achieved through the use of LDAP-based provider of OC4J Java Authentication and Authorization Service (JAAS).

UDDI-specific OID groups are located under the `cn=uddi_groups` subtree of the group subtree of the OID default subscriber.

In other words, users are located under the user subtree of the OID default subscriber.

The types of UDDI users are summarized in Table 10–1.

Table 10–1 Default UDDI Groups

Group	Description
uddipublisher	Can access the publishing end point and save, update, or delete UDDI entities in the registry.
uddipublisher	Can create UDDI subscriptions.
uddiadmin	Can access the administration end points and perform administrative activities. Can perform all activities specified in uddipublisher group.
uddireplicator	Can perform replication activities based on the replication schedule: send replication requests such as get_changeRecords to other UDDI nodes and apply the changeRecords received.

Note: Do not remove any of these default UDDI Groups.

In addition to these groups, there are also a set of default groups for user quota purposes. Those groups can be added, updated, or removed based on the specific user quota policy administrators need to enforce.

By default, the following users are created in an installation. Administrators can add or remove users to or from these corresponding groups as shown in Table 10–2.

Table 10–2 Default UDDI Users

Group	User Names	Comments
uddiadmin	ias_admin	Typically, Enterprise Manager administrators also login as ias_admin to publish to the UDDI registry through the Enterprise Manager integrated J2EE Web Services deployment and publishing wizard.
uddipublisher	uddi_publisher, uddi_publisher1	These are sample users for demonstrating publishing and different default quota groups.
uddireplicator	uddi_replicator	The default user used for performing the UDDI replication activities in the background. This user should not be removed. If you do need to remove this user, make sure you add another user to the uddireplicator group. The user to start the Replication Client module must be updated as well by modifying the `orion-application.xml` file in the `oraudrepl.ear` archive file.

Generic user management, such as creation, deletion, suspension, and so forth, is handled by Oracle Internet Directory and its Delegated Administration Service. Refer to *Oracle Internet Directory Administrator’s Guide* for more information.

User management, including operations such as creation, deletion, suspension, role management, and so forth, is handled by OC4J Java Authentication and Authorization (JAAS) service. Refer to *Oracle Application Server Containers for J2EE Services Guide* for more information.

In general, user management is handled by the OC4J JAAS service and OID. However, to find out the authorized name of a user, use the `-getUsers` option of the `uddiadmin.jar` command-line tool described as follows:

getUsers

Parameters: `<registry admin URL> <username> <password> [-verbose] -getUsers`

Description: Lists all existing users who have entities in the registry.

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>  [-verbose]
-getUsers
```

getUserDetail

Parameters: `<registry admin URL> <username> <password> [-verbose] -getUserDetail <username_to_retrieve>`

Description: Retrieves the details of the named user, currently the authorizedName of each user.

Quota Enforcement

OracleAS UDDI Registry provides a mechanism to enforce the number of entities a publisher can own. A publisher can own at most a specific number of tModels, publisherAssertions, businessEntities, businessServices per businessEnitity, and bindingTemplates per businessService depending upon the quota group associated with the publisher, which is guided by the user group to which the publisher is assigned.

OracleAS UDDI Registry uses a group-based mechanism for assigning quota limits to a publisher. When a new publisher is added, the OracleAS UDDI Registry administrator must associate the publisher with a quota group. Table 10–3 shows the predefined quota groups and quota limits for each entity that a publisher can own.

Table 10–3 Predefined Quota Groups

Quota Group	Quota Limits per Entity				
	business Entities	businessServices per businessEntity	bindingTemplates per businessService	tModels	publisherAssertions
Default	1	4	2	100	10
uddi_ unlimited_ quota_group	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited
uddi_lowlimits_ quota_group	2	2	1	3	3
<Implicit>UDDI _Administrators	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited

The explicit `Default` quota group cannot be deleted. Users who are UDDI administrators always get unlimited quota.

The OracleAS UDDI Registry administrator can also update a quota group, add a new quota group, delete a quota group, view the lists of quota groups and their quota limits, and associate a publisher with a quota group. The following sections describe each of these administrator tasks.

Associating a Publisher with a Quota Group

When a user is added to the user store (OID or `jazn-data.xml`), the user should be placed in a group so that it gets the appropriate quota group. For example, with the pre-defined settings, administrators can assign a user to have the low quota limits by assigning the user to the `uddi_lowlimits_quota_group` group.

If a user does not belong to a particular group, the user gets the quota limits from the `Default` group. A UDDI administrator always has unlimited quota.

Viewing the Lists of Quota Groups and Their Limits

Use the `-getRoleQuotaLimits` option of the command-line tool `uddiadmin.jar`, described as follows:

getRoleQuotaLimits

Parameter: `getRoleQuotaLimits`

Description: Displays all the J2EE-role-to-quota-limits mappings that are currently set in the registry.

Parameter type/allowable values: long

Initial value: 0

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -getRoleQuotaLimits
```

Updating the Limits of a Quota Group

Use the `-setRoleQuotaLimits` option of the command-line tool `uddiadmin.jar`, described as follows:

setRoleQuotaLimits

Parameter: `setRoleQuotaLimits`

Description: Sets the quota limit value for the specified quota group. This option can be used to create a new group-to-quota-limit mapping or to update an existing mapping. The parameters are defined as follows:

- roleName -- name of the quota group to map to the specified limits
- maxBE -- maximum number of businessEntity data structures allowed
- maxBSperBE -- maximum number of businessService data structures per businessEntity allowed
- maxBTperBS -- maximum number of bindingTemplate data structures per businessEntity allowed
- maxTM -- maximum number of tModel data structures allowed
- maxPA -- maximum number of publisherAssertion data structures allowed

The value -1 means unlimited.

Parameter type/allowable values: N / A

Initial value: N / A

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setRoleQuotaLimits <roleName> <maxBE> <maxBSperBE> <maxBTperBS>
<maxTM> <maxPA>
```

Adding a New Quota Group (Advanced Operation)

To add a new quota group, perform the following steps:

1. Add the group to the user store, typically OID.
2. Define the corresponding J2EE security role partnerGroup for the new group name you want to create in the `orauddi` application. The settings must be added in both the `application.xml` file of the `orauddi.ear` file and the `web.xml` file of the `orauddi.ear` file.
3. Define the J2EE security role to the user store mapping in the `orion-application.xml` file of the `orauddi.ear` file.
4. Define the actual limits of the quota group using the `-setRoleQuotaLimits` option of the command-line tool `uddiaddmin.jar`. See the `-setRoleQuotaLimits` option in Updating the Limits of a Quota Group on page 10-28 for more information.

Deleting a Quota Group (Advanced Operation)

To remove a quota group, perform the following steps:

1. Remove the J2EE security role for the partnerGroup you want to remove from the `orauddi` application. The settings must be removed from both the `application.xml` file of the `orauddi.ear` file and the `web.xml` file of the `orauddi.ear` file.
2. Remove the J2EE security role to the user store mapping in the `orion-application.xml` file of the `orauddi.ear` file.
3. Remove the actual limits of the quota group using the `-deleteRoleQuotaLimits` option of the command-line tool `uddiadmin.jar`. See the `-deleteRoleQuotaLimits` option described after Step 4 for more information.
4. Remove the group from the user store, typically OID.

deleteRoleQuotaLimits

Parameter: `deleteRoleQuotaLimits`

Description: Deletes the group-to-quota-limits mappings for the specified quota groups.

Parameter type/allowable values: N / A

Initial value: N / A

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -deleteRoleQuotaLimits <roleName> [<roleName>...]
```

Administrative Entity Management

The following parameters are used for administrative entity management:

deleteEntity

Parameters: `<registry admin URL> <username> <password> [-verbose] -deleteEntity [-businessKey <businessKey> | -serviceKey <serviceKey> | -bindingKey <bindingKey> | -tModelKey <tModelKey>]`

Description: Deletes the named entity irrespective of the owner of the entity. Note that this operation performs a nonpermanent delete (hide) operation in the case of a tModel entity.

destroyTModel

Parameters: `<registry admin URL> <username> <password> [-verbose] -destroyTModel <tModelKey>`

Description: Permanently deletes the named tModel from the registry (as opposed to the UDDI-defined delete_tModel call, which is just hiding the tModel entity).

Parameters: `<registry admin URL> <username> <password> [-verbose] -changeOwner <new username> [-businessKey <businessKey> | -tModelKey <tModelKey>]`

Description: Changes the ownership of the named entity to the new specified user.

Import Operation

The following parameter is used for importing entities:

import

Parameters: `<registry admin URL> <username> <password> [-verbose] [-s|-m] -import [-businesses <filename> | -tmodels <filename> | -publisherAssertion <filename> -fromBusinessCheck [true|false] -toBusinessCheck [true|false]]`

Description: Imports all businessEntity and tModel data structures, and a publisherAssertion in the named file. For importing the businessEntity data structure, the named file (`<filename>`) for importing should contain a UDDI businessDetail XML document. For importing tModel data structures, the named file should contain a UDDI tModelDetail XML document. By importing them, entityKeys (such as, businessKey, serviceKey, bindingKey, tModelKey) are preserved. The operatorName and authorizedName fields, however, are not preserved. The operatorName field will be replaced by the operatorName configuration parameter of the registry. The owner of the imported entities is the administrator; hence, the authorizedName field will be the authorizedName of the administrator. Importing can be done in single mode (-s), which does not allow partial success (some entities are imported and some are not due to some error condition), or in multiple mode (-m), which does allow partial success.

The import parameter is particularly useful in importing the well-known service interface specification tModel and classification tModel data structures from some authoritative sources.

Because the entity keys are preserved, administrators should be careful in evaluating the source of the entities to ensure there will not be a collision in entity keys.

For importing a publisher assertion, two Boolean values are required. These Boolean values are used to indicate from which side (or both sides when two Boolean values are true) the publisher assertion is going to be inserted.

Set Operational Information

The `-setOperationalInfo` parameter is used for setting some operational information of entities, such as the modified timestamp. Note there are two options.

setOperationalInfo

Parameters: `Option 1: <registry admin URL> <username> <password> [-verbose] - setOperationalInfo [[-businessKey key | -tModelKey key] [-newOperator OperatorName] [-newAuthorizedname authName] [-newTime timestamp]]`

`Option 2: <registry admin URL> <username> <password> [-verbose] - setOperationalInfo [[-serviceKey key | -bindingKey key] -newTime timestamp]`

Description: Sets some operational information, such as the operator name, authorized name, or timestamp of a businessEntity or tModel specified by a key, for example, following an import operation. Any combination of these three options is allowed to be set using the `-setOperationalInfo` option.

The syntax option `[[-businessKey key | -tModelKey key] [-newOperator OperatorName] [-newAuthorizedname authName] [-newTime timestamp]]` lets you change either the operator name, the authorized name, or the timestamp, or all three options of a business entity or tModel specified by a key.

The syntax option `[[-serviceKey key | -bindingKey key] -newTime timestamp]` lets you change only the timestamp of a business service or binding template.

Note: The format of a timestamp is defined as ’yyyy-mm-dd hh.mm:ss.fffffffff’ by java.sql.Timestamp. For example,

```
’2002-12-01 00:00:00’
```

Because there is a blank space in the timestamp value between ’yyyy-mm-dd and hh.mm:ss.fffffffff, the entire value must be placed inside a pair of quotation marks on the command line.

Warning: This feature should not be invoked when replication is set to on. In general, the `-setOperationalInfo` option should not be used when replication is enabled.

UDDI Replication

The OracleAS UDDI Registry allows administrators to create a logical registry that comprises one or more Oracle UDDI implementations and UDDI implementations from other vendors that also implement the UDDI v2.0 Replication Specification. See the UDDI v2.0 Replication Specification for more information.

This section briefly describes the data replication process and the program interface required to achieve complete data replication among UDDI operator nodes that form a UDDI service. UDDI replication ensures that all operator nodes see all the changes that have originated at individual operator nodes. In addition, any inquiries made at any operator node within the UDDI service yield results consistent to those made at any other operator node within the UDDI service, hence the logical OracleAS UDDI Registry.

For detailed technical descriptions of concepts and definitions involved with UDDI replication, including replication processing, how to bring new UDDI operators online, checking and validation of replicated data, see the UDDI v2.0 Replication Specification. The sections that follow describe the Oracle implementation of UDDI replication.

Enabling UDDI Replication

To enable UDDI replication, an administrator must perform the following steps:

1. Participate with and agree to the replication topology with UDDI administrators of other operator nodes. This involves editing the replication configuration (in the format specified in the UDDI v2.0 replication specification) accordingly, and using the `-downloadReplicationConfiguration` and `-uploadReplicationConfiguration` options of the command-line tool `uddiadmin.jar`.
2. Enable replication scheduling by setting the following server property, `oracle.uddi.server.scheduler.status`, to the value 1.
3. Enable update journal storage by setting the following property, oracle.uddi.server.replication.startMaintainingUpdateJournal, to true.

After UDDI replication is started, the UDDI administrator can suspend or resume replication operations by stopping or starting the `oraudrepl.ear` application.

If HTTPS client-certification is used, UDDI administrators must do the following:

1. Obtain an exported Oracle wallet file using Oracle Wallet Manager and specify the exported wallet location by setting the server property

`oracle.uddi.server.replication.walletLocation`. This option only needs to be set once.

2. Use the `-setWalletPassword` option to supply the wallet password, whenever the `oraudrepl.ear` application is started or restarted. The password is not persistent for security reasons.

See Replication Configuration Management on page 10-36 for a description of useful parameter options that are provided to assist OracleAS UDDI Registry administrators in the day-to-day operations during replication, using the command-line tool `uddiadmin.jar`.

In some cases, the administrator of the source of the error must correct an invalid changeRecord operation that caused the error. The administrator can use the `-correctChangeRecord` option of the command-line tool `uddiadmin.jar` to supply the correct changeRecord data. See Replication Exception Handling on page 10-38 for more information.

Replication Configuration Management

The following parameters are used in replication configuration management:

uploadReplicationConfiguration

Parameters: `<registry admin URL> <username> <password> [-verbose] -uploadReplicationConfiguration <xml_file_ containing_replication_configuration>`

Description: Uploads the specified replication configuration to a particular UDDI node within an OracleAS UDDI Registry. The application must be restarted for the new replication configuration to be used.

downloadReplicationConfiguration

Parameters: `<registry admin URL> <username> <password> [-verbose] -downloadReplicationConfiguration`

Description: Downloads the currently used replication configuration from a specified UDDI node within the OracleAS UDDI Registry.

Miscellaneous Operations

The following parameters are used in miscellaneous operations:

doPing

Parameters: `<registry admin URL> <username> <password> [-verbose] -doPing replicationEndPointSoapUrl [-password walletPassword]`

Description: Sends a UDDI replication do_ping message to the replication end-point URL specified. This is similar to the ping command in TCP/IP that is used to check if the other end point is alive. The optional walletPassword is useful when the JVM, which receives the do_ping message, does not have a valid wallet password set.

replicationEndPointSoapUrl

Parameters: `<registry admin URL> <username> <password> [-verbose] -replicationEndPointSoapUrl [-password walletPassword]`

Description: Gets the high-water marks vector from the specified UDDI node. The optional walletPassword is useful when the JVM, which receives the do_ping message, does not have a valid wallet password set.

getChangeRecord

Parameters: `<registry admin URL> <username> <password> [-verbose] -getChangeRecord local_usn`

Description: Gets the detail of a change record specified by local_usn (an integer). This API is used in conjunction with the `-CorrectChangeRecord` option to correct wrong or inconsistent data across different UDDI nodes with the OracleAS UDDI Registry.

HTTPS Setup

The following parameter is used in HTTPS setup operations:

setWalletPassword

Parameters: `<registry replication wallet admin URL> <username> <password> [-verbose] -setWalletPassword walletPassword`

Description: Sets the wallet password to be used for HTTPS communication among UDDI nodes for UDDI replication. Each time the application is restarted, this option must be invoked because the wallet password is not stored persistently, for security reasons. The registry replication wallet admin URL is `http://`*`<OracleAS-host>`*`:`*`<OracleAS-port>`*`/uddirepl/admin/wallet`.

Custody Transfer

The following parameter is used in replication custody transfer operations:

transferCustody

Parameters: `<registry admin URL> <username> <password> [-verbose] -transferCustody oldOperatorName newOperatorName newAuthorizedName [-tModelKey tModelKey | -businessKey businessKey]`

Description: Transfers the custody of a tModel or a business entity to a new operator and a new authorized name. This option is part of custody transfer as defined by the UDDI specification.

Replication Exception Handling

If any errors occur during replication operations, the OracleAS UDDI Registry logs the error in the `application.log` file of the `oraudrepl.ear` file. The administrator should investigate the cause of the error and correct each problem accordingly.

The following parameter is used in replication exception handling:

correctChangeRecords

Parameters: `<registry admin URL> <username> <password> [-verbose] -correctChangeRecord <changeRecordCorrectionfile> <changeRecordNewDatafile>`

Description: Applies the changeRecordCorrectionfile file contents and changeRecordNewDatafile file contents to the UDDI node. The content of these files must conform to the UDDI replication XML schema. This option is part of UDDI replication error recovery.

Advanced Configuration and Tuning

See UDDI Replication Properties on page 10-51 for a description of a set of server properties provided for advanced tuning and configuration of the replication operations.

Registry-Based Category Validation

OracleAS UDDI Registry for 10*g* (9.0.4) can perform a spell-check form of category value validation. An administrator can add or remove the set of categories that will be validated by the registry. Refer to the v2.0 UDDI specification for more information.

Adding a New Category for Registry-Based Validation

To add a new category, you must load the category values into the database and register the category with the registry. Perform the following steps:

1. Publish the category to the registry by saving a new tModel data structure. For example, look at the tModel data structure named `ntis-gov:naics:1997`. You can use the included sample Web applications link `http://<OracleAS-host>:<OracleAS-port>/uddi/` or a third-party tool.

   If the tModel data structure has been defined in some other registry, you can also import it (instead of creating a new one, which results in different tModelKeys entities) using the `uddiadmin.jar` utility. See Import Operation on page 10-33 for more information on the import operation.

   The tModel data structure published should be classified as *"unvalidatable"* in uddi-org:types taxonomy. Specifically, the following keyedReference should appear in the categoryBag element of the tModel data structure:
   `<keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4" keyName="" keyValue="unvalidatable" />`

2. Load the category values into the database. To do this, all the category values should be in a file using the following format:

   - Each line of the file describes one category value in the category. It should be in the following format:

     ```
     <category value> | <description of category value>
      | <category value of the parent>
     ```

   - If a category value is a root value, for example, it has no parent, the category value of the parent should be set to itself.

   - The line in the file for a category value should occur before the lines for all of its descendants.

     Examples can be found in the `uddi/taxonomy` directory for UNIX and in the `uddi\taxonomy` directory for Windows. Excerpts from the NAICS file are as follows:

     ```
     22|Utilities|22
     ```

```
221|Utilities|22
2211|Electric Power Generation, Transmission|221
```

If your files use different characters from different languages, it is recommended that you save the file with UTF-8 encoding to avoid any problems that may arise, such as character corruption.

3. Create a SQL*Loader control file to load the category file. An example is `uddi/admin/naics-97.ctl` for UNIX and `uddi\admin\naics-97.ctl` for Windows. Copy the file and replace the category file name in the control file with the one you create. Refer to the v2.0 UDDI specification for more information about generating a unique ID for the new category tModel.
4. Load the category file to the database using SQL*Loader. Refer to *Oracle9i Database Utilities* for more information about using SQL*Loader.
5. Configure the registry so that it recognizes the category that must be validated by using the command-line administrative tool, `uddiadmin.jar`. For example, to add a new tModel entity with key `UUID:FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFF0`, issue the setProperty command for the property `oracle.uddi.server.categoryValidationTModelKeys` as follows:

   ```
   java -jar uddiadmin.jar <registry admin URL> <username>
    <password> -setProperty
   "oracle.uddi.server.categoryValidationTModelKeys=
   'UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4',
   'UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88',
   'UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2',
   'UUID:CD153257-086A-4237-B336-6BDCBDCC6634',
   'UUID:FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFF0' "
   ```

   Notice that because the setProperty command defines all categories that need to be validated, to add a new category, you must set the property with all the existing tModelKey values plus the new tModelKey value.
6. Allow the registry users to use the category tModel published by removing the *"unvalidatable"* categorization done in Step 1. Specifically, the following keyedReference element should be removed from the categoryBag element of the tModel data structure: `<keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4" keyName="" keyValue="unvalidatable" />`

Removing a Category from Registry-Based Validation

To remove a category from registry-based validation, you should unregister the category with the registry and remove the category values in the database. Perform the following steps:

1. To unregister the category with the registry, you should remove it from the list of validated categories using the `uddiadmin.jar setProperty` command for the property `oracle.uddi.server.categoryValidationTModelKeys`.

   You do not have to (and in general should not) delete the tModel data structure from the registry.

2. To remove the category values from the database, use the SQL*Plus script `wurvcrm.sql` in the `uddi/admin` directory for UNIX and in the `uddi\admin` directory for Windows. For example:

   ```
   sqlplus sys/<sys-password> @wurvcrm.sql
   ```

   When running this script, you will be prompted for the tModelKey value of the category to be removed. You should see that a set of rows has been deleted. If the result shows that 0 rows were deleted, you entered an invalid tModelKey value.

External Validation

Third parties can register new category and identifier schemes, and then control the validation process used by the OracleAS UDDI Registry to perform external validation or checking. This enables a third-party category provider to validate the UDDI entities to be saved when the entity is categorized, or identified with the category, by providing a validate_values SOAP Web service.

The operator that is calling the validate_values service will pass a businessEntity, a businessService, or a tModel element as the sole argument to this callout. This is the same data that is being passed within a save_business, save_service, or save_tModel API call. External validation is performed for any third-party category provider and identifier scheme that is classified as checked. A tModel element marked as checked asserts that it represents a categorization, identifier, or namespace tModel element that has a properly registered validation service.

If no error is found, the response is a dispositionReport message returning an errorCode value of E_success and an errno value of 0. If any error is found, or the called service needs to signal that the information being saved is not valid based on the validation algorithm chosen by the external service provider, then the service should raise a SOAP Fault and indicate either an errorCode value of E_invalidValue or E_valueNotAllowed. In either case, the error text indicates the keyedReference data that is being rejected, and the reason why.

Use the command-line tool `uddiadmin.jar` with the `-setProperty` option to:

- Enable external validation
- Add an externally validated category to the registry
- Remove an externally validated category from the registry

Enabling External Category Validation

To enable external category validation, issue the `-setProperty` option for the following property `oracle.uddi.server.externalValidation` as follows:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> -setProperty
oracle.uddi.server.externalValidation=true
```

Adding an Externally Validated Category to the Registry

To add an externally validated category to the registry, perform the following steps:

1. Publish the new category as a tModel data structure to the registry. This data structure must be categorized as checked under uddi-org:types category.
2. Register the external validation service of the category with the registry by updating the following server property: `oracle.uddi.server.externalValidationTModelList` using the `-setProperty` option as follows:

   ```
   java -jar uddiadmin.jar <registry admin URL> <username> <password>
   -setProperty
   oracle.uddi.server.externalValidationTModelList=<key-value>,<URL-validation-
   service>
   ```

   For example, if the category tModel published has the key "uuid:acme-taxonomy-key", and the URL of the validation service is `http://acme.com/externalValidation`, the command with the entry is as follows:

   ```
   java -jar uddiadmin.jar <registry admin URL> <username> <password>
   -setProperty
   oracle.uddi.server.externalValidationTModelList=uuid:acme-taxonomy-key,http:
   //acme.com/externalValidation
   ```

In addition, the timeout limit (in milliseconds) can be tuned for calls to the external validation service using the server property `oracle.uddi.server.externalValidationTimeout` as follows:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
-setProperty oracle.uddi.server.externalValidationTimeout=5000
```

Removing an Externally Validated Category from the Registry

To remove an externally validated category from the registry, perform the following steps:

1. Update the following server property: `oracle.uddi.server.externalValidationTModelList` using the `-setProperty` option by supplying a null value for the `<URL-validation-service>` as follows:

   ```
   java -jar uddiadmin.jar <registry admin URL> <username> <password>
   -setProperty oracle.uddi.server.externalValidationTModelList=<key-value>,""
   ```

   For example, if the category tModel published has the key "uuid:acme-taxonomy-key", and the URL of the validation service is `http://acme.com/externalValidation`, the command with the null entry will be as follows:

   ```
   java -jar uddiadmin.jar <registry admin URL> <username> <password>
   -setProperty
   oracle.uddi.server.externalValidationTModelList=uuid:acme-taxonomy-key,""
   ```

2. Deprecate or update the corresponding tModel data structure. If the tModel is not updated, the registry will reject any new UDDI entries that are categorized or identified by the category that was removed in subsequent save calls to the save_business, save_service, or save_tModel API.

Performance Monitoring and Tuning

On the back end of an Oracle database, UDDI servlets, and the associated JDBC connection pools, can all be monitored using Oracle Enterprise Manager and other standard database monitoring and tuning utilities.

In an OC4J standalone environment, performance information is typically available at

```
http://<oc4j-host-name>:<port-number>/dmsoc4j/Spy
```

Data Backup and Restore Operations

Registry data backup and restore operations can be done by using the standard Oracle database backup and restore operations. See *Oracle9i Backup and Recovery Concepts*.

Additional Information

The following sections are some additional OracleAS UDDI Registry administration information.

UUID Generation

The UUID generation algorithm that is used generates version 4 UUID, which creates UUIDs from random numbers.

All built-in tModel data structures as specified in the UDDI v2.0 specification are included. An additional tModel data structure `uddi-org:operators`, defined in the UDDI v2.0 specification, is also included to classify the bootstrap node businessEntity that represents the OracleAS UDDI Registry itself.

Database Configuration

The following sections describe some database-specific configuration information.

Database Character Set Should Be UTF-8 The database character set should be UTF-8 to accommodate all possible characters. However, if a customer is absolutely certain that the data to be stored in the registry contains characters of a specific country or region (such as western Europe), the customer may use the appropriate database character set.

Functional Index Should Be Enabled The functional index must be enabled to support index-based, case-insensitive search. The following `init.ora` parameter is involved: `query_rewrite_enabled=true`

In addition, the cost-based optimizer must be turned on for analyzing all tables or indexes in the UDDISYS schema. For example:

```
execute dbms_stats.gather_schema_stats(ownname=>'UDDISYS',cascade=>true);
```

Accuracy of Modified Timestamps of UDDI Entities The accuracy of modified timestamps of UDDI entities is dependent on the version and compatibility of the database. If the database compatibility is release 9.0.1 or higher, the modified timestamps are of

SQL type TIMESTAMP, with accuracy up to microseconds. If the database compatibility is below release 9.0.1, the modified timestamps are of SQL type DATE, with accuracy up to seconds.

