

Developer's Guide to Web Services

Sun™ ONE Application Server

Version 7, Enterprise Edition

817-2151-10
September 2003

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.

Copyright © 2003 Sun Microsystems, Inc. All rights reserved.

THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF SUN MICROSYSTEMS, INC. USE, DISCLOSURE OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF SUN MICROSYSTEMS, INC. U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Sun™ ONE, the Java Coffee Cup logo and the Sun™ ONE logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

This product is covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

CE LOGICIEL CONTIENT DES INFORMATIONS CONFIDENTIELLES ET DES SECRETS COMMERCIAUX DE SUN MICROSYSTEMS, INC. SON UTILISATION, SA DIVULGATION ET SA REPRODUCTION SONT INTERDITES SANS L'AUTORISATION EXPRESSE, ÉCRITE ET PRÉALABLE DE SUN MICROSYSTEMS, INC. Droits du gouvernement américain, utilisateurs gouvernementaux - logiciel commercial. Les utilisateurs gouvernementaux sont soumis au contrat de licence standard de Sun Microsystems, Inc., ainsi qu'aux dispositions en vigueur de la FAR (Federal Acquisition Regulations) et des suppléments à celles-ci. L'utilisation est soumise aux termes de la Licence.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, Sun™ ONE, le logo Java Coffee Cup et le logo Sun™ ONE sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Ce produit est soumis à la législation américaine en matière de contrôle des exportations et peut être soumis à la réglementation en vigueur dans d'autres pays dans le domaine des exportations et importations. Les utilisations, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers les pays sous embargo américain, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exhaustive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

Contents

About This Document	7
Who Should Use This Guide	7
Using the Documentation	8
Documentation Conventions	10
General Conventions	10
Conventions Referring to Directories	11
How This Guide Is Organized	12
Reference Information	13
Product Support	13
Chapter 1 About Web Services	15
What are Web Services?	15
Messaging Models Used in Web Services	16
Synchronous Model	16
Asynchronous Model	17
Standards and Interoperability in Web Services	17
SOAP	17
WSDL	18
UDDI	18
ebXML	18
A Simple Web Service Scenario	19
Support for Web Services in Sun ONE Application Server	20
Java APIs for XML and Web Services	21
JAXP	22
JAX-RPC	23
JAXR	23
SAAJ	24

JAXM	24
Implementing Web Services Using Java APIs	25
The Coffee Break Example	26
Preparing for Developing Web Services and Clients	28
Using Ant Tasks	28
Setting Up the Client Environment	29
 Chapter 2 Services and Clients Using JAX-RPC	33
JAX-RPC Implementation	33
Developing JAX-RPC Web Services	35
JAX-RPC Web Services Using a WSDL	37
Assembling and Deploying JAX-RPC Web Services	38
Invoking JAX-RPC Web Services	42
Creating Clients Using Generating Stubs Method	43
Generating the Stubs	45
Coding the Client	45
Compiling the Client Code	46
Assembling the Client Classes into a JAR file	46
Running the Client	47
Creating Clients Using Dynamic Invocation Interface	47
Creating JAX-RPC Client Using a Dynamic Proxy	48
Creating a JAX-RPC Client Using the Call Interface	49
Assembling and Deploying a JAX-RPC Client	51
Sample Applications	52
JAX-RPC Client Invoking an EJB	52
Building Security into JAX-RPC Web Services	54
Basic Authentication Over SSL	55
Adding Security Elements to web.xml	56
Setting Security Properties in the Client Code	57
Mutual Authentication Over SSL	58
Setting Up Client Certificate Authentication for Web Services	59
JAX-RPC Tools	63
wscompile Tool	63
wscompile Command Options	64
Configuration File	66
wsdeploy Tool	68
wsdeploy Command Options	68
Namespace Mappings	70
SOAP Handlers	71
Java Language Types Supported By JAX-RPC	72

Chapter 3 SOAP Clients and Services Using SAAJ and JAXM	75
SOAP Clients	75
SOAP Client Messaging Models	75
Client Without a Messaging Provider	76
Client With a Messaging Provider	77
SOAP Messages	78
Parts of a SOAP Message	78
Accessing Elements of a Message	80
Developing a SOAP Client	82
How SOAP Messaging Occurs?	82
Creating a SOAP Client	83
Assembling and Deploying a SOAP Client	89
SOAP Service	90
Creating a SOAP Service	90
Exception and Fault Handling	91
Fault Handling	91
Defining SOAP Fault	93
Assembling and Deploying a SOAP Service	94
Sample Clients and Services	94
 Chapter 4 Clients Using JAXR	 95
Developing a JAXR Client	95
Getting Access to a Registry	96
Accessing an ebXML Registry	96
Establishing a Connection	96
Querying a Registry	100
Finding Organizations by Name	101
Finding Organizations by Classification	101
Finding Organizations by WSDL Descriptions	102
Finding Services and Service Bindings	103
Managing Registry Data	104
Getting Authorization from the Registry	104
Creating an Organization	105
Adding Classifications	106
Using Taxonomies	107
Defining Taxonomies	107
Specifying Postal Address	109
Adding Services and Service Bindings to an Organization	111
Publishing a Web Service to a UDDI Registry	112
Assembling and Deploying a JAXR Client	117
Sample JAXR Client	118

Appendix A XML Schema Definitions 119
XML Schema for wscompile Configuration File 119
XML Schema for Deployment Descriptors 132
XML Schema for Exported wscompile Model Files 138
XML Schema for Runtime Descriptors 141

About This Document

This guide describes how to create and run Web services and Java™ based clients that invoke them on Sun™ Open Net Environment (Sun ONE) Application Server. In addition to describing programming concepts and tasks, this guide offers sample code, implementation tips, reference material, and a glossary.

This preface contains information about the following topics:

- [Who Should Use This Guide](#)
- [Using the Documentation](#)
- [Documentation Conventions](#)
- [How This Guide Is Organized](#)
- [Reference Information](#)
- [Product Support](#)

Who Should Use This Guide

The intended audience for this guide are the information technology developers in a corporate enterprise who develop and publish Web services, and build clients that invoke them.

This guide assumes you are familiar with the following topics:

- Java(2) Platform, Enterprise Edition™ specification
- HTML
- Java™ and XML programming
- Java APIs as defined in specifications for EJB™, JSP™, and JDBC™

- Software development processes, including debugging and source code control

Using the Documentation

The Sun ONE Application Server 7, Enterprise Edition manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML).

The following table lists tasks and concepts described in the Sun ONE Application Server manuals.

Table 1 Sun ONE Application Server Documentation Roadmap

For information about	See the following
Late-breaking information about the software and the documentation.	<i>Release Notes</i>
Comprehensive, table-based summary of supported hardware, operating system, JDK, and JDBC/RDBMS.	<i>Platform Summary</i>
Sun ONE Application Server 7 overview, features available with each product edition.	<i>Product Overview</i>
Diagrams and descriptions of server architecture, benefits of the Sun ONE Application Server architectural approach.	<i>Server Architecture</i>
New enterprise, developer, and operational features of Sun ONE Application Server 7.	<i>What's New</i>
How to get started with the Sun ONE Application Server 7 product. Includes new features, architectural overview, and sample application tutorial.	<i>Getting Started Guide</i>
Installing the Sun ONE Application Server software and its components, such as sample applications, the Administration interface, and the high-availability components. Instructions for implementing a basic high-availability configuration are included.	<i>Installation Guide</i>
Evaluating your system needs and enterprise to ensure that you deploy Sun ONE Application Server in a manner that best suits your site. General issues and concerns that you must be aware of when deploying an application server are also discussed.	<i>System Deployment Guide</i>
Best practices for HTTP session availability that application architects and developers can use.	<i>Application Design Guidelines for Storing Session State</i>

Table 1 Sun ONE Application Server Documentation Roadmap (*Continued*)

For information about	See the following
Creating and implementing Java™ 2 Platform, Enterprise Edition (J2EE™ platform) applications intended to run on the Sun ONE Application Server 7 that follow the open Java standards model for J2EE components such as servlets, Enterprise JavaBeans™ (EJBs™), and JavaServer Pages™ (JSPs™). Includes general information about application design, developer tools, security, assembly, deployment, debugging, and creating lifecycle modules. A comprehensive Sun ONE Application Server glossary is included.	<i>Developer's Guide</i>
Creating and implementing J2EE web applications that follow the Java™ Servlet and JavaServer Pages (JSP) specifications on the Sun ONE Application Server 7. Discusses web application programming concepts and tasks, and provides sample code, implementation tips, and reference material. Topics include results caching, JSP precompilation, session management, security, deployment, SHTML, and CGI.	<i>Developer's Guide to Web Applications</i>
Creating and implementing J2EE applications that follow the open Java standards model for enterprise beans on the Sun ONE Application Server 7. Discusses Enterprise JavaBeans (EJB) programming concepts and tasks, and provides sample code, implementation tips, and reference material. Topics include container-managed persistence, read-only beans, and the XML and DTD files associated with enterprise beans.	<i>Developer's Guide to Enterprise JavaBeans Technology</i>
Creating Application Client Container (ACC) clients that access J2EE applications on the Sun ONE Application Server 7.	<i>Developer's Guide to Clients</i>
Creating web services in the Sun ONE Application Server environment.	<i>Developer's Guide to Web Services</i>
Java™ Database Connectivity (JDBC™), transaction, Java Naming and Directory Interface™ (JNDI), Java™ Message Service (JMS), and JavaMail™ APIs.	<i>Developer's Guide to J2EE Services and APIs</i>
Creating custom NSAPI plug-ins.	<i>Developer's Guide to NSAPI</i>
Information and instructions on the configuration, management, and deployment of the Sun ONE Application Server subsystems and components, from both the Administration interface and the command-line interface. Topics include cluster management, the high-availability database, load balancing, and session persistence. A comprehensive Sun ONE Application Server glossary is included.	<i>Administrator's Guide</i>
Editing Sun ONE Application Server configuration files, such as the <code>server.xml</code> file.	<i>Administrator's Configuration File Reference</i>
Configuring and administering security for the Sun ONE Application Server operational environment. Includes information on general security, certificates, and SSL/TLS encryption. HTTP server-based security is also addressed.	<i>Administrator's Guide to Security</i>

Table 1 Sun ONE Application Server Documentation Roadmap (*Continued*)

For information about	See the following
Configuring and administering service provider implementation for J2EE™ Connector Architecture (CA) connectors for the Sun ONE Application Server 7. Topics include the Administration Tool, Pooling Monitor, deploying a JCA connector, and sample connectors and sample applications.	<i>J2EE CA Service Provider Implementation Administrator's Guide</i>
Migrating your applications to the new Sun ONE Application Server 7 programming model, specifically from iPlanet Application Server 6.x and from Netscape Application Server 4.0. Includes a sample migration.	<i>Migrating and Redploying Server Applications Guide</i>
How and why to tune your Sun ONE Application Server to improve performance.	<i>Performance Tuning Guide</i>
Information on solving Sun ONE Application Server problems.	<i>Troubleshooting Guide</i>
Messages that you may encounter while running Sun ONE Application Server 7. Includes a description of the likely cause and guidelines on how to address the condition that caused the message to be generated.	<i>Error Message Reference</i>
Utility commands available with the Sun ONE Application Server; written in manpage style.	<i>Utility Reference Manual</i>
Using the Sun™ Open Net Environment (Sun ONE) Message Queue software.	The Sun ONE Message Queue documentation at: http://docs.sun.com/db?p=prod/sl.slmsgqu

Documentation Conventions

This section describes the types of conventions used throughout this guide:

- [General Conventions](#)
- [Conventions Referring to Directories](#)

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX® format (with forward slashes separating directory names).
- **URLs** are given in the format:

`http://server.domain/path/file.html`

In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.

- **Font conventions** include:
 - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
 - *Italic* type is used for code variables.
 - *Italic* type is also used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
 - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.
- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in [“Conventions Referring to Directories” on page 11](#).

By default, the location of *install_dir* on **most** platforms is:

```
/opt/SUNWappserver7
```

For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See [“Conventions Referring to Directories” on page 11](#) for exceptions and additional information.

- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following:

```
default_config_dir/domains/domain/instance
```

Conventions Referring to Directories

By default, when using the Solaris™ 8 and 9 installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 8 and 9 installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:

- *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
- *default_config_dir* refers to `/var/opt/SUNWappserver7/domains` which is the default location for any domains that are created.
- *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

How This Guide Is Organized

This guide provides instructions for the development and the deployment of Web services to Sun ONE Application Server. The guide also provides information on developing client applications that can invoke Web services.

- [Chapter 1, “About Web Services”](#)

This chapter introduces you to Web services and the standards used in implementing Web services. Also discusses about the working of Web services in the Sun ONE Application Server environment.

- [Chapter 2, “Services and Clients Using JAX-RPC”](#)

This chapter describes the procedure to develop, deploy, execute JAX-RPC Web services and clients that access such services.

- [Chapter 3, “SOAP Clients and Services Using SAAJ and JAXM”](#)

This chapter introduces you to the SAAJ and JAXM APIs, how to use these APIs to develop message-oriented services and clients in Sun ONE Application Server environment.

- [Chapter 4, “Clients Using JAXR”](#)

This chapter introduces you to the JAXR API, how to use the API to build clients, and how to manage the registry data.

- [Appendix A, “XML Schema Definitions”](#)

This appendix provides XML Schema Definitions for the various configuration files used in developing JAX-RPC Web services and clients.

Finally, [Index](#) are provided.

Reference Information

In addition to the information in the Sun ONE Application Server documentation collection listed in “[Using the Documentation](#)” on page 8, we recommend the following resources:

General J2EE Information:

Core J2EE Patterns: Best Practices and Design Strategies by Deepak Alur, John Crupi, & Dan Malks, Prentice Hall Publishing

Java Security, by Scott Oaks, O'Reilly Publishing

Web Services:

Java Web Services, by David Chappell and Tyler Jewell, O'Reilly Publishing

Programming with EJB components:

Enterprise JavaBeans, by Richard Monson-Haefel, O'Reilly Publishing

Java API Specifications:

<http://java.sun.com/xml/download.html>

Java Web Services Tutorial:

<http://java.sun.com/webservices/docs/1.0/tutorial/index.html>

Product Support

If you have general feedback on the product or documentation, please send this to appserver-feedback@sun.com.

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

<http://www.sun.com/supporttraining/>

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation

- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

About Web Services

This chapter introduces you to Web services, standards used in implementing Web services, and summarizes the process of building such services.

This chapter describes the following topics:

- [What are Web Services?](#)
- [Messaging Models Used in Web Services](#)
- [Standards and Interoperability in Web Services](#)
- [Support for Web Services in Sun ONE Application Server](#)
- [A Simple Web Service Scenario](#)
- [Java APIs for XML and Web Services](#)
- [Implementing Web Services Using Java APIs](#)
- [Preparing for Developing Web Services and Clients](#)

What are Web Services?

A Web service is a modular application that you can describe, publish, locate, and invoke across the web. A Web service perform functions, which can be anything from simple requests to complicated business processes. Once a Web service is deployed, other applications or other Web services can discover and invoke the deployed service.

Web services are invoked using Simple Object Access Protocol (SOAP) messages. SOAP is a lightweight messaging protocol that allows objects of any kind, on any platform, written in any language to cross-communicate. SOAP messages are encoded in eXtensible Markup Language (XML) and typically transported over HTTP. Unlike other distributed computing technologies, Web services are loosely coupled and can dynamically locate and interact with other components on the internet to provide services.

A Web service is invoked using an XML message such as a SOAP message through a well-defined message exchange pattern. The message exchange pattern is defined in a Web Services Description Language (WSDL) document by a description of the data required to invoke the service.

Messaging Models Used in Web Services

This section describes the two principal messaging models used in Web services. The two Web service messaging models are distinguished by their request-response operation handling.

This section describes the following Web services models:

- [Synchronous Model](#)
- [Asynchronous Model](#)

Synchronous Model

Synchronous means that every time a client accesses a Web service application, the client receives a SOAP response. Synchronous is request-response operation. Synchronous services are designed when client applications require a more immediate response to a request. Web services that rely on synchronous communication are usually Remote Procedure Call (RPC)-oriented. Java™ API for XML-based RPC (JAX-RPC) and SOAP with Attachments API for Java™ (SAAJ) use the synchronous model for communication.

Asynchronous Model

Asynchronous means that the client which invokes a Web service, does not or can not wait for a response. Thus, asynchronous is one-way operation. The client sends a request in the form of an XML message. The Web service receives the message and processes it, sending the results when it completes its processing. An asynchronous send requires a messaging provider which is not supported in Sun ONE Application Server. Asynchronous receive is supported in the application server using the JavaTM API for XML Messaging (JAXM).