Transport Security

The Inquiry API in general does not require authentication. However, if the inquiry end point needs to be protected, transport-level authentication, such as HTTP BASIC authentication and HTTPS SSL client authentication, can be enabled by configuring the `web.xml` file. A security role, `uddiguest`, is reserved for accessing the protected inquiry end point. Refer to *Oracle Application Server Containers for J2EE Services Guide* and *Oracle Application Server Containers for J2EE User's Guide* for more information about security roles and related security configuration.

For the Publishing endpoint URL, you may want to allow HTTPS access only. To disable HTTP access, edit the `web.xml` file of the `orauddi` application to enforce data confidentiality and make adjustments to HTTP servers accordingly. Refer to the chapter on security in *Oracle Application Server Containers for J2EE User's Guide* and to *Oracle Application Server Containers for J2EE Services Guide* for more information. For example, to disable HTTP access in the `web.xml` file, use the following code:

```
<user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

Similarly, you can set up HTTPS access for the Administrative endpoint and the UDDI Replication endpoint in the same way.

Standard Classification Support

The OracleAS UDDI Registry uses the following three standard classifications:

- North American Industry Classification System (NAICS)

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

  ```
  http://www.census.gov/epcd/www/naics.html
  ```

- Universal Standard Products and Services Codes (UNSPSC)

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

- ISO-3166 Geographic classification (ISO-3166)

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

`http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html`

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

> **See Also:** "Database Character Set and Built-in ISO-3166 Classification" on page 10-46.

Database Character Set and Built-in ISO-3166 Classification

The UDDI specification mandates that the registry support the full UTF-8 character set. Oracle recommends, though does not require, using UTF-8 as the character set for the Oracle Application Server infrastructure database if the OracleAS UDDI Registry is used.

If the database is not configured with the UTF-8 character set or its equivalent or superset, there could be data corruption and error due to loss in character set conversion to or from UTF-8. Refer to *Oracle9i Globalization Support Guide* for details.

In particular, the descriptions in the UDDI built-in ISO-3166 classification contains descriptions with non-ASCII characters, such as some Western European characters and some Eastern European characters for the names of cities or regions. In order to support the non-UTF-8 database, all non-ASCII characters in the descriptions are replaced with ASCII characters as an approximation.

If you do have a UTF-8 database, you can upgrade the built-in ISO-3166 classification to the one with accurate descriptions using the following instructions:

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

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

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

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

Considerations in a Production Environment

The following information describes some postinstallation configuration steps that you should do immediately after the installation. These steps are not mandatory, but are highly recommended in a production environment.

- **Security for publishing the end point:** By default, HTTP access is enabled. However, HTTPS access is recommended for security concerns. See Transport Security on page 10-45 for more information about disabling HTTP access.
- **Database connection pool sizing and statement caching:** Database connection pool parameters, such as maximum number of database connections and usage of statement caching, should be configured to accommodate the actual database server load.

  If you are using an Oracle database of your choice as the backend storage, the parameters can be configured by editing the data source `jdbc/OracleUddi`. Refer to the chapter on data sources in *Oracle Application Server Containers for J2EE Services Guide* for more information.

  If you are using the Oracle Application Server infrastructure database as the backend storage, the parameters can be configured by modifying the following UDDI server configuration parameters:

  – oracle.uddi.server.db.minConnections
  – oracle.uddi.server.db.maxConnections
  – oracle.uddi.server.db.jdbcDriverType
  – oracle.uddi.server.db.stmtCacheType
  – oracle.uddi.server.db.stmtCacheSize

  Refer to Server Configuration on page 10-25 and Server Configuration Properties Reference Information on page 10-48 for more information.
- **Change of the operatorName and businessEntity discoveryURL prefix:** In some cases, administrators may want to change either the operatorName or businessEntity discoveryURL prefixes, or both parameter values, when moving a system from a staging environment to a production environment.

The SQL script `${ORACLE_HOME}/uddi/admin/uddirpic.sql` on UNIX or `%ORACLE_HOME%\uddi\admin\uddirpic.sql` on Windows can be used to change the these parameter values.

Server Configuration Properties Reference Information

This section describes reference information for some UDDI server configuration properties. It is divided into the following sections:

- Installation or First-Use Properties
- External Classification Validation Properties
- UDDI Replication Properties
- UDDI Replication Scheduler Properties
- Registry-Based Validation Properties
- Database Connection Properties

These server configuration parameters are referenced in Server Configuration on page 10-25. As each example shows, these configuration parameters can be changed only by using the command-line administration tool, `uddiadmin.jar`, which is described in Using the Command-Line Tool uddiadmin.jar on page 10-25.

Installation or First-Use Properties

The following two properties `operatorName` and `businessEntityURLPrefix` should be changed immediately after an installation, but should not be changed afterward:

operatorName

Property name: `operatorName`

Description: Provides the name of the operator of the OracleAS UDDI Registry. This name appears in the operator attribute of responses. Setting this parameter applies in a retroactive fashion to existing entities in the database. For example, changing the operator name results in all business and tModel data structures that currently have the old operator name to be changed to the new operator name.

Note: Be sure to set this parameter before enabling replication.

Property type/allowable values: A non-null string.

Initial value: OracleUddiServer

Typical value: <domain of the UDDI registry>/uddi

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.operatorName=OracleUddiServer
```

businessEntityURLPrefix

Property name: `businessEntityURLPrefix`

Description: Provides the prefix of the generated discoveryURL, which is automatically generated for each businessEntity data structure saved in the registry. The prefix should be customized for your deployment environment. Setting this parameter applies in a retroactive fashion to existing entities in the database. For example, changing the discoveryURL prefix results in all discoveryURLs of usetype "businessEntity" that begin with the old URL prefix to be changed to the new URL prefix.

> **Note:** Be sure to set this parameter before enabling replication.

Property type/allowable values: A valid URL.

Initial value: The OracleAS UDDI Registry will prompt an administrator for an initial value upon server initialization.

Typical value: The host name and port should be the host name and port of the Web server (which may or may not be the same as the servlet container).

Guideline: N/A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.businessEntityURLPrefix=
```

defaultLang

Property name: `defaultLang`

Description: Provides the default language of the registry for the purpose of filling in UDDI v1.0 description elements, which lack a language qualification. Language defaults are not done for UDDI v2.0 requests. Valid values are the values of the xml:lang attribute.

Property type/allowable values: Values of xml:lang.

Initial value: en

Typical value: The location of the primary region the registry serves.

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.defaultLang=en
```

External Classification Validation Properties

The following UDDI server properties can be used with external classification validation:

externalValidation

Property name: `externalValidation`

Description: Determines if external validation occurs.

Property type/allowable values: Boolean (true, false)

Initial value: false

Typical value: false

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.externalValidation=true
```

externalValidationTModelList

Property name: `externalValidationTModelList`

Description: Provides the list of tModel key-URL pairs that represents the categorization and identifier tModel data structures that will be validated by an external SOAP service. The tModelKey and URL values within a pair are separated by a comma (,), and pairs of values are separated by a semicolon (;).

Property type/allowable values: N / A

Initial value: null value ""

Typical value: null value ""

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.externalValidationTModelList=uuid:acme-taxonomy-key,
http://acme.com/externalValidation
```

externalValidationTimeout

Property name: `externalValidationTimeout`

Description: Defines the amount of time, in milliseconds, before timeout occurs for external validation.

Property type/allowable values: long

Initial value: 5000

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.externalValidationTimeout=5000
```

UDDI Replication Properties

The following UDDI server properties can be used with replication:

taskExecutionPeriod

Property name: `taskExecutionPeriod`

Description: Controls the period of time during which replication task should be executed (in milliseconds).

Property type/allowable values: long

Initial value: 5000

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.replication.taskExecutionPeriod=5000
```

maxChangeRecordsSentEachTime

Property name: `maxChangeRecordsSentEachTime`

Description: Controls the maximum number of change records sent out in response to an incoming getChangeRecords request.

Property type/allowable values: integer

Initial value: 100

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.replication.maxChangeRecordsSentEachTime=100
```

pushTaskExecutionPeriod

Property name: `pushTaskExecutionPeriod`

Description: Controls the push task execution period (in milliseconds).

Property type/allowable values: long

Initial value: 45000

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.replication.pushTaskExecutionPeriod=45000
```

pushEnabled

Property name: `pushEnabled`

Description: Controls whether or not push should be performed for UDDI replication.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.replication.pushEnabled=true
```

soapRequestTimeout

Property name: `soapRequestTimeout`

Description: Controls the timeout value for each SOAP replication request (in milliseconds).

Property type/allowable values: long

Initial value: 180000

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
 oracle.uddi.server.replication.soapRequestTimeout=180000
```

soapRequestAuthMethod

Property name: `soapRequestAuthMethod` (Authentication property)

Description: Controls the authentication method the registry node will try to use in sending replication SOAP requests to other nodes. If CLIENT-CERT is used, the administrator must set the wallet password each time the registry node gets started or restarted.

Property type/allowable values: one of {NONE, CLIENT-CERT}

Initial value: NONE

Typical value: CLIENT-CERT

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.replication.soapRequestAuthMethod=NONE
```

walletLocation

Property name: `walletLocation` (Authentication property)

Description: Defines the wallet file name. The wallet file will be located in the same place as uddiserver.config.

Property type/allowable values: N / A

Initial value: `ewallet.p12`

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.replication.walletLocation=ewallet.p12
```

startMaintainingUpdateJournal

Property name: `startMaintainingUpdateJournal` (Advanced use property)

Description: Controls whether or not the update journal will be maintained for UDDI replication. This property must be set to true for replication to occur.

Note: Be sure to upload a correct replication configuration before you set this property to `true`.

Note: Once you set this property to `true`, you should only set it back to `false` if you no longer want to participate in UDDI replication. Setting this property haphazardly from true to false will result in fatal loss of change records.

Property type/allowable values: Boolean (true, false)

Initial value: false

Typical value: false

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.replication.startMaintainingUpdateJournal=false
```

changeRecordWantsAck

Property name: `changeRecordWantsAck` (Advanced use property)

Description: Controls whether or not ACK is required for the change records sent out from the local node.

Property type/allowable values: Boolean (true, false)

Initial value: false

Typical value: false

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
oracle.uddi.server.replication.changeRecordWantsAck=false
```

UDDI Replication Scheduler Properties

The following UDDI server properties can be used to set UDDI replication scheduler properties:

timer_pool_size

Property name: `timer_pool_size`

Description: Specifies the number of concurrently active threads used by the scheduler.

Property type/allowable values: N / A

Initial value: 1

Typical value: 1

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.scheduler.timer_pool_size=1
```

status

Property name: `status`

Description: Indicates whether or not the scheduler is enabled to send out replication requests.

Property type/allowable values: Boolean (0=off, 1=on)

Initial value: 1

Typical value: 1

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.scheduler.status=1
```

Registry-Based Validation Properties

The following UDDI server properties can be used for registry-based validation and quota limit checking:

categoryValidationTModelKeys

Property name: `categoryValidationTModelKeys` (Advanced use property)

Description: Represents the categorization and identifier tModel keys, which will be validated by the registry during an attempted save operation.

Property type/allowable values: A list in the form of '<tModelKey1>', '<tModelKey2>', '<tModelKey3>'.

Initial value: 'UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4', which represents (uddi-org:types classification). The preinstalled value, however, is the UDDI types classification plus the three classifications defined in the UDDI v1.0 specification: (uddi-org:types, uddi-org:iso-ch:3166-1999, ntis-gov:naics:1997, unspsc-org:unspsc).

Note: The uddi-org:types classification *should not* be removed from the list.

Typical value: The preinstalled value.

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty
"oracle.uddi.server.categoryValidationTModelKeys=
'UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4',
'UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88',
'UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2',
'UUID:CD153257-086A-4237-B336-6BDCBDCC6634' "
```

identifierValidation

Property name: `identifierValidation` (Advanced use property)

Description: Controls validation for all IdentifierBag entities. The following flag settings are allowed:

- full -- all validation conditions will be checked
- tmodel_existence -- only tModelKey existence will be validated
- none -- no condition will be checked

Property Type/allowable values: full, tmodel_existence, none

Initial value: full

Typical value: full

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.identifierValidation=full
```

operatorCategory

Property name: `operatorCategory` (Advanced use property)

Description: Determines whether or not additional entities may be categorized as an operator node, if categoryValidation is true.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
  oracle.uddi.server.categoryValidation.operatorCategory=true
```

categoryValidation

Property name: `categoryValidation` (Advanced use property)

Description: Controls validation for all CategoryBag entities. The following flag settings are allowed:

- full -- all validation conditions will be checked
- tmodel_existence -- only tModelKey existence will be checked
- none -- no condition will be checked

Property type/allowable values: full, tmodel_existence, none

Initial value: full

Typical value: full

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.categoryValidation=full
```

assertionKeyedRefValidation

Property name: `assertionKeyedRefValidation` (Advanced use property)

Description: Controls validation for all publisher assertion KeyedReference entities. The following flag settings are allowed:

- full -- all validation conditions will be checked
- tmodel_existence -- only tModelKey existence will be validated
- none -- no condition will be checked

Property type/allowable values: full, tmodel_existence, none

Initial value: full

Typical value: full

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
oracle.uddi.server.assertionKeyedRefValidation=full
```

tModelInstanceInfoKeyValidation

Property name: `tModelInstanceInfoKeyValidation` (Advanced use property)

Description: Determines if tModelKey existence validation occurs within tModelInstanceInfo elements.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
```

```
 [-verbose] -setProperty oracle.uddi.server.tModelInstanceInfoKeyValidation=true
```

addressTModelKeyValidation

Property name: `addressTModelKeyValidation` (Advanced use property)

Description: Determines if tModelKey existence validation occurs within address elements.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.addressTModelKeyValidation=true
```

hostingRedirectorValidation

Property name: `hostingRedirectorValidation` (Advanced use property)

Description: Determines if hostingRedirector validation occurs within bindingTemplate elements. Validation ensures that the referenced bindingTemplate element exists and does not contain a hostingRedirector element.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.hostingRedirectorValidation=true
```

Miscellaneous Properties

The following UDDI server properties are miscellaneous.

quotaLimitChecking

Property name: `quotaLimitChecking`

Description: Determines whether or not publishing quotas, the limits on the number of entities that can be created in the registry per user, are enforced.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password>
 [-verbose] -setProperty oracle.uddi.server.quotaLimitChecking=true
```

schemaValidationUponIncomingRequests

Property name: `schemaValidationUponIncomingRequests` (Advanced use property)

Description: Determines whether or not the server will validate incoming requests against the UDDI XML schema.

Property type/allowable values: Boolean (true, false)

Initial value: true

Typical value: true

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
oracle.uddi.server.schemaValidationUponIncomingRequests=true
```

Database Connection Properties

The following UDDI server properties can be used for configuring database connection properties:

minConnections

Property name: `minConnections` (Advanced use property)

Description: Determines the minimum number of database connections in the connection pool. This property is applicable only if the Oracle Application Server infrastructure database is used as the backend storage.

> **Note:** In a cluster environment, this property must be set for each OC4J instance.

Property type/allowable values: A nonnegative integer that is smaller than the value for `maxConnections`.

Initial value: 1

Typical value: 1

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
  oracle.uddi.server.db.minConnections=1
```

maxConnections

Property name: `maxConnections` (Advanced use property)

Description: Determines the maximum number of database connections in the connection pool. This property is applicable only if the Oracle Application Server infrastructure database is used as the backend storage.

> **Note:** In a cluster environment, this property must be set for each OC4J instance.

Property type/allowable values: A positive integer.

Initial value: 8

Typical value: Depends on the maximum number of concurrent requests and the desired performance.

Guideline: The estimated maximum number of concurrent requests plus a percentage of the buffer.

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
  oracle.uddi.server.db.maxConnections=12
```

jdbcDriverType

Property name: `jdbcDriverType` (Advanced use property)

Description: Defines the type of JDBC driver to be used to access the Oracle Application Server infrastructure database. This property is applicable only if the Oracle Application Server infrastructure database is used as the backend storage.

> **Note:** In a cluster environment, this property must be set for each OC4J instance.

Property type/allowable values: {thin, oci}

Initial value: thin

Typical value: N / A

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
  oracle.uddi.server.db.jdbcDriverType=thin
```

stmtCacheType

Property name: `stmtCacheType` (Advanced use property)

Description: Defines the type of statement caching. This property is to be used with the Oracle Application Server infrastructure database and JDBC driver only.

> **Note:** In a cluster environment, this property must be set for each OC4J instance.

Property type/allowable values: {NONE, IMPLICIT, EXPLICIT}

Initial value: NONE

Typical value: EXPLICIT

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
  oracle.uddi.server.db.stmtCacheType=NONE
```

stmtCacheSize

Property name: `stmtCacheSize` (Advanced use property)

Description: Defines the size (number of statements cached) of statement caching per connection. This property is to be used with the Oracle Application Server infrastructure database and JDBC driver only.

> **Note:** In a cluster environment, this property must be set for each OC4J instance.

Property type/allowable values: integer

Initial value: 50

Typical value: 50

Guideline: N / A

Example:

```
java -jar uddiadmin.jar <registry admin URL> <username> <password> [-verbose]
-setProperty
  oracle.uddi.server.db.stmtCacheSize=50
```

OracleAS UDDI Server Error Message Reference Information

The error codes listed are used by UDDI administrators. In general, UDDI error code E_fatalError can represent various server-side errors that an administrator has to handle.

The specific server-side error is captured in the J2EE application log file. The log file is typically located at `<`*`J2EE_HOME`*`>/application-deployments/orauddi/application.log`. The reference provides additional information for an administrator to diagnose and resolve problems.

WUR-00010: An attempt was made to update a configuration parameter that does not exist "{0}".

Cause: The named UDDI server configuration parameter does not exist.

Action: Correct the spelling of the name of the configuration parameter to be updated. Refer to the configuration parameter reference information for details.

WUR-00011: An attempt was made to update a configuration parameter "{0}" in uddiserver.config. That file cannot be found.

Cause: The UDDI server configuration file uddiserver.config could not be found.

Action: Make sure that the JVM property oracle.home of the OC4J instance is defined properly.

WUR-00012: The specified user name, "{0}", is not a name that is known to the registry.

Cause: The named user does not exist in the registry.

Action: Correct the spelling of the named user.

WUR-00013: The 'Default' role for publishing limits may not be deleted.

Cause: An attempt was made to remove the system-defined user quota role 'Default.'

Action: Do not delete the user quota role 'Default.' If the 'Default' user quota role is not desirable, set the quota limits to zero to disable it.

WUR-00050: Unable to retrieve subscription management configuration parameter "{0}": Internal database schema configuration error encountered.

Cause: An internal database configuration error occurred while retrieving the configuration parameter for the subscription management module.

Action: Identify the database error message embedded in the details of the error. Correct the database configuration according to the database error message.

WUR-00051: Unable to set subscription management configuration parameter "{0}": Internal database schema configuration error encountered.

Cause: An internal database configuration error occurred while setting the configuration parameter for the subscription management module.

Action: Identify the database error message embedded in the details of the error. Correct the database configuration according to the database error message.

WUR-00100: An internal error occurred while marshaling the response.

Cause: An unexpected internal error occurred in writing the response to a client.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00101: An internal error occurred while unmarshaling the request.

Cause: An unexpected internal error occurred in parsing the request sent by a client.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00104: The value of the configuration parameter named "{0}" is invalid.

Cause: The value of the named UDDI server configuration parameter was invalid.

Action: Refer to the configuration parameter reference information for the valid values. Use the UDDI administration tool to update the configuration parameter.

WUR-00105: A database error with SQL code "{0}" occurred while trying to "{1}".

Cause: An unexpected database error occurred in carrying out the named action.

Action: Identify and correct the database error. The database error is embedded in the details of the error.

WUR-00106: An internal error caused the request to fail to make the specified updates. While rolling back the changes, another error occurred; this leaves data in an unpredictable state.

Cause: An unexpected database error occurred in rollback phases of error processing.

Action: Identify and correct the database error. The database error is embedded in the details of the error.

WUR-00107: An internal error occurred while committing the requested changes to the registry; this leaves data in an unpredictable state.

Cause: An unexpected database error occurred in committing the requested changes.

Action: Identify and correct the database error. The database error is embedded in the details of the error.

WUR-00108: An internal error occurred while trying to get a connection to the underlying database.

Cause: An unexpected database error occurred in obtaining a database connection to serve the request.

Action: Identify and correct the database error. The database error is embedded in the details of the error.

WUR-00109: An internal error occurred while trying to close a connection to the underlying database.

Cause: An unexpected database error occurred during the release of the database connection after the request was served.

Action: Identify and correct the database error. The database error is embedded in the details of the error.

WUR-00110: An internal error occurred while trying to create and set up a data source abstraction for the underlying database.

Cause: An unexpected internal error occurred while creating the database connection pool.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00111: An internal error occurred while trying to perform a JNDI lookup and locate of the object "{0}".

Cause: An internal error occurred in obtaining the named object from the JNDI context. Examples of possible objects include database connection pools, message queues, and so forth.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00113: An internal error occurred while trying to access the repository API to set up a data source abstraction.

Cause: An unexpected internal error occurred while creating the database connection pool using Oracle Application Server metadata repository access API.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00114: An internal error occurred while trying to generate a Universal Unique Identifier (UUID).

Cause: An unexpected internal error occurred while generating a UUID.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00115: The registry was unable to retrieve OC4J-specific environment settings from the J2EE container; the user "{0}" cannot be authenticated.

Cause: An unexpected internal error occurred while authenticating the user. The error is usually due to incorrect settings in web.xml or using an unsupported version of the OC4J container.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00116: An internal error occurred while performing the automatic postinstallation configuration for the UDDI registry. Regular registry operations cannot proceed if the registry is not properly configured.

Cause: An unexpected internal error occurred in performing the automatic postinstallation configuration for the UDDI registry.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00117: Cannot close data source properly.

Cause: An unexpected internal error occurred while closing the database connection pool during shutdown of the UDDI registry.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00200: An internal error occurred during external validation.

Cause: An unexpected internal error occurred while making a validation call to an external validation service.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00201: An internal error occurred during external validation while processing the in-memory request.

Cause: An unexpected internal error occurred while processing the UDDI entities in the request before they were sent for external validation.

Action: Identify and correct the internal error. The internal error is embedded in the details of the error.

WUR-00202: An internal error occurred during external validation because the tModel list property, "{0}", has the wrong format.

Cause: The value of the UDDI server configuration property, oracle.uddi.server.externalValidationTModelList, was invalid.

Action: Correct the value. Refer to the configuration parameter reference information for details.

WUR-00203: An internal error occurred during external validation because the timeout property, "{0}", is not the right integer format.

Cause: The value of the UDDI server configuration property, oracle.uddi.server.externalValidationTimeout, was invalid.

Action: Correct the value. Refer to the configuration parameter reference information for details.

WUR-00204: An internal error occurred during external validation because the response is not a correct DispositionReport.

Cause: DispositionReport returned by the external validation service was invalid. For example, DispositionReport was empty.

Action: Contact the external validation service provider.

WUR-00205: An internal error occurred during external validation because the response is not expected. The response is of code "{0}" with message "{1}".

Cause: DispositionReport returned by the external validation service contained an unexpected DispositionReport error number.

Action: Contact the external validation service provider.

WUR-00300: DB schema version is missing. Please check DB for VERSION table.

Cause: The version of the database schema for persistent storage was missing.

Action: Contact Oracle Support Services.

WUR-00301: DB schema version "{0}" is incompatible with mid-tier version. DB schema must be updated to make the UDDI registry function.

Cause: The version of the database schema for persistent storage was not supported by the version of the registry being used.

Action: Upgrade the database schema to the latest version. Refer to the UDDI database schema upgrade documentation for details.

WUR-00302: An internal error occurred while trying to retrieve and load the UDDI DELTA server property file.

Cause: An internal error occurred while initializing the UDDI registry in the backward compatibility mode with an older version of the database schema.

Action: Contact Oracle Support Services.

WUR-00303: This operation is not allowed by DB schema version "{0}". You must upgrade DB schema to the latest version to carry out this operation.

Cause: The requested operation was not supported because the UDDI registry was running in the backward compatibility mode with an older version of the database schema.

Action: Upgrade the database schema to the latest version. Refer to the UDDI database schema upgrade documentation for details.

WUR-05001: Cannot find the UDDI entity just saved.

Cause: An unexpected internal error occurred in updating the update journal.

Action: Contact Oracle Support Services.

WUR-05002: Cannot perform custody transfer for an entity that is not businessEntity or tModel. The key of the offending entity is "{0}".

Cause: In the custody transfer change record, the specified UDDI entity is not businessEntity or tModel.

Action: Contact the administrator of the UDDI node where the change record originated.

WUR-05003: Warning: Received a duplicate change record originating from node "{0}" with usn "{1}".

Cause: A duplicate change record sent from the named UDDI node was detected.

Action: No action is needed. This is merely an informational message.

WUR-05004: Received an out-of-order change record originating from node "{0}" with usn "{1}". The change record with usn "{2}" has been processed.

Cause: The named change record was received after a change record with a larger update sequence number (USN) had been processed.

Action: Contact the administrator of the UDDI node where the change record originated.

WUR-05005: The change record originating from node "{0}" with usn "{1}" is invalid because the named node is not recognized.

Cause: The originating node of the named change record was not recognized. In other words, the node was not recorded in the replication communication graph.

Action: Contact the administrator of the UDDI node that provided the change record.

OracleAS UDDI Content Syndication UI Implementation Error Message

The following error message is associated with the UDDI content syndication UI implementation. This error is returned to the user, non administrator, as a message within the GUI.

Error Code OSS-00301:

The requested action can be only done by an administrator.

Cause: The logged in user does not have enough privileges to perform the requested action.

Action: Login as administrator and request the action again.

UDDI Open Database Support

In addition to the Oracle Application Server infrastructure database, the following databases are supported:

- Microsoft SQL Server
- IBM DB2
- Oracle (non-Oracle Application Server infrastructure database)

For Microsoft SQL Server and IBM DB2, the Oracle Application Server DataDirect Connect JDBC driver is needed.

The following installation steps for SQL Server, DB2, and Oracle assume that the relevant database server has been installed. These instructions also assume that Oracle Application Server Portal has been installed, which should copy the relevant

UDDI files to `${ORACLE_HOME}/uddi/admin` on UNIX or `%ORACLE_HOME%\uddi\admin` on Windows.

Microsoft SQL Server

The following sections describe installation and configuration information.

Script Source Directory

Installation must be performed from a Windows machine. If the `%ORACLE_HOME%\uddi\admin\mssql` directory is not accessible from the SQL Server machine, then copy this directory to a location that is accessible. This directory (or the original `%ORACLE_HOME%\uddi\admin\mssql` if no copying is necessary) will be referred to as *%MSSQL_HOME_DB%*.