Standards and Interoperability in Web Services

Web services are based on a set of standard protocols and technologies, so that all the components of a Web service understand how to communicate. This section describes the following standards.

- [SOAP](#)
- [WSDL](#)
- [UDDI](#)
- [ebXML](#)

SOAP

Simple Object Access Protocol (SOAP) is a lightweight protocol that allows exchange of information in a distributed environment. It plays a very important role in the communication mechanism for Web services. It provides a standard packaging structure for transporting XML documents using a variety of standard internet technologies including SMTP, HTTP, and FTP.

For more information about SOAP 1.1 specification, visit the following URL:

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

WSDL

Web Services Description Language (WSDL) is an XML-based specification schema for describing a Web service. WSDL defines Web services as a set of endpoints or ports operating on messages. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. The message can be either message-style or RPC-style. WSDL is extensible to allow the description of endpoints and their associated messages regardless of what message formats or network protocols are used to communicate.

For more information about WSDL specification, visit the following URL:

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

UDDI

Universal Description, Discovery, and Integration (UDDI) standard provides a mechanism for businesses to describe themselves and the types of services they provide and allows these to register themselves in a UDDI registry. Using SOAP messages, other businesses can search, query, or discover registered businesses. Having discovered other suitable businesses to partner with, businesses can then integrate their services with their partners and provide service to their customers.

For more information about UDDI 2.0 specification, visit the following URL:

<http://www.uddi.org>

ebXML

electronic business eXtensible Markup Language (ebXML) defines core components, business processes, registry and repository, messaging services, trading partner agreements, and security. ebXML defines standards by extending all three of the previous standards to achieve e-business partner interoperability for document exchange. ebXML message service extends SOAP 1.1 with attachment for use as the base messaging protocol to achieve reliability and other quality of service aspects.

ebXML Collaboration Partner Profile and Agreement (CPP and CPA) describe partner interactions for the e-business scenario in a complete manner.

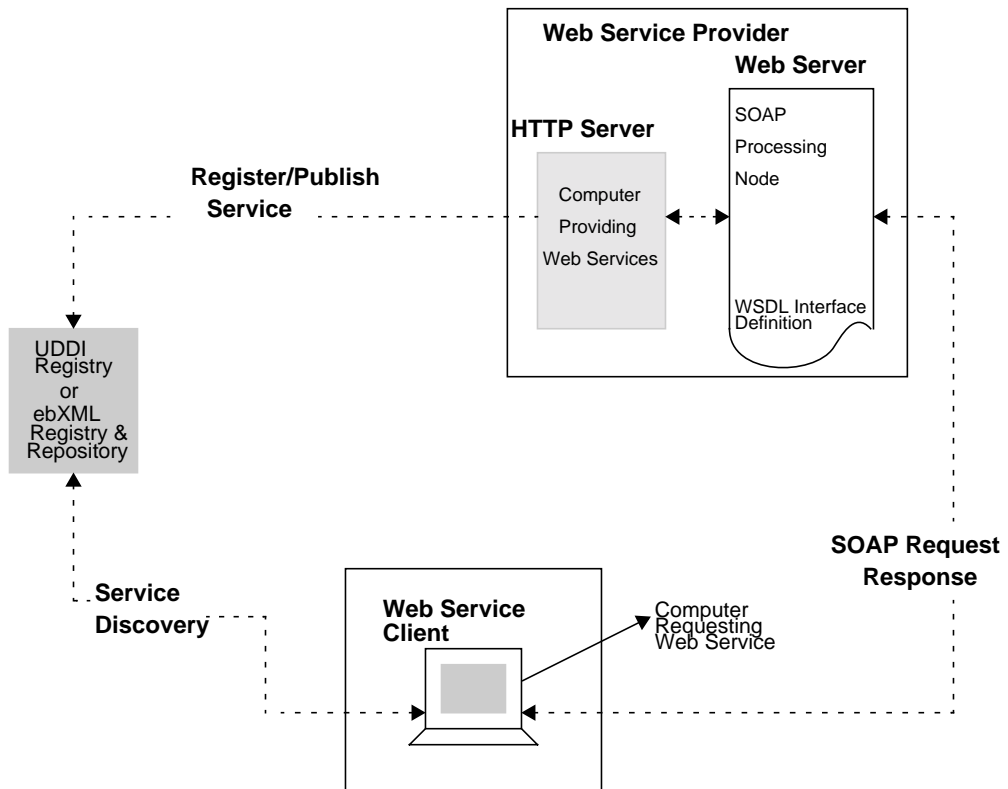
An ebXML Registry and Repository enables the storing and sharing of information between parties to allow e-business collaboration. Sun ONE Application Server supports clients to access an ebXML registry through a third-party provider.

For more information on ebXML, visit the following URL:
<http://www.ebxml.org>

A Simple Web Service Scenario

The following figure illustrates the working of a simple Web service in Sun ONE Application Server environment.

Figure 1-1 Web services in Sun ONE Application Server Environment



The typical working of Web services is explained in the following steps:

1. Once the Web service application is ready to accept requests, the Web service is registered with a registry, such as a UDDI registry or an ebXML registry and repository. Describe the Web service using a WSDL.

2. Another service or a user locates this registered service and requests by querying the registry.
3. The requesting service or a user writes an application to bind the registered service using SOAP.
4. The client discovers the Web service that is registered with the registry.
5. The request from a client to a Web service arrives in the form of an XML document.
6. The Web service receives the request and processes the request.
7. The Web service calls one or more Enterprise JavaBeans (EJBs) components to perform business data processing.
8. The EJB components perform their processing calling external systems.
9. The EJB components return data to the service.
10. The Web service then marshals this return value into an XML document.
11. The Web service returns the XML document to the client on a response.

Support for Web Services in Sun ONE Application Server

Support for Web services in Sun ONE Application Server, is primarily through the implementation of different JAX* APIs. Sun ONE Application Server delivers the runtime environment and tools to develop, deploy, and host RPC and document-oriented Web services. Sun ONE Application Server provides the facility to publish Web services into a UDDI registry and discover the registered services. In addition, Sun ONE Application Server enables integration of reliable messaging into Web services using Sun ONE MQ 3.0.1 (Message Queue).

For more information on the Java APIs, see [“Java APIs for XML and Web Services” on page 21](#).

Sun ONE Application Server includes the following features:

- Sun ONE Application Server includes command-line tools to perform the following tasks:

- Generate client side stubs and server-side tie classes which interface with the application server at runtime. You can generate stubs and tie-classes using the `wscompile` and `wsdeploy` tools provided with Sun ONE Application Server. For more information on using these tools, see “JAX-RPC Tools” on page 63.
- Convert Java interfaces-to-WSDL and vice versa.
- Ant tasks to develop and deploy Web services to Sun ONE Application Server.
- Provides Sun ONE Studio 4, a graphical user interface, that helps in the development and deployment of Web services and clients.

For instructions on how to use Sun ONE Studio to build Web services and clients, see the *Sun ONE Studio Building Web Services* documentation. The Sun ONE Studio documentation is available at the following URL:

<http://docs.sun.com/source/816-7862/index.html>

Unsupported Features in Sun ONE Application Server

Sun ONE Application Server does not support the following features:

- Does not include any registry servers but, is certified with the Sun ONE Registry Server as well as tested with external UDDI registry.
- Does not support any Web services security protocols, instead relies on web container’s security.
- Does not include a messaging provider that enforces reliable messaging between the client and the server.

Java APIs for XML and Web Services

Support for developing Web services on Sun ONE Application Server is primarily based on the implementation of Java APIs for XML and Web services. These APIs provide specific XML and/or SOAP capabilities required to access or deliver Web services from the Java platforms. This section describes each API as delivered in Sun ONE Application Server.

For detailed information on the Java APIs and the programming concepts, visit the following URL:

<http://java.sun.com/webservices/docs/1.0/tutorial/index.html>

The following table lists the Java APIs supported by Sun ONE Application Server. The first column lists the Java APIs and the second column shows the version number of the Java API.

Table 1-1 Java APIs for XML and Web Services supported in Sun ONE Application Server

Java API	Version Number
JAXP	1.1, 1.2
JAX-RPC	1.0
JAXR	1.0
SAAJ	1.1
JAXM	1.1

JAXP

The JavaTM API for XML Processing (JAXP) supports the processing of XML documents using Simple API for XML (SAX) and Document Object Model (DOM), along with a pluggable interface to an XML Stylesheet Language Transformations (XSLT) engine. JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation. Depending on the needs of the application, developers have the flexibility to swap between XML processors, such as high performance vs. memory conservative parsers, without making application code changes. Thus, application and tools developers can rapidly and easily XML-enable their Java applications for e-commerce, application integration, and dynamic web publishing. JAXP 1.2 implementation in Sun ONE Application Server has support for XML schema and an XML compiler (XSLTC).

JAXP 1.2 is required for the other Java APIs for XML and Web services in Sun ONE Application Server. JDK 1.4 bundles an implementation of JAXP 1.1. Sun ONE Application Server bundles JAXP 1.2 implementation. To override the classes in JAXP implementation in JDK 1.4, see “[Overriding the JAXP Implementation](#)” on [page 31](#).

For more information about JAXP, visit the following URLs:

http://java.sun.com/xml/tutorial_intro.html

http://java.sun.com/xml/xml_jaxp.html

JAX-RPC

The Java API for XML-based RPC (JAX-RPC) enables developers to build SOAP based interoperable and portable Web services. JAX-RPC provides an easy to develop programming model for the development of SOAP based synchronous Web services. Developers use the RPC programming model to develop clients and endpoints. For typical scenarios, developers are not exposed to the complexity of the underlying runtime mechanisms, such as SOAP protocol level mechanisms, marshalling, and unmarshalling.

A JAX-RPC runtime system or a library abstracts these runtime mechanisms for programming Web services. A JAX-RPC client can use stubs-based, dynamic proxy, or dynamic invocation interface (DII) programming models to invoke a heterogeneous Web services application. JAX-RPC provides support for document based messaging. Using JAX-RPC, any MIME encoded content can be carried as part of a SOAP message with attachments. This enables exchange of XML document, images, and other MIME types across Web services. JAX-RPC supports HTTP level session management and SSL based security mechanisms. This enables in the development of secured Web services.

Sun ONE Application Server provides support for development and deployment of JAX-RPC Web services and clients. In addition to providing implementation for JAX-RPC API, application server provides tools support for WSDL to Java and Java to WSDL mapping as part of the development of clients and services.

For detailed information on JAX-RPC, visit the following URL:

<http://java.sun.com/xml/jaxrpc/index.html>

JAXR

The JavaTM API for XML Registries (JAXR) provides standard Java API for accessing different kinds of XML registries in a uniform manner. An XML registry is an enabling infrastructure for building, deploying, and discovering Web services.

Currently, there are a variety of specifications for XML registries including pre-eminently, the ebXML Registry and Repository standard, which is being developed by OASIS and U.N./CEFACT and the UDDI specification, which is being developed by a vendor consortium.

JAXR enables Java software programmers to use a single, easy-to-use abstraction API to access a variety of XML registries. Simplicity and ease of use are facilitated within JAXR by a unified JAXR information model, which describes content and metadata within XML registries.

JAXR provides rich metadata capabilities for classification and association, as well as rich query capabilities. As an abstraction-based API, JAXR gives developers the ability to write registry client programs that are portable across different target registries.

Sun ONE Application Server provides implementation for JAXR 1.0 version. This version of the JAXR specification includes detailed bindings between the JAXR information model and both the ebXML Registry and the UDDI Registry v2.0 specifications.

For detailed information on JAXR visit the following URL:

<http://java.sun.com/xml/jaxr/index.html>

SAAJ

The SOAP with Attachments API for Java™ (SAAJ) enables developers to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments note. This API is derived from the `java.xml.soap` package originally defined in the JAXM 1.0 specification.

Sun ONE Application Server provides support for SAAJ API 1.1. For more information on SAAJ 1.1 API, visit the following URL:

<http://java.sun.com/xml/downloads/saaaj.html>

JAXM

Java™ API for XML Messaging (JAXM) defines the API for xml messaging using a messaging provider. JAXM API enables applications to send and receive document-oriented XML messages. JAXM implements SOAP 1.1 with attachments messaging so that developers can focus on building, sending, receiving, and retrieving messages, avoiding programming low level communication routines.

Sun ONE Application Server implements the JAXM 1.1 API that enables applications to send and receive asynchronous messages using a messaging provider.

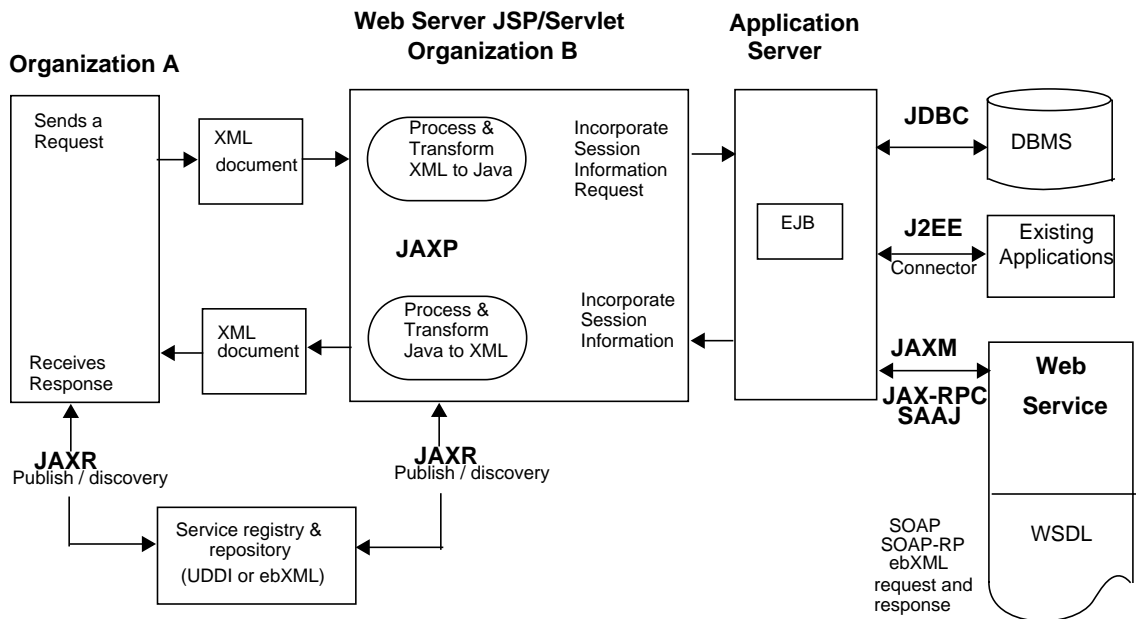
For detailed information on JAXM visit the following URL:

<http://java.sun.com/xml/jaxm/index.html>

Implementing Web Services Using Java APIs

The [Figure 1-2](#) illustrates the role of standard Java APIs in implementing Web services.

Figure 1-2 Implementing Web Services Using Java APIs



SOAP messaging is the protocol for Web services. JAXP API allows you to access and parse XML data. The main goal of JAXP is to provide an interface that lets the you create, manipulate, and use standard XML parsers without reference to the underlying implementation, allowing you to create parser-neutral code, and deferring parser selection to runtime.

JAX-RPC aids you in building XML-based requests such as SOAP requests, used for sending and receiving method calls using XML-based protocols. JAX-RPC API provides the high level framework to expose Java functionality as Web services that can be consumed by SOAP clients or as the way to consuming SOAP services and clients. In the typical JAX-RPC use case, the developer does not have to deal with XML and SOAP programming, thus enabling rapid application development. This not only does XML to Java mapping and vice-versa, also avoids you to interact directly with the XML representation of the call.

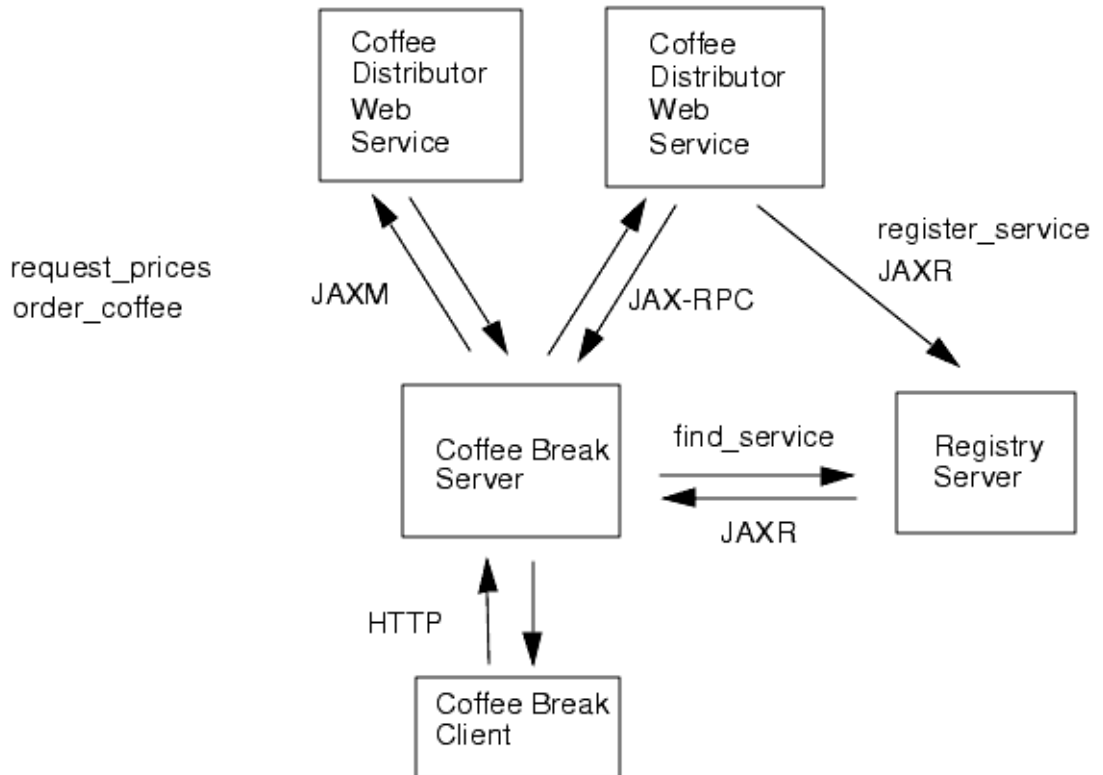
JAXM provides the framework for such XML data to interact between two applications. That is, JAXM allows the transfer of complete XML documents between two separate Web services. SOAP provides the underlying format of messages transferred between the applications. JAXM being a lightweight API, abstracts away the underlying message infrastructure. Hence it is easy to develop JAXM messages packaged via SOAP, which allows you to access SOAP messages quickly and easily.

SAAJ API allows you to manipulate simple SOAP messages. It can be used in combination with JAX-RPC, which is the J2EE standard API for sending and receiving SOAP messages, to represent literal XML document fragments. SAAJ is an integral part of the JAX-RPC, but you can also use it with other APIs such as JAXM.

The difference between JAXM and JAX-RPC is, JAXM supports the development of message-oriented middle ware-type applications, that allows you to focus on sending and receiving messages. Whereas, JAX-RPC supports the application for RPC behavior. JAX-RPC provides the Java interface to XML RPC calls as defined in SOAP.

The Coffee Break Example

The Coffee Break example demonstrates how each of the Java APIs for XML and Web services can be used.

Figure 1-3 Coffee Break Web Service Using Java APIs

The following steps describes the working of the Coffee Break Web service.

- The Coffee Break server obtains the information about the various types of coffees it sells and the prices associated, by querying the distributors at startup and on demand.
- The Coffee Break server uses JAXM messaging to communicate with one of its distributors. It has been dealing with this distributor for some time and has previously made the necessary arrangements for performing request-response JAXM messaging. The two parties have agreed to exchange four kinds of XML messages and have set up the DTDs that these messages will follow.
- The Coffee Break server uses JAXR to send a query searching for coffee distributors that support JAX-RPC to the Registry Server.

- The Coffee Break server requests the price lists from each of the coffee distributors. The server makes the appropriate remote procedure calls and waits for the response, which is a JavaBean component representing a price list. The JAXM distributor returns the price lists as XML documents.
- Upon receiving the responses, the Coffee Break server processes the price lists from the JavaBean components returned by calls to the distributors.
- The Coffee Break Server creates a local database of distributors.
- When an order is placed, suborders are sent to one or more distributors using the distributor's preferred protocol.

The code examples for the coffee break server is installed at *install_dir/samples/webservices/cb*. For more information on using the sample, see the sample application document.

Preparing for Developing Web Services and Clients

This section describes the pre-requisites to develop Web services and clients. This section presents the following topics:

- [Using Ant Tasks](#)
- [Setting Up the Client Environment](#)

Using Ant Tasks

You can use the automated assembly features available through Ant, a Java-based build tool available through the Apache Software Foundation:

<http://jakarta.apache.org/ant/>

Ant is a java-based build tool that is extended using Java classes. Instead of using shell commands, the configuration files are XML-based, calling out a target tree where tasks get executed. Each task is run by an object that implements a particular task interface.

In order to use ant tasks in your client environment, perform the following tasks:

- Include *install_dir/appserver7/bin* in the PATH environment variable. The Ant script provided with Sun ONE Application Server, *asant*, is located in this directory. For details on how to use *asant*, see the sample applications documentation in the

install_dir/appserver7/samples/docs/ant.html

For more information on using Ant tasks in Sun ONE Application Server environment, see the *Sun ONE Application Server Developer's Guide*.

Creating the build.xml File

Ant commands operate under the control of a build file, normally called *build.xml*, that defines the processing steps required.

The *build.xml* file provides several targets that support optional development activities. This build file includes targets for compiling the application, deploying the application to the application server, redeploying the modified application to the application server, and removing old copies of the application to regenerate their content.

For more information on creating an ant build file, see the Apache Ant Manual at:

<http://jakarta.apache.org/ant/manual/index.html>

Setting Up the Client Environment

A client uses various jar files that are bundled with Sun ONE Application Server. This section describes how to setup your client environment:

If you are developing a client application in the system where you have installed Sun ONE Application Server, the required jar files are included to help the development of a client.

If your client development environment is different from that of the system where Sun ONE Application Server is installed, you must perform the following steps:

If you are using version 1.3 of Java 2 SDK, perform the following steps:

1. Copy the following jar files to your client development environment.
 - *mail.jar* - JavaMail API. Installed at *install_dir/share/lib*
 - *activation.jar* - JavaBeans Activation Framework. Installed at *install_dir/share/lib*.
 - *fscontext.jar* - Contains the file system service provider. Installed at *install_dir/share/lib*.

- `jaxm-api.jar` - Java API for XML Messaging. Installed at *install_dir/share/lib*.
 - `jaxrpc-api.jar` - Java API for XML-based RPC. Installed at *install_dir/share/lib*.
 - `jaxrpc-impl.jar` - Java API for XML-based RPC implementation. Installed at *install_dir/share/lib*.
 - `jaxr-api.jar` - Java API for XML Registry. Installed at *install_dir/share/lib*.
 - `jaxr-impl.jar` - Java API for XML Registry implementation. Installed at *install_dir/share/lib*.
 - `saaj-api.jar` - SOAP runtime API. Installed at *install_dir/share/lib*.
 - `saaj-impl.jar` - SOAP implementation. Installed at *install_dir/share/lib*.
 - `commons-logging.jar` - Contains a logging library package. Installed at *install_dir/share/lib*.
 - `jaxp-api.jar` - The `javax.xml.parsers` and `javax.xml.transform` components of JAXP. These packages contain the APIs that give applications a consistent way to obtain instances of XML processing implementations.
 - `sax.jar` - The APIs and helper classes for the Simple API for XML (SAX), used for serial access to XML data.
 - `dom.jar` - The APIs and helper classes for the Document Object Model (DOM), used to create an in-memory tree structure from the XML data.
 - `xercesImpl.jar` - The implementation classes for the SAX and DOM parsers, as well as `xerces`-specific implementations of the JAXP APIs.
 - `xalan.jar` - The "classic" (interpreting) XSLT processor.
 - `xsltc.jar` - The compiling XSLT processor.
2. Add the following jar files to the starting of your classpath. These jar files must appear first in the classpath to avoid using any other parser:
- `jaxp-api.jar`
 - `dom.jar`
 - `sax.jar`
 - `xercesImpl.jar`

- xalan.jar
- xsltc.jar

3. Add the rest of the jar files also to your classpath.

If you are using version 1.4 of Java 2 SDK for developing clients, perform the following step:

Copy all the jar files listed in [Step 1](#), except the following to your client development environment and add them to your classpath.

- jaxp-api.jar
- dom.jar
- sax.jar
- xercesImpl.jar
- xalan.jar
- xsltc.jar

The J2SE 1.4 is the first version of the JDK that bundles an implementation of JAXP 1.1. This allows developers to write applications without having to provide a parser and XSLT processor with their application. However, to override this implementation of JDK with a newer version, you need to use the ‘Endorsed Standards Override Mechanism’.

Overriding the JAXP Implementation

To use the JAXP 1.2 implementation, copy the following jar files into *Java_home/jre/lib/endorsed/* directory:

- dom.jar
- sax.jar
- xercesImpl.jar
- xalan.jar
- xsltc.jar

If the */endorsed* directory does not exist, you must create it.

NOTE The *jaxp-api.jar* file should not be copied, because it contains high-level APIs that are not subject to change.

The jar files must exist in *Java_home*/jre/lib/endorsed/ directory to override earlier versions of the Xalan libraries that are a standard part of the 1.4 platform. Because of that special requirement, it is not possible to specify these libraries using the `-classpath` option on the `java/javac` command line.

Alternatively, you can use the `java.endorsed.dirs` system property to dynamically add those jar files to the JVM when you start your client development. Using that system property gives you flexibility of using different implementations for different applications.

For more information on how to use 'Endorsed Standards Override Mechanism', visit the following URL:

<http://java.sun.com/j2se/1.4/docs/guide/standards>

Services and Clients Using JAX-RPC

This chapter describes the procedure to develop, assemble, and deploy RPC-based Web services in Sun ONE Application Server environment; how to build clients that invoke such services.

This chapter contains the following sections:

- [JAX-RPC Implementation](#)
- [Developing JAX-RPC Web Services](#)
- [Assembling and Deploying JAX-RPC Web Services](#)
- [Invoking JAX-RPC Web Services](#)
- [JAX-RPC Client Invoking an EJB](#)
- [Building Security into JAX-RPC Web Services](#)
- [JAX-RPC Tools](#)
- [Java Language Types Supported By JAX-RPC](#)

JAX-RPC Implementation

Java™ API for XML-based RPC (JAX-RPC) is an API for building Web services and clients that use remote procedure calls (RPC) and XML. The RPC mechanism enables clients to execute procedures on other systems in a distributed environment. In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines envelope structure, encoding rules, and a convention for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages over HTTP. For more information on SOAP messages, see [“SOAP Messages” on page 78](#).

JAX-RPC uses technologies HTTP, SOAP, and the WSDL defined by the World Wide Web Consortium (W3C), which makes it possible for a JAX-RPC client to access a Web service that is not running on the Java platform and vice versa. Sun ONE Application Server implementation of the JAX-RPC API uses HTTP as the transport protocol. The implementation also provides necessary tools to generate stubs, ties, and other artifacts needed on the client-side and the server-side. See [“JAX-RPC Tools” on page 63](#).

Implementation of JAX-RPC in Sun ONE Application Server provides the following benefits to the developers:

- Enables JAX-RPC clients to invoke Web services developed across heterogeneous platforms.
- Developers are not exposed to the complexity of the underlying runtime mechanisms such as, SOAP protocol level mechanisms, marshalling, and unmarshalling. A JAX-RPC runtime system or a library abstracts these runtime mechanisms for the Web services programming model. This simplifies Web service development.
- Provides support for WSDL to Java and Java to WSDL mapping as part of the development of Web service’s endpoints and clients. (A Web service’s endpoint is the address at which the Web service can be reached using a specific protocol or a data format, from where its methods can be invoked.)
- Supports the J2SE SDK classes, application classes that you have written, and JavaBean components. For more information, see [“Java Language Types Supported By JAX-RPC” on page 72](#).
- Enables a Web service endpoint to be developed using the Servlet model. A Web service endpoint is deployed on the application server. These endpoints are described using a WSDL document.
- A JAX-RPC client can use stubs-based, dynamic proxy, or dynamic invocation interface (DII) programming models to invoke a heterogeneous Web service endpoint. See [“Invoking JAX-RPC Web Services” on page 42](#).
- Provides `wscompile` and `wsdeploy` tools to help in the development of Web services and clients. See [“JAX-RPC Tools” on page 63](#).

Developing JAX-RPC Web Services

JAX-RPC Web services are synchronous services which means that, every time a client invokes a JAX-RPC Web services operation, it always receives a SOAP response, even if the method that implements the operation returns void. For more information on the Web services operation, see [“Messaging Models Used in Web Services” on page 16](#).

Web services deployed to Sun ONE Application Server can be accessed by any type of client such as an application client, any J2EE component performing the role of a client, any J2SE-based client, or a .net client.

The following steps describe the procedure to create JAX-RPC Web services using the Java interface and its implementation:

1. Define a class that represents the remote interface to the service; this is the service endpoint interface. This class contains the signature for the methods that a client may invoke on the service. The service endpoint interface extends the `java.rmi.Remote` interface and its methods must throw `java.rmi.RemoteException`. The following code illustrates the creation of a service endpoint interface.

```
package hello;

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

public interface HelloIF extends Remote{

    public String sayHello(String S) throws RemoteException;

}
```

In the code illustration above, the name of the package file is `hello`, and the service definition interface is `HelloIF.java`.

A service endpoint is deployed in a container that implements the JAX-RPC runtime system.

2. Write the service implementation class. The service implementation class is an ordinary Java class. Invocation is done inside the servlet container. The code illustration below shows how to write the service implementation class.

```
package hello;

public class HelloImpl implements HelloIF {

    public String message = "Hello";

    public String sayHello(String S) {
```

```

        return message + S;
    }
}

```

3. In order to handle the communication between the client and the service endpoint, JAX-RPC needs various classes, interfaces, and other files on both the client-side and the server-side. JAX-RPC implementation in Sun ONE Application Server provides the `wscompile` tool to generate these artifacts.

The `wscompile` tool uses the configuration file, `config.xml` to read the interface and implementation class, for generating client-side and server-side artifacts. The `wscompile` tool also creates the WSDL description for the service.

The configuration file of the example is given below:

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="HelloWorld"
    targetNamespace="http://hello.org/HelloWorld.wsdl"
    typeNamespace="http://hello.org/hello/type"

    packageName="hello">
    <interface name="hello.HelloIF"
      servantName="hello.HelloImpl"/>
    </service>
  </configuration>

```

For information about the configuration file, see [“Configuration File” on page 66](#).

For information about the XML schema for creating a configuration file, see [Appendix A, “XML Schema Definitions.”](#)

The following is the syntax to run the `wscompile` tool:

```

wscompile -gen:both -d build/client -classpath build/shared
config.xml

```

Stubs and ties are the most important artifacts that the `wscompile` tool generates. Stubs and ties are the classes that enable the communication between a service endpoint and a client. The stub class sits on the client side, between the service client and the JAX-RPC client runtime system. The stub class is responsible for converting a request from a JAX-RPC service client to a SOAP message and sending it across to the service endpoint using the

specified protocol. It also converts the response from the service endpoint, which it receives in the form of a SOAP message, to the format required by the client. Converting a client request to SOAP format is called *marshalling*; converting back from SOAP format to a client response is *unmarshalling*.

Similarly, the tie class resides on the server side, between the service endpoint and the JAX-RPC runtime system. The tie class handles marshalling and unmarshalling the data between the service endpoint class and the SOAP format. A stub is a local object that acts as a proxy for the service endpoint.

You can use an ant build file (`build.xml`) to compile the service, generate server-side artifacts and create a portable war file. You can find a sample `build.xml` file at the following location:

```
install_dir/samples/webservices/jaxrpc/simple/src
```

For more information on creating a `build.xml` file, see [“Creating the build.xml File” on page 29](#).

4. Assemble and deploy the service to Sun ONE Application Server. See [“Assembling and Deploying JAX-RPC Web Services” on page 38](#).
5. Write the client side application that invokes the service. See [“Invoking JAX-RPC Web Services” on page 42](#).

JAX-RPC Web Services Using a WSDL

You can create a JAX-RPC Web service using an existing WSDL document. In this method, the `wscompile` tool generates the service definition interface for the Web service using the WSDL. The WSDL `portType` is mapped to the Java service definition interface. To generate the service interface from the WSDL, use the `wscompile` command with `-import` option, passing it the location of the WSDL document. Alternatively, you can store the information required to generate the service definition interface in a configuration file by name `config.xml`. The `config.xml`, typically stores the location of the WSDL that you wish to access.

The following `wscompile` command reads the `config.xml` to generate the service definition interface:

```
wscompile -gen:server -import <config.xml>
```

The configuration file with a WSDL document has the following format:

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration
xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
```

```

<wsdl location="[1]"
  packageName="[2]">
  <typeMappingRegistry>[3] </typeMappingRegistry>
</wsdl>

</configuration>

```

The configuration file with a WSDL document has the following attributes:

- `wsdl location` - URL pointing to a WSDL document.
- `packageName` - Specifies a fully qualified name of the Java package for the generated classes/interfaces.
- `typeMappingRegistry` - The type mapping registry used for this service.