Create the Database and User

The `%MSSQL_HOME_DB%\wurcreatedb_mssql.sql` script has been provided to create the `uddisys` database and `uddisys` user for a SQL Server instance in `mixed-authentication` mode. If you are using Windows authentication or wish to alter some of the settings in this script, you may do so as long as all the following requirements are met:

- The collation for the `uddisys` database must be case-sensitive.
- Recursive triggers must be enabled on the `uddisys` database.
- The `uddisys` user must have the `uddisys` database as its default database.
- The `uddisys` user must be a member of the `db_owner` role for the `uddisys` database.

To run the script with the Microsoft `osql` utility, use the administrator login and password (`sa/sa`):

```
osql -S <server> -U sa -P sa  -i wurcreatedb_mssql.sql
```

where *<server>* is the server hosting the SQL Server instance.

Install the Schema

Go to the *%MSSQL_HOME_DB%* directory. Use the `osql` utility to execute the SQL script `wurinst_mssql.sql` using the `uddisys/uddisys` account created in Create the Database and User on page 10-72.

The syntax is as follows:

```
osql -S <server> -U <user> -P <password> -d <database> -i wurinst_mssql.sql
```

where `<server>` is the server hosting the SQL Server instance.

For example:

```
osql -S server-machine -U uddisys -P uddisys -d uddisys -i wurinst_mssql.sql
```

Import BUILTIN_CHECKED_CATEGORY Table Entries

Import the `iso3166-99_tModelKey.txt`, `naics-97_tModelKey.txt`, and `unspsc-73_tModelKey.txt` files into the BUILTIN_CHECKED_CATEGORY table as follows:

1. Select the **Import and Export Data** option from the SQL Server Start menu options. Click **Next**.
2. For the Data Source, select the last option, **Text File**. Then, provide the name and location of the appropriate text file, *%MSSQL_HOME_DB%*`\iso3166-99_tModelKey.txt`. Click **Next**.
3. The default file format should be **Delimited**. Accept this by clicking **Next**.
4. Set the delimiter to the ("**|**") character. Click **Next**.
5. Select the **uddisys** database for the destination. Provide the appropriate authentication mechanism and credentials, which are `SQL Server Authentication` with user `uddisys` and password `uddisys`, by default. Make sure that the selected database is **uddisys**. Click **Next**.
6. Click the **Destination** and select the **BUILTIN_CHECKED_CATEGORY** table.
7. Click **Transform**. Map TMODEL_KEY to Col001, KEY_NAME to Col003, KEY_VALUE to Col002, and PARENT_VALUE to Col004. Click **OK**.
8. Click **Next**.
9. Click **Next** to run immediately and click **Finish** to start.
10. Repeat this process for the `naics-97_tModelKey.txt` and `unspsc-73_tModelKey.txt` files.

Note: If the character set of your database is not UTF-8, do not use the script `iso3166-99.txt` to load the ISO3166 taxonomy because the taxonomy contains characters from different languages. Instead, use the script iso3166-99-ascii.txt to load an ASCII-only version of the taxonomy.

Configure OracleAS OC4J to Use SQL Server

Define a data source with the name and location set to `jdbc/OracleUddi` to reflect that SQL Server is the desired database, like the following:

```
<data-source
    class="com.evermind.sql.DriverManagerDataSource"
    name="jdbc/OracleUddi"
    location="jdbc/OracleUddi"
    connection-driver="com.oracle.ias.jdbc.sqlserver.SQLServerDriver"
    username="uddisys"
    password="uddisys"
url="jdbc:oracle:sqlserver://<servername>:1433;SelectMethod=cursor;User=uddisys;
Password=uddisys"
/>
```

Note that *`<servername>`* is the network name or IP address of the server hosting the SQL Server instance used for UDDI.

The data source needs to be accessible by the `orauddi.ear` and `oraudrepl.ear` files.

Refer to the Data Sources chapter in the *Oracle Application Server Containers for J2EE Services Guide* for more information.

Restart the UDDI server for these changes to take effect.

IBM DB2

The following sections describe installation and configuration information.

Script Source Directory

If the $*`{ORACLE_HOME}`*`/uddi/admin/db2` directory is not accessible from the machine with the relevant DB2 tools, then copy this directory to a location that is accessible. This directory will be referred to as $*`{DB2_HOME_DB}`* on UNIX or *`%DB2_HOME_DB%`* on Windows.

Create the Database and User

Go to the $*{DB2_HOME_DB}* directory on UNIX or the *%DB2_HOME_DB%* directory on Windows. The `wurcreatedb_db2.sql` script is provided for creating the `uddisys` database. The user is responsible for creating a `uddisys` user with password `uddisys` based on the authentication scheme that is being used for DB2. By default, this requires creating a `uddisys` user at the operating system level.

If you wish to alter some of the settings in this script, you may do so as long as both the following requirements are met:

- The default tablespace for the `uddisys` database must be at least 8 KB pages. This also requires providing a buffer pool that will support a page size of at least 8 KB.
- The `applheapsz` parameter must be increased to approximately 12800 pages.

To run the script, start the DB2 Command Line Processor by entering `db2` in UNIX or `db2cmd` in Windows. Then, execute the script:

```
db2 -t +p < wurcreatdb_db2.sql
```

where `-t` allows the use of semicolons to terminate SQL statements and `+p` suppresses prompting.

Install the Schema

Run the `wurinst_db2.sql` script. This also triggers the `wurcreat.sql`, `wurdbsql.sql`, and `wurpopul.sql` scripts. To run these scripts, do the following:

Launch the command-line processor as previously described, then enter the following:

```
db2 -t +p < wurinst_db2.sql
```

Import BUILTIN_CHECKED_CATEGORY Table Entries

Import the `iso3166-99_tModelKey.txt`, `naics-97_tModelKey.txt`, and `unspsc-73_tModelKey.txt` files into the BUILTIN_CHECKED_CATEGORY table as follows:

1. Right click the table **BUILTIN_CHECKED_CATEGORY** from the Control Center and select **IMPORT**.
2. Specify the Import file as $*{DB2_HOME_DB}*`/iso3166-99_tModelKey.txt` for UNIX or *%DB2_HOME_DB%*`\iso3166-99_tModelKey.txt` for Windows.

3. Select **Delimited ASCII format (DEL)**. Click **Options** and select ('|') as the delimiter.
4. Use the INSERT import mode (the default).
5. Set the Commit records equal to 500.
6. For the Message file, enter `${DB2_HOME_DB}/uddi/admin/db2/iso3166-99_tModelKey.log` for UNIX or `%DB2_HOME_DB%\uddi\admin\db2\iso3166-99_tModelKey.log` for Windows.
7. Go to the **Columns** tab. Select **Include Columns by Position**. Map TMODEL_KEY to 1, KEY_NAME to 3, KEY_VALUE to 2, and PARENT_VALUE to 4.
8. Click **OK** to run the import process.
9. Repeat this process for the `naics-97_tModelKey.txt` and `unspsc-73_tModelKey.txt` files.

Note: If the character set of your database is not UTF-8, do not use the script `iso3166-99.txt` to load the ISO3166 taxonomy because the taxonomy contains characters from different languages. Instead, use the script iso3166-99-ascii.txt to load an ASCII-only version of the taxonomy.

Configure OracleAS OC4J to Use DB2

The following sections describe how to create the DB2 package and modify the URL for regular use.

Create a DB2 Package Define a data source with the name and location set to `jdbc/OracleUddi` to reflect that DB2 is the desired database, like the following:

```
<data-source
    class="com.evermind.sql.DriverManagerDataSource"
    name="jdbc/OracleUddi"
    location="jdbc/OracleUddi"
    connection-driver="com.oracle.ias.jdbc.db2.DB2Driver"
    username="uddisys"
    password="uddisys"
url="jdbc:oracle:db2://<servername>:50000;databaseName=UDDISYS;PackageName=JDBCP
KG;DynamicSections=512;CreateDefaultPackage=TRUE;ReplacePackage=true"
/>
```

Note that *<servername>* is the network name or IP address of the server hosting the DB2 instance used for UDDI.

The data source needs to be accessible by the `orauddi.ear` and `oraudrepl.ear` files.

Refer to the Data Sources chapter in the *Oracle Application Server Containers for J2EE Services Guide* for more information.

Now, launch the UDDI server so that these initial URL connection strings will be used to create the appropriate default package in DB2.

Modify the URL for Regular Use Now that the DB2 package has been created, update the data source defined in the previous step (see Create a DB2 Package on page 10-76) and change the URL attribute from:

```
url="jdbc:oracle:db2://<servername>:50000;databaseName=uddisys;PackageName=JDBCP
KG;DynamicSections=512;CreateDefaultPackage=TRUE;ReplacePackage=true"
```

to:

```
url="jdbc:oracle:db2://<servername>:50000;databaseName=uddisys;PackageName=JDBCP
KG;DynamicSections=512"
```

Note that the last two parameters, `CreateDefaultPackage` and `ReplacePackage`, have been removed from the final URL attribute.

Once these changes have been made to both `data-sources.xml` files, restart the UDDI server for the changes to take effect.

Oracle (Non-OracleAS Infrastructure Database)

The following sections describe installation and configuration information.

Script Source Directory

If the $*{ORACLE_HOME}*`/uddi/admin` directory is not accessible from the server with the relevant Oracle tools, then copy this directory to a location that is accessible. This directory will be referred to as $*{ORACLE_HOME_ORACLE}* on UNIX or *%ORACLE_HOME_ORACLE%* on Windows.

Create the Database and User

The following steps describe how to create the `uddisys` database and the `uddisys` user:

1. Go to the $*{ORACLE_HOME_ORACLE}* directory on UNIX or the *%ORACLE_HOME_ORACLE%* directory on Windows.
2. Use SQL*Plus to execute the SQL script `wurinst.sql` using the `sys` user account. For example:

   ```
   sqlplus "sys/change_on_install as sysdba" @wurinst.sql
   ```

The schema `uddisys` is created with the password `uddisys`. A log file `wurinst.log` is produced.

Populate the Validated Taxonomy Codes

Populate the validated taxonomy codes using SQL*Loader with the three control scripts: `naics-97.ctl`, `iso3166-99.ctl`, and `unspsc-73.ctl`. For example:

```
sqlldr userid=uddisys/uddisys control=naics-97.ctl
sqlldr userid=uddisys/uddisys
   control=unspsc-73.ctl
sqlldr userid=uddisys/uddisys  control=iso3166-99.ctl
```

Note: If the character set of your database is not UTF-8, do not use the script `iso3166-99.ctl` to load the ISO3166 taxonomy because the taxonomy contains characters from different languages. Instead, use the script to load an ASCII-only version of the taxonomy:

```
sqlldr userid=uddisys/uddisys control=iso3166-99-ascii.ctl
```

Configure OracleAS OC4J to Use the Non-OracleAS Infrastructure Database

Define a data source with the name and location set to `jdbc/OracleUddi` to reflect that non-Oracle Application Server infrastructure database is the desired database, like the following:

```
<data-source
    class="oracle.jdbc.pool.OracleConnectionCacheImpl"
    name="jdbc/OracleUddi"
    location="jdbc/OracleUddi"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="uddisys"
    password="uddisys"
url="jdbc:oracle:thin:@<servername>:1521:<oracle sid>"
```

```
/>
```

Note that `<servername>` is the network name or IP address of the server hosting the non-Oracle Application Server infrastructure database instance used for UDDI.

The data source needs to be accessible by the `orauddi.ear` and `oraudrepl.ear` files.

Refer to the Data Sources chapter in the *Oracle Application Server Containers for J2EE Services Guide* for more information.

Restart the UDDI server for these changes to take effect.

UDDI Subscription Service

The OracleAS UDDI Registry, leveraging OracleAS Syndication Services, provides a subscription service allowing publishers in the registry to monitor or obtain changes in the registry. By specifying a specific query or a set of entities, an administrator can define an offer that provides changes in the entities that are interesting to some users or scenarios. For example:

- An administrator can define an offer that provides changes to any businessService entities classified in the NAICS classification scheme, for example, under the category named `mining` .
- Publishers in the registry are interested in these types of services and can subscribe to the offer.

For example, if you want to find all changes to any businessServices entities that are classified under the topic name *mining* in the NAICS classification scheme, as a user of the registry, you can subscribe to such an offer and automatically receive periodic updates to the content.

Defining Offers

An administrator defines an offer using the OracleAS Syndication Services administrators tool.

1. Determine the content connector to be used based on the type of filtering criteria.
2. Create a content provider resource defining the specific filtering criteria.
3. Create an offer.
    a. Define the contract (such as licensing terms, delivery rules) of the offer.

b. Grant the offer to `uddi_syndication` application user, so that regular UDDI publishers can subscribe to the offer using the UDDI Content Subscription Manager.

> **Note:** The offer must be granted to uddi_syndication user. It is the user used in UDDI Content Subscription Manager.

See *Oracle Application Server Syndication Services Developer's and Administrator's Guide* for more information.

For example, to provide an offer of changes to businessService entities classified under the mining category in the NAICS classification scheme, the administrator would do the following:

1. Determine the content connector to be used. First, the administrator determines the content connector to be used based on the type of filtering criteria. In this example, the type of filtering criteria is a finding services by categoryBag. Therefore, the content connector to be used is UddiFindServiceByCategoryBagCPAdaptor.
2. Create a content provider resource. Secondly, the administrator creates a content provider resource using the content connector selected. The content provider resource defines the specific filtering criteria. In this example, the UDDI Subscription Service specific filtering criteria is that mining category in the NAICS classification scheme.
3. Create an offer. Finally, the administrator creates an offer (to which regular UDDI publishers can subscribe) using the content provider resource. In addition, to the filtering criteria, the administrator does the following:
   a. Defines the contract (such as licensing terms, delivery rules) of the offer.
   b. Grants the offer to `uddi_syndication` application user, so that regular UDDI publishers can subscribe to the offer using the UDDI Content Subscription Manager.

OracleAS Syndication Services comes configured with the following content UDDI connectors. A description of each connector is provided along with the specified input arguments or properties each contains.

- UddiFindBusinessByCategoryBagCPAdaptor -- find businesses by category bag; the category bag contains only one keyed reference with the following properties: tModel key, key name, and key value.

- UddiFindServiceByCategoryBagCPAdaptor -- find services by category bag; the category bag contains only one keyed reference with the following properties: tModel key, key name, and key value.
- UddiFindServiceByTModelBagCPAdaptor -- find services by TModel bag; the tModel bag contains only one keyed reference with the following properties: tModel key, key name, and key value.
- UddiFindTModelByCategoryBagCPAdaptor -- find TModels by category bag; the category bag contains only one keyed reference with the following properties: tModel key, key name, and key value.
- UddiGetBusinessDetailCPAdaptor -- get the full businessEntity information for one business identified by a BusinessKey.
- UddiGetServiceDetailCPAdaptor -- get full details for a registered businessService identified by a ServiceKey.
- UddiGetBindingDetailCPAdaptor -- get full bindingTemplate information suitable for making one or more service requests identified by a BindingKey.
- UddiGetTModelDetailCPAdaptor -- get full details for a registered tModel data structure identified by a TModelKey.
- UddiFindBusinessByNameCPAdaptor -- find one business by name; the name is the business name prefix.

The OracleAS Syndication Services administrator:

1. May register a content provider for any of these preconfigured connectors by specifying its properties, selecting the desired UDDI content connector, and specifying settings to access the content repository and its resources.
2. May create an offer for a content provider by selecting the content provider resource, specifying its offer properties, and choosing users or groups to which to grant access to this offer.

Once the offers are created, the UDDI Content Subscription Administrator uses the Web-based UDDI Content Subscription Manager to:

1. Manage UDDI application subscription properties, such as configuring the UDDI Content Subscription Manager with OracleAS Syndication Services.
2. Subscribe to available offers as well as cancel his own subscriptions and those belonging to any user.

See Subscribing to an Offer on page 10-83 for more information about using this administrative tool to create OracleAS UDDI Registry UDDI Registry-based subscriptions. See *Oracle Application Server Syndication Services Developer's and Administrator's Guide* for more information about managing and registering content providers and creating offers and associated offer contracts with content providers.

Advanced Topic: Creating New UDDI Content Connectors

OracleAS UDDI Registry provides a command-line tool to facilitate the automatic generation of custom OracleAS Syndication Services content connectors for various UDDI inquiry requests. The command-line tool can be described as follows:

`generateCPA`

Parameter: `<registry admin URL> <username> <password> [-verbose] -generateCPA <javaClassName> <uddiRequestXMLTemplate>`

Description: Given the UDDI request template XML, generates an OracleAS Syndication Services content connector, in the format of a Java class file. The generated Java class file will have the name as specified by the `javaClassName` parameter and a fixed Java package of `oracle.uddi.server.subscription.cp`. In order for OracleAS Syndication Services to find it, the Java class file should be incorporated into the existing JAR file located in the following directory:

```
For UNIX:

<ORACLE_HOME>/syndication/lib/cp/uddicpas.jar

For Windows:

<ORACLE_HOME>\syndication\lib\cp\uddicpas.jar
```

For example, given the following XML file `findbiz.xml` as a UDDI request template, perform the following steps:

```
<find_business xmlns='urn:uddi-org:api_v2' generic='2.0'>
   <findQualifiers>
     <findQualifier>sortByNameDesc</findQualifier>
     <findQualifier>sortByDateAsc</findQualifier>
     <findQualifier>caseSensitiveMatch</findQualifier>
   </findQualifiers>
   <name>$(BusinessName,"Test")</name>
</find_business>
```

Note that parameter definitions are allowed in this XML template. The syntax is as follows: `$(parameterName)` or `$(parameterName,"default_value")`. For the find_business request template, a parameter with a preset value is defined. An Oracle Application Server UDDI Content Subscription Administrator can generate offers by setting different values to the `BusinessName` parameter after loading the content connector generated from this XML template. Note that this UDDI request XML template *must* have a UDDI v2 namespace.

1. Execute the `generateCPA` command as follows:

   ```
   java -jar uddiadmin.jar <registry admin URL> <username> <password>
    [-verbose] -generateCPA UddiFindBizCPAdaptor findbiz.xml
   ```

   A `UddiFindBizCPAdaptor.class` file will be generated and must be incorporated into the `uddicpas.jar` file.

2. Navigate to the directory where the `uddicpas.jar` file is located.

   a. Create under this current directory the subdirectory `oracle/uddi/server/subscription/cp` on UNIX or the subdirectory `oracle\uddi\server\subscription\cp` on Windows.

   b. Copy your class file, UddiFindBizCPAdaptor.class into the *<ORACLE_HOME>*/`syndication/lib/cp/oracle/uddi/server/subscription/cp` directory on UNIX or the *<ORACLE_HOME>*`\syndication\lib\cp\oracle\uddi\server\subscription\cp` directory on Windows.

   c. Execute the following JAR command:

      ```
      On UNIX:
      jar -uf uddicpas.jar
      oracle/uddi/server/subscription/cp/UddiFindBusinessByNameCPAdaptor.class

      On Windows:
      jar -uf uddicpas.jar
      oracle\uddi\server\subscription\cp\UddiFindBusinessByNameCPAdaptor.class
      ```

3. Register the connector. Refer to *Oracle Application Server Syndication Services Developer's and Administrator's Guide* for more information.

Subscribing to an Offer

The UDDI Content Subscription Manager is a Web-based application that allows users (publishers and administrators) to subscribe to offers from content providers through specialized UDDI content connectors managed by OracleAS Syndication Services. As subscribers to the OracleAS UDDI Registry syndicated by OracleAS

Syndication Services, users can create subscriptions to obtain changes in UDDI Registry content delivered to them through e-mail. Users can also cancel their own subscriptions.

The UDDI Content Subscription Manager recognizes two types of users, the regular user or publisher, who has the UDDI publisher privilege, and the administrator of the UDDI Content Subscription Manager, who logs in as a UDDI administrator, for example `ias_admin`.

The regular user can do the following (see Using the UDDI Content Subscription Manager as a Publisher on page 10-84):

- Subscribe to offers (create subscriptions).
- Cancel only their own subscriptions.

The administrator (for example, `ias_admin`) can do the following (see Using the UDDI Content Subscription Manager as a UDDI Administrator on page 10-91):

- Subscribe to offers (create subscriptions).
- Cancel their own subscriptions as well as all subscriptions belonging to all users.
- Enter or change UDDI subscription application properties, such as configuring the UDDI Content Subscription Manager with OracleAS Syndication Services. These configurable properties include specifying:
  - The syndication services URL.
  - The syndication subscriber user name and password for UDDI (the syndication user name and password for the special UDDI application subscriber).
  - The syndication connection pool size. This is the pool size for syndication connections held by the subscription application.
  - The logging level or the level of detail to record in the log file.

Using the UDDI Content Subscription Manager as a Publisher

To use the UDDI Content Subscription Manager as a UDDI publisher, perform the following steps:

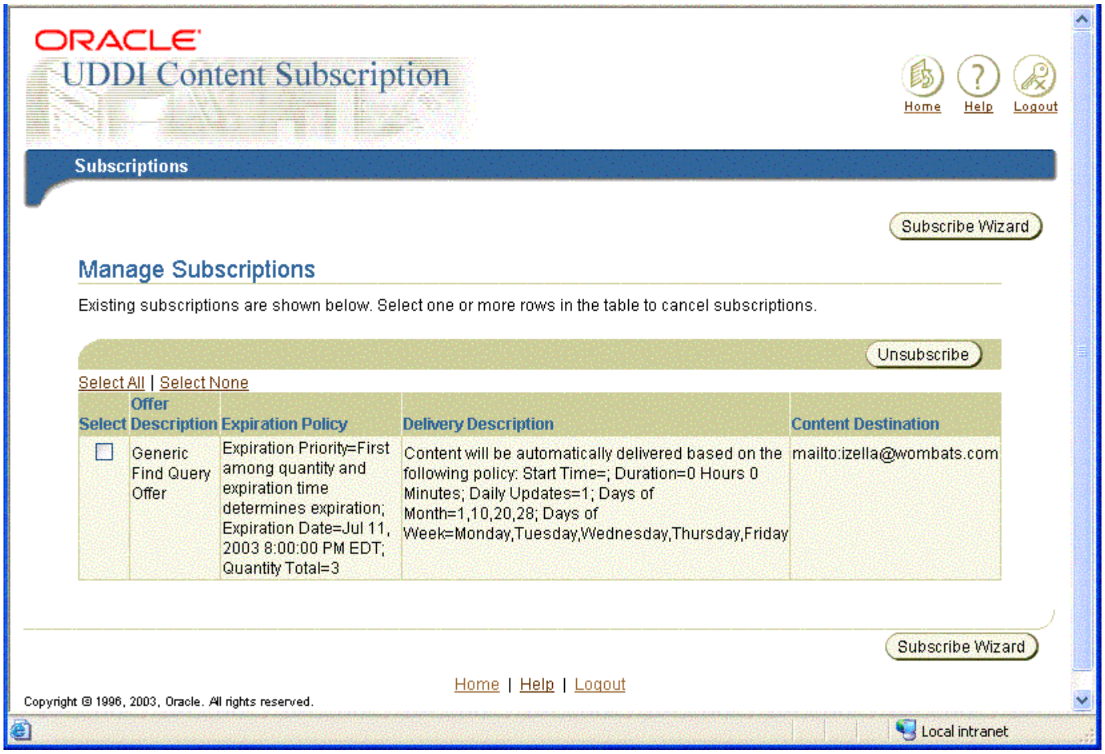
1. Start the UDDI Content Subscription Manager by entering the following URL:

   ```
   http://<host>:<port>/uddisub/subscription/ui
   ```

where the `<host>` parameter indicates the system on which the UDDI Content Subscription Manager is installed and the `<port>` parameter specifies the port number on which it is running.

2. Next, log in as a UDDI publisher (for example, `uddi_publisher`/`<publisher-password>`). The UDDI Content Subscription Manager home page or **Subscriptions** page is displayed, as shown in Figure 10–2.

Figure 10–2 Subscriptions Page



Manage Subscriptions

Existing subscriptions are shown below. Select one or more rows in the table to cancel subscriptions.

Select	Offer Description	Expiration Policy	Delivery Description	Content Destination
☐	Generic Find Query Offer	Expiration Priority=First among quantity and expiration time determines expiration; Expiration Date=Jul 11, 2003 8:00:00 PM EDT; Quantity Total=3	Content will be automatically delivered based on the following policy: Start Time=; Duration=0 Hours 0 Minutes; Daily Updates=1; Days of Month=1,10,20,28; Days of Week=Monday,Tuesday,Wednesday,Thursday,Friday	mailto:izella@wombats.com

As the UDDI publisher, you can do any of the following tasks:

a. Create a subscription.

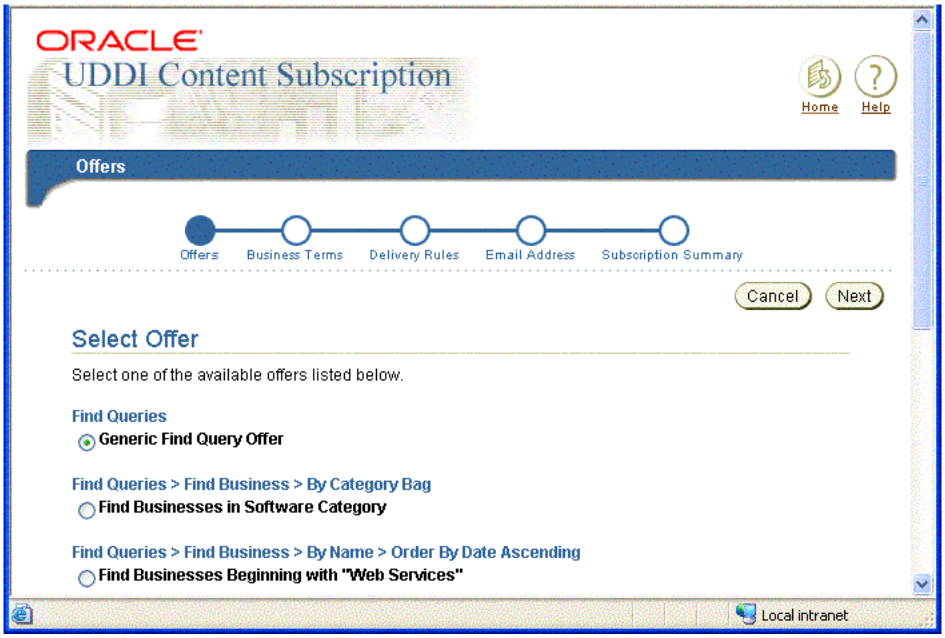
Click **Subscribe Wizard** to launch a 5-step subscribe wizard that lets you choose an offer, accept the business terms of the offer, select the delivery rules for delivering content to you, specify the e-mail address to where content is to be delivered, and review a summary of the specified subscription information before you create the subscription.

b. Cancel a subscription.

Select an existing subscription by selecting its corresponding box in the **Select** column, then click **Unsubscribe**.

3. To subscribe to an offer, click **Subscribe Wizard**. In the first of 5 steps of the subscribe wizard, the **Offer**s page is displayed, as shown in Figure 10–3.