For information on the XML schema for creating a configuration file, see [Appendix A, “XML Schema Definitions.”](#)

The code below is the configuration file of the sample and is located at:

install_dir/samples/webservices/jaxrpc/simple

```

<?xml version="1.0" encoding="UTF-8"?>

<configuration
xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">

  <wsdl location="HelloWorld.wsdl"

    packageName="samples.webservices.jaxrpc.simple"/>

</configuration>

```

After you generate the service interface, perform [Step 2](#) to [Step 5](#) under the section, [“Developing JAX-RPC Web Services” on page 35.](#)

Assembling and Deploying JAX-RPC Web Services

A JAX-RPC Web service application can be assembled and deployed to Sun ONE Application Server as a WAR file. A WAR file contains the files needed for the web application in compressed form.

The following steps describe the procedure to assemble and deploy a Web services application to Sun ONE Application Server.

1. **Create the WAR file.** To create a WAR file that contains the service code, create a `build.xml` file, specifying the `create-war` command for the `target-name`. The following code is a sample `build.xml` file that creates a WAR file:

```
<target name="create-war" depends="compile-server"
    description="Packages the WAR file">
    <echo message="Creating the WAR..." />
    <delete file="../${portable-war}" />
    <delete dir="${assemble}/WEB-INF" />
    <copy todir="${assemble}/WEB-INF/classes/">
        <fileset dir="${build}/shared/" includes="**/*.class" />
    </copy>
    <copy file="web.xml" todir="${assemble}/WEB-INF" />
    <copy file="jaxrpc-ri.xml" todir="${assemble}/WEB-INF" />
    <jar jarfile="${assemble}/${portable-war}" >
        <fileset dir="${assemble}" includes="WEB-INF/**" />
    </jar>
    <move file="${assemble}/${portable-war}" todir=".." />
</target>
```

This XML file when executed, bundles the files into a WAR file named `hello-portable.war`. This WAR file is not ready for deployment because it does not contain the tie classes. A WAR (web application archive) file contains a complete web application in compressed form.

A special directory under the document root, `WEB-INF`, contains everything related to the application that is not in the public document tree of the application. No file contained in `WEB-INF` can be served directly to the client. The contents of `WEB-INF` include:

- `/WEB-INF/classes/*`, the directory for servlet and other classes.
- `/WEB-INF/lib/*.jar`, the directory for JAR files containing beans and other utility classes.

- /WEB-INF/web.xml and /WEB-INF/sun-web.xml, XML-based deployment descriptors that specify the web application configuration, including mappings, initialization parameters, and security constraints.

The web application directory structure follows the structure outlined in the J2EE specification.

In the example, the `hello-portable.war` contains the following files:

- WEB-INF/classes/hello/HelloIF.class
 - WEB-INF/classes/hello/HelloImpl.class
 - WEB-INF/jaxrpc-ri.xml
 - WEB-INF/web.xml
2. Define the configuration file that specifies the name of the service and its service endpoint interface and the class. The name of the configuration file must be `jaxrpc-ri.xml`. The following configuration file is the configuration file of the example.

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
    version="1.0"
    targetNamespaceBase="http://hello.org/wsdl"
    typeNamespaceBase="http://hello.org/types"
    urlPatternBase="/ws">
    <endpoint
        name="HelloWorld"
        displayName="HelloWorld Service"
        description="A simple web service"
        interface="samples.webservices.jaxrpc.simple>HelloIF"
        implementation="samples.webservices.jaxrpc.simple.
HelloImpl"/>
    <endpointMapping
        endpointName="HelloWorld"
        urlPattern="/simple"/>
</webServices>
```

The configuration file contains the following `webServices` attributes:

- The `webServices` element includes `name`, `typeNamespace`, and `targetNamespace` attributes.

- The `name` attribute is used to generate the WSDL file for publication in a public registry.
- The `typeNamespace` attribute defines the namespace in WSDL document for types generated by the `wscompile` tool.
- The `targetNamespace` attribute is used for qualifying everything else in the WSDL document.

For information about the XML schema for creating a runtime configuration file, see [Appendix A, “XML Schema Definitions.”](#)

3. Create `web.xml` deployment descriptor file to include the information required for deploying a service, such as mapping the service to an URL, specifying the location of the configuration file in the WAR file, etc. For more information on the deployment descriptors, see the *Sun ONE Application Server Developer's Guide*.

For general information about DTD files and XML, see the XML specification at:

<http://www.w3.org/TR/REC-xml>

The following is the deployment descriptor of the example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Hello World Application</display-name>
    <description>A web application containing a simple JAX-RPC
    endpoint</description>
    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

For information about the XML schema for creating deployment descriptors, see [Appendix A, “XML Schema Definitions.”](#)

Web services applications have a directory structure, all accessible from a mapping to the application's document root (for example, `/hello`).

4. Use the `wsdeploy` tool to create a deployable WAR module. The `wsdeploy` tool executes the `wscompile` tool to generate the stubs, tie classes and other necessary classes.

```
wsdeploy -keep tmpdir tempdir -o hello.war hello-portable.war
```

The `wsdeploy` command when executed, performs the following tasks:

- Reads the `hello-portable.war` file as input
- Gets information from the `jaxrpc-ri.xml` file that's inside the `hello-portable.war` file
- Generates the tie classes for the service
- Generates a WSDL file named `HelloWorld.wsdl`
- Assembles the tie classes, the `HelloWorld.wsdl` file, and the contents of `hello-portable.war` file into a deployable WAR file.

See “[wsdeploy Tool](#)” on page 68 for information on using the `wsdeploy` command-line tool.

5. Use the `asadmin deploy` command to deploy the WAR module.

For example,

```
asadmin> deploy --user admin --password admin --host localhost
--port 4848 --type web --instance server1
/sun/appserver7/samples/webservices/jaxrpc/simple/Hello.war
```

For more information on using the `asadmin` command-line tool, see the *Sun ONE Application Server Developer's Guide*.

Invoking JAX-RPC Web Services

Invoking a Web service essentially refers to the actions that a client application performs to access a Web service. Web services deployed to Sun ONE Application Server can be accessed by any client. That is, any J2EE component within the application server can take the role of a client. Any application or an application client can access Web services. A client can use Apache SOAP libraries to make a call to a Web service or it could be a .net client.

This section describes the procedures to develop JAX-RPC clients that can invoke JAX-RPC Web services deployed to Sun ONE Application Server.

JAX-RPC clients are applications that use the JAX-RPC APIs and runtime for invoking a Web service. These clients import the service using WSDL and can invoke a service that has been defined and deployed on a non-Java platform. JAX-RPC defines the `javax.xml.rpc.Service` interface to model the Web service from a client's perspective. You can use either J2SE or J2EE client programming model to develop JAX-RPC clients.

The main steps in invoking a Web service are listed below:

1. Add the Java client JAR files to the client jar path. For more information on how to add the jar files to the classpath, see [“Setting Up the Client Environment” on page 29](#).
2. Create a Java-based service client.
3. Assemble and deploy your client application. See [“Assembling and Deploying a JAX-RPC Client” on page 51](#).
4. Execute your Java client to invoke the Web service.

You can create JAX-RPC clients using the stubs method, a dynamic proxy, or the call interface method. This section discusses the following topics:

- [Creating Clients Using Generating Stubs Method](#)
- [Creating Clients Using Dynamic Invocation Interface](#)
 - [Creating JAX-RPC Client Using a Dynamic Proxy](#)
 - [Creating a JAX-RPC Client Using the Call Interface](#)
- [Assembling and Deploying a JAX-RPC Client](#)

Creating Clients Using Generating Stubs Method

Stubs are used when a JAX-RPC client knows what method to call and how to call it, such as what parameters to pass. Invoking a remote method through a stub is like invoking a remote method using the Java Remote Method Invocation (RMI) system. A stub simplifies the remote method calls by making them appear like local method calls. A local stub object is used to represent a remote object. To make a remote method call, a JAX-RPC client makes the method call on the local stub.

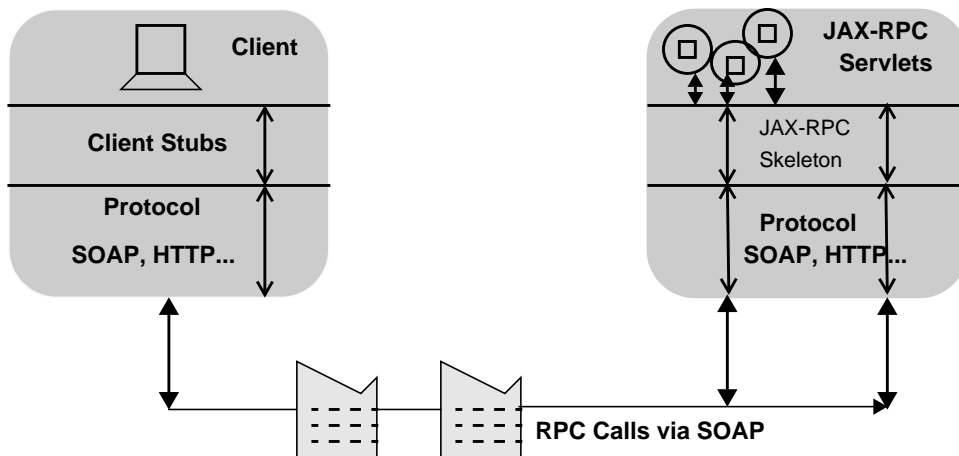
A stub class is a mapping of a port in the WSDL that describes the Web service. It must therefore implement the service definition interface that reflects the methods of the associated `portType`. Thus the client has strongly typed, early-bound access to the Web service endpoint.

The stub must also implement the `javax.xml.rpc.Stub` interface, which provides the facility for the client to configure the stub dynamically.

Typically, a JAX-RPC client performs the following steps. These steps are illustrated in the figure “[JAX-RPC Client Model](#)”.

1. The client calls the stub.
2. The stub redirects the call to the appropriate Web service.
3. The server catches the call and redirects it to a framework.
4. The framework wraps the actual implementation of the service, then calls the Web service on behalf of the client.
5. The framework returns the call to the server.
6. The Web service, in turn, returns the information to the originating client stub.
7. Finally, the client stub returns the information to the client application.

JAX-RPC Client Model



The following sections describe these steps:

- [Generating the Stubs](#)
- [Coding the Client](#)
- [Compiling the Client Code](#)
- [Assembling the Client Classes into a JAR file](#)

- [Running the Client](#)

Generating the Stubs

You can use the `wscompile` tool to generate the stubs for the client. In addition to generating the stubs, the `wscompile` tool also generates the ties for the server. To generate stubs, set the `PATH` to the `install_dir/share/bin` directory. Run the tool using the following syntax:

```
wscompile -gen:client -d build/client -classpath build/shared
config.xml
```

For more information on `wscompile` tool, see [“wscompile Tool” on page 63](#).

There are two ways to generate the stubs. The stubs can be generated from the service endpoint definition or from a WSDL document. The `wscompile` command above uses the service endpoint definition to generate the stubs.

Coding the Client

Make sure to add the necessary jar files to the classpath. For more information, see [“Setting Up the Client Environment” on page 29](#).

The client performs the following steps:

1. Obtain an instance of the interface stub.
2. Set the endpoint property of the stub to point to the service endpoint of the Web service.
3. Call the method.

In this example, the stubs are generated using the service endpoint definition. You provide the configuration information using the JAX-RPC implementation's client-side API `javax.xml.rpc.Stub`.

The following code illustrates the above mentioned steps:

```
package hello;

import javax.xml.rpc.Stub;

public class HelloClient {
    public static void main(String[] args) {
        try {
            HelloIF_Stub stub =
                (HelloIF_Stub)(newHelloWorld_Impl().getHelloIFPort());

            stub._setProperty( javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                args[0]);
```

```

System.out.println(stub.sayHello("Duke!"));
command-line}
catch (Exception ex) {
    ex.printStackTrace();
}
}
}

```

In the code illustration above, the `HelloClient` is a stand-alone program that calls the `sayHello` method of the `HelloWorld` service. It makes this call through a stub, a local object that acts as a proxy for the remote service. In the code listing, note the names of the `HelloIF_Stub` and `HelloWorldImpl` classes, which were generated by the `wscompile` tool. The `HelloIF` prefix matches the name of the service definition interface and the `HelloWorld` prefix corresponds to the service name specified in the configuration file. The `HelloWorldImpl` class is the implementation of a service as described in the JAX-RPC specification. The client gets a reference to the stub by calling the `getHelloIF` method of the `HelloWorldImpl` class, which was created when you ran the `wscompile` tool.

The `args[0]` parameter of the `stub._setProperty` method is a URI that denotes the address of the target service port.

Compiling the Client Code

NOTE Be sure to run the `wscompile` and the `wsdeploy` tools before you compile the client code. The client code refers to the generated by the `wscompile` tool.

```
wscompile -gen:client -d build/client -classpath
build/shared config.xml
```

To compile the client, go to the directory where you have the client code saved and type the following command:

```
asant compile
```

This command compiles the Java source code.

Assembling the Client Classes into a JAR file

You can use the `asant` tool to assemble the client classes into a JAR file. `asant` is a command-line interface tool. Type the following command:

```
asant jar
```

This command creates the client jar file.

Running the Client

Use the `asant` tool to run the client. Type the following command:

```
asant run
```

The `run` target of `asant` executes this command:

```
java -classpath cpath client endpoint
```

cpath - The classpath includes the client jar file that you have created, as well as several other JAR files that are part of the JAX-RPC implementation.

NOTE In order to run the client remotely, all of these JAR files must reside on the remote client system.

endpoint- `http://localhost:8080/jaxrpc-hello/jaxrpc>HelloIF`

The `jaxrpc-hello` portion of the URL is the context of the servlet that implements the `HelloWorld` service. This portion corresponds to the prefix of the `jaxrpc-hello.war` file. The `jaxrpc` string matches the value of the `<url-pattern>` element of the `sun-web.xml` deployment descriptor. And finally, `HelloIF` is the name of the interface that defines the service.

You can accomplish the task of compiling, assembling and deploying, and running a client through a `build.xml` file. The `build.xml` file for the sample bundled with Sun ONE Application Server is available at the following location:

```
install_dir/samples/webservices/jaxrpc/simple/src
```

Creating Clients Using Dynamic Invocation Interface

Using Dynamic Invocation Interface (DII), a client can call a service or a remote procedure. The client can discover the name of the service or the procedure at runtime, making use of a service broker that can dynamically look up the service and its remote procedures.

The `javax.xml.rpc.Service` encapsulates two types of dynamic invocation that do not require any generated code. This section describes the procedure to create dynamic clients.

- [Creating JAX-RPC Client Using a Dynamic Proxy](#)
- [Creating a JAX-RPC Client Using the Call Interface](#)

Creating JAX-RPC Client Using a Dynamic Proxy

A JAX-RPC client can interact with a Web service using a dynamic proxy. A dynamic proxy is a class that dynamically supports service endpoints at runtime, without having pre generated stubs. A client creates dynamic proxies by calling the `getPort()` method of the `javax.xml.rpc.Service` interface. The client calls its `getPort()` method, passing in the Java service definition interface and the corresponding Web service port name. It passes back a dynamically built and configured implementation of the service definition interface—a dynamically built stub.

For more information on dynamic proxies, visit the following URL:

<http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>

The steps given below explain the procedure to create a dynamic proxy client.

1. Make sure to add necessary jar files to the classpath. For more information on adding jar files to the classpath, see [“Setting Up the Client Environment” on page 29](#).
2. Create a client class that uses a dynamic proxy to invoke the service.

```
public class HelloClient {
    ....
    ....
}
```

3. Define the name of the service, the port name and the name of the WSDL that contains the information about the Web service you wish to access.

```
String urlString = endpoint;
String namespaceUri = "http://proxy.org/wsd/HelloWorld";
String serviceName = "HelloWorld";
String portName = "HelloIFPort";
URL helloWsdUrl = new URL(urlString);
```

4. Obtain an instance of the default implementation for `ServiceFactory` object.