Figure 10–3 Offers Page



Select Offer

Select one of the available offers listed below.

Find Queries

[● Generic Find Query Offer]

Find Queries > Find Business > By Category Bag

○ Find Businesses in Software Category

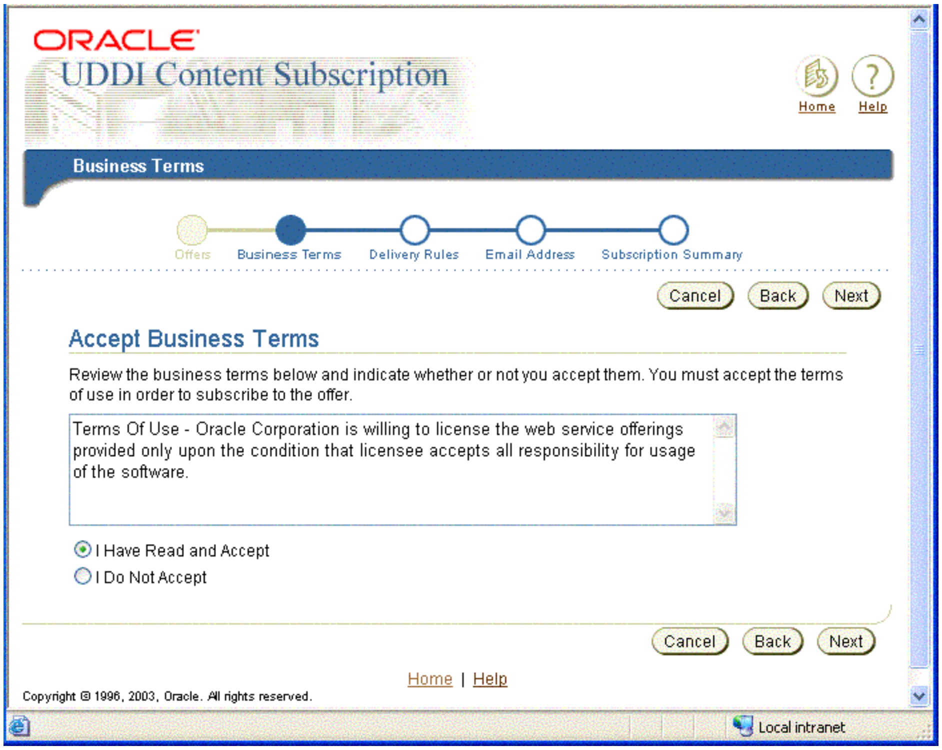
Find Queries > Find Business > By Name > Order By Date Ascending

○ Find Businesses Beginning with "Web Services"

a. At the **Offers** page, select one of the available offers from the list, then click **Next** to continue to the next step.

b. At the **Business Terms** page, as shown in Figure 10–4, review the business terms of the offer. If the business terms are acceptable, click the radio button **I Have Read and Accept**, then click **Next** to continue to the next step. If the business terms are not acceptable, click **Back** to return to the previous **Offers** page and find another offer whose business terms are acceptable.

Figure 10–4 Business Terms Page



Accept Business Terms

Review the business terms below and indicate whether or not you accept them. You must accept the terms of use in order to subscribe to the offer.

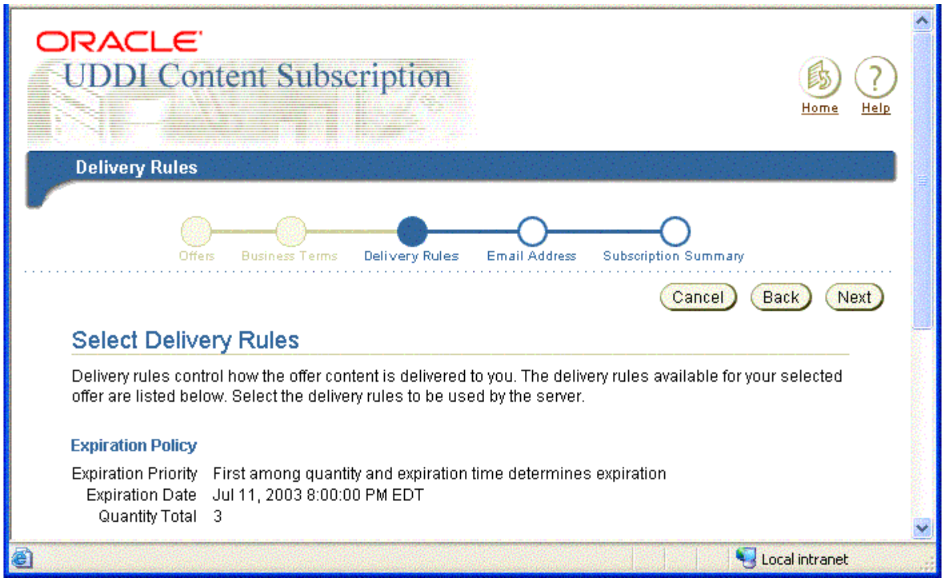
Terms Of Use - Oracle Corporation is willing to license the web service offerings provided only upon the condition that licensee accepts all responsibility for usage of the software.

[✓ I Have Read and Accept]
I Do Not Accept

Copyright © 1996, 2003, Oracle. All rights reserved.

c. At the **Delivery Rules** page, as shown in Figure 10–5 and Figure 10–6, select the delivery rules to be used by the OracleAS Syndication Services to deliver content to you by clicking its box, then, click **Next** to continue to the next step. The expiration policy information is displayed. Only push delivery rules are available for selection.

Figure 10–5 Delivery Rules Page (Top Half of Page)



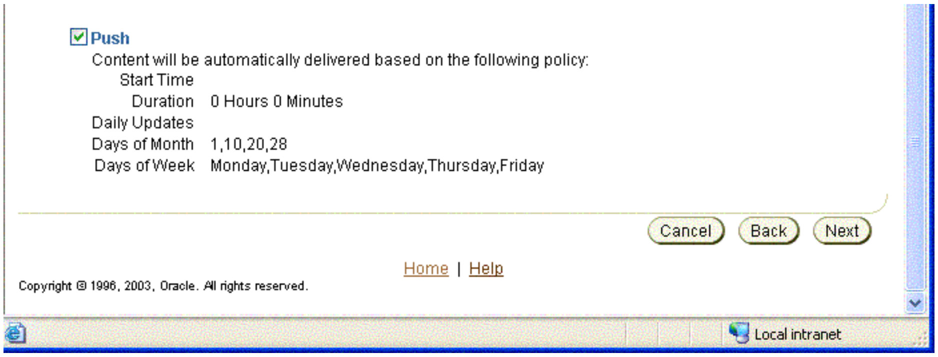
Select Delivery Rules

Delivery rules control how the offer content is delivered to you. The delivery rules available for your selected offer are listed below. Select the delivery rules to be used by the server.

Expiration Policy

Expiration Priority	First among quantity and expiration time determines expiration
Expiration Date	Jul 11, 2003 8:00:00 PM EDT
Quantity Total	3

Figure 10–6 Delivery Rules Page (Bottom Half of Page)



✓ Push

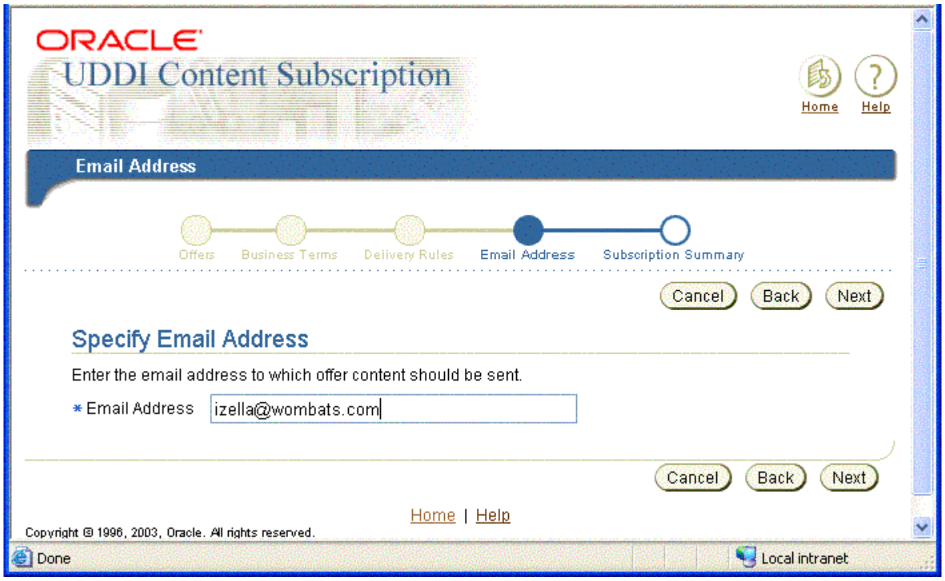
Content will be automatically delivered based on the following policy:

Start Time	
Duration	0 Hours 0 Minutes
Daily Updates	
Days of Month	1,10,20,28
Days of Week	Monday,Tuesday,Wednesday,Thursday,Friday

Copyright © 1996, 2003, Oracle. All rights reserved.

d. At the **Email Address** page as shown in Figure 10–7, enter the e-mail address to whom this offer content is to be sent, then click **Next** to continue to the next step.

Figure 10–7 Email Address Page



Specify Email Address

Enter the email address to which offer content should be sent.

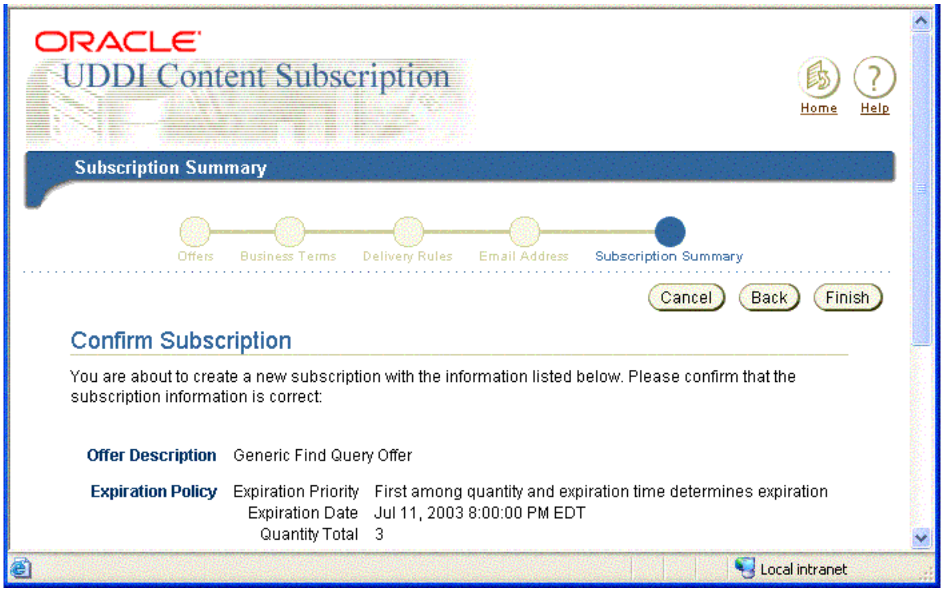
* Email Address: izella@wombats.com

Copyright © 1996, 2003, Oracle. All rights reserved.

e. At the **Subscription Summary** page, as shown in Figure 10–8 and Figure 10–9, review the subscription information. The following information is displayed: offer description, expiration policy, push delivery rules, and e-mail address to where the content is to be pushed. If the information is correct, click **Finish** to complete the subscription process. A confirmation message is shown at the top of the **Subscriptions** page, indicating that your subscription was successfully created.

If the information is not correct, click **Back** to return to the appropriate subscribe wizard page where you can make the necessary change, then click **Next** to return to this Subscription Summary page to review a summary of the subscription information again.

Figure 10–8 Subscription Summary Page (Top Half of Page)

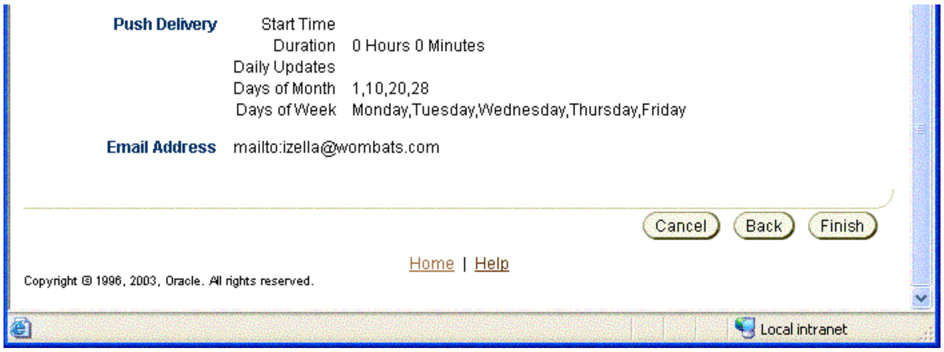


Confirm Subscription

You are about to create a new subscription with the information listed below. Please confirm that the subscription information is correct:

Offer Description	Generic Find Query Offer	
Expiration Policy	Expiration Priority	First among quantity and expiration time determines expiration
	Expiration Date	Jul 11, 2003 8:00:00 PM EDT
	Quantity Total	3

Figure 10–9 Subscription Summary Page (Bottom Half of Page)



Push Delivery	Start Time	
	Duration	0 Hours 0 Minutes
	Daily Updates	
	Days of Month	1,10,20,28
	Days of Week	Monday,Tuesday,Wednesday,Thursday,Friday
Email Address	mailto:izella@wombats.com	

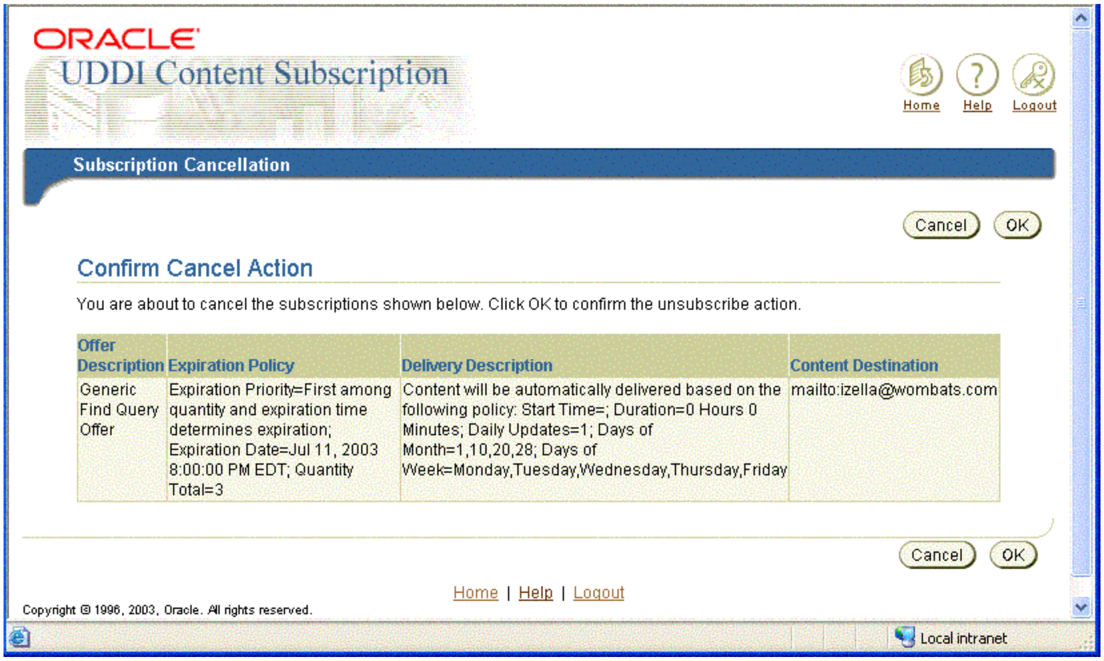
Copyright © 1996, 2003, Oracle. All rights reserved.

This completes the tasks that a UDDI publisher can perform using the UDDI Content Subscription Manager.

Canceling a Subscription

To cancel a subscription, at the **Subscriptions** page as shown in Figure 10–2, select the subscriptions that you want to cancel by clicking their boxes in the **Select** column, then click **Unsubscribe**. A **Subscription Cancellation** page is displayed, as shown in Figure 10–10. Click **OK** to confirm the unsubscribe action.

Figure 10–10 Subscription Cancellation Page



Confirm Cancel Action

You are about to cancel the subscriptions shown below. Click OK to confirm the unsubscribe action.

Offer Description	Expiration Policy	Delivery Description	Content Destination
Generic Find Query Offer	Expiration Priority=First among quantity and expiration time determines expiration; Expiration Date=Jul 11, 2003 8:00:00 PM EDT; Quantity Total=3	Content will be automatically delivered based on the following policy: Start Time=; Duration=0 Hours 0 Minutes; Daily Updates=1; Days of Month=1,10,20,28; Days of Week=Monday,Tuesday,Wednesday,Thursday,Friday	mailto:izella@wombats.com

Using the UDDI Content Subscription Manager as a UDDI Administrator

To use the UDDI Content Subscription Manager as a UDDI administrator, perform the following steps:

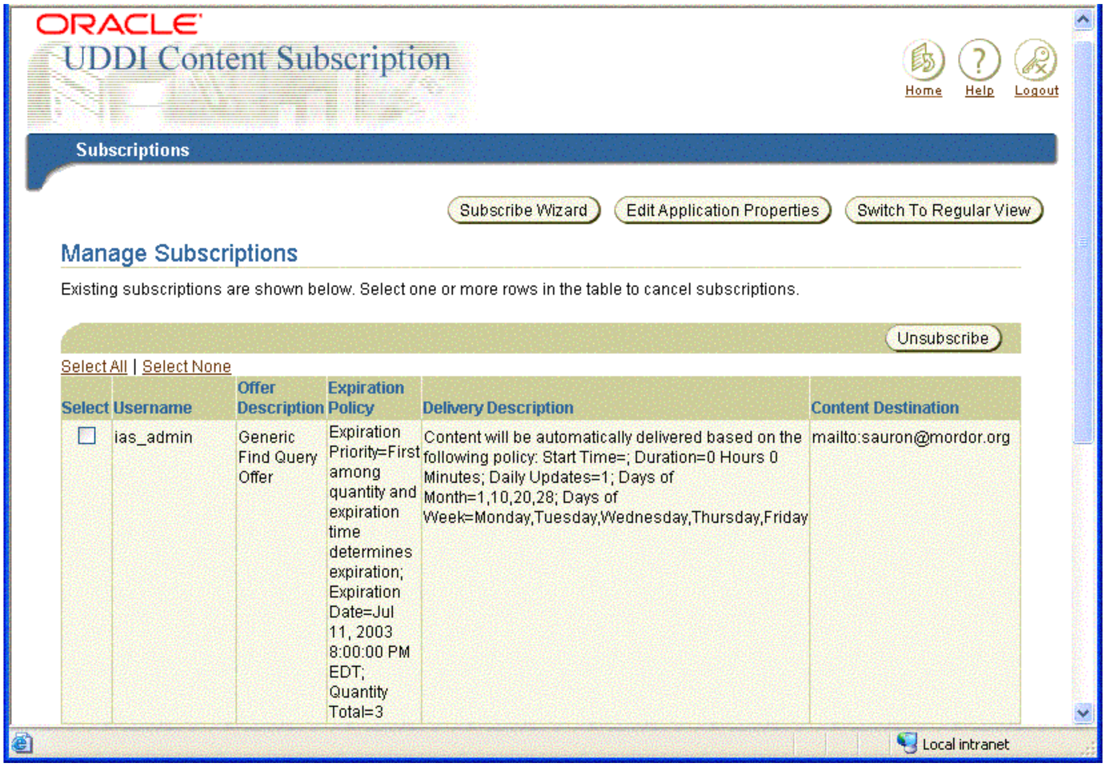
1. Start the UDDI Content Subscription Manager by entering the following URL:

   ```
   http://<host>:<port>/uddisub/subscription/ui
   ```

   where the *<host>* parameter indicates the system on which the UDDI Content Subscription Manager is installed and the *<port>* parameter specifies the port number on which it is running.

2. Next, log in as a UDDI administrator (for example, `ias_admin`/*<ias_admin-password>*). The UDDI Content Subscription Manager home page or **Subscriptions** page is displayed, as shown in Figure 10–11.

Figure 10–11 Subscriptions Page



Manage Subscriptions

Existing subscriptions are shown below. Select one or more rows in the table to cancel subscriptions.

Select	Username	Offer Description	Expiration Policy	Delivery Description	Content Destination
☐	ias_admin	Generic Find Query Offer	Expiration Priority=First among quantity and expiration time determines expiration; Expiration Date=Jul 11, 2003 8:00:00 PM EDT; Quantity Total=3	Content will be automatically delivered based on the following policy: Start Time=; Duration=0 Hours 0 Minutes; Daily Updates=1; Days of Month=1,10,20,28; Days of Week=Monday,Tuesday,Wednesday,Thursday,Friday	mailto:sauron@mordor.org

As the administrator, you can do any of the following tasks:

a. Create a subscription.

   Click **Subscribe Wizard** to launch a 5-step subscribe wizard that lets you select an offer, accept the business terms of the offer, select the delivery rules for delivering content to you, specify the e-mail address to where content is to be delivered, and review a summary of the specified subscription information before you create the subscription.

b. Cancel a subscription.

Select an existing subscription by selecting its corresponding box in the **Select** column, then click **Unsubscribe**.

c. Edit application properties.

Click **Edit Application Properties** to edit the UDDI subscription application properties, such as configuring the UDDI Content Subscription Manager with OracleAS Syndication Services.

d. Switch to regular view.

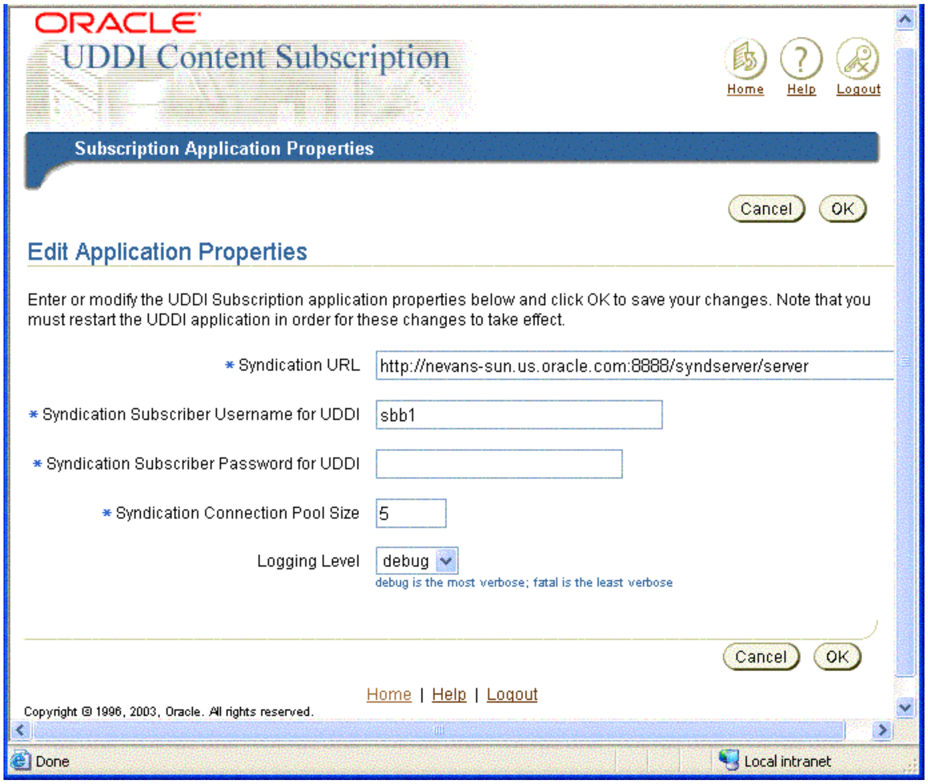
Click **Switch to Regular View** if you have logged in as the administrator and want to just view or manage your own subscriptions as a regular user would.

3. Click **Subscription Application Properties** to view or edit the UDDI subscription application properties, as shown in Figure 10–12. The UDDI content subscription administrator may need to change the default properties only if one or more settings need to be changed.

For example, if the instance of OracleAS Syndication Services is installed on the same system as this UDDI Content Subscription Manager, then the syndication URL should be correct; if OracleAS Syndication Services is not on the same system, then the UDDI Content Subscription administrator must specify the syndication URL. You can edit any properties that may need to be changed. Usually, the default settings will be fine. If no changes are necessary, click **Cancel**; if changes are necessary, make your changes, then click **OK**. Then, restart the UDDI content subscription application in order for these changes to take effect.

Note: You must enter values for *all* fields in order to make changes.

Figure 10–12 Subscription Application Properties Page


ORACLE
UDDI Content Subscription
Home
Help
Logout
Subscription Application Properties
Cancel
OK

Edit Application Properties

Enter or modify the UDDI Subscription application properties below and click OK to save your changes. Note that you must restart the UDDI application in order for these changes to take effect.

* Syndication URL	http://nevans-sun.us.oracle.com:8888/syndserver/server
* Syndication Subscriber Username for UDDI	sbb1
* Syndication Subscriber Password for UDDI	
* Syndication Connection Pool Size	5
Logging Level	debug

debug is the most verbose; fatal is the least verbose

Copyright © 1996, 2003, Oracle. All rights reserved.

Note: If you change any subscription application properties, you must restart the UDDI application in order for these changes to take effect.

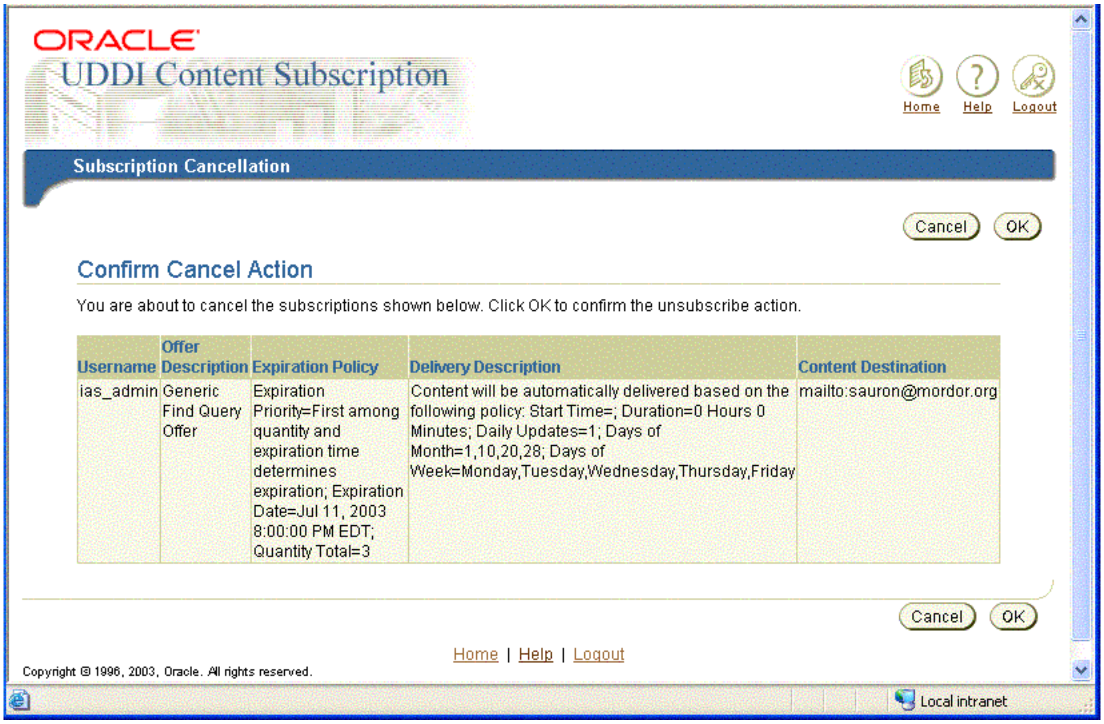
Having checked the subscription application properties, regular users and administrators can now begin subscribing to offers and managing subscriptions.

4. You can subscribe to an offer just like a regular publisher by following the procedure described beginning at Step 3 in Using the UDDI Content Subscription Manager as a Publisher on page 10-84.

Canceling a Subscription

To cancel one of your own subscriptions or a subscription belonging to any user, at the **Subscriptions** page as shown in Figure 10–11, select the subscriptions from that you want to cancel by clicking their boxes in the **Select** column, then click **Unsubscribe**. A **Subscription Cancellation** page is displayed, as shown in Figure 10–13. Click **OK** to confirm the unsubscribe action.

Figure 10–13 Subscription Cancellation Page



Confirm Cancel Action

You are about to cancel the subscriptions shown below. Click OK to confirm the unsubscribe action.

Username	Offer Description	Expiration Policy	Delivery Description	Content Destination
ias_admin	Generic Find Query Offer	Expiration Priority=First among quantity and expiration time determines expiration; Expiration Date=Jul 11, 2003 8:00:00 PM EDT; Quantity Total=3	Content will be automatically delivered based on the following policy: Start Time=; Duration=0 Hours 0 Minutes; Daily Updates=1; Days of Month=1,10,20,28; Days of Week=Monday,Tuesday,Wednesday,Thursday,Friday	mailto:sauron@mordor.org

11 Consuming Web Services in J2EE Applications

This chapter describes how to consume Web Services in Java 2 Platform, Enterprise Edition (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 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

Oracle Application Server Containers for J2EE (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 Oracle Application Server.

Sample Use Scenarios

This section describes two sample use cases for a better understanding of how to use the Web Service HTML/XML Stream Processing Wizard.

Handling an XML or HTML Stream Accessed Through a Static URL

The following steps generate the Java stubs that consume a static XML or HTML stream.

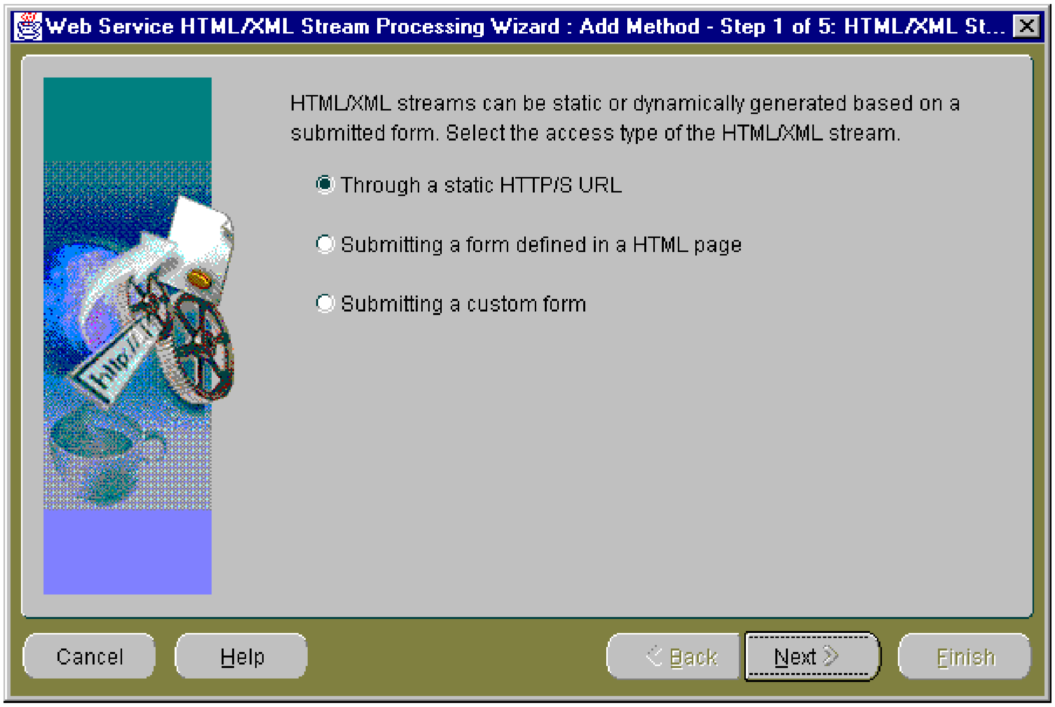
1. Invoke the Web Service HTML/XML Stream Processing Wizard using the following command:

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

Note: The `WebServicesHtmlXmlWizard.jar` file is located in your *`$ORACLE_HOME`*`/webservices/lib` installation directory for UNIX or *`%ORACLE_HOME%`*`\webservices\lib` installation directory for Windows.

---

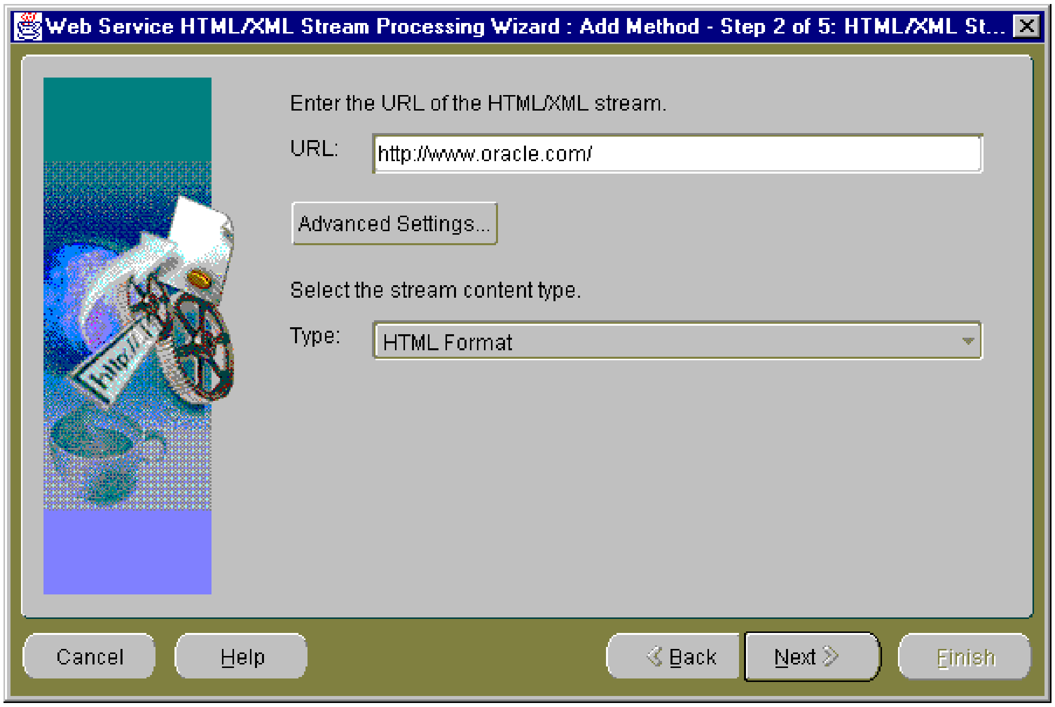
2. In **Step 1 of 5: HTML/XML Stream Type**, select the first option **Through a static HTTP/S URL**, then click **Next** to continue to the next step.


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.

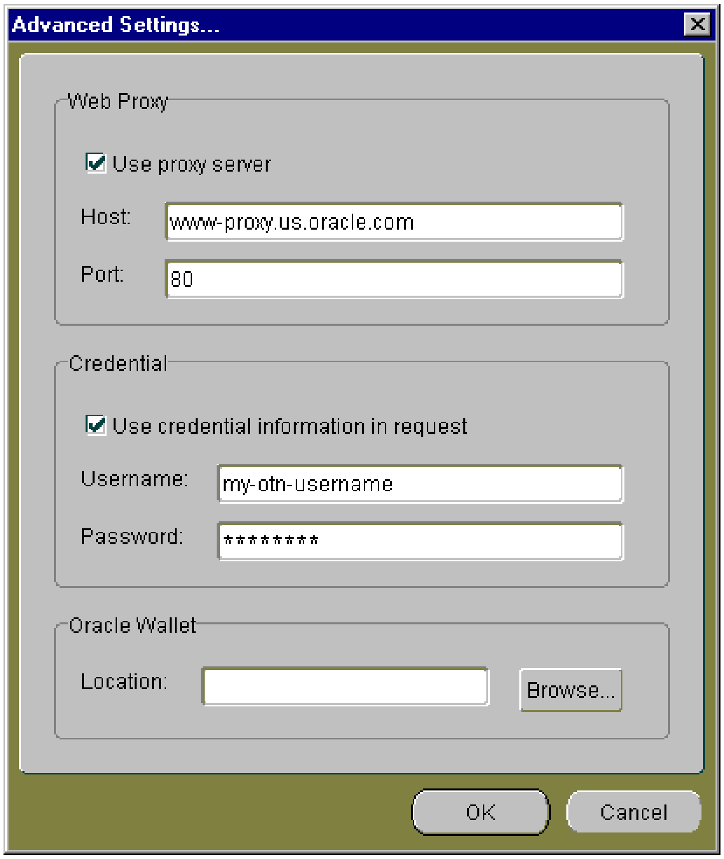
URL: http://www.oracle.com/

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.

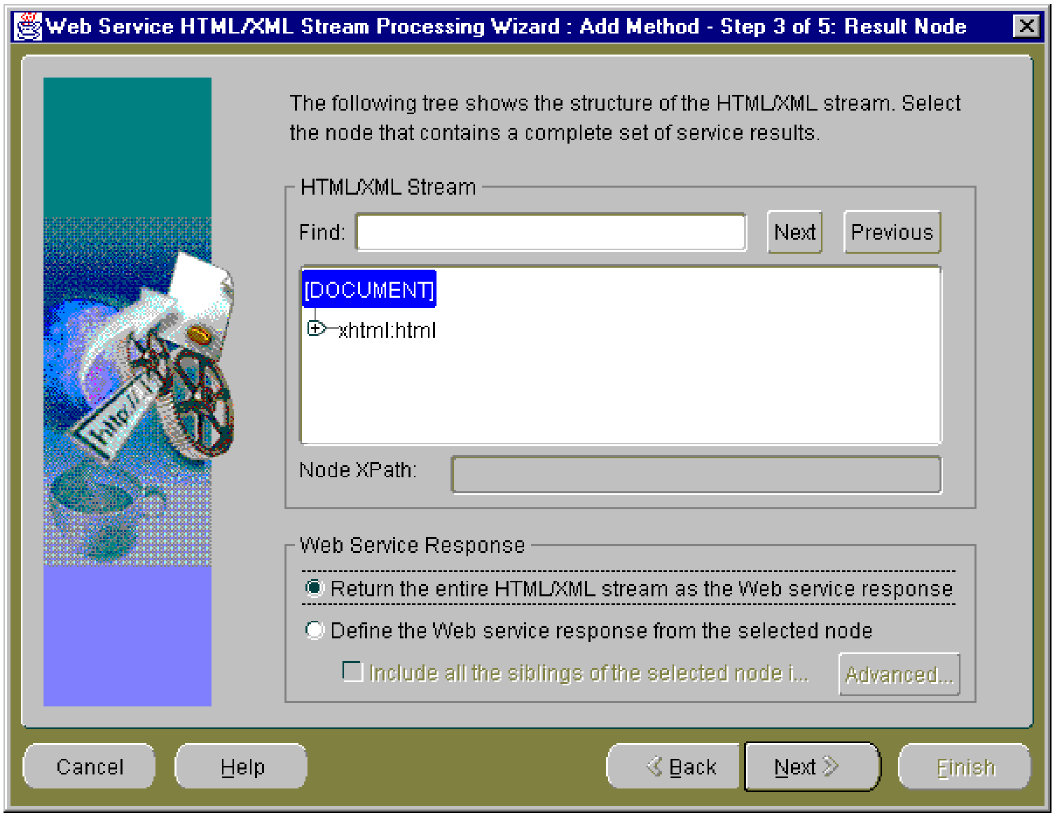
If the URL you are accessing requires basic HTTPS authentication, use the **Oracle Wallet** section of the **Advanced Settings** pop-up window to enter the Wallet location.

---

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 8 for more information about the **Service Response Fields** window.

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

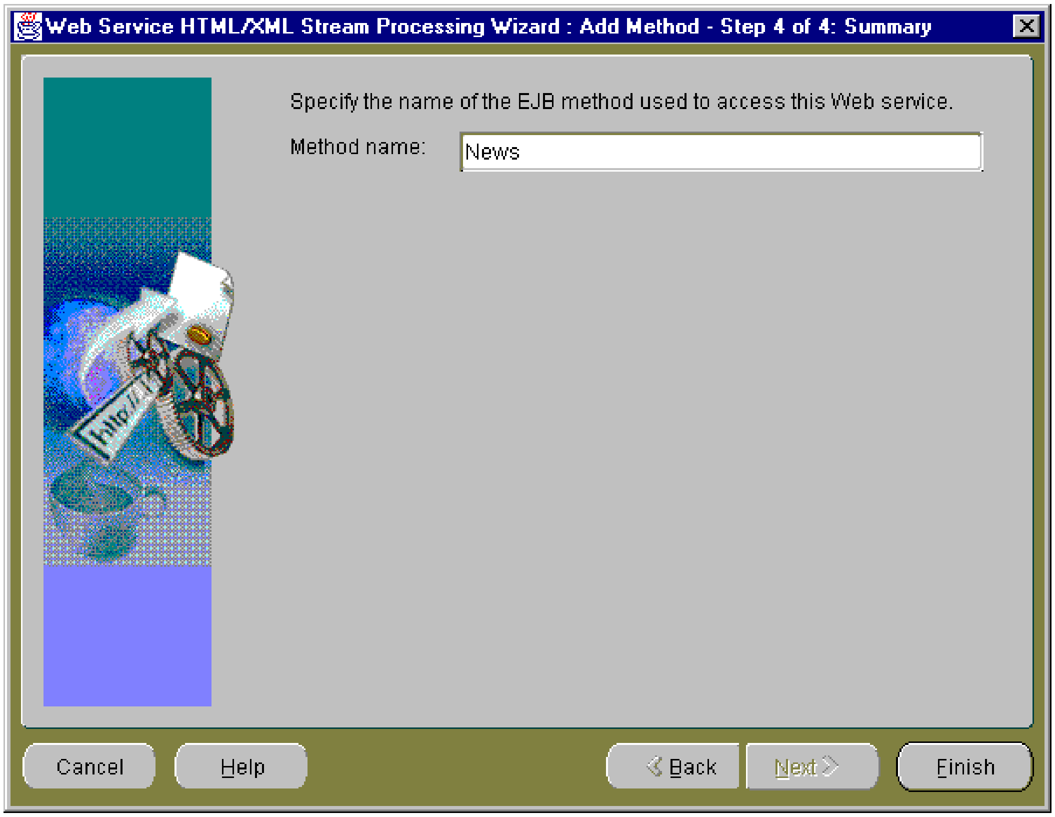
6. In **Step 4 of 4: Summary**, you must specify your EJB method name.

   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.

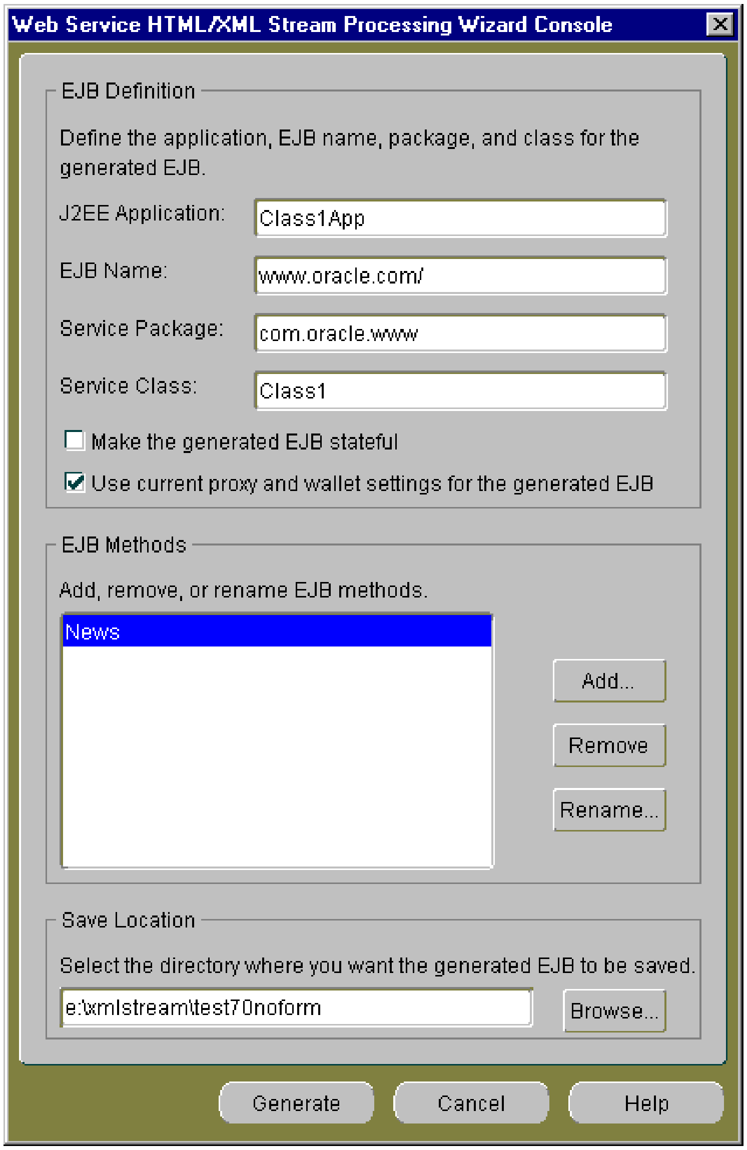
Method name: News

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.


Web Service HTML/XML Stream Processing Wizard Console

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 Oracle Application Server) is located within the parent directory you specified in Step 7. The .ear file contains the generated EJB, JAR, and XML files for your application, where the `application.xml` file located in the `/deploy/META-INF` directory for UNIX or the `\deploy\META-INF` directory for Windows serves as the EAR manifest file.

- A JAR file, containing your EJB application class files is located within the `/deploy` directory for UNIX or the `\deploy` directory for Windows. The JAR file includes all EJB application class files and the deployment descriptor file.
- A standard J2EE EJB deployment descriptor (`ejb-jar.xml`), for all the beans in the module, is located within the `/classes/META-INF` directory for UNIX or the `\classes\META-INF` directory for Windows. The XML deployment descriptor describes the application components and provides additional information to enable the container to manage the application.
- The source code of a set of Java classes that you can use in your Java applications is located within the `/src` directory for UNIX or the `\src` directory for Windows. The generated JavaBean and EJB Java source code is contained in subdirectories according to their Java package names.
- The `/classes` directory for UNIX or the `\classes` directory for Windows contains the compiled generated classes and additional XML resources used by the generated code.

The following code is generated in the `src/com/oracle/www/Class1.java` file on UNIX or the `src\com\oracle\www\Class1.java` file on Windows showing the remote interface (Class1) of the generated EJB. In this case, a method (news) with no parameters that return an org.wc3.dom.Element is generated because the HTML stream was selected as a static HTML page.

```
public interface Class1 extends EJBObject
{
  public org.w3c.dom.Element news()
    throws RemoteException;
}
```

Handling an XML or HTML Stream Accessed Through a Form

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

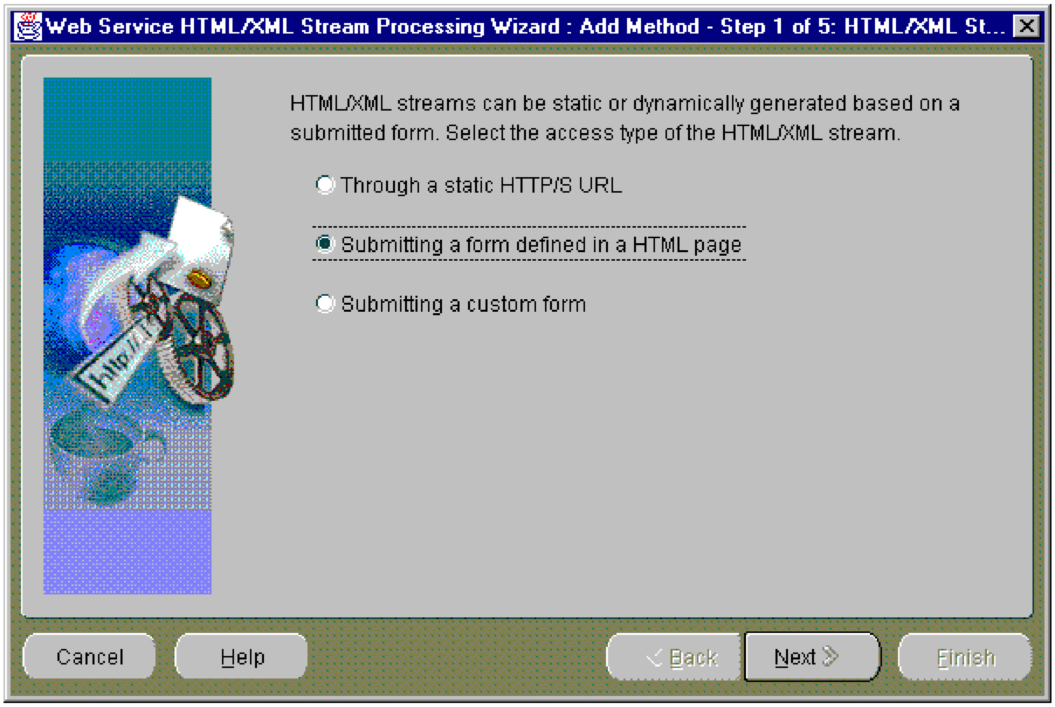
1. Invoke the Web Service HTML/XML Stream Processing Wizard using the following command:

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

Note: The WebServicesHtmlXmlWizard.jar file is located in your *$ORACLE_HOME*`/webservices/lib` installation directory for UNIX or *%ORACLE_HOME*`\webservices\lib` installation directory for Windows.

---

2. In **Step 1 of 5: HTML/XML Stream Type**, select the second option, **Submitting a form defined in an HTML page**, then click **Next** to continue to the next step.


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.

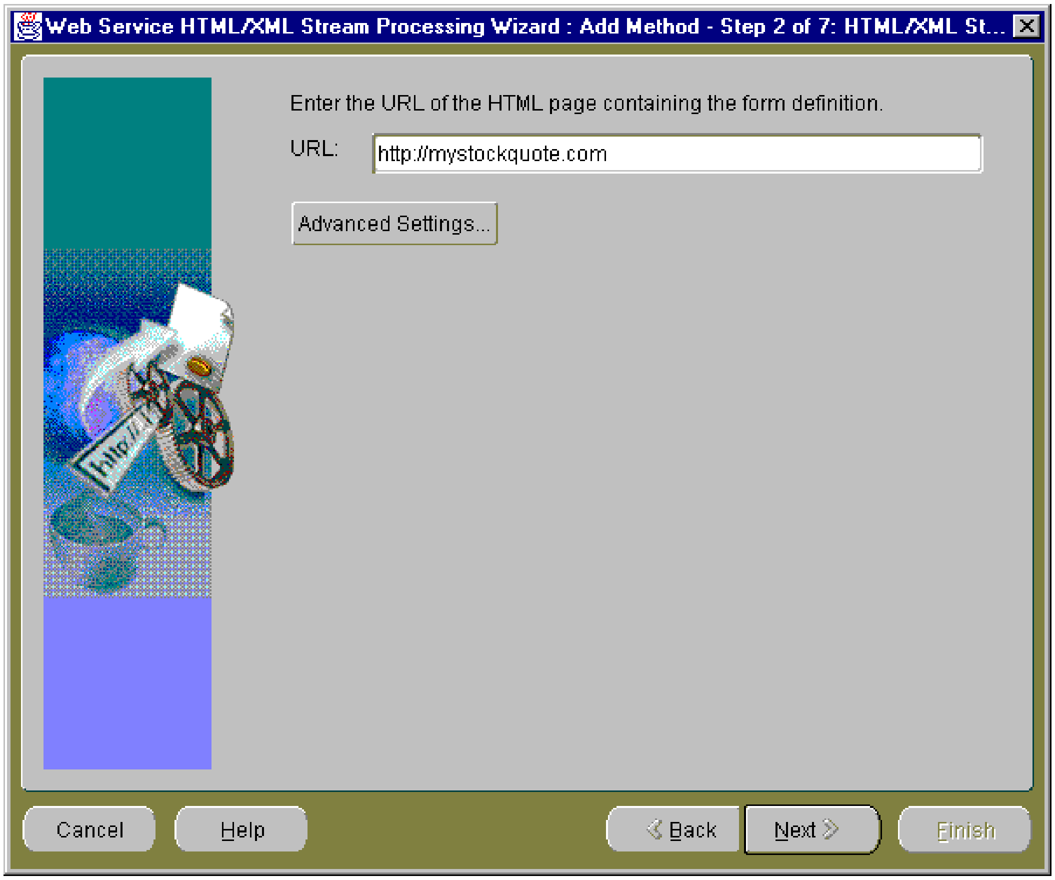
( ) Through a static HTTP/S URL

(●) Submitting a form defined in a HTML page

( ) Submitting a custom form

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...
http://
Cancel
Help
< Back
Next >
Finish

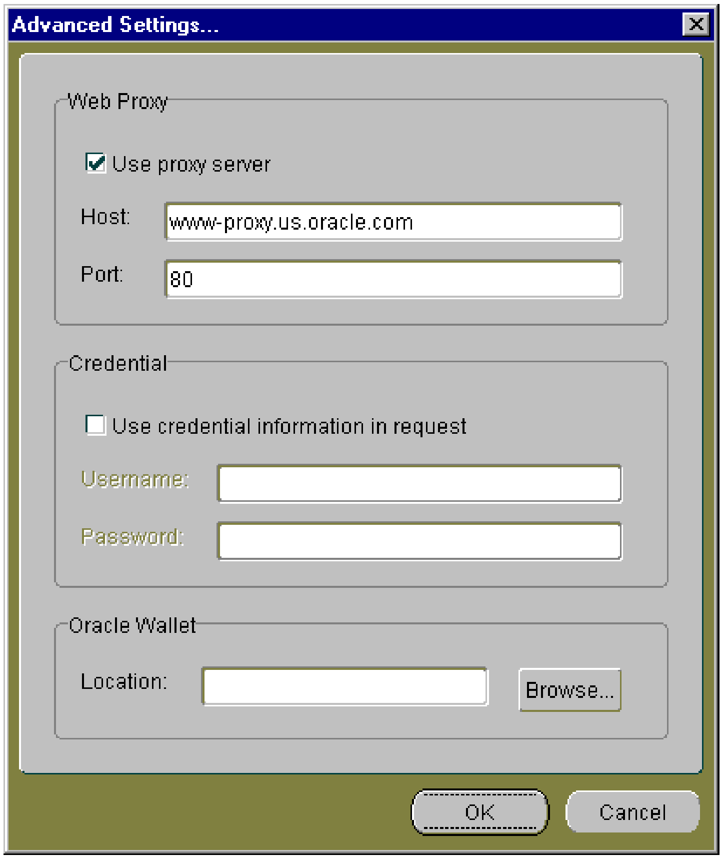
Enter the URL of the HTML page containing the form definition.