```
ServiceFactory serviceFactory = ServiceFactory.newInstance();

Service helloService = serviceFactory.createService(helloWsdUrl,
    new QName(namespaceUri, serviceName));
```

5. Get a dynamic proxy for the object.

```
HelloIF myProxy =(HelloIF)helloService.getPort( new
QName(namespaceUri,portName), proxy>HelloIF.class);
```

In the code illustration, the `getPort()` method is passed in an interface definition that will be used as a template for building a runtime instance of a dynamic proxy.

6. Invoke the service using

```
java.lang.reflect.InvocationHandler.Invoke();

System.out.println(myProxy.sayHello("Buzz"));
```

Creating a JAX-RPC Client Using the Call Interface

In the Call Interface approach, a client dynamically discovers services, configures the remote calls, and executes the calls. The client uses the `javax.xml.rpc.Call` interface for the dynamic invocation of a JAX-RPC service. At runtime, the client uses the DII to call remote procedures on the Web service.

DII `Call` object method supports two types of invocation, namely, synchronous request-response and one-way mode. In the synchronous request-response mode, the client uses the `invoke` method of the call object to make a remote method. The client then waits until the operation is complete, that is, until a response is returned. In one-way method, the client uses the `invokeOneWay` method of the call object to make a remote call.

The steps given below explains the procedure to create a client that can invoke a Web service using the call interface approach:

1. Make sure to include the necessary jar files to the classpath. For more information, see [“Setting Up the Client Environment” on page 29](#).
2. When you create a dynamic client, define the name of the service that you wish to access and the port name. Then, you create a service factory using the `ServiceFactory.newInstance()` method. The `ServiceFactory.newInstance()` method is supported by the JAXR API to define a service. For more information, see [“Adding Services and Service Bindings to an Organization” on page 111](#).

```
private static String qnameService = "HelloWorld";
private static String qnamePort = "HelloIF";

ServiceFactory factory = ServiceFactory.newInstance();

Service service = factory.createService(new
QName(qnameService));
```

3. Create a Service object from the factory.

```
Service service = Factory.createService(new
QName(qnameService));
```

4. Create a Call object from the service and pass the name of the port and the operation you want to execute.

```
QName port = new QName(qnamePort);

Call call = service.createCall();
call.setPortTypeName(port);
call.setTargetEndpointAddress(endpoint);
```

5. Set the property prior to making the actual method call. The setProperty method is used to set standard properties that are listed in the JAX-RPC specification.

```
call.setProperty(Call.SOAPACTION_USE_PROPERTY, new
Boolean(true));

call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);
QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
call.setReturnType(QNAME_TYPE_STRING);
```

6. Set the operation name.

```
call.setOperationName(new
QName(BODY_NAMESPACE_VALUE, "sayHello"));
call.addParameter("String_1", QName_TYPE_STRING,
ParameterMode.IN);
String[] params = { new String("Duke!") };
```

The `addParameter` method is used to add a parameter and the type for the operation specified in the `setOperationName` method. Note that the values of these parameters are obtained from the WSDL document for the service.

7. Use the `Call.invoke()` method to invoke the service.

```
String result = (String)call.invoke(params);
```

The `invoke` method invokes the operation specified in the `setOperationName` method using a synchronous request-response interaction mode. The method call specifies the input parameters for the invocation.

Creating a JAX-RPC Client Using a WSDL

The following steps describes the procedure to create a dynamic client that uses the WSDL to locate a Web service and invoke the service.

1. **Create a service factory using the `ServiceFactory.newInstance()` method.**

```
ServiceFactory serviceFactory = ServiceFactory.newInstance();
```
2. **Create a service object from the factory. Pass the name of the WSDL.**

```
String namespaceUri = "http://hello.org/wsdl";
URL helloWsdURL = new URL(namespaceUri);
```
3. **Create a `serviceName` object and pass the name of the service that you wish to invoke.**

```
String serviceName = "HelloWorld";
```
4. **Create a `portName` object and specify the port name.**

```
String portName = "HelloIFport";
```
5. **Create an `OperationName` object and specify the name of the operation in the service that you wish to execute.**

```
String operationName = "sayHello";
```
6. **Create a service, passing it the WSDL location and the name of the service that you want to invoke.**

```
Service helloService =
    serviceFactory.createService(helloWsdURL, new
        QName(namespaceUri, serviceName));
```
7. **Create a `Call` object, pass it the name of the port and the operation that you want to execute.**

```
Call call = helloService.createCall(portName, operationName);
```
8. **Invoke the service using the `Call.invoke()` method.**

```
String result = String call.invoke(helloService);
```

Assembling and Deploying a JAX-RPC Client

JAX-RPC Clients can be bundled into a deployable WAR file using the `wsdeploy` command tool. The `wsdeploy` command reads the JAX-RPC runtime descriptor `jaxrpc-ri.xml` file and the web application deployment descriptor `web.xml` file. Assembling and deploying a JAX-RPC client involves the following steps:

1. **Create the JAX-RPC runtime descriptor file. The name of the file must be `jaxrpc-ri.xml`. See [“The jaxrpc-ri.xml File” on page 69](#).**

2. JAX-RPC client is a web module. Create a web module deployment descriptor `web.xml`. For information on `web.xml` file, see the *Sun ONE Application Server Developer's Guide to Web Applications*.
3. Use the `wsdeploy` command tool to create a deployable WAR file. For information about `wsdeploy` command tool, see [“wsdeploy Tool” on page 68](#).
4. Deploy the WAR file using `asadmin deploy` command.

You can accomplish the task of assembling and deploying, and running a JAX-RPC client through an ant `build.xml` file. The `build.xml` file for the samples are bundled with Sun ONE Application Server which is available at the following location:

`install_dir/samples/webservices/jax-rpc/simple/src`

Sample Applications

- `install_dir/samples/webservices/jaxrpc/proxy` - contains a sample dynamic proxy client application that illustrates the basics of creating, deploying, and accessing a Web service.
- `install_dir/samples/webservices/jaxrpc/dynamic` - contains a dynamic invocation interface client that illustrates the basics of creating, deploying, and accessing a Web service.

JAX-RPC Client Invoking an EJB

This section describes the procedure to create a stand-alone JAX-RPC client that makes a remote call on a JAX-RPC service. This service locates a stateless session bean and invokes a method on the bean.

NOTE These instructions apply to the development of JAX-RPC services only in the J2EE 1.3.1 environment.

The main steps to invoke an EJB are listed below:

1. Create a stateless session bean. See the *Sun ONE Application Server Developer's Guide to Enterprise Java Beans* for detailed instructions on creating a stateless session bean.

2. Create a JAX-RPC Web service that performs a lookup on the EJB. The following code illustrates how a Web services application can make a remote call on the EJB:

```
public String sayHello(String name) {
    Context initial = new InitialContext();
    Context myEnv = (Context)initial.lookup("java:comp/env");
    Object objref = myEnv.lookup("ejb/SimpleGreeting");
    GreetingHome home =
        (GreetingHome)PortableRemoteObject.narrow(objref,GreetingHome.cl
        ass);
}
```

3. Create a stand-alone client that makes a remote call on the JAX-RPC service. The following code is an example of the client that makes a remote call on the EJB.

```
package samples.webservices.jaxrpc.toejb.client;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import samples.webservices.jaxrpc.toejb.ejb.*;

public class GreetingClient {

    public static void main(String[] args) {
        try {

            Context initial = new InitialContext();
            Context myEnv = (Context)initial.lookup("java:comp/env");
            Object objref = myEnv.lookup("ejb/SimpleGreeting");

            GreetingHome home =
                (GreetingHome)PortableRemoteObject.narrow(objref,GreetingHome.cl
                ass);

            Greeting salutation = home.create();

            System.out.println(salutation.sayHey("Buzz"));

            System.exit(0);

        } catch (Exception ex) {

            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }

        // main
    }
}
```

You can find the complete code listing for the sample at:

install_dir/samples/webservices/jax-rpc/toejb/src/

4. Assemble the service and the client. See [“Assembling and Deploying JAX-RPC Web Services” on page 38](#) and [“Assembling and Deploying a JAX-RPC Client” on page 51](#).
5. Deploy the session bean by performing the following steps:
 - Edit the deployment descriptor files (`ejb-jar.xml` and `sun-ejb-jar.xml`).
 - Execute an Ant build command (such as `build jar`) to reassemble the JAR module.
 - Use the `asadmin deploy` command to deploy the JAR module. For example, the following command deploys an EJB application as a stand-alone module:

```
asadmin deploy --type ejb --instance inst1 myEJB.jar
```
6. Deploy the JAX-RPC service. See [“Assembling and Deploying JAX-RPC Web Services” on page 38](#).
7. Run the JAX-RPC client using the `asant` command.

```
asant run
```

Building Security into JAX-RPC Web Services

This section describes the procedure to provide security to a JAX-RPC service application, by providing security to the web container that contains the application, using HTTP/SSL for basic and mutual authentication. For more information on authentication, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

This section presents the following topics:

- [Basic Authentication Over SSL](#)
- [Adding Security Elements to web.xml](#)
- [Setting Security Properties in the Client Code](#)
- [Mutual Authentication Over SSL](#)
- [Setting Up Client Certificate Authentication for Web Services](#)

You must perform the following steps to configure a JAX-RPC Web service endpoint for HTTP/S basic and mutual authentication:

- Use `keytool`, which is part of the J2SE SDK, to generate certificates and keystrokes.
- Add security elements to `sun-web.xml` deployment descriptor.
- Add some properties in the client code.
- Build and run the Web service.

Basic Authentication Over SSL

Follow the steps below to configure a Web service for basic authentication over HTTP/S:

1. Configure a certificate and enable SSL on HTTP listener for your server. For more information on configuring a certificate and enabling SSL on HTTP, see “Administering Certificates” and “Turning Security On” sections respectively, in the *Sun ONE Application Server Administrator’s Guide to Security*.

A HTTP client uses a repository of trusted Certificate Authorities (CA) during the SSL handshake to validate server certificate. It is important that the CA of your server certificate be a trusted CA for the client.

2. For the J2SE 1.4 based clients which includes JSSE based clients, such as Web services applications, you need to import the certificate of your server’s CA into JSSE cacerts database. Use the command line tool, the `keytool` to import the trusted CA certificate.

The following code illustrates how you can import the certificate of your CA into the trusted CA database of your J2SE-based client:

```
keytool -import -v -alias "CMS-CA" -file cmsca.cer -keystore
cacerts
```

For more information on using `keytool`, run `keytool` command with `-help` option or visit the following URL:

```
http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/keyto
ol.html
```

3. Enter keystore password: changeit

Owner: CN=Certificate Manager, OU=AppServices, O=Sun Microsystems, L=SCA, ST=California, C=US

Issuer: CN=Certificate Manager, OU=AppServices, O=Sun Microsystems, L=SCA, ST=California, C=US

Serial number: 1

Valid from: Mon Jun 03 12:00:00 PDT 2002 until: Thu Jun 03 12:00:00 PDT 2004

Certificate fingerprints:

MD5: 6C:8D:A6:E4:55:52:1A:FF:9D:19:44:D7:0F:62:66:95

SHA1:89:B1:0E:7E:8F:56:B2:34:65:46:15:86:53:7E:3E:6B:4F:9D:84:63

Trust this certificate? [no]: yes

Certificate was added to keystore

[Saving cacerts]

4. Configure the server instance to use appropriate realm and make sure the realm has users you would like to permit the Web services access.

For Example, to set up the flat file of users, follow the steps below in the Administration interface:

- Select the server instance and click on the Security node in the left pane.
- Select the drop-down box for the Default Realm in the right pane and choose the option “file”.
- Select the Realms in the left pane and click on the file realm to add the users to the file realm.
- Apply your changes. Now, the server’s flat file user database is ready to use.

For detailed information on Configuring the server instance to use the realm, see the *Sun ONE Application Server Administrator’s Guide*.

Adding Security Elements to web.xml

Enable Basic Authentication for the web application and specify a security constraint to enforce authentication. For more information on security elements in `web.xml`, see the *Sun ONE Application Server Developer’s Guide to Web Applications*.

Here is an example of how you can configure the Basic Authentication for the Web service servelt-based end point. This security-constraint allows principals with the role “ServiceUser” which is mapped to user “bob” in the `sun-web.xml`.

The `WEB-INF/web.xml` :

```
<security-constraint>

    ....
    <web-resource-collection>
        <web-resource-name>SecureHello</web-resource-name>
        <url-pattern>/security</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>

</security-constraint>

<login-config>

    <auth-method>BASIC</auth-method>

</login-config>
```

The `WEB-INF/sun-web.xml`:

```
<sun-web-app>
<security-role-mapping>

<role-name>ServiceRole</role-name>
<principal-name>bob</principal-name>

</security-role-mapping>
</sun-web-app>
```

5. Set the security properties for the J2SE-based client. For step-by-step instructions, see [“Setting Security Properties in the Client Code”](#) on page 57.

Setting Security Properties in the Client Code

For basic authentication over SSL, the client code must set several security-related properties.

trustStore Property

The client specifies the `trustStore` property as follows:

```
System.setProperty("javax.net.ssl.trustStore", trustStore);
```

trustStorePassword Property

The `trustStorePassword` property is the password of the J2SE SDK keystore. In the previous section, you specified the default password `changeit` when running `keytool`. The client sets the `trustStorePassword` property in the following code illustration:

```
System.setProperty("javax.net.ssl.trustStorePassword",  
trustStorePassword);
```

Username and Password Properties

The *username* and *password* values correspond to the manager role. The client sets the `username` and `password` properties as follows:

```
stub._setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY, username);  
stub._setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY, password);
```

Mutual Authentication Over SSL

To configure and create a JAX-RPC service with mutual authentication, follow all of the steps in the [“Basic Authentication Over SSL” on page 55](#). Then, follow these steps:

1. Export the generated client certificate.

The following code illustrates how to export the client certificates.

```
$Java_home/bin/keytool -export -alias -client -storepass changeit  
-file client.cer -keystore client.keystore
```

2. Import the client certificate into the server's keystore.

```
$Java_home/bin/keytool -import -v -trustcacerts -alias -client  
-file client.cert -keystore server.keystore -keypass changeit  
-storepass changeit
```

3. Run the application using `asant` tool.

```
asant run
```

Setting Up Client Certificate Authentication for Web Services

This section describes how you can configure a Web service that uses Client Certificate Authentication with Sun ONE Application Server.

NOTE For clients, using HTTPS, follow the steps 1-6 given below.

1. Follow the steps 1 and 2 described under “[Basic Authentication Over SSL](#)” on [page 55](#).
2. For J2SE-based clients, generate a key-pair for the client certificate using the `keytool`. The key-pairs are stored in a keystore. The following code line illustrates how to generate a key-pair:

```
> keytool -genkey -v -alias clcert -dname 'CN=Test User,
OU=testOU, O=testO, L=SCA, S=California, C=US'\
-keystore clcerts -keypass changeit
```

The command execution, displays the following information:

```
Generating 1,024 bit DSA key pair and self-signed certificate
(SHA1WithDSA)

for: CN=Test User, OU=testOU, O=testO, L=SCA, ST=California, C=US

Enter key password for <clcert1>

(RETURN if same as keystore password):
[Saving clcerts]
```

3. Generate a certificate request to be sent to a trusted CA. Use the following `keytool` command to generate a certificate request:

```
>keytool -certreq -v -alias clcert -file clreq -keystore clcerts
```

Enter keystore password: changeit

Certification request stored in the file: clreq

Submit this to your CA

```
>more clreq
```

This command generates the following message:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
```

```
MIICazCCAigCAQAwZTElMAkGA1UEBhMCVVMxEzARBgNVBAGTCkNhbGlm3JuaWExDDAKBgNVBACTA1NDQTEOMAwGA1UEChMFdGVzdE8xDzANBgNVBASTBnRlc3RPVTEsMBAGA1UEAxMJVGZvdCBVc2Vy
```

```
..
```

```
GqdqJvx+mVcgcQkdzap+ykIEcOZ/qQq60rPddf6yK9wnWkez32Y2btGWe598oAAwCwYHKoZIZjgE
AwUAAzAAMC0CFQCFf7TuJLJ+zS/HH+fCcKnFAtmKYwIUZM10c56KrScgledImxGzLIKMsGE=
```

```
-----END NEW CERTIFICATE REQUEST-----
```

4. Send the generated certificate request to a trusted CA and request a signed certificate. For more information on sending the request to a trusted CA, see *CA Server Administration* documentation.