URL: http://mystockquote.com

Advanced Settings...

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

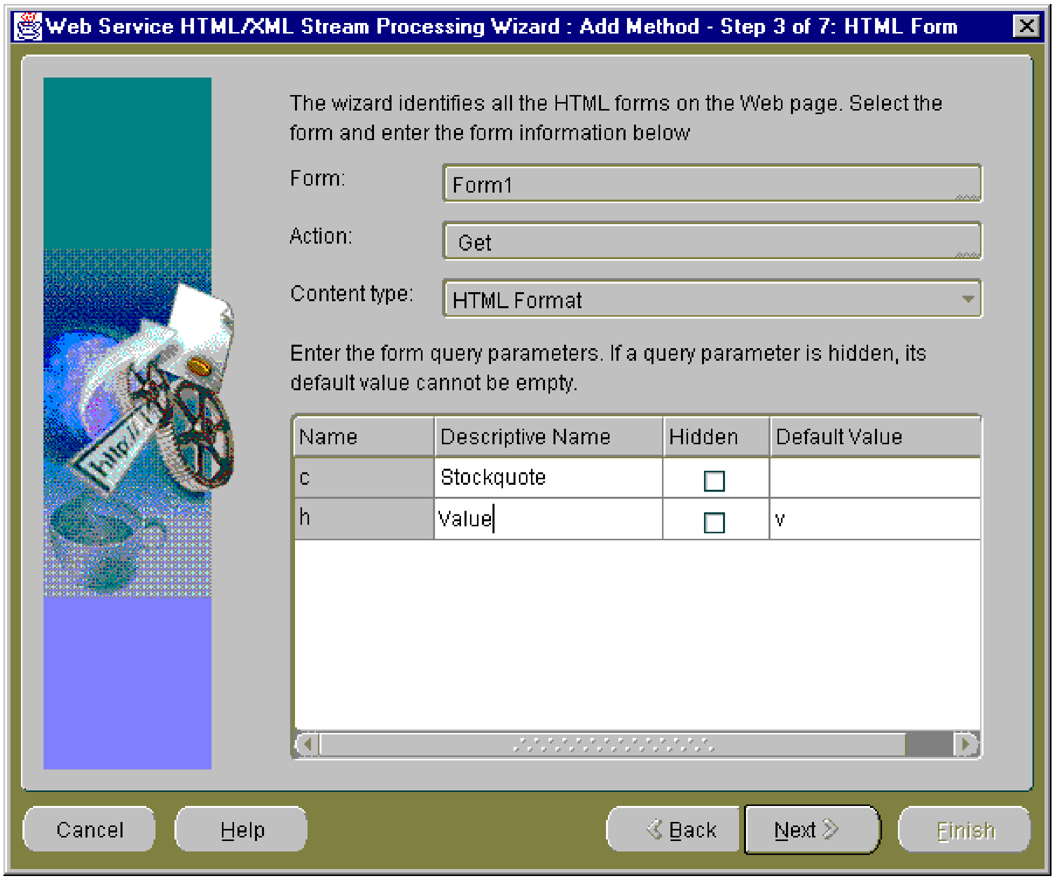
Location:

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

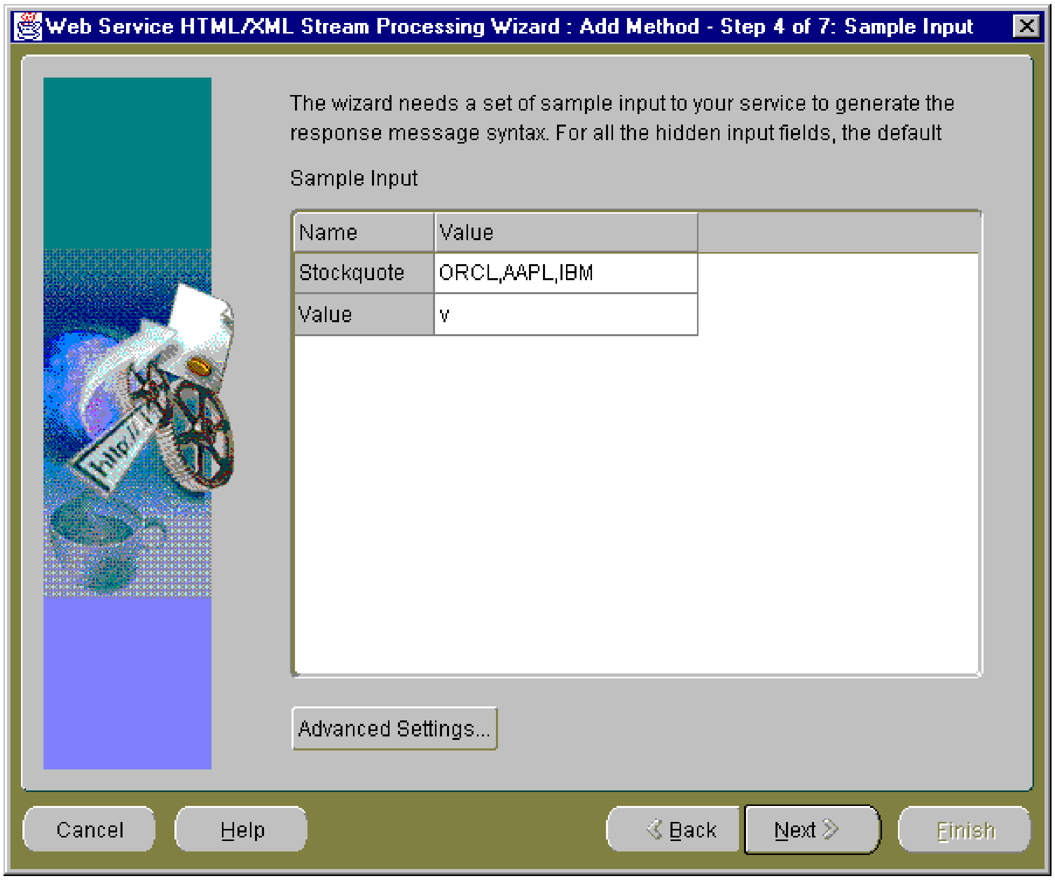
Enter the form query parameters. If a query parameter is hidden, its default value cannot be empty.

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

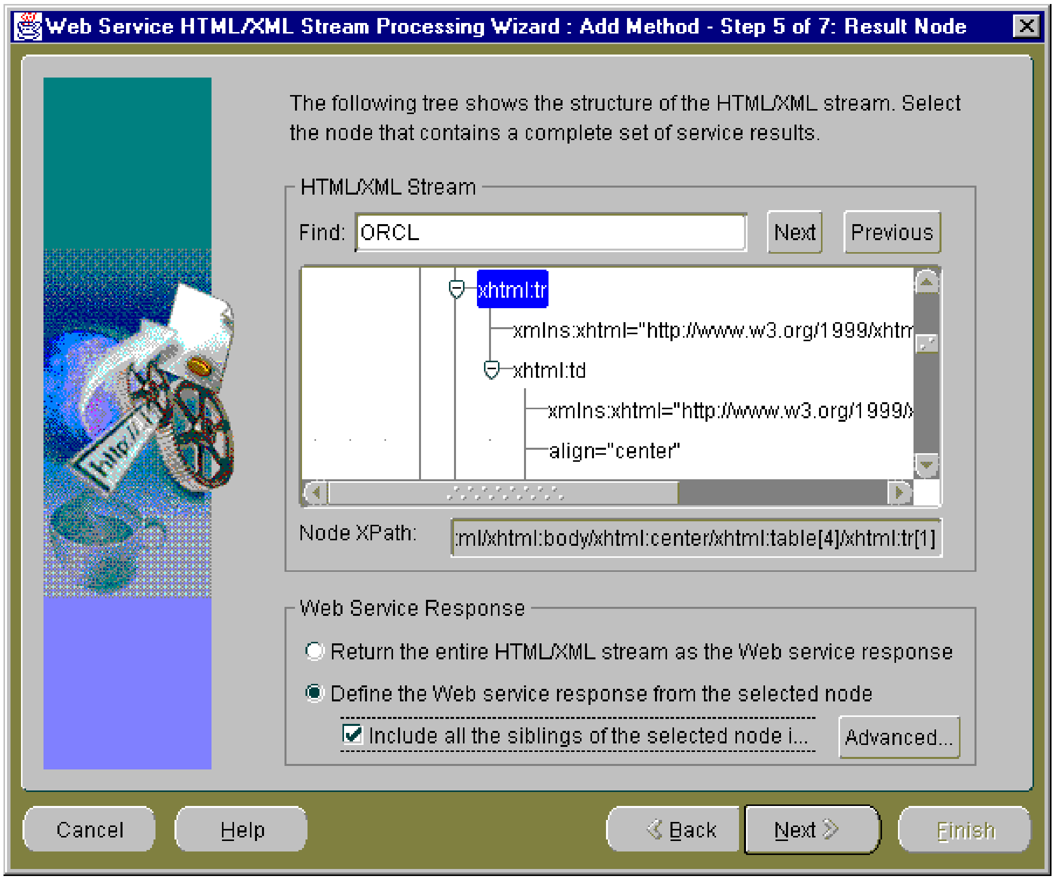
Name	Value
Stockquote	ORCL,AAPL,IBM
Value	v

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/
    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.

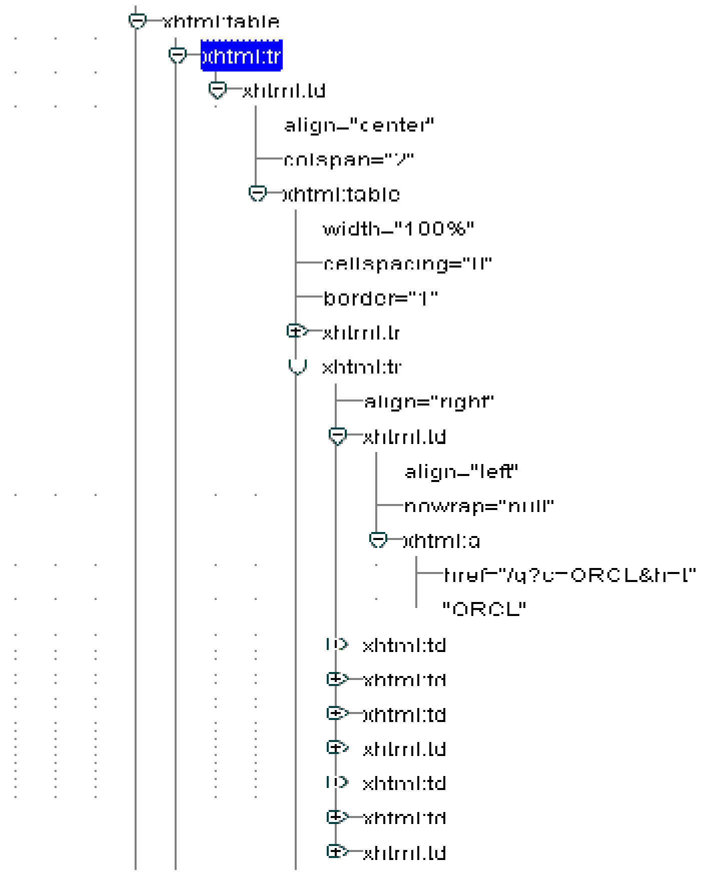
If you want to select just the node that contains the complete set of information you are interested in, select the second option **Define the Web service response from the selected node**. Then, navigate to the node you want by moving down the HTML / XML stream tree.

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 11–1, Figure 11–2, and Figure 11–3 together show an excerpt of what the xhtml tree would appear like when expanded. The selected node `xhtml:tr` is located in the next lower table node, which is one level lower than the `xhtml:tr` nodes for ORCL and its two siblings AAPL and IBM.

Figure 11–1 Expanded xhtml Tree Showing the Selected Node of Interest Relative to the Nodes for ORCL and Sibling Nodes AAPL and IBM (Part1)



```
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
```

Figure 11–2 Expanded xhtml Tree Showing the Selected Node of Interest Relative to the Nodes for ORCL and Sibling Nodes AAPL and IBM (Part2)

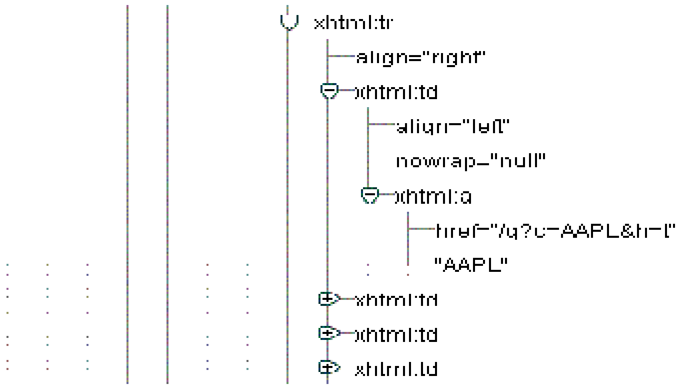
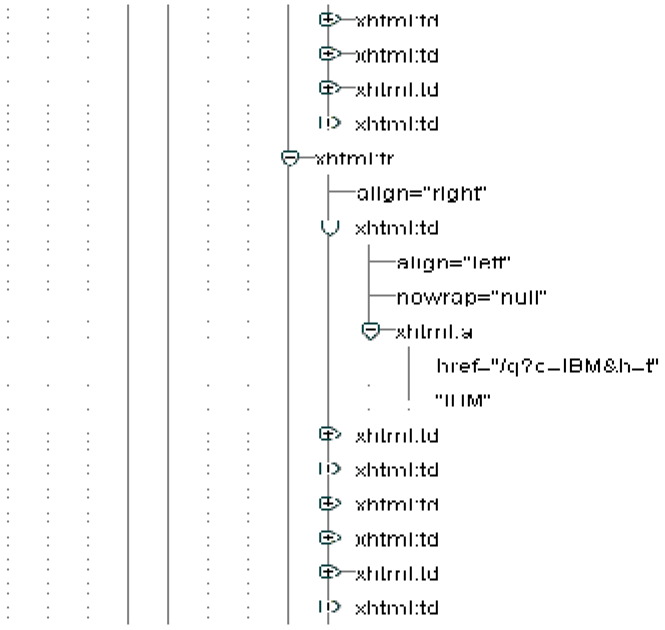


Figure 11–3 Expanded xhtml Tree Showing the Selected Node of Interest Relative to the Nodes for ORCL and Sibling Nodes AAPL and IBM (Part 3)



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.

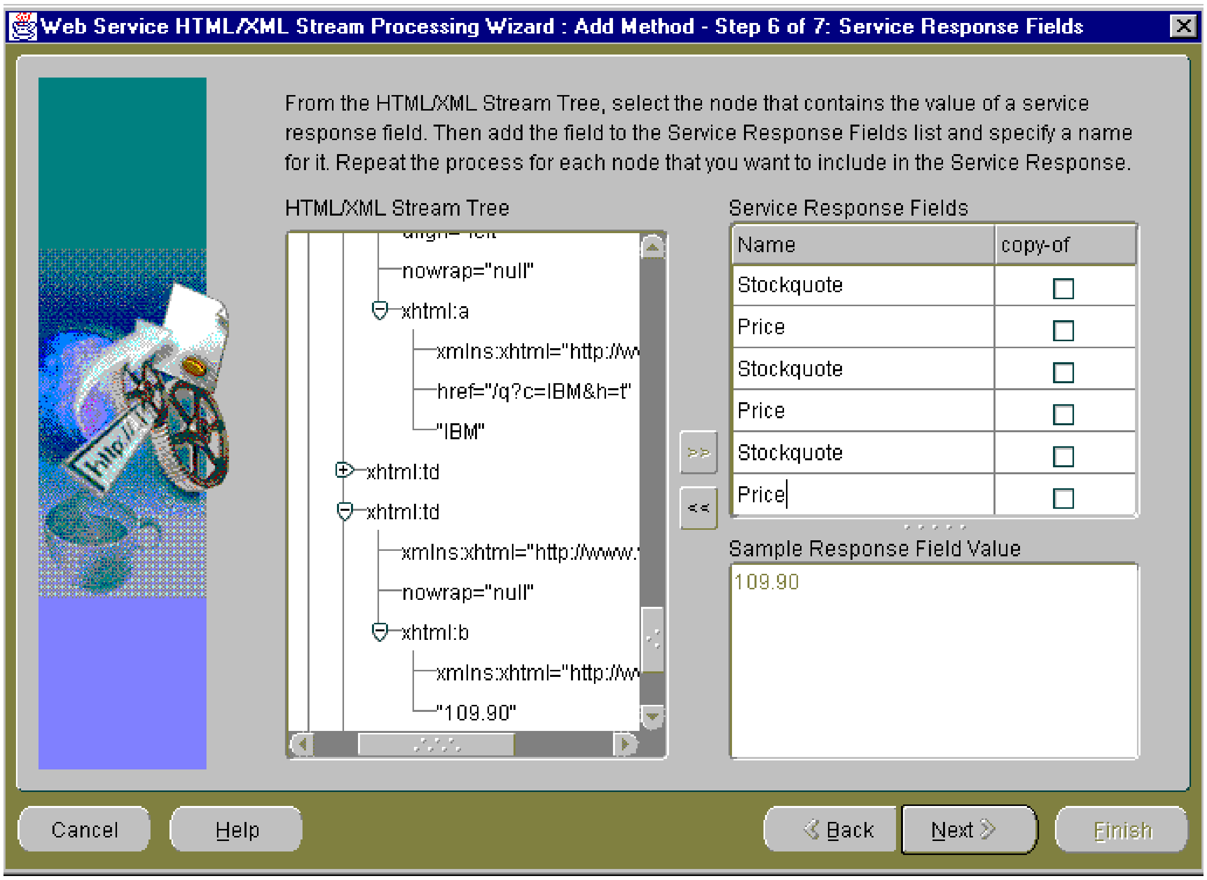
The following predicate filters out the first position: position() != 1. Enter this predicate expression in the **Predicate that filters the set of sibling nodes** field of the **Advanced Settings** pop-up window to filter the first sibling from the Web Service response.

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.

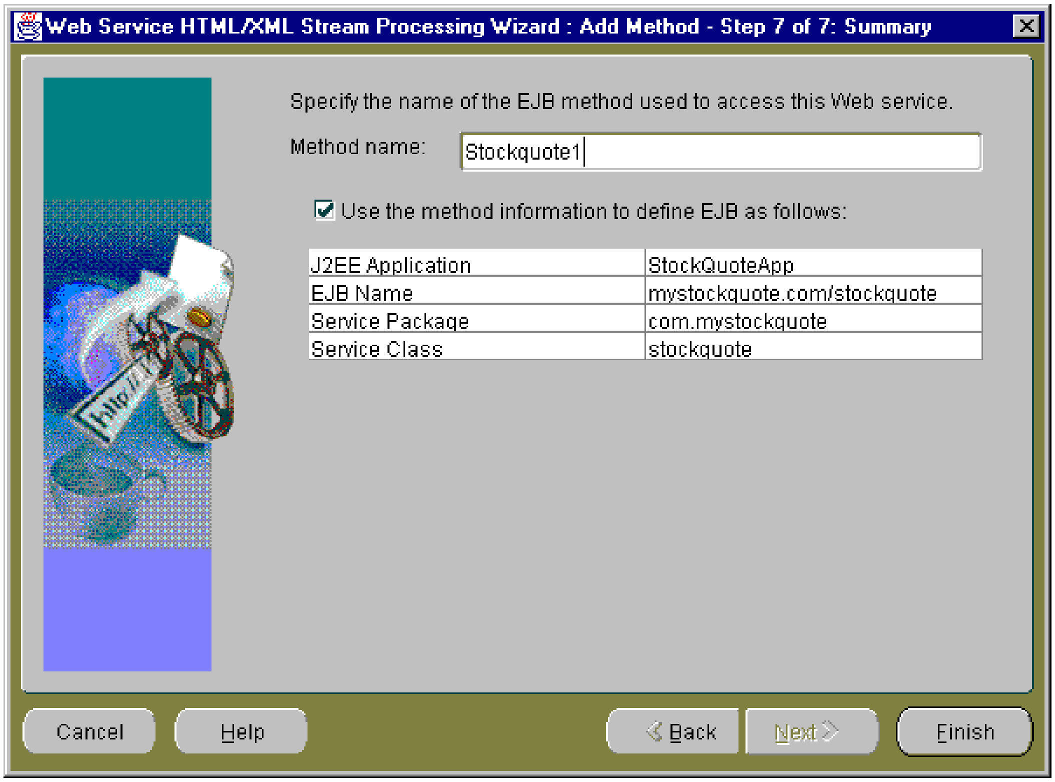
9. In **Step 7 of 7: Summary**, you must specify your EJB method name.

   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:

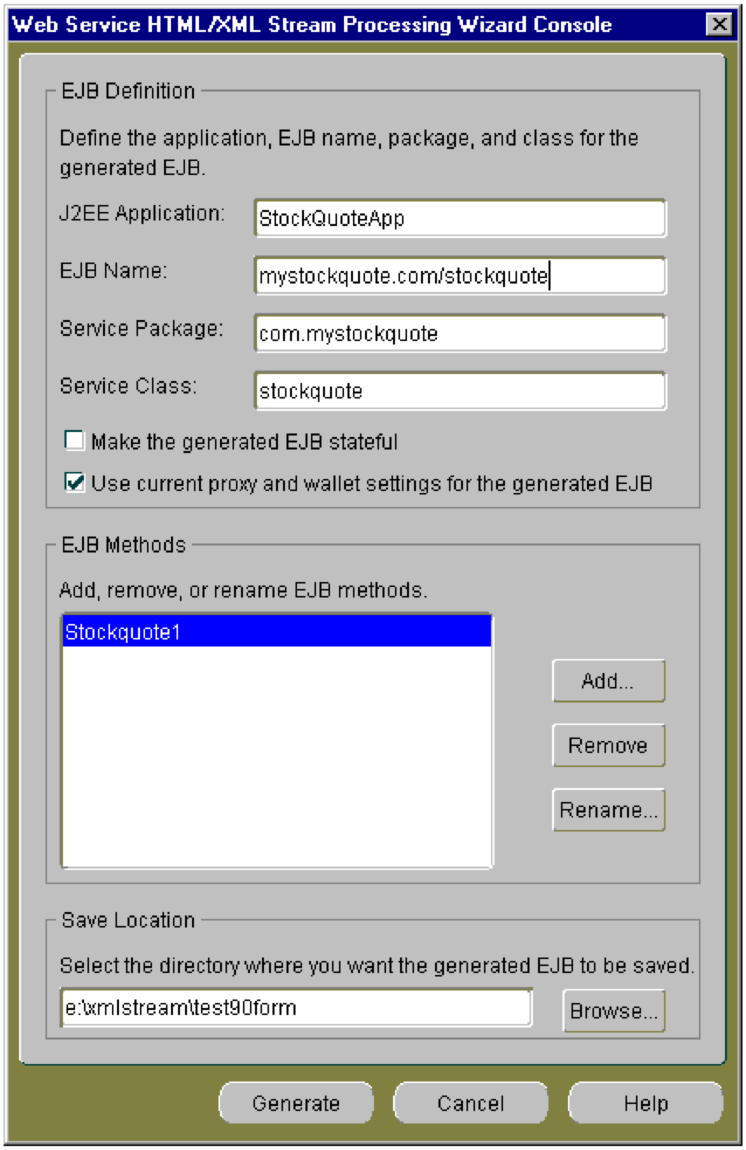
J2EE Application	StockQuoteApp
EJB Name	mystockquote.com/stockquote
Service Package	com.mystockquote
Service Class	stockquote

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 Oracle Application Server) 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 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 SOAP Mapping Registry is used to map XML types to the corresponding Java types.

Additional references regarding WSDL and SOAP can be found in the following locations:

- The WSDL 1.1 specification is available at

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

- The SOAP 1.1 specification is available at

  `http://www.w3.org/TR/SOAP/`

The command-line options for running the `wsdl2ejb` utility are described in Table 11–1.

Table 11–1 wsdl2ejb Utility Command-Line Options

Option	Description
-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`*`/webservices/lib` installation directory for UNIX or *`%ORACLE_HOME`*`\webservices\lib` installation directory for Windows.

If an HTTP URL is used to supply the location of the WSDL document and an HTTP proxy is required to access it, the following command and syntax must be used to run the utility:

```
java -Dhttp.ProxyHost=myProxyHost -Dhttp.proxyPort=80 -jar wsdl2ejb.jar -d
<destDir> http://myhost/mydoc.wsdl
```

In this example, the utility uses the supplied WSDL to generate the EJB EAR file in the destination directory (<*`destDir`*>). The EJB class name, Java Naming and Directory Interface (JNDI) binding key, and Java package name are derived from the location of the SOAP service described in the WSDL.

In this command syntax, the `wsdl2ejb` utility maps the XML types, which are supported by default by the Oracle SOAP Mapping Registry.

The `wsdl2ejb` utility generates the following sets of files located within the destination directory name (<*`destDir`*>) that you specify in the command line. The utility saves the generated files using the following directory layout:

```
 Root /
      + app.ear
      + src/
        + ... generated java sources ...
      + classes/
        + META-INF/
          + ejb-jar.xml
        + ... compiled classes and xml resources ....
      + deploy/
        + ejb.jar
        + META-INF/
          + application.xml
```

- An .ear file (which is a JAR archive containing the J2EE application that can be deployed in OC4J) 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 WSDL SOAP location will be managed separately from the
    WSDL document.
 -->
<!ELEMENT wsdl (location, ((service-name, service-port) | (service-binding, soap-location))?)>

<!-- Specify the location of the source WSDL document (for example, "/home/mywsdl.wsdl",
"http://myhost/mywsdl.wsdl") -->
<!ELEMENT location (#PCDATA)>


<!-- Specify the name of the WSDL service to be used for the generation.
     It is the name of one of the services defined in the source WSDL. -->
<!ELEMENT service-name (#PCDATA)>


<!-- Specify the service port of the WSDL service to be used for the generation.
     It is the name of one ports of the service name defined above in the source WSDL. -->
<!ELEMENT service-port (#PCDATA)>


<!-- Specify the name of the WSDL binding to be used for the generation.
     It is the name of one of the bindings defined in the source WSDL. -->
<!ELEMENT service-binding (#PCDATA)>

<!-- Specify the SOAP location service port of the WSDL service to be used for the generation.
     It is the name of one ports of the service name defined above in the source WSDL. -->
<!ELEMENT soap-location (#PCDATA)>

<!-- Specify the properties related to the generated EJB. -->
<!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 definition 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 11–2 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 11–2 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 11–2 (Cont.) Elements, Subelements, and Attributes of the wsdl2ejb XML Configuration File as Defined in the DTD

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 11–3 shows how the supported XML Schema types are mapped to the corresponding Java type by default. Any other required type will have to be supported though the custom type mapping described previously.

Table 11–3 Supported XML Schema Types and Corresponding Java Type

Supported XML Schema Type	Corresponding Java Type
string	java.lang.String
int	int
decimal	BigDecimal
float	float
double	double
Boolean	Boolean
long	long
short	short
byte	byte
date	GregorianCalendar
timeInstant	java.util.Date

Note: Arrays of supported types, shown in Table 11–3 are also supported.

Known Limitations of the `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 $ORACLE_HOME/webservices/`demo/basic/wsdl2ejb` directory. The demonstration (demo) will use some sample WSDL documents as sources and generate EJB that can be used to invoke the Web Service operations.

The demos can be run using Jakarta ant. Review the `build.xml` file to make sure that the initial properties (RMI_HOST, RMI_PORT, RMI_ADMIN, RMI_PWD) are set correctly according to your configuration. The `build.xml` file will execute the `wsdl2ejb` utility on the demo WSDL documents, deploy the generated EJB, and execute the EJB clients.

Note: IIf you are executing the demos behind a firewall and need to set proxy information to access external HTTP sites, make sure this proxy information is specified in the `wsdl2ejb` configuration files (rpc_doc_conf.xml, base_conf.xml).

Note: The demos are based on WSDL/SOAP interoperability test suites. They access live SOAP services available on the Internet as SOAP interoperability test cases. The successful execution of these demos depends on the availability of these services.

The directory structure of the demos is as follows:

```
demo/web_services/wsdl2ejb:
   - README.txt                : Readme file
   - build.xml                 : Jakarta ant build file to run all the demos
   - rpc_doc                  : directory for simple RPC and document style operations
       - rpc_doc_conf.xml      : wsdl2ejb configuration file for the rpc_doc demo
       - TestRpcDocClient.java : client for the rpc_doc demo
       - DocAndRpc.wsdl        : sample WSDL for the rpc_doc demo
       - (generated)           : directory where the EJB will be generated
   - base
       - base_conf.xml         : wsdl2ejb configuration file for the base interoperability demo
       - TestInteropBaseClient.java : client for the base interoperability demo
       - InteropTest.wsdl           : WSDL document for the base interoperability demo
       - MySoapStructBean.java      : bean utilized to map the custom type used
                                        in the example defined in the WSDL document
       - MySoapStructBean.jar       : packaged-compiled custom type bean
       - (generated)                : directory where the EJB will be generated
```

RPC and Document Style with Simple Types Example

This example uses a simple WSDL document that shows a couple of operations: Add and Multiply. Add is using the document-style operation using literal parts, while Multiply is RPC-style and uses encoded parts.

To generate the EJB stub, use the following command:

```
On UNIX
cd $ORACLE_HOME/webservices/demo/basic/wsdl2ejb
java -jar ../../../lib/wsdl2ejb.jar -conf rpc_doc/rpc_doc_conf.xml

On Windows
cd %ORACLE_HOME%\webservices\demo\basic\wsdl2ejb
java -jar ..\..\..\lib\wsdl2ejb.jar -conf rpc_doc\rpc_doc_conf.xml
```

The utility generates the `TestApp.ear` file containing the definition of a stateless EJB, which can be used as a proxy for the Web Service. The EAR file can be deployed in OC4J as any standard EJB. Refer to *Oracle Application Server 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 follow to access the EJB

      //
      // Add test
      //
      Document  doc = new XMLDocument();
      Element elAdd = doc.createElementNS("http://soapinterop.org", "s:Add");
      Element   elA = doc.createElementNS("http://soapinterop.org", "s:a");
      Element   elB = doc.createElementNS("http://soapinterop.org", "s:b");
      elA.appendChild(doc.createTextNode("4"));
      elB.appendChild(doc.createTextNode("3"));
      elAdd.appendChild(elA);
      elAdd.appendChild(elB);
      doc.appendChild(elAdd);

      Element elAddResponse = service.add(elAdd);
      Node tNode = elAddResponse.getFirstChild().getFirstChild();
      System.out.println("AddResponse: "+tNode.getNodeValue());

      //
      // Multiply Test
      //
      int a = 4;
      int b = 3;
      int iMultiplyResponse = service.multiply(a, b);
      System.out.println("MultiplyResponse: "+iMultiplyResponse);


    }
    catch (Throwable ex) {
      ex.printStackTrace();
    }
  }
}
```

The result of the execution of the client is the following:

```
AddResponse: 7
MultiplyResponse: 12
```

Round 2 Interop Services: Base Test Suite Example

This example starts from a subset of the WSDL document defined by the base test suite of the second round of SOAP interoperability tests. The purpose of this demo example is to show the usage of built-in types in the SOAP Mapping Registry as well as how to add custom types mapping.

Start by looking at the WSDL portType in the `InteropTest.wsdl` file.

```
<types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://soapinterop.org/xsd">
      <complexType name="ArrayOfstring">
         <complexContent>
            <restriction base="SOAP-ENC:Array">
               <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="string[]"/>
            </restriction>
         </complexContent>
       </complexType>
       <complexType name="ArrayOfint">
          <complexContent>
            <restriction base="SOAP-ENC:Array">
               <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="int[]"/>
            </restriction>
          </complexContent>
       </complexType>
       <complexType name="ArrayOffloat">
          <complexContent>
             <restriction base="SOAP-ENC:Array">
                <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="float[]"/>
             </restriction>
          </complexContent>
       </complexType>
       <complexType name="ArrayOfSOAPStruct">
          <complexContent>
             <restriction base="SOAP-ENC:Array">
                <attribute ref="SOAP-ENC:arrayType"
wsdl:arrayType="s:SOAPStruct[]"/>
             </restriction>
         </complexContent>
      </complexType>
      <complexType name="SOAPStruct">
            <all>
                <element name="varString" type="string"/>
                <element name="varInt" type="int"/>
                <element name="varFloat" type="float"/>
            </all>
        </complexType>
    </schema>
</types>

<message name="echoStringRequest">
   <part name="inputString" type="xsd:string"/>
</message>
<message name="echoStringResponse">
   <part name="return" type="xsd:string"/>
</message>
<message name="echoStringArrayRequest">
   <part name="inputStringArray" type="s:ArrayOfstring"/>
</message>
<message name="echoStringArrayResponse">
   <part name="return" type="s:ArrayOfstring"/>
</message>
<message name="echoIntegerRequest">
   <part name="inputInteger" type="xsd:int"/>
</message>
<message name="echoIntegerResponse">
   <part name="return" type="xsd:int"/>
</message>
<message name="echoIntegerArrayRequest">
   <part name="inputIntegerArray" type="s:ArrayOfint"/>
</message>
<message name="echoIntegerArrayResponse">
   <part name="return" type="s:ArrayOfint"/>
</message>
<message name="echoFloatRequest">
   <part name="inputFloat" type="xsd:float"/>
</message>
<message name="echoFloatResponse">
   <part name="return" type="xsd:float"/>
</message>
<message name="echoFloatArrayRequest">
   <part name="inputFloatArray" type="s:ArrayOffloat"/>
</message>
<message name="echoFloatArrayResponse">
   <part name="return" type="s:ArrayOffloat"/>
</message>
<message name="echoStructRequest">
   <part name="inputStruct" type="s:SOAPStruct"/>
</message>
<message name="echoStructResponse">
```

```
      <part name="return" type="s:SOAPStruct"/>
   </message>
   <message name="echoStructArrayRequest">
      <part name="inputStructArray" type="s:ArrayOfSOAPStruct"/>
   </message>
   <message name="echoStructArrayResponse">
      <part name="return" type="s:ArrayOfSOAPStruct"/>
   </message>
   <message name="echoVoidRequest"/>
   <message name="echoVoidResponse"/>
   <message name="echoBase64Request">
      <part name="inputBase64" type="xsd:base64Binary"/>
   </message>
   <message name="echoBase64Response">
      <part name="return" type="xsd:base64Binary"/>
   </message>
   <message name="echoDateRequest">
      <part name="inputDate" type="xsd:dateTime"/>
   </message>
   <message name="echoDateResponse">
      <part name="return" type="xsd:dateTime"/>
   </message>
   <message name="echoDecimalRequest">
      <part name="inputDecimal" type="xsd:decimal"/>
   </message>
   <message name="echoDecimalResponse">
      <part name="return" type="xsd:decimal"/>
   </message>
   <message name="echoBooleanRequest">
      <part name="inputBoolean" type="xsd:boolean"/>
   </message>
   <message name="echoBooleanResponse">
      <part name="return" type="xsd:boolean"/>
   </message>

   <portType name="InteropTestPortType">
      <operation name="echoString" parameterOrder="inputString">
         <input message="tns:echoStringRequest"/>
         <output message="tns:echoStringResponse"/>
      </operation>
      <operation name="echoStringArray" parameterOrder="inputStringArray">
         <input message="tns:echoStringArrayRequest"/>
         <output message="tns:echoStringArrayResponse"/>
      </operation>
      <operation name="echoInteger" parameterOrder="inputInteger">
         <input message="tns:echoIntegerRequest"/>
         <output message="tns:echoIntegerResponse"/>
      </operation>
      <operation name="echoIntegerArray" parameterOrder="inputIntegerArray">
         <input message="tns:echoIntegerArrayRequest"/>
      <output message="tns:echoIntegerArrayResponse"/>
      </operation>
      <operation name="echoFloat" parameterOrder="inputFloat">
         <input message="tns:echoFloatRequest"/>
      <output message="tns:echoFloatResponse"/>
      </operation>
      <operation name="echoFloatArray" parameterOrder="inputFloatArray">
         <input message="tns:echoFloatArrayRequest"/>
         <output message="tns:echoFloatArrayResponse"/>
      </operation>
      <operation name="echoStruct" parameterOrder="inputStruct">
         <input message="tns:echoStructRequest"/>
         <output message="tns:echoStructResponse"/>
      </operation>
      <operation name="echoStructArray" parameterOrder="inputStructArray">
         <input message="tns:echoStructArrayRequest"/>
         <output message="tns:echoStructArrayResponse"/>
      </operation>
      <operation name="echoVoid">
         <input message="tns:echoVoidRequest"/>
         <output message="tns:echoVoidResponse"/>
      </operation>
      <operation name="echoBase64" parameterOrder="inputBase64">
         <input message="tns:echoBase64Request"/>
         <output message="tns:echoBase64Response"/>
      </operation>
      <operation name="echoDate" parameterOrder="inputDate">
         <input message="tns:echoDateRequest"/>
         <output message="tns:echoDateResponse"/>
      </operation>
      <operation name="echoDecimal" parameterOrder="inputDecimal">
         <input message="tns:echoDecimalRequest"/>
         <output message="tns:echoDecimalResponse"/>
      </operation>
      <operation name="echoBoolean" parameterOrder="inputBoolean">
         <input message="tns:echoBooleanRequest"/>
         <output message="tns:echoBooleanResponse"/>
      </operation>
   </portType>
```

Notice that the WSDL document contains more complex types than the previous demo. Array of primitives types are now used as well as the struct primitive types. With the exception of the SOAPStruct complex type, every other type is supported as built-in type in the SOAP Mapping Registry. You then need to add a new complex type definition to the SOAP Mapping Registry to handle the SOAPStruct complex type.

The SOAPStruct schema definition is the following:

```
<complexType name="SOAPStruct">
      <all>
            <element name="varString" type="string"/>
            <element name="varInt" type="int"/>
            <element name="varFloat" type="float"/>
      </all>
</complexType>
```

In the `MySoapStructBean.java` file, this SOAPStruct complex type can be mapped to a simple JavaBean class such as the following, and have the marshalling and unmarshalling actions handled by the BeanSerializer.

```
public class MySoapStructBean implements java.io.Serializable
{
  private String m_varString = null;
  private int m_varInt = 0;
  private float m_varFloat = 0;

  public MySoapStructBean() {}
  public MySoapStructBean(String s, int i, float f) {
    m_varString = s;
    m_varInt    = i;
    m_varFloat  = f;
  }

  public String getVarString () { return m_varString; }
  public int getVarInt() { return m_varInt; }
  public float getVarFloat() { return m_varFloat; }

  public void setVarString (String s) { m_varString = s; }
  public void setVarInt(int i) { m_varInt = i; }
  public void setVarFloat(float f) { m_varFloat = f; }
}
```

With the mapping JavaBean class ready, and having identified what serializer and deserializer to use, you can now configure the `wsdl2ejb` utility so that a new schema to Java map is added. This can be achieved by adding the following to the `wsdl2ejb` configuration file, `base_conf.xml`:

```
<mapTypes jar="base/MySoapStructBean.jar" >
  <map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       local-name="SOAPStruct"
       namespace-uri="http://soapinterop.org/xsd"
       java-type="MySoapStructBean"
       java2xml-class-name="org.apache.soap.encoding.soapenc.BeanSerializer"
       xml2java-class-name="org.apache.soap.encoding.soapenc.BeanSerializer" />
</mapTypes>
```

The `MySoapStructBean.jar` file contains the definition of the MySoapStructBean class. With this map, the SOAPStruct complex type, belonging to the `http://soapinterop.org/xsd namespace`, will be mapped to the MySoapStructBean JavaBean class and the converse is true as well. For more information about SOAP serializers and deserializers, see the Oracle SOAP documentation.

With this additional configuration, you can now run the `wsdl2ejb` utility with the following command:

```
On UNIX
cd $ORACLE_HOME/webservices/demo/basic/wsdl2ejb
java -jar ../../../lib/wsdl2ejb.jar -conf base/base_conf.xml
On Windows
cd %ORACLE_HOME%/webservices/demo/basic/wsdl2ejb
java -jar ..\..\..\lib\wsdl2ejb.jar -conf base\base_conf.xml
```

The `wsdl2ejb` utility generates the `InteropLabApp.ear` file that contains the definition of a stateless EJB, which can be used as a proxy for the Web Service. The EAR file can be deployed in OC4J as any standard EJB. See *Oracle Application Server 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 Platform Enterprise Edition (J2EE) application acquires a WSDL document at runtime, the dynamic invocation API is used to invoke any SOAP operation described in the WSDL document. The dynamic invocation API describes a WebServiceProxyFactory factory class that can be used to build instances of a WebServiceProxy. Each created WebServiceProxy instance is based on the location of the WSDL document, (and optionally on additional qualifiers), that identify which service and port should be used. The WebServiceProxy class exposes methods to determine the WSDL portType, including the syntax and signatures of all operations exposed by the WSDL service and to invoke the defined operations.

This section briefly describes the dynamic invocation API and how to use it.

For Java samples, refer to the code supplied with Oracle Application Server Web Services in `$ORACLE_HOME/webservices/demo/basic/java_services/dynamicproxy` on UNIX or in `%ORACLE_HOME%\webservices\demo\basic\java_services\dynamicproxy` on Windows. For EJB samples, refer to the code supplied in the directory `$ORACLE_HOME/webservices/demo/basic/stateless_ejb` on UNIX or `%ORACLE_HOME%\webservices\demo\basic\stateless_ejb` on Windows.

Dynamic Invocation API

The dynamic invocation API contains two packages, oracle.j2ee.ws.client and oracle.j2ee,ws.client.wsdl, which contain additional classes grouped by interface, class, and exception, as shown in Table 11–4 and Table 11–5.

Table 11–4 The oracle.j2ee.ws.client Package

Classes	Description
Classes	
WebServiceProxyFactory	This class creates a WebServiceProxy class given a WSDL document.
Interfaces	
WebServiceProxy	This interface represents a service defined in a WSDL document.
WebServiceMethod	This interface invokes a Web Service method.
Exceptions	
WebServiceProxyException	This class describes exceptions raised by the WebServiceProxy API.

Table 11–5 The oracle.j2ee.ws.client.wsdl Package

Classes	Description
Interfaces	
PortType	This interface represents a port type.
Operation	This interface represents a WSDL operation.
Input	This interface represents an input message, and contains the name of the input and the message itself.
Output	This interface represents an output message, and contains the name of the output and the message itself.
Fault	This interface represents a fault message, and contains the name of the fault and the message itself.
Message	This interface describes a message used for communication with an operation.
Part	This interface represents a message part and contains the part's name, elementName, and typeName.

Table 11–5 (Cont.) The oracle.j2ee.ws.client.wsdl Package

Classes	Description
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 Oracle Application Server 10*g* Documentation Library as Proxy API Reference (Javadoc) under Oracle Application Server Web Services, which is located under the J2EE and Internet Applications tab.

The WebServiceProxyFactory class contains methods that can instantiate a WebServiceProxy class given either the URL or the Java input stream of the WSDL document. Four methods let you use either the first service and its first port in the supplied WSDL document or use the name of one of services and the name of one of the ports of the service to create a WebServiceProxy instance. Two methods also let you create a WebServiceProxy instance for a WSDL document, which has been authored following the UDDI best practices for WSDL. A method lets you supply additional optional initialization parameters to the WebServiceProxy instance.

Table 11–6 briefly describes the WebServiceProxyFactory factory class methods and the required parameters for each method. See the JavaDoc for more detailed information about this factory class and its methods.

Table 11–6 WebServiceProxyFactory Factory Methods and Parameters

Methods	Parameters

`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`

Table 11–6 (Cont.) WebServiceProxyFactory Factory Methods and Parameters

Methods	Parameters

`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 11–2 describes the WebServiceProxy interface. The WebServiceProxyFactory factory methods optionally take additional parameters that are provided in the WebServiceProxy interface that can be used to dynamically invoke an operation in a WSDL document.

Table 11–7 WebServiceProxy Interface Methods and Parameters

Methods	Parameters	Description
`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 11–8 describes the WebServiceMethod interface, which is used to invoke a Web Service method.

Table 11–8 WebServiceMethod Interface Methods and Parameters

Methods	Parameters	Description
`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.

12 Advanced Topics for Web Services

This `chapter` covers advanced Oracle Application Server Web Services topics, including the following topics:

- Setting the Web Services Debugging Property ws.debug
- Untyped Request Handling Options
- SOAP Header Support
- Using the WSDL Analyzer Utility

Setting the Web Services Debugging Property ws.debug

To obtain Oracle Application Server Web Services debugging information, use the Java property `ws.debug`, and set its value to `true`. To set the `ws.debug` value to `true`, use Oracle Enterprise Manager to specify OC4J startup options. Debugging output is sent to the OC4J instance log file corresponding to the island where Oracle Application Server Web Services is running.

Example 12–1 provides sample debugging output.

Example 12–1 Web Services Debug Output

```
WS Debug: initQnameMap('null')
WS Debug: operation name is: helloWorld
WS Debug: QueryString is: invoke=helloWorld&param0=test
WS Debug: Operation Name is: helloWorld
WS Debug: Port Type Local name is: StatelessExamplePortType
WS Debug: Port Type Namespace URI is: http://oracle.j2ee.ws_
example/StatelessExample.wsdl
WS Debug: Operation Local name is: helloWorld
WS Debug: Operation Namespace URI is: http://oracle.j2ee.ws_
example/StatelessExample.wsdl
WS Debug: Operation Get parameter order: null
```

See Also: *Oracle Application Server Containers for J2EE User's Guide* for information on setting debugging options and showing debugging output.

Untyped Request Handling Options

Oracle Application Server Web Services supports requests for RPC style Web Services in the following cases:

- Typed requests where an incoming RPC request with SOAP encoded parameters includes type attributes that specify type information for every incoming parameter. Example 12–2 shows a sample typed RPC request.
- Untyped requests where an incoming RPC request with SOAP encoded parameters may not include a type attribute for every incoming parameter. Example 12–3 shows a sample un-typed RPC request. This type of RPC request provides improved interoperability with .NET clients.

Oracle Application Server Web Services client-side applications and tools do not generate untyped requests, but some external tools or applications may generate such requests. Due to the performance cost for supporting untyped requests, by default such support is not enabled.

To support requests with untyped parameters, use the optional `<accept-untyped-request>` tag with the WebServicesAssembler. This tag applies as a sub-tag with the `<stateful-java-service>` and `<stateless-java-service>` tags when the corresponding `<message-style>` tag is set to the value RPC. The `<accept-untyped-request>` tag also applies as a sub-tag for the `<stateless-session-ejb-service>` tag.

Table 12–1 shows `<accept-untyped-request>` tag specification.

Example 12–2 Sample Typed RPC Request

```
 <?xml version='1.0' encoding='UTF-8'?>
 <SOAP-ENV:Envelope xmlns:SOAP-ENV="
    http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:sayHello xmlns:ns1="urn:Hello"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
       <param0 xsi:type="xsd:string">Scott</param0>
       <param1 xsi:type="xsd:int">27</param1>
    </ns1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 12–3 Sample Un-Typed RPC Request

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="
      http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
     <ns1:sayHello xmlns:ns1="urn:Hello"
       SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
       <param0>Scott</param0>
       <param1>27</param1>
     </ns1:sayHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 12–1 WebServicesAssembler <accept-untyped-request> Tag

Tag	Description
`<accept-untyped-request>` *`value`* `</accept-untyped-request>`	Setting *value* to `true` tells `WebServicesAssembler` to allow the Web Service to accept untyped requests. When the value is `false`, the Web Service does not accept untyped-request. Valid values: `true`, `false` (case is not significant; TRUE and FALSE are also valid) Default value: `false`

SOAP Header Support

This section covers Oracle Application Server Web Services support for SOAP request headers sent from a Web Services client to an endpoint. This section covers the following topics:

- Client Side SOAP Request Header Support
- Server Side SOAP Request Header Support
- Limitations for SOAP Header Support

Client Side SOAP Request Header Support

Oracle Application Server Web Services generated client-side proxy code provides methods to use SOAP request headers. A SOAP request message, including the SOAP request headers is transmitted to a service endpoint when Web Services proxy code is invoked.

When Oracle Application Server Web Services generates a proxy, either from WSDL for a Web Services Document or RPC style service, the proxy code provides two SOAP request header support methods:

```
void _setSOAPRequestHeaders(org.apache.soap.Header headers)
org.apache.soap.Header _getSOAPRequestHeaders()
```

These methods provide access to an `org.apache.soap.Header` object. By default the org.apache.soap.Header object's value is set to `null` which signifies there are no headers in the SOAP request message. When a request header is needed, use the `_setSOAPRequestHeaders()` method to specify the `Header` object to be sent with the SOAP request message.

Note: When proxies are generated for Stored Procedure or JMS Document Style Web Services the `_setSOAPRequestHeaders()` and `_getSOAPRequestHeaders()` methods are not supplied.

The SOAP request header information is shared for all proxy operations. After the headers are set using `_setSOAPRequestHeaders()`, all subsequent operation invocations using the proxy use the same header value. To set a new header value, call `_setSOAPRequestHeaders()` using a new `Header` object or with a `null` value.

Note: After setting the SOAP request header, the same header object is used for each subsequent operation invocation until the object is reset using `_setSOAPRequestHeaders()`.

To create and manipulate SOAP request headers you need to populate the header object. The `org.apache.soap.Header` object provides a method for specifying the contents of one or more SOAP header blocks. It is defined as:

```
public void setHeaderEntries(java.util.Vector headerEntries)
```

The vector is populated with `org.w3c.dom.Element` objects which specify individual SOAP header blocks.

When a header entry includes the `mustUnderstand` attribute set to the value 1, the recipient must process the header entry. If the recipient cannot process the header entry, then a SOAP fault is returned with the value `FAULT_CODE_MUST_UNDERSTAND`.

See Also: Section 4.2, "SOAP Header", in for information on header entries in SOAP 1.1 `http://www.w3.org/TR/SOAP/`.

Setting SOAP Headers in a Client-Side Proxy

This section shows a sample that uses the proxy class `EmployeeProxy`. The complete sample containing this code is available in the directory `$ORACLE_HOME/web_services/demo/header_demo/client`. In the sample, a single header block is added to the `Header` object. The `Header` object is then supplied as an argument to the proxy's `_setSOAPRequestHeaders()` method.

Example 12–4 Segment of Client Using Message with SOAP Request Header

```
.
.
.
// Create an intance of the proxy
EmployeeProxy proxy = new EmployeeProxy();
// Create a Header objecy
Vector v = new Vector();
v.add (e);
Header hdr = new Header();
hdr.setHeaderEntries(v);

// Set the Header
proxy._setSOAPRequestHeaders(hdr);
// Invoke the request
System.out.println("Salary of MILLER is: " + proxy.getEmployeeSalary("MILLER"));
```

Server Side SOAP Request Header Support

To process a SOAP request header on the server side, a Web Service needs to implement the `oracle.j2ee.ws.HeaderCallback` interface that is part of the Oracle Application Server Web Services supplied `wsserver.jar`. This interface includes one method that takes a single `org.apache.soap.Header` argument.

The Oracle Application Server Web Services infrastructure calls the `processHeaders()` method before every associated service method.

When an incoming SOAP request header includes one or more header entries with the `mustUnderstand` attribute set to either 1, `true`, or `TRUE` values, then the Web Service implementation must implement the `oracle.j2ee.ws.HeaderCallback` interface. If this interface is not implemented, Oracle Application Server Web Services throws a SOAP fault with the fault code set to `FAULT_CODE_MUST_UNDERSTAND`.

If a Web Service implementation implements the `HeaderCallback` interface, the implementation can throw a SOAP exception with the fault code set to `FAULT_CODE_MUST_UNDERSTAND` if the service does not know how to process a header entry with the mustUnderstand attribute set to 1, `true`, or `TRUE`. Oracle Application Server Web Services then translates the exception and Oracle Application Server Web Services throws a SOAP fault with the fault code set to `FAULT_CODE_MUST_UNDERSTAND`.