You may verify the returned signed certificate in Base64 encoded X.509 format (saved for example in `calcert.txt` file). Note that this certificate is chained as it is signed by the CA. If you use a pkcs7 format, the `keytool` may complain that the certificate is not in X.509 format.

```
> keytool -printcert -v -file clcert.txt
```

```
Certificate[1]:
```

```
Owner: CN=Certificate Manager, OU=AppServices, O=Sun
Microsystems, L=SCA, ST=California, C=US
```

```
Issuer: CN=Certificate Manager, OU=AppServices, O=Sun
Microsystems, L=SCA, ST=California, C=US
```

```
Serial number: 1
```

```
Valid from: Mon Jun 03 12:00:00 PDT 2002 until: Thu Jun 03
12:00:00 PDT 2004
```

```
Certificate fingerprints:
```

```
MD5: 6C:8D:A6:E4:55:52:1A:FF:9D:19:44:D7:0F:62:66:95
```

```
SHA1:
```

```
89:B1:0E:7E:8F:56:B2:34:65:46:15:86:53:7E:3E:6B:4F:9D:84:63
```

```
Certificate[2]:
```

```
Owner: CN=Test User, OU=testOU, O=testO, L=SCA, ST=California,
C=US
```

```
Issuer: CN=Certificate Manager, OU=iWSQA, O=Sun Microsystems,
L=SCA, ST=California, C=US
```

```
Serial number: 19
```

```
Valid from: Thu Sep 12 18:33:54 PDT 2002 until: Fri Sep 12
18:33:54 PDT 2003
```

```
Certificate fingerprints:
```

```
MD5: 82:09:8A:DC:E2:85:82:B5:56:98:93:81:97:A9:D5:32
SHA1:
1D:7C:F2:F2:ED:79:A3:62:0A:A2:1B:22:74:11:BF:52:CB:8D:9E:BB
```

5. Update the keystore with the signed client certificate.

```
> keytool -import -v -trustcacerts -alias clcert -file clcert.txt
-keystore clcerts -keypass changeit
```

Displays the following message:

```
Certificate was added to keystore
[Saving clcerts]
```

6. Now, you must set up the client Java virtual machine to use this keystore/password: `-Djavax.net.ssl.keyStore=<path-to/clcerts>` and `-Djavax.net.ssl.keyStorePassword`.

If you are running the client within `<java>` ant target, you can use the `<sysproperty>` element to specify the keystore/password.

For example:

```
<sysproperty key="javax.net.ssl.keyStore"
value="C:/security/clcerts"/>
<sysproperty key="javax.net.ssl.keyStorePassword"
value="changeit"/>
```

7. Enable the certificate realm for the server instance.

- In the Administration interface, select the server instance in the left pane.
- Click the Security node and choose Default Realm to Certificate.

For more information on enabling the certificate alarm, see the *Sun ONE Application Server Administrator's Guide*.

8. Edit the `web.xml` deployment descriptor to configure the Web service application to use CLIENT-CERT authentication.

```
security-constraint>
<web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
    <url-pattern>/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
</web-resource-collection>
<auth-constraint>
    <role-name>TesterRole</role-name>
</auth-constraint>
</security-constraint>
<login-config>
    <auth-method> CLIENT-CERT </auth-method>
</login-config>
```

9. Edit the Sun ONE Application Server specific deployment descriptor (`sun-web.xml`) to map the role to the X.509 principal name DN of the client certificate.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Sun
ONE Application Server 7.0 Servlet 2.3//EN"
'http://www.sun.com/software/sunone/appserver/dtds/sun-web-app_2
_3-0.dtd'
<sun-web-ap>
<security-role-mapping>
    <role-name>TesterRole</role-name>
    <principal-name>CN=Test User, OU=testOU, O=testO, L=SCA,
ST=California, C=US</principal-name>
</security-role-mapping>
</sun-web-app>
```

10. Alternatively, you may want to set an optional `assign-groups` property that can be set for the certificate realm configuration to which all certificate users belong. This will allow you to do the role-mapping to this group name, instead of having to list the principal name DN's.

- In the Administration interface, select the server instance in the left pane.
- Click and expand the Security node and the Realms node.
- Click on the certificate realm and on the right pane, click on the properties link to add the property names `cert-users` with the value *assign-groups*.
- Save and Apply the changes. The `server.xml` reflects the configuration settings:

```
<security-service>

<auth-realm name="certificate"
classname="com.ipplanet.ias.security.auth.realm.certificate
.CertificateRealm">

<property value="cert-users" name="assign-groups"/>

</auth-realm>

</security-service>
```

For more information on configuring the certificate realm, see the *Sun ONE Application Server Administrator's Guide*.

11. Restart the server and run the client to verify the working of client certificate authentication.

JAX-RPC Tools

JAX-RPC implementation of Sun ONE Application Server includes the following tools that helps in the development of JAX-RPC clients.

- [wscompile Tool](#)
- [wsdeploy Tool](#)

wscompile Tool

`wscompile` is a mapping tool that is bundled with Sun ONE Application Server which generates stubs, ties, serializers, and other artifacts. You can also use this tool to generate a WSDL document from the service endpoint definition or produce a service endpoint definition from the WSDL document.

You have the option of generating only the client-side artifacts such as stubs, server-side artifacts such as ties, or both client and server-side artifacts. The tool reads the configuration file that contains information needed to generate the artifacts.

The syntax of the `wscompile` command is as follows:

```
wscompile [options] config_file_name
```

wscompile Command Options

The following table lists the options that you can use with the `wscompile` command. The first column specifies the option that you can use with the command, and the second column describes the option.

Table 2-1 wscompile Tool Options

Option	Description
<code>-gen:client</code>	Generates client-side artifacts such as stubs, service interface, implementation classes, and remote interface.
<code>-gen:server</code>	Generates server-side artifacts such as ties, server configuration file, WSDL file, service definition interface. If you are using <code>wsdeploy</code> tool, you must not use this option.
<code>-gen:both</code>	Generates both client and server-side artifacts at the same time.
<code>gen</code>	Same as <code>gen:client</code> .
<code>-define</code>	Defines a service.
<code>-classpath<classpath_string></code>	Specify the path of input class files.
<code>cp<classpath_string></code>	Same as <code>classpath</code> .
<code>-d<directory_name></code>	Sets the output directory for all generated files.
<code>-s<directory></code>	Specifies the path where the generated files will be stored.

Table 2-1 wscompile Tool Options

Option	Description
<code>-f:<features>/-features:<features></code>	Enable the listed features. Features are separated by comma. List of features supported are: <ol style="list-style-type: none"> 1. <code>datahandleronly</code> - always map the attachments to data handler types 2. <code>explicitcontext</code> - turn on explicit service mapping context 3. <code>infix=<name></code> - Specify an infix to use for generated serializers 4. <code>nodatabindings</code> - turn off data bindingss for literal encoding 5. <code>noencodedtypes</code> - turn off encoding type information 6. <code>nomultirefs</code> - turn off support for multiple references 7. <code>novalidation</code> - turn off validation for imported WSDL file 8. <code>searchschema</code> - search schema aggressively for subtypes 9. <code>serializeinterfaces</code> - turn on direct servialization of interface type
<code>-g</code>	Generates debugging info.
<code>-httpproxy:<host>:<port></code>	Specify HTTP proxy server.
<code>-import</code>	Generate interfaces and value types.
<code>-keep</code>	Keep the generated .java files after the compilation is complete.
<code>-model<file></code>	Write internal model to the file.
<code>-nd<directory></code>	Specify the path to store non class generated files.
<code>-O</code>	Optimize the generated code.
<code>-verbose</code>	Output messages about the compiler action.
<code>-version</code>	Print version information.

You must use the `wscompile` command with one of the `-import`, `-gen`, and `-define` options. Invoking `wscompile` command with no options will display the usage information.

Configuration File

The `wscompile` tool reads the configuration file, which contains information that describes the Web service. The basic structure of the configuration file is given below:

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration

    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">

    <service> or <wsdl> or <modelfile>

</configuration>
```

If you use the `<service>` element in your configuration file, the `wscompile` tool reads the RMI interface that describes the service and generates a WSDL file.

If you use the `<wsdl>` element in your configuration file, the `wscompile` tool reads the service's WSDL file and generates the service's RMI interface.

If your configuration file contains a `<service>` or `<wsdl>` element, the `wscompile` tool generates a model file that contains the internal data structure that describe the service. If you have already generated a model file in this manner, then you can reuse it the next time you run the `wscompile` tool.

For information on the XML schema to create a configuration file, see [Appendix A, "XML Schema Definitions."](#)

Configuration File with RMI Interfaces

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration
xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">

<service name="[1]"

    targetNamespace="[2]"
    typeNamespace="[3]">
        packageName="[4]">

            <interface name="[5]"

                servantName="[6]"
                soapAction="[7]"
                soapActionBase="[8]"/>

            <typeMappingRegistry> [9] </typeMappingRegistry>

        </service>

</configuration>
```

The configuration file contains the following Web services attributes:

- `servicename` - This attribute is used to generate a properties file that the servlet-based JAX-RPC runtime uses for dispatching the request to tie-and-servant combination.
- `targetNamespace` - This attribute specifies the target name space for the generated WSDL document.
- `typeNameSpace` - This attribute specifies the target name space for the schema portion of the generated WSDL document.
- `packageName` - Specifies the package name for the generated Java classes. For example, the service interface extending `javax.xml.rpc.Service`.
- `interface name` - Specifies the fully qualified name of a Java interface.
- `servantName` - Specifies the fully qualified name of a servant class.
- `soapAction` - String used as the `SOAPAction` for all operations in the corresponding port. This is optional.
- `soapActionBase` - String used as a prefix for the `SOAPAction` strings for the operations in the corresponding port.
- `typeMappingRegistry` - Specifies the type mapping information.

NOTE One generic servlet class `com.sun.xml.rpc.server.http.JAXRPCServlet` is used for all JAX-RPC endpoints.

Configuration file with a WSDL document

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration
xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">

  <wsdl location="[1]"
    packageName="[2]">
    <typeMappingRegistry>[3] </typeMappingRegistry>
  </wsdl>
</configuration>
```

The configuration file with a WSDL document has the following attributes:

- `wsdl location` - URL pointing to a WSDL document.

- `packageName` - Specifies a fully qualified name of the Java package for the generated classes/interfaces.
- `typeMappingRegistry` - type mapping information.

wsdeploy Tool

The `wsdeploy` tool generates a deployable WAR file for a service. This tool takes as input a WAR file for the service.

Syntax of the `wsdeploy` command is as follows:

`wsdeploy [options] war file`

wsdeploy Command Options

The following table lists the options that you can use with the `wsdeploy` command. The first column lists the options, and the second column describes the option.

Table 2-2 wsdeploy Tool Options

Option	Description
<code>-classpath<classpath_string></code>	Specify the path of input class files.
<code>cp<classpath_string></code>	Same as <code>classpath</code> .
<code>-tmp<directory_name></code>	Specify the path of the temporary directory.
<code>-o<output war file></code>	Specify the path where the generated WAR file will be stored. This option is required.
<code>-keep</code>	Keep the generated .java files after the compilation is complete.
<code>-verbose</code>	Output messages about the compiler action.
<code>-version</code>	Print version information.

`war file`- Typically, you create the WAR file with a development tool or with the `asant war` task. The following are the contents of a simple WAR file:

```

META-INF/MANIFEST.MF
WEB-INF/classes/hello/HelloIF.class
WEB-INF/classes/hello/HelloImpl.class
WEB-INF/jaxrpc-ri.xml
WEB-INF/web.xml

```

In the example, `HelloIF` is the service's RMI interface and `HelloImpl` is the class that implements the interface. The `web.xml` file is the deployment descriptor of a web component.

The `wsdeploy` tool examines the deployment descriptor `web.xml` and `jaxrpc-ri.xml` to generate the WAR file. If the deployment descriptor identifies a model file, the information in the model file is used for generating a WAR file. If the deployment descriptor does not identify a model file, `wsdeploy` generates a model. For information about the XML schema for creating a model file, see [Appendix A, "XML Schema Definitions."](#)

Behind the scene, `wsdeploy` tool runs the `wscompile` tool with `-gen:server` option. In other words, the tool generates the server-side artifacts such as ties. This tool can also generate the service endpoint definition, or a WSDL document.

The `jaxrpc-ri.xml` File

The `jaxrpc-ri.xml` file is the JAX-RPC implementation specific configuration file. This configuration file is read by the `wsdeploy` tool. The following code lists the contents of a `jaxrpc-ri.xml` file for a simple HelloWorld Service.

```

<?xml version="1.0" encoding="UTF-8"?>
<webServices
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
    version="1.0
    targetNamespaceBase="http://com.test/wsdl"
    typeNamespaceBase="http://com.test/types"
    urlPatternBase="/ws">
  <endpoint
    name="HelloWorld"
    displayName="Hello Service"
    description="A simple web service"
    interface="samples.webservices.jaxrpc.dynamic>HelloIF"
    implementation="samples.webservices.jaxrpc.dynamic>HelloImpl"/>
  <endpointMapping
    endpointName="HelloWorld"
    urlPattern="/dynamic"/>
</webServices>>

```

The `<webServices>` element must contain one or more `<endpoint>` elements. In this example, note that the interface and implementation attributes of `<endpoint>` specify the service's interface and implementation class. The `<endpointMapping>` element associates the service port with an element of the endpoint URL path that follows the `urlPatternBase`.

For information about the XML schema for creating the runtime descriptor, see [Appendix A, "XML Schema Definitions."](#)

Namespace Mappings

This section is for developers who are familiar with WSDL, SOAP, and the JAX-RPC specifications.

Here is a schema type name example:

```
schemaType="ns1:SampleType"
xmlns:ns1="http://echoservice.org/types"
```

When generating a Java type from a schema type, `wscompile` gets the class name from the local part of the schema type name.

To specify the package name of the generated Java classes, you define a mapping between the schema type namespace and the package name. You define this mapping by adding a `<namespaceMappingRegistry>` element to the `config.xml` file. For example:

```
<service>
    ...
    <namespaceMappingRegistry>
        <namespaceMapping
            namespace="http://echoservice.org/types"
            packageName="echoservice.org.types" />
        </namespaceMappingRegistry>
    ...
</service>
```

SOAP Handlers

A handler accesses a SOAP message that represents an RPC request or response. Handler class must implement the `javax.xml.rpc.handler` interface. A handler can manipulate a SOAP message with the APIs of the `javax.xml.soap` package.

The following are the examples of the tasks performed by a handler:

- Encryption and decryption
- Logging and auditing
- Caching
- Application-specific SOAP header processing

A handler chain is a list of handlers. You may specify one handler chain for the client and one for the server. On the client, you include the `<handlerChains>` element in the `jaxrpc-ri.xml` file. On the server, you include this element in the `config.xml` file.

Here is an example of the `<handlerChains>` element in the `config.xml` file:

```
<handlerChains>
  <chain runAt="server"
    roles= "http://acme.org/auditing
           http://acme.org/morphing"
    xmlns:ns1="http://foo/foo-1">
    <handler className="acme.MyHandler"
      headers ="ns1:foo ns1:bar"/>
      <property name="property" value="xyz"/>
    </handler>
  </chain>
</handlerChains>
```

Java Language Types Supported By JAX-RPC

JAX-RPC maps types of the Java programming language to XML/WSDL definitions. For example, JAX-RPC maps the `java.lang.String` class to the `xsd:string` XML data type. You as application developers need not know the details of these mappings, but you must be aware that not every class in the Java 2 Standard Edition (J2SE) can be used as a method parameter or return type in JAX-RPC.

J2SE SDK Classes

JAX-RPC supports the following J2SE SDK classes:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.String`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.util.Calendar`
- `java.util.Date`

This release of JAX-RPC also supports several implementation classes of the `java.util.Collection` interface. The following table displays the supported classes. The first column lists the `java.util.Collection` subinterfaces, the second column lists the classes supported by the subinterface.

Table 2-3 Supported Classes

java.util.collection Subinterfaces	Classes Supported
List	Array List
	LinkedList
	Stack
	Vector

Table 2-3 Supported Classes

java.util.collection Subinterfaces	Classes Supported
Map	HashMap
	Hashtable
	Properties
	TreeMap
Set	HashSet
	TreeSet

Primitives

JAX-RPC supports the following primitive types of the Java programming language:

- boolean
- byte
- double
- float
- int
- long
- short

Arrays

JAX-RPC also supports arrays with members of supported JAX-RPC types. Examples of supported arrays are `int []` and `String []`. Also supports multidimensional arrays, such as `BigDecimal [][]`.

Application Classes

JAX-RPC also supports classes that you have written for your applications. In an order processing application, for example, you might provide classes named `Order`, `LineItem`, and `Product`. The JAX-RPC Specification refers to such classes as value types, because their values (or states) may be passed between clients and remote services as method parameters or return values.