This section shows server-side Web Services code that provides the implementation for the `Employee` service. The complete sample containing this Web Service is available in the directory `$ORACLE_HOME/web_services/demo/basic/header_demo/client` (after unzipping `$ORACLE_HOME/webservices/demo/demo.zip`).

Example 12–5 shows an interface that extends `HeaderCallback`.

Example 12–6 shows a section of the service implementation for the sample `getEmployeeSalery` interface, including the `processHeaders()` method that can handle incoming SOAP request headers of the form:

```
<SOAP-ENV:Header>
  <credentials>
    <username>scott</username>
    <password>tiger</password>
   <datasource>jdbc/OracleCoreDS</datasource>
  </credentials>
</SOAP-ENV:Header>
```

Example 12–5 Employee Interface Extending HeaderCallback

```
import oracle.j2ee.ws.HeaderCallback;
/**
 *  Employee java class being exposed as Web Services
 *  This service also extends HeaderCallback so as to
 *  access Headers.
 */
public interface Employee
  extends  HeaderCallback
{
  // Get the salary for a given Employee
  int getEmployeeSalary(String  ename);
}
```

Example 12–6 Including A HeaderCallback processHeaders() Implementation

```
  public void processHeaders(Header header)
     throws java.io.IOException,
            oracle.xml.parser.v2.XSLException
  {
    // Get all the Elements
    Vector entries = header.getHeaderEntries();
    Element e = (Element) entries.firstElement();
    System.out.println("Element received is: " );
    ((XMLElement)e).print(System.out);

    // Get independent nodes and retrieve node values.
    Node userNode;
    userNode = ((XMLNode)e).selectSingleNode("username");
    userName = ((XMLElement)userNode).getText();

    Node passwordNode;
    passwordNode = ((XMLNode)e).selectSingleNode("password");
    password = ((XMLElement)passwordNode).getText();

    Node dsNode;
    dsNode = ((XMLNode)e).selectSingleNode("datasource");
    datasourceName = ((XMLElement)dsNode).getText();

    System.out.println("User name is: " + userName);
    System.out.println("Password is: " + password);
    System.out.println("Datasource is: " + datasourceName);
  }
```

Limitations for SOAP Header Support

The following list contains limitations related to SOAP header support:

1. Oracle Application Server Web Services does not provide support for processing or translating header information that is specified in a WSDL definition.
2. Oracle Application Server Web Services does not provide validation, XML or otherwise, for SOAP request header information provided in the `org.apache.soap.Header` object. The user is responsible for populating this object with well-formed XML.
3. Oracle Application Server Web Services does not provide support for SOAP response headers.
4. When proxies are generated for JMS Document Style Web Services, the SOAP request header `_setSOAPRequestHeaders()` and `_getSOAPRequestHeaders()` methods are not supplied. Using JMS Web Services there are no server-side facilities for processing SOAP request headers.
5. When proxies are generated for Stored Procedure Web Services, the SOAP request header `_setSOAPRequestHeaders()` and `_getSOAPRequestHeaders()` methods are not supplied. Using Stored

Procedure Web Services there are no server-side facilities for processing SOAP request headers.

Using the WSDL Analyzer Utility

The `wsdlAnalyzer` is a sample Oracle Application Server Web Services utility that checks WSDL files and invokes Web Services. The utility enables you to analyze a WSDL file from a given URL that you supply, or from a file.

The `wsdlAnalyzer.ear` file is supplied on the Oracle Technology Network Web site,

http://otn.oracle.com/sample_code/tech/java/web_services/content.html

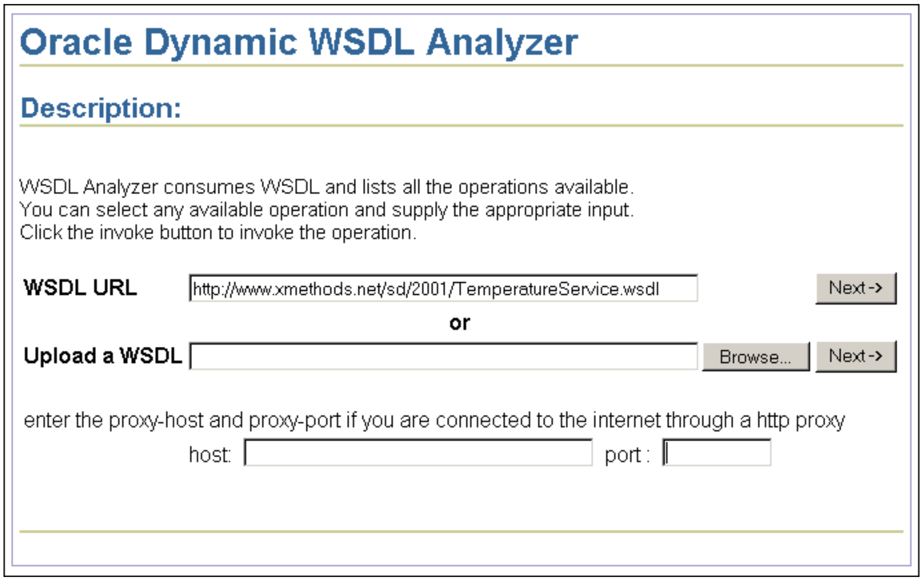
The `README.txt` file in the directory describes how to deploy the utility. Add the `dsv2.jar` file as a library element in `$ORACLE_HOME/config/application.xml` before running `wsdlAnalyzer`.

After deploying the wsdlAnalyzer open your browser and point to the URL:

```
http://host:port/wsdlAnalyzer
```

Figure 12–1 shows the `wsdlAnalyzer` page where you enter the WSDL location. Figure 12–2 shows a result page after entering parameters and selecting the Invoke button.

Figure 12–1 wsdlAnalyzer Web Service Result Page



Oracle Dynamic WSDL Analyzer

Description:

WSDL Analyzer consumes WSDL and lists all the operations available.
You can select any available operation and supply the appropriate input.
Click the invoke button to invoke the operation.

WSDL URL: http://www.xmethods.net/sd/2001/TemperatureService.wsdl

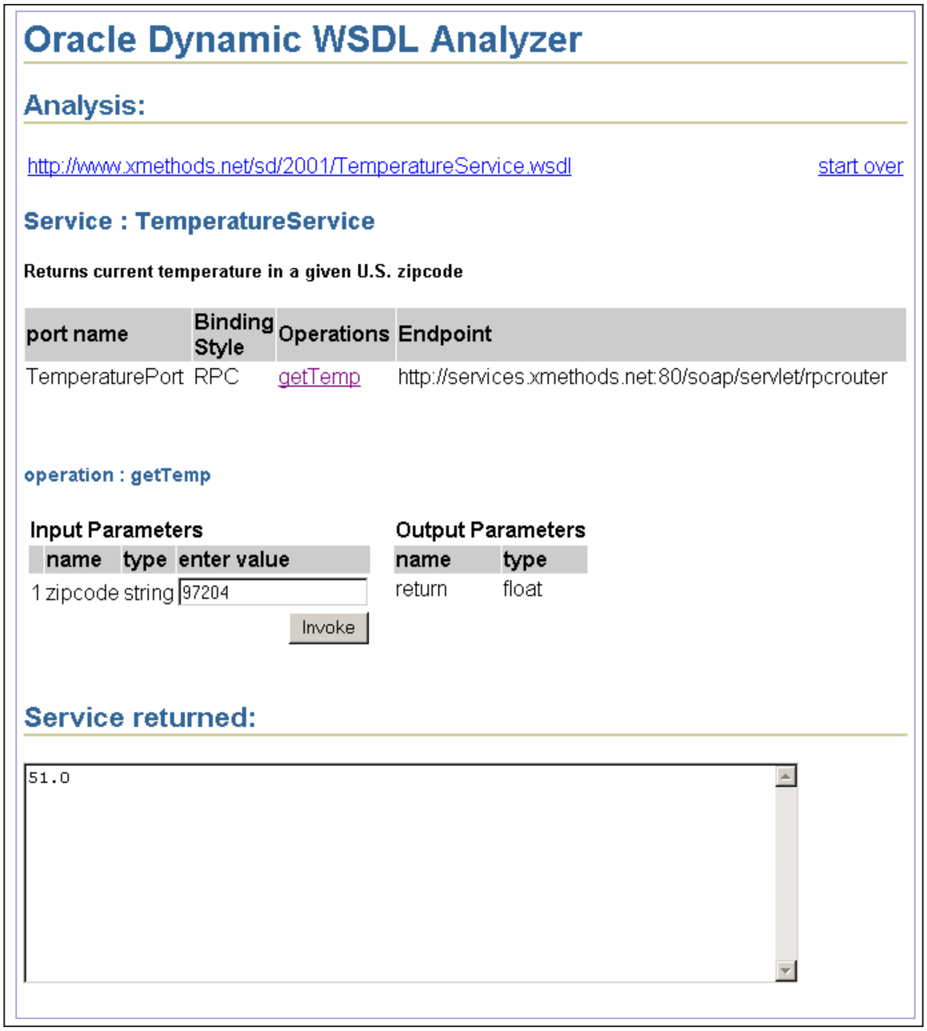
or

Upload a WSDL

enter the proxy-host and proxy-port if you are connected to the internet through a http proxy

host: port :

Figure 12–2 wsdlAnalyzer Test Page



Oracle Dynamic WSDL Analyzer

Analysis:

http://www.xmethods.net/sd/2001/TemperatureService.wsdl

start over

Service : TemperatureService

Returns current temperature in a given U.S. zipcode

port name	Binding Style	Operations	Endpoint
TemperaturePort	RPC	getTemp	http://services.xmethods.net:80/soap/servlet/rpcrouter

operation : getTemp

Input Parameters

	name	type	enter value
1	zipcode	string	97204

Invoke

Output Parameters

name	type
return	float

Service returned:

```
51.0
```

A
Using Oracle SOAP

This appendix covers the following topics:

- Understanding Oracle Application Server SOAP
- Apache SOAP Documentation
- Configuring the SOAP Request Handler Servlet
- Using OracleAS SOAP Management Utilities and Scripts
- Deploying OracleAS SOAP Services
- Using OracleAS SOAP Handlers
- Using OracleAS SOAP Audit Logging
- Using OracleAS SOAP Pluggable Configuration Managers
- Working With OracleAS SOAP Transport Security
- Using OracleAS SOAP Sample Services
- Using the OracleAS SOAP EJB Provider
- Using PL/SQL Stored Procedures With the SP Provider
- SOAP Troubleshooting and Limitations
- OracleAS SOAP Differences From Apache SOAP
- Apache Software License, Version 1.1

Understanding Oracle Application Server SOAP

In addition to the Oracle Application Server Web Services previously described in this chapter, that use a unique Servlet interface and J2EE deployment for Web Services, Oracle Application Server also provides Oracle Application Server SOAP (OracleAS SOAP) that is derived from Apache 2.3.1 SOAP and includes a number of enhancements.

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

OracleAS SOAP is a modified version of Apache SOAP 2.3.1. Most of the documentation that applies to Apache SOAP 2.3.1 also applies to OracleAS SOAP. The Apache SOAP 2.3.1 documentation can be found at the following site:

http://xml.apache.org/soap/docs/index.html

Configuring the SOAP Request Handler Servlet

The OracleAS 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 `soap.ear` file in the directory `$SOAP_HOME/lib` on UNIX or `%SOAP_HOME%\lib` 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, expand the `soap.ear` file. In the directory `webapps/soap/WEB-INF` on UNIX or `webapps\soap\WEB-INF` on Windows, modify the path name specified for the `SoapConfig` parameter in the `soap.properties` file. Then, redeploy the updated `soap.ear` 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`. OracleAS 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. OracleAS 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 OracleAS 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 OracleAS SOAP Management Utilities and Scripts

To use the OracleAS 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 OracleAS SOAP Services

This section covers the following topics related to deploying and undeploying OracleAS 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 OracleAS 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 OracleAS SOAP Web Service in Oracle Application Server Containers for J2EE (OC4J) by performing the following steps:

1. Create service and provider deployment descriptors.
2. Expand the `soap.ear` file found in `$SOAP_HOME/lib` on UNIX or `%SOAP_HOME\lib` on Windows.
3. Copy Java classes and Jars implementing the service to the correct locations in the expanded `soap.ear` directories.

   Copy Java .class files to `WEB-INF/classes`.

   Copy Java .jar files to `WEB-INF/libs`.
4. Redeploy the updated `soap.ear` file.
5. Deploy the provider descriptor by executing the command:

   ```
   providerMgr.sh deploy FileName
   ```

   where *FileName* is the name of the provider descriptor xml file.
6. 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

It is necessary to install an OracleAS SOAP service on every machine in a cluster. If the service is not installed on all machines in a 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 OracleAS 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 OracleAS SOAP Audit Logging

The OracleAS 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).

OracleAS SOAP stores audit trails as XML documents. Using XML documents, OracleAS SOAP creates portable audit trails and enables the transformation of complete audit trails or individual audit records to different formats.

By default, OracleAS 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`

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.

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

Audit Event Filter Attributes	Description
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.
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 Oracle 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`
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 OracleAS SOAP Handlers" on page A-13

Using OracleAS SOAP Pluggable Configuration Managers

OracleAS SOAP supports pluggable configuration managers similar to those supported in Apache SOAP 2.3.1. Since OracleAS SOAP supports provider deployment descriptors separate from service deployment descriptors, the interface details using OracleAS SOAP are slightly different from Apache SOAP 2.3.1. In OracleAS 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 OracleAS 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 OracleAS SOAP Transport Security

Oracle Application Server uses the security capabilities of the underlying transport that sends SOAP messages. Oracle 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)

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

In an OracleAS 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 Oracle Application Server supports overriding the values specified for system properties using Oracle 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.authRealm	Specifies the realm for which the HTTP authentication username/password is specified. This property is mandatory when using basic authentication.
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.proxyAuthRealm	Specifies the realm for which the proxy authentication username/password is specified.
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.proxyUsername	Specifies the HTTP proxy authentication username.
http.username	Specifies the HTTP authentication username.

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

Property	Description
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 Oracle 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.
1022ContentType
	Specifies the value for the Content-Type HTTP header in Oracle9*i*AS, and in Oracle Application Server 10*g*. 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 or 10g (9.0.4). This property provides interoperablity between Oracle9iAS Release 9.0.x Oracle SOAP clients or Oracle Application Server 10*g* (9.0.4) 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 Oracle Application Server version 9.0.x 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`.

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.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.
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 Oracle Application Server SOAP Client

This section describes how to use SSL with the OracleAS 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`).

OracleAS 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 OracleAS SOAP Client side available, it is still possible to use HTTPS as a transport with OracleAS 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 OracleAS SOAP Sample Services

The section lists the samples included with OracleAS SOAP. The class files for all of the samples are in `$SOAP_HOME/lib/samples.jar` on UNIX or in `%SOAP_HOME%\lib\samples.jar` on Windows.

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 OracleAS SOAP EJB Provider

This section compares the OracleAS 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.

OracleAS SOAP uses the same mechanism to support Stateless Session EJBs as Apache SOAP.

Stateful Session EJB Provider in Apache SOAP

On receiving a first time SOAP request, the Apache SOAP Stateful Session EJB provider first locates the Home Interface through a JNDI lookup and using a subsequent create obtains an object reference to a Stateful Session EJB. The provider then invokes the requested method on the object reference.

In the next step the provider serializes the `EJBHandle` of the specified EJB reference and appends it to the targetURI with an "@" delimiter. The Stateful Session EJB provider then sends this modified target URI back to the requesting SOAP client. If the client wants to reuse the same EJB instance, it must retrieve this "modified" target URI for the service from the Response and set it in the next SOAP Call.

Upon receiving this request, the Stateful EJB provider extracts the stringified EJB reference and deserializes it into an EJBHandle from which it can obtain the EJB reference. It can then invoke the method on the specified EJB.

The drawback of the Apache SOAP implementation is that the client must be EJB aware and that it could not operate with other SOAP servers.

OracleAS SOAP offers an alternative solution for Stateful Session EJBs that allows for client interoperablity.

Stateful Session EJB Provider in OracleAS SOAP

The OracleAS SOAP Stateful Session EJB provider binds the EJB reference to the current session, if none is bound, otherwise, it merely retrieves the EJB reference from the session. In order for the client to access the same Stateful Session EJB, the client has to simply maintain 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 OracleAS SOAP

In order for a SOAP client to run a business method on an entity EJB, it first needs to either "create" a new EJB upon which to run the method or *find* an already existing EJB which suit some criteria. Access to an entity EJB occurs within a session. At the start of the session the SOAP client must invoke a "create" or "find" (in order to specify the bean object interest). While maintaining the same session, all other business methods are directed to that EJB. A subsequent "find" or "create" within the same or different session directs business method execution requests to the newly "created" (or "found") EJB.

Another issue is that EJB specification provides that some "find" methods can return either a Collection of EJB refs or single EJB ref.

The Oracle solution for Entity EJBs embraces the following solution for this problem:

It disallows find methods that return "Collections". This allows for the provider to uniquely specify an Entity EJB to target subsequent business method requests.

Deployment and Use of the OracleAS SOAP EJB Provider

To install an EJB provider and deploy Web Services to the provider under OC4J, where the application server hosts both the SOAP servlet and the deployed EJB's, follow these steps:

1. Deploy an EJB provider to SOAP using a provider descriptor.

   The provider descriptor specifies the following:

   - EJB access credentials by the middle tier
   - JNDI context factory class
   - JNDI context factory URL
   - Provider class name
   - Provider id

2. Create the EJB Web Service:
   - Define the associated EJB classes and package the EJB into an EAR file as defined by J2EE spec.
   - Define the service descriptor which specifies following details of the EJB Web Service:
     * JNDI Location
     * Home interface class name
     * Application Deployment Name of this EJB Web Service in OC4J
     * The provider id to which this service is to be associated
3. Deploy ear file in OC4J. Modify the OC4J specific EJB descriptor to correct the JNDI location for the EJB (as described in sample README).

Current Known EJB Provider Limitations

All service methods can only take primitive Java types as arguments to the methods. User-defined Java types are currently not supported.

Using PL/SQL Stored Procedures With the SP Provider

The OracleAS 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 Oracle JPublisher to translate from PL/SQL to Java; hence, it inherits all of the restrictions of Oracle JPublisher.

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 Oracle 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)

Using Object Types

Oracle 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 Oracle 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 Oracle 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 Oracle Application Server Web Services, including:

- Tunneling Using the TcpTunnelGui Command
- Setting Configuration Options for Debugging
- Using DMS to Display Runtime Information

- SOAP Limitations for Java Type Precedence with Overloaded Methods

Tunneling Using the TcpTunnelGui Command

SOAP provides the `TcpTunnelGui` command to display messages sent between a SOAP client and a SOAP server. `TcpTunnelGui` listens on a TCP port, which is different than the SOAP server, and then forwards requests to the SOAP server.

Invoke `TcpTunnelGui` as follows:

```
java org.apache.soap.util.net.TcpTunnelGui TUNNEL-PORT SOAP-HOST SOAP-PORT
```

Table A–7 lists the command line options for `TcpTunnelGui`.

Table A–7 TcpTunnelGui Command Arguments

Argument	Description
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 in the file `soap.xml.` This file is placed in `soap.ear` file in the directory `$SOAP_HOME/lib` on UNIX or in `%SOAP_HOME%\lib` on Windows.

To modify the debugging option, expand the `soap.ear` file and modify the file `soap.xml` in the directory `webapps/soap/WEB-INF` on UNIX or in `webapps\soap\WEB-INF` on Windows, then redeploy the updated `soap.ear` file.

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

Oracle Application Server Web Services is instrumented with DMS to gather information on the execution of the SOAP Request Handler Servlet, the Java Provider, and on individual services.

DMS information includes execution intervals from start to stop for the following:

- Total time spent in SOAP request and response (includes time in providers and services)
- Total time spent in the Java Provider (includes time in services)
- Total time executing services (`soap/java-provider/`*`service-URI`*)

To view the DMS information, go to the following site:

```
http://hostname:port/soap/servlet/Spy
```

SOAP Limitations for Java Type Precedence with Overloaded Methods

OracleAS 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()`.

OracleAS SOAP Differences From Apache SOAP

This section covers differences between Apache Soap and OracleAS SOAP.

Service Installation Differences

Additional instructions are provided for installing services when OracleAS SOAP is used in conjunction with OC4J.

Optional Provider Enhancements

OracleAS 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 OracleAS 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 OracleAS 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 OracleAS 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 OracleAS 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 OracleAS 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 OracleAS 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.

Apache Software License, Version 1.1

This program contains third-party code from the Apache Software Foundation (Apache). Under the terms of the Apache license, Oracle is required to provide the following notices. Note, however, that the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle AS IS and without warranty or support of any kind from Oracle or Apache.

B

Web Services Security

The ability to control user access to Web content and to protect your site against people breaking into your system is critical. This appendix describes the architecture and configuration of security for Oracle Application Server Web Services, including the Oracle Application Server UDDI Registry.

This chapter covers the following topics:

- About Web Services Security
- Configuring Web Services Security
- About Oracle Application Server UDDI Registry Security
- Configuring UDDI Security

See Also:

- *Oracle Application Server 10g Security Guide*
- *Oracle Identity Management Concepts and Deployment Planning Guide*

About Web Services Security

SOAP is the messaging protocol for Oracle Application Server Web Services. Oracle Application Server Web Services only supports HTTP (S) for a transport protocol for SOAP messages. Oracle Application Server security that applies for HTTP(S) can be leveraged for Oracle Application Server Web Services.

Oracle Application Server Web Services supports the following security features:

- Secure Connection: By securing the connection using SSL (HTTPS), one can invoke a Web Service securely.
- Authentication: Basic and Digest Access Authentication can be enforced using HTTP (S) headers. This method is not secure unless the authentication is specified in conjunction with SSL.
- Authorization: Authorization is supported by retrieving the Principal using a User Manager such as the Oracle Application Server Java Authentication and Authorization Service (JAZN) User Manager.

All the HTTP(S) transport security features are applicable to all types of Oracle Application Server Web Services implementations (including stateless and stateful java classes, stateless session bean and stateless stored procedures). In addition, if a stateless session bean is exposed as a Web Service, ACL policies can be enforced on the bean when the connection is authorized by a User Manager and a Principal object is obtained.

If a stored procedure is exposed as a Web Service, then it is secure to encrypt the password of the corresponding data source in the data-sources.xml file.

See Also:

- *Oracle Application Server Containers for J2EE Security Guide*
- Chapter 8, "Configuring EJB Application Security" in the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*

Configuring Web Services Security

When you run a client-side application that uses Oracle Application Server Web Services, you can access secure Web Services by setting properties in the client application. Table B–1 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 B–1 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 B–1 Web Services HTTP Transport Security Properties

Property	Description
`http.authRealm`	Specifies the realm for which the HTTP authentication username / password is specified. This property is mandatory when using basic authentication.
`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.proxyAuthRealm`	Specifies the realm for which the proxy authentication username / password is specified.
`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.proxyUsername`	Specifies the HTTP proxy authentication username.
`http.username`	Specifies the HTTP authentication username.

Table B–1 (Cont.) Web Services HTTP Transport Security Properties

Property	**Description**
`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 Oracle 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.

Table B–1 (Cont.) Web Services 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.

About Oracle Application Server UDDI Registry Security

This section covers the following topics:

- Protecting Oracle Application Server UDDI Registry Resources
- Managing and Enforcing Protected UDDI Resources
- Using Oracle Application Server Security Services

See Also: "OracleAS UDDI Registry Administration" on page 10-25

Protecting Oracle Application Server UDDI Registry Resources

Oracle Application Server UDDI resources are protected as follows.

Oracle Application Server UDDI Registry

For the OracleAS UDDI Registry, the following resources are protected:

- Data – Write access to the data stored in the OracleAS UDDI Registry is protected; this is typically metadata of Web Services.
- Functions – Administrative operations to the OracleAS UDDI Registry.
- Passwords – N / A. User passwords are protected by JAZN.

Oracle Application Server Content Subscription Manager Application

For the Oracle Application Server UDDI Content Subscription Manager application, the following resource is protected:

- Passwords – Password for the UDDI syndication subscriber are protected.

Managing and Enforcing Protected UDDI Resources

Protection for the following OracleAS UDDI Registry resources are managed and enforced as follows.

Oracle Application Server UDDI Registry

Oracle Application Server Java Authentication and Authorization Service (JAZN) and the UDDI application manages and enforces write access to the data stored in the OracleAS UDDI Registry. JAZN determines the identity and the security role of a user. Only the owner has rights to update data.

For administrative operations for the OracleAS UDDI Registry JAZN also manages and enforces access; in addition, JAZN protects the servlets that provide administrative operations.

Oracle Application Server Content Subscription Manager Application

The application manages the UDDI syndication subscription password used to access Oracle Application Server Syndication Services. The password, which is persistently stored in the database, is further protected by the database `DBMS_OBFUSCATION PL/SQL` package.

Update of the UDDI syndication subscriber password is available through a UDDI Web-based tool. The web-based tool uses JAZN to query the security role of the authenticated user. The password update facility is available only if the authenticated user has the `uddiadmin` security role.

> **See Also:** "Using the UDDI Content Subscription Manager as a UDDI Administrator" on page 10-91

Using Oracle Application Server Security Services

UDDI leverages the JAZN User level security features and uses SSL encryption, both server side and client side, for accessing OracleAS Infrastructure 10*g* options.

Configuring UDDI Security

To configure UDDI for security, consider the following areas:

- Configuring the Oracle Application Server UDDI Registry
- Configuring the UDDI Content Subscription Manager
- Configuring the UDDI Client

Configuring the Oracle Application Server UDDI Registry

To ensure the confidentiality of the communication between the OracleAS UDDI Registry and clients, do the following:

1. Configure the Oracle HTTP Server / SSL listener to provide HTTPS access.
2. Configure OC4J to prohibit HTTP access.
3. To ensure the communication to a UDDI replication endpoint is authorized, configure the Oracle HTTP Server / SSL listener to enable HTTPS client-certificate based authentication.

Configure all security-sensitive UDDI endpoints, including: publishing, administration, replication wallet administration, and subscription management (typically, the inquiry endpoint does not need to be confidential).

Configuring the UDDI Content Subscription Manager

In order to make the Oracle Application Server Content Subscription Manager functional, you must supply the proper password of the UDDI syndication subscriber.

> **See Also:** "Using the UDDI Content Subscription Manager as a UDDI Administrator" on page 10-91

Configuring the UDDI Client

If you use the UDDI Client Library to develop applications to communicate with the OracleAS UDDI Registry, you can use the Oracle Application Server Web Services security features to configure the HTTP transport properties.

> **See Also:** "Configuring Web Services Security" on page B-2

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 OracleAS UDDI Registry before accessing the service.

SOAP

SOAP is the name of 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 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 OracleAS UDDI Registry.

Stored Procedure Web Service

Oracle Application Server 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 Oracle Application Server 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