To be supported by JAX-RPC, an application class must conform to the following rules:

- It must have a public default constructor.
- It must not implement (either directly or indirectly) the `java.rmi.Remote` interface.
- Its fields must be supported JAX-RPC types.

The class may contain public, private, or protected fields. For its value to be passed (or returned) during a remote call, a field must meet these requirements:

- A public field cannot be final or transient.
- A non-public field must have corresponding getter and setter methods.

JavaBeans Components

JAX-RPC also supports JavaBeans components, which must conform to the same set of rules as application classes. In addition, a JavaBeans component must have a getter and setter method for each bean property. The type of the bean property must be a supported JAX-RPC type.

SOAP Clients and Services Using SAAJ and JAXM

This chapter describes how to use the SOAP with Attachments API for Java™ (SAAJ) and the Java™ API for XML Messaging (JAXM) to build clients that can send and receive messages, and deploy them to Sun ONE Application Server. This chapter contains the following sections:

- [SOAP Clients](#)
- [SOAP Service](#)

SOAP Clients

This section describes the two messaging models in which SOAP clients can be used; the procedure to develop and deploy such clients. This section describes the following topics:

- [SOAP Client Messaging Models](#)
- [Developing a SOAP Client](#)
- [Assembling and Deploying a SOAP Client](#)

SOAP Client Messaging Models

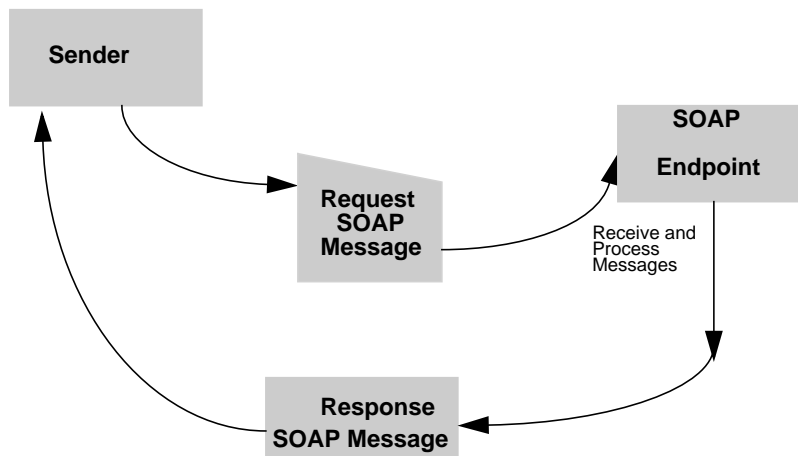
You can build SOAP clients using the following two messaging models.

- [Client Without a Messaging Provider](#)
- [Client With a Messaging Provider](#)

Client Without a Messaging Provider

An application that does not use a messaging provider can exchange only synchronous messages. That is, an application operates in a client role and can send only request-response messages. This type of client uses the `SOAPConnection` method of the SAAJ API. The following figure illustrates how synchronous messages are exchanged between the sender and the receiver without using a messaging provider.

Figure 3-1 SOAP Message Without Using a Messaging Provider



Clients not using a messaging provider have the following advantages:

- The application can be written using the J2SE platform.
- You do not need to deploy the application in a servlet or a J2EE container.
- No messaging provider configuration is necessary.

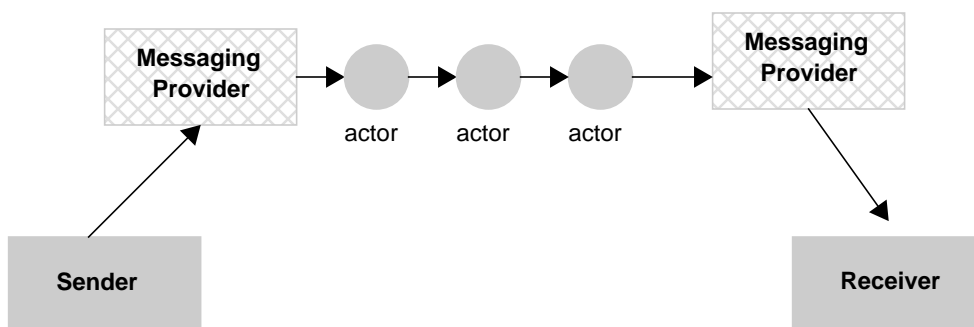
Clients not using a messaging provider have the following limitations:

- The client can send only request-response messages.
- The client can act in the client role only.

Client With a Messaging Provider

You must use a messaging provider if you want to be able to get and save requests that are sent to you at any time. The JAXM API provides the framework to send and receive messages using a messaging provider. You will need to run the client in a container, which provides the messaging infrastructure used by the provider. The [Figure 3-2](#) illustrates how asynchronous messages are exchanged between the sender and the receiver using a messaging provider.

Figure 3-2 SOAP Message Using a Messaging Provider



Clients using messaging provider have the following advantages:

- Clients can assume the roles of a client and a service.
- Clients can hand off message delivery to a provider.
- Clients can send messages to one or more destinations before it delivers the message to the final recipient. These intermediate message recipients are called *actors* and they are specified in the `SOAPHeader` object of the message.
- Clients can take advantage of any provider-supported SOAP messaging protocols and 'Quality of Service' affecting the types of messages and the reliability and the quality of service of message delivery.

NOTE Sun ONE Application includes a sample JAXM provider that illustrates how a provider is used to enable ‘fire and forget’ messaging for the sending client. See the Samples documentation for complete details on how to enable, deploy, and use it. The sample applications are available at the following location:

install_dir/samples/webservices/jaxm/jaxm-provider/

Future release of Sun ONE Application Server will include JAXM Providers that support reliable SOAP messaging as well as ebXML messaging.

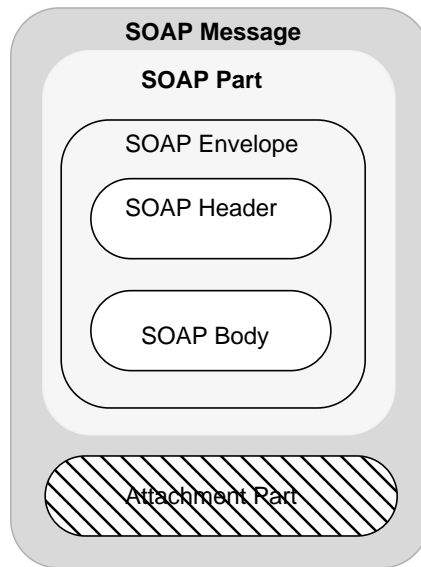
SOAP Messages

This section introduces you to the structure and parts of a SOAP message, how you access these, and how you process SOAP messages. This section describes the following topics:

- [Parts of a SOAP Message](#)
- [Accessing Elements of a Message](#)

Parts of a SOAP Message

A SOAP message is an XML document that consists of a `SOAPEnvelope`, an optional `SOAPHeader`, and a `SOAPBody`. The SOAP message header contains information that allows the message to be routed through one or more intermediate nodes before it reaches its final destination.

Figure 3-3 Parts of a SOAP Message

The [Figure 3-3](#), shows the structure and parts of a SOAP message. Different objects represents each part of a SOAP message.

The `SOAPMessage` object contains

- A `SOAPPart` object that contains
 - A `SOAPEnvelope` object that contains
 - An empty `SOAPHeader` object - is optional, included for convenience because, most messages will use it.
 - An empty `SOAPBody` object - can hold the content of the message and can also contain fault messages that contain status information or details about a problem with the message.
- `AttachmentPart` that may contain plain text, or an image file.

The `SOAPEnvelope` is the root element of the XML document representing the message. It defines the framework for how the message should be handled and by whom. XML content starts at the `SOAPEnvelope`.

The `SOAPHeader` is a generic mechanism for adding features to a SOAP message. It can contain any number of child elements that define extensions to the base protocol. For example, header child elements might define authentication information, transaction information, locale information, and so on. The software that handles the message may, without prior agreement, use this mechanism to define who should deal with a feature and whether the feature is mandatory or optional.

The `body` is a container for mandatory information intended for the ultimate recipient of the message. A SOAP message may also contain an attachment, which need not necessarily be an XML document.

Accessing Elements of a Message

You need to access parts of a message when you create the message body or the attachment part or when you are processing the message.

The `SOAPMessage` object `message` contains a `SOAPPart` object. Use the `message` object to retrieve it.

```
SOAPPart soapPart = message.getSOAPPart();
```

Next you can use `SOAPPart` to retrieve the `SOAPEnvelope` object that it contains.

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

You can now use `envelope` to retrieve its empty `SOAPHeader` and `SOAPBody` objects.

```
SOAPHeader header = envelope.getHeader();
```

```
SOAPBody body = envelope.getBody();
```

```
SOAPBody objects are initially empty.
```

Namespaces

An XML namespace is a means of qualifying elements and attribute names to disambiguate from other names in the same document. An explicit XML Namespace declaration takes the following form:

```
<prefix:myElement
xmlns:prefix = "URI">
```

The declaration defines `prefix` as an alias for the specified URI. In the element `myElement`, you can use `prefix` with any element or attribute to specify that the element or attribute name belongs to the namespace specified by the URI. The following line of code is an example of a namespace declaration:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
```

This declaration defines SOAP_ENV as an alias for the namespace

```
http://schemas.xmlsoap.org/soap/envelope/
```

After defining the alias, you can use it as a prefix to any attribute or element in the Envelope element.

Pre-defined SOAP Namespaces

SOAP defines two namespaces:

- The SOAPEnvelope, the root element of a SOAP message, has the following namespace identifier "http://schemas.xmlsoap.org/soap/envelope".
- The SOAP serialization, the URI defining SOAP serialization rules, has the following namespace identifier:
"http://schemas.xmlsoap.org/soap/encoding".

When you use SAAJ or JAXM to construct or consume messages, you are responsible for setting or processing namespaces correctly and for discarding messages that have incorrect namespaces.

Using Namespaces when Creating a SOAP Message

When you create the body elements or header elements of a SOAP message, you must use the Name object to specify a well-formed name for the element. You obtain a Name object by calling the method SOAPEnvelope.createName.

When you call this method, you can pass a local name as a parameter or you can specify a local name, prefix, and a URI. For example, the following line of code defines a Name object bodyName.

```
Name bodyName = MyEnvelope.createName("TradePrice", "GetLTP",
"http://foo.eztrade.com");
```

This would be equivalent to the namespace declaration:

```
<GetLTP:TradePrice xmlns:GetLTP="http://foo.eztrade.com">
```

The following code shows how you can create a name and associate it with a SOAPBody element. Note the use and placement of the createName method.

```
SoapBody body = envelope.getBody();//get body from envelope
Name bodyName = envelope.createName("TradePrice", "GetLTP",
"http://foo.eztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

Developing a SOAP Client

Using SAAJ, a client can create and send SOAP messages in a point-to-point model. JAXM defines the API for xml messaging using a messaging provider. JAXM depends on the SOAP with Attachments API for Java (SAAJ), which defines the API for operating on the SOAP with attachments message model in Java. Sun ONE Application Server does not include a supported JAXM messaging provider. It does however include along with the sample applications, a simple JAXM provider that demonstrates how a messaging provider handles asynchronous SOAP messaging from a client. This section describes the following topics:

- [How SOAP Messaging Occurs?](#)
- [Creating a SOAP Client](#)
- [Assembling and Deploying a SOAP Client](#)

How SOAP Messaging Occurs?

SOAP messaging occurs when a SOAP message, produced by a message factory is sent to an `Endpoint` via a `Connection`. This section describes the following topics:

- [Endpoint](#)
- [Connection](#)

Endpoint

An endpoint identifies the final destination of a message. An endpoint is defined by the `URLEndpoint` class. If you do not use a provider, you must construct or find an endpoint to which the message is sent.

Constructing an Endpoint

You can initialize an endpoint either by calling its constructor or by looking it up in a naming service.

The following code uses a constructor to create an `URLEndpoint`:

```
myEndpoint = new URLEndpoint("http://host/myServlet")
```

Using the Endpoint to Address a Message

Specify the endpoint as a parameter to the `SOAPConnection.call` method, which you use to send a SOAP message.

Sending a Message to Multiple Endpoints

Administered objects are objects that encapsulate provider-specific configuration and naming information. If you are using an administered object to define an endpoint, note that it is possible to associate that administered object with multiple URLs—each URL capable of processing incoming SOAP messages.

The code sample below associates the endpoint whose lookup name is `myEndpoint` with two URLs: `http://www.myServlet1/` and `http://www.myServlet2/`. This syntax allows you to use a SOAP connection to publish a SOAP message to multiple endpoints.

```
imgobjmgr add
  -t e
  -l "cn=myEndpoint"
  -o "imgSOAPEndpointList=http://www.myServlet1/
    http://www.myServlet2/"
```

Connection

To send a SOAP message using SAAJ or JAXM, you must obtain a `SOAPConnection` or a `ProviderConnection` respectively. You can also transport a SOAP message using the Message Queue; for more information, see the *Sun ONE Message Queue Developer's Guide*.

SOAP Connection

A `SOAPConnection` allows you to send messages directly to a remote party. You can obtain a `SOAPConnection` object simply by calling the static method `SOAPConnectionFactory.newInstance()`. Neither reliability nor security are guaranteed over this type of connection.

Provider Connection

A `ProviderConnection`, which you get from a `ProviderConnectionFactory`, creates a connection to a particular messaging provider. When you send a SOAP message using a provider, the message is forwarded to the provider, and then the provider is responsible for delivery to its final destination thus ensuring reliable and secure messaging.

Creating a SOAP Client

Before creating a SOAP client, make sure to set up your client environment. For more information on setting up your client environment, see [“Setting Up the Client Environment” on page 29](#).

If you are creating a point-to-point client, you must import the `javax.xml.soap` package of the SAAJ API. If you are creating a pub/sub client, import the `javax.xml.messaging` package of the JAXM API. In addition, you must import the following packages:

```
import java.net.*;
import java.io.*;
import java.util.*;

import javax.servlet.http.*;
import javax.servlet.*;

import javax.activation.*;
import javax.naming.*;
```

Creating a SOAP client and accessing the message involves the following steps:

- [Getting a Connection](#)
- [Creating a Message](#)
- [Adding Content to the Header](#)
- [Adding Content to a Message](#)
- [Adding an Attachment to the Message](#) (optional)
- [Sending a Message](#)
- [Retrieving the Content from a Response Message](#)
- [Accessing Attachment Part of the Message](#)

Getting a Connection

A client that does not use a messaging provider uses the `SOAPConnection` object to create a connection. The message sent using `SoapConnection` object goes directly from the sender to the URL that the sender specifies.

You must obtain a `SOAPConnectionFactory` object that you can use to create your connection. The SAAJ API makes this easy by providing the `SOAPConnectionFactory` class with a default implementation. The following code illustrates how to get an instance of the implementation:

```
SOAPConnectionFactory scf = SOAPConnectionFactory.
    newInstance();
```

Now you can use `scf` to create `SOAPConnection` object.

```
SOAPConnection con = scf.createConnection();
```

Creating a Message

To create a message, use a `MessageFactory` object. If you are creating a stand-alone client, that is, a client that does not use a messaging provider and also does not run in a container, you can use the implementation of the `MessageFactory` class that the SAAJ API provides. The following code illustrates getting an instance of the default message factory and then using it to create a message:

```
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

Message creation takes care of creating the `SOAPPart`, a required part of the message as per the SOAP 1.1 specification.

```
SOAPPart sp = msg.getSOAPPart();
```

NOTE Generally, the default message factory is used to create a message. However, you can write your own implementation of a message factory, and plug it in through system properties as explained below:

- Implement a message factory class by extending `javax.xml.soap.MessageFactory`. `javax.xml.soap` is the package defined in the SAAJ API.
 - Indicate the desired message factory class to be instantiated by setting the System Property `javax.xml.soap.MessageFactory` to the full classname of the message factory class `mypackage.MySOAPMessageFactoryImpl`.
-

For information on structure and parts of a SOAP message, see [“Parts of a SOAP Message” on page 78](#).

Adding Content to the Header

You create a `SOAPHeaderElement` object to add content to the header. The following code illustrates how to create a `SOAPHeaderElement` using the `SOAPEnvelope` object.

```
SOAPHeader hdr = envelope.getHeader();

Name headerName = envelope.createName("Purchase Order", "PO",
"http://www.sonata.com/order");

SOAPHeaderElement headerElement =
hdr.addHeaderElement(headerName);
```

The `headerElement` is identified by the `Name` object `headerName`. The `addHeaderElement` method is used to add or create a header element.

To add content to `headerElement`, use the `addTextNode` method as shown in the code illustration below:

```
headerElement.addTextNode("order");
```

The `SOAPHeader` object contains a `SOAPHeaderElement` object whose content is "order".

Adding Content to a Message

You can add content to a `SOAPPart` object or to one or more `AttachmentPart` object or to both parts of a message.

To add content to the body, create a `SOAPBodyElement` object and add an XML element that you build with the method `SOAPElement.addTextNode`. The following code illustrates adding content to the message:

```
SOAPEnvelope envelope = sp.getSOAPEnvelope();
SOAPBody bdy = envelope.getSOAPBody();
SOAPBodyElement gltp =
bdy.addBodyElement(envelope.createName("GetLastTradePrice",
"ztrade", "http://wombat.ztrade.com"));
gltp.addChildElement(envelope.createName("symbol", "ztrade",
"http://wombat.ztrade.com")).addTextNode("SUNW");
```

The first three lines of the code access the `SOAPBody` object `body`, that is used to create a new `SOAPBodyElement` object and add it to `body`. The `CreateName` method has the argument `Name` object that identifies the `SOAPBodyElement` that is being added. The last line adds the XML string passed to the method `addTextNode`.

Adding an Attachment to the Message

The procedure to add attachments to a message is same for both the clients with and without using a messaging provider. The `AttachmentPart` object is used to add attachment part to a message.

You use the `SOAPMessage` object to create an `AttachmentPart` object. The `SOAPMessage` class has three methods for creating an `AttachmentPart` object. The first method allows you to create an `AttachmentPart` with no content. That is, the `AttachmentPart` method `setContent` is later used to add content to the attachment.

```
URL url = new URL(data);
AttachmentPart ap = msg.createAttachmentPart(new
DataHandler(url));
```

The `setContent` method takes two parameters, a Java object for the content and a `String` object that identifies the content type. Content is the `SOAPBody` part of a message that has a `Content-Type` header with the value “text/xml” because the content has to be in XML format. In the `AttachmentPart`, the type of the content has to be specified as this object can take any type of content.

Each `AttachmentPart` has one or more headers associated with it. In the `setContent` method, the type of the method used is the type for the header `Content-Type`. This is the only header that is required. You can also set other optional headers, such as `Content-Id` and `Content-Location`. For convenience, JAXM and SAAJ APIs provides `get` and `set` methods for the headers `Content-type`, `Content-Id`, and `Content-Location`. These headers are helpful in accessing a particular attachment when a message has multiple attachments.

The following code illustrates how you can use the `setContent` method:

```
String stringContent = "Update address for Sunny Skies " + "Inc.,  
to 10 Upbeat Street, Pleasant Grove, CA 95439";  
  
ap.setContent(stringContent, "text/html");  
ap.setContentId("update_address");  
msg.addAttachmentPart(ap);
```

If you also want to attach a jpeg image, the second argument for the `setContent` method must be “image/jpeg”. The following code illustrates the use of `setContent` method to attach an image:

```
AttachmentPart ap2 = msg.createAttachmentPart();  
  
byte[] jpegData = . . .;  
  
ByteArrayInputStream stream = new  
ByteArrayInputStream(jpegData);  
  
ap2.setContent(stream, "image/jpeg");  
  
msg.addAttachmentPart(ap2);
```

The other two `AttachmentPart` methods allow you to create an `AttachmentPart` object with content. One of the two methods is very similar to the `AttachmentPart.setContent` method. It takes a Java object containing the content and a `String` giving the content type. The object may be a `String`, a stream, a `javax.xml.transform.Source` object, or a `javax.activation.DataHandler` object.

The other method for creating an `AttachmentPart` object with content takes a `DataHandler` object, which is part of the JavaBeans Activation Framework (JAF).

The following code illustrates how you can use `DataHandler` in content. First you create a `java.net.URL` object for the file you want to add as content. Create the `DataHandler` object `javax.activation.DataHandler` object `dh` initialized with the `URL` object and pass it to the method `createAttachmentPart`.

```
URL url = new URL("http://greatproducts.com/gizmos/img.jpg");
DataHandler dh = new DataHandler(url);
AttachmentPart ap = msg.createAttachmentPart(dh);
ap.setContentId("gyro_image");
msg.addAttachmentPart(ap);
```

Sending a Message

Stand-alone Client

To send a message, a stand-alone client uses the `SOAPConnection` method call. This method takes two arguments, the message being sent and the destination to which the message should go which is an `Endpoint` object that contains the URL of the receiver.

When using `SoapConnection`, you send message using `javax.xml.soap.SOAPConnection.call()` method.

For example:

```
URL urlEndpoint = new URL(to);
SOAPMessage reply = con.call(msg, urlEndpoint);
```

Retrieving the Content from a Response Message

To retrieve message content, the client uses the `onMessage` method. The client accesses `SOAPBody` object, using the message to get the envelope and the envelope to get the body. Access its `SOAPBodyElement` object because that is the element to which content was added. To retrieve the content, which was added with the method `Node.addTextNode`, you call the method `Node.getValue`. The `getValue` returns the value of the immediate child of the element that calls the method. To access `bodyElement`, you need to call the method `getChildElement` on `body`. The following code illustrates how to retrieve contents from a response message:

```
public SOAPMessage onMessage(SOAPMessage message)
{
    SOAPEnvelop env = msg.getSOAPPart().getEnvelope();
    env.getBody()
        .addChildElement(env.createName("Response"))
        .addTextNode("This is a Response");
    return msg;
}
```

```
}
```

To retrieve the contents from the message that contains an attachment, you need to access the attachment. When it is given no argument, the method

`SOAPMessage.getAttachments` returns a `java.util.Iterator` object over all the `AttachmentPart` objects in a message. The following code prints content of each `AttachmentPart` object in the `SOAPMessage` object `message`.

```
java.util.Iterator it = message.getAttachments();
while (it.hasNext()) {
    AttachmentPart attachment = (AttachmentPart)it.next();
    Object content = attachment.getContent();
    String id = attachment.getContentId();
    System.out.print("Attachment " + id + " contains: " + content);
    System.out.println("");
}
```

Accessing Attachment Part of the Message

When you receive a message with an attachment, or you wish to change an attachment to a message, you need to access the attachment part of the message. The `SOAPMessage.getAttachments` method without any attachment returns a `java.util.Iterator` object over all `AttachmentPart` objects in a message. The following code illustrates accessing the attachment part to get the content of each `AttachmentPart` object in the `SOAPMessage` object `message`.

```
java.util.Iterator it = msg.getAttachments();
while (it.hasNext()) {
    AttachmentPart ap = it.next();
    Object content = ap.getContent();
    String id = ap.getContentId();
    System.out.print("Attachment " + id + " contains: " + content);
    System.out.println("");
}
```

Assembling and Deploying a SOAP Client

Applications created using JAXM API and SAAJ API are assembled as web applications (WAR) or J2EE platform based applications (EAR). For more information on assembling and deploying a web application, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

SOAP Service

This section describes how you can write SOAP service and describes how you can handle exceptions and faults in SOAP messages. This section describes the following topics:

- [Creating a SOAP Service](#)
- [Exception and Fault Handling](#)

Creating a SOAP Service

A SOAP service is the final recipient of a SOAP message and is implemented as a servlet. You can either create your own servlet or you can extend the `JAXMServlet` class which is bundled in the `javax.xml.messaging` package. This section describes the procedure to create a SOAP service based on the `JAXMServlet` class.

To create a SOAP service, your servlet must implement either the `ReqRespListener` or `OneWayListener` interfaces. A `ReqRespListener` requires that you return a reply.

```
public class MyServlet extends JAXMServlet implements
    ReqRespListener{

    ...

    ...

}
```

Using any of the interfaces, implement a method called `onMessage(SOAPMsg)`.

```
public SOAPMessage onMessage(SOAP Message msg)
```

The following code is the complete listing of the SOAP consumer using `JAXMServlet`.

```
public class MyServlet extends JAXMServlet implements
    ReqRespListener {

    public SOAPMessage onMessage(SOAP Message msg) {

        //Process message here

    }

}
```

`JAXMServlet` will call `onMessage` after receiving a message using the HTTP `POST` method. This saves you the work of implementing your own `doPost()` method to convert the incoming message into a SOAP message.

The `onMessage` method needs to disassemble the SOAP message that is passed to it by the servlet and process its contents. Processing the message involves accessing the parts of a SOAP message. If there are problems in the processing of the message, the service needs to create a SOAP fault object and send it back to the client. For more information on handling faults, see [“Fault Handling” on page 91](#).

The following code illustrates the processing of a SOAP message:

```
{http://xml.coverpages.org/dom.html
SOAPEnvelope env = reply.getSOAPPart().getEnvelope();
SOAPBody sb = env.getBody();

// create Name object for XElement that we are searching for Name
ElName = env.createName("XElement");

//Get child elements with the name XElement
Iterator it = sb.getChildElements( ElName );

//Get the first matched child element.
//We know there is only one.
SOAPBodyElement sbe = (SOAPBodyElement) it.next();

//Get the value for XElement
MyValue = sbe.getValue(); }
```

Exception and Fault Handling

On the client's side, JAXM and SAAJ uses a SOAP exception to handle errors that occur during the generation of the SOAP request or unmarshalling of the response. This section describes the following topics:

- [Fault Handling](#)
- [Defining SOAP Fault](#)

Fault Handling

Server-side code must use the `SOAPFault` object to handle errors that occur on the server-side when unmarshalling the request, processing the message, or marshalling the response. The `SOAPFault` interface extends the `SOAPBodyElement` interface.

SOAP messages have a specific element and format for error reporting on the server side: a SOAP message body can include a `SOAPFault` element to report errors that occur during the processing of a request. Created on the server side and sent from the server back to the client, the SOAP message containing the `SOAPFault` object reports any unexpected behavior to the originator of the message.

The `SOAPFault` element defines the following four subelements:

faultcode

A code that identifies the error. The code is intended for use by software to provide an algorithmic mechanism for identifying the fault. This element is required.

faultstring

A string that describes the fault identified by the fault code. This element provides an explanation of the error that is understandable to a human. This element is required.

faultactor

A URI specifying the source of the fault: who caused the fault. This element is not required if the message is sent to its final destination without going through any intermediaries. If a fault occurs at an intermediary, then that fault must include a `faultactor` element.

detail

This element carries specific information related to the `body` element. It must be present if the contents of the `body` element could not be successfully processed. Thus, if this element is missing, the client should infer that the `body` element was processed. While this element is not required for any error except a malformed payload, you can use it in other cases to supply additional information to the client.

Predefined Fault Codes

The SOAP specification lists the following four predefined `faultcode` values:

VersionMismatch

The processing party found an invalid namespace for the SOAP envelope element; that is, the namespace of the `SOAPEnvelope` element was not `http://schemas.xmlsoap.org/soap/envelope/`.

MustUnderstand

An immediate child element of the `SOAPHeader` element was either not understood or not appropriately processed by the recipient. This element's `mustUnderstand` attribute was set to 1 (true).

Client

The message was incorrectly formed or did not contain the appropriate information. For example, the message did not have the proper authentication or payment information. The client should interpret this code to mean that the message must be changed before it is sent again. If this is the code returned, the `SOAPFault` object should probably include a `detailEntry` object that provides additional information about the malformed message.

Server

The message could not be processed for reasons that are not connected with its content. For example, one of the message handlers could not communicate with another message handler that was upstream and did not respond. Or, the database that the server needed to access is down. The client should interpret this error to mean that the transmission could succeed at a later point in time.

Defining SOAP Fault

You can specify the value for `faultcode`, `faultstring`, and a `faultactor` using methods of the `SOAPFault` object. The following code illustrates the creation of a `SOAPFault` object and sets the `faultcode`, `faultstring`, and `faultactor` attributes:

```
SOAPFault fault;
reply = factory.createMessage();
envp = reply.getSOAPPart().getEnvelope(true);
someBody = envp.getBody();
fault = someBody.addFault():
fault.setFaultCode("Server");
fault.setFaultString("Some Server Error");
fault.setFaultActor("http://xxx.me.com/list/endpoint.esp/");
reply.saveChanges();
```

The server can return this object in its reply to an incoming SOAP message in case of a server error.

The following code illustrates how to define a detail and detail entry object. Note that you must create a name for the detail entry object.

```
SOAPFault fault = someBody.addFault();
fault.setFaultCode("Server");
fault.setFaultActor("http://foo.com/uri");
```

```

fault.setFaultString ("Unkown error");

DetailEntry entry =
detail.addEntry(envelope.createName("125detail", "m",
"Someuri"));

entry.addTextNode("the message cannot contain the string //");

reply.saveChanges();

```

Assembling and Deploying a SOAP Service

Applications created using JAXM API and SAAJ API are assembled as web applications (WAR) or J2EE platform based applications (EAR). For more information on assembling and deploying a web application, see the *Sun ONE Application Server Developer's Guide to Web Applications*.

Sample Clients and Services

Sample client and services applications are bundled with Sun ONE Application Server. These samples demonstrate the creation of services and clients that send and receive XML messages. You can find the samples at the following location.

install_dir/samples/webservices/jaxm

install_dir/imq/demo/jaxm

Clients Using JAXR

Application Server provides the ability for the clients to publish, discover, and manage content within XML registries using the implementation of Java™ API for XML Registry (JAXR).

This chapter describes the procedure to develop clients that can interact with the registry to perform various operations on the registry. This chapter contains the following sections:

- [Developing a JAXR Client](#)
- [Managing Registry Data](#)
- [Publishing a Web Service to a UDDI Registry](#)
- [Assembling and Deploying a JAXR Client](#)

Developing a JAXR Client

This section describes the steps required to implement a JAXR client that can perform queries and update a registry.

Before you develop a JAXR client, make sure to setup your client environment. For detailed information on setting up your client environment, see [“Setting Up the Client Environment” on page 29](#).

Implementing a JAXR client involves the following steps:

- [Getting Access to a Registry](#)
- [Establishing a Connection](#)
- [Querying a Registry](#)

Getting Access to a Registry

You must obtain permission from the registry to access the registry. A JAXR client can then perform queries, add data to registry, or update registry data. To register with one of the public UDDI version 2 registries, go to one of the following Web sites and follow the instructions:

<http://uddi.microsoft.com/> (Microsoft)

<https://uddi.ibm.com/ubr/registry.html> (IBM)

When you register, you will obtain a user name and password. To run samples bundled with Sun ONE Application Server, you may register with IBM's UDDI registry.

Accessing an ebXML Registry

An ebXML registry allows you to publish and discover Web services. Unlike a UDDI registry, an ebXML registry can store the metadata about a service and arbitrary content such as, the actual Web service description, that is the WSDL document.

For more information on ebXML, visit the following URL:

<http://www.ebxml.org>

Sun ONE Application Server supports JAXR clients to access an ebXML registry through a third-party JAXR provider. The `ebxmlrr-client` is a package that provides an implementation of the JAXR API that is compatible with the OASIS ebXML Registry V2.x (version 2.0 and 2.1) standard. The `ebxmlrr-client` package also includes a Registry Browser application that can graphically browse any OASIS ebXML V2.x registry.

For more information, visit the following URL:

<http://ebxmlrr.sourceforge.net>

Establishing a Connection

The first task a JAXR client must perform is, to establish a connection to a registry. This connection contains the client state and preference information used when the JAXR provider invokes methods on the registry provider.

NOTE In order to add data to the registry or to update registry data, the client must set authentication information on the connection. The establishment of this authentication information with the registry provider is specific to that registry provider.

Create a connection from a connection factory. A JAXR provider may supply one or more pre configured connection factories that clients can look up using the JNDI API.

The following code illustrates how to establish a connection to a JAXR provider:

```
import javax.xml.registry.*;

...

public void makeConnection(String queryURL, String publishURL)
{
    ConnectionFactory factory = ConnectionFactory.newInstance();
    .....
}
```

In the code above, `queryURL` and `publishURL` are the URL of the query and publish registries respectively.

Setting Properties

The implementation of JAXR API in Sun ONE Application Server allows you to set a number of properties on a JAXR connection. The following table list the standard JAXR connection properties and properties specific to implementation of JAXR in Sun ONE Application Server. The first column shows the name of the property and the description of that property, the second column shows the data type that you can use with the property, and the third column shows the default value associated with the property.

Table 4-1 Standard JAXR Connection Properties

Property Name and Description	Data type	Default Value
<code>javax.xml.Registry.queryManagerURL</code>	String	None
Specifies the URL of the query manager service within the target registry provider		

Table 4-1 Standard JAXR Connection Properties

Property Name and Description	Data type	Default Value
<code>javax.xml.registry.lifeCycleManagerURL</code> Specifies the URL of the life cycle manager service within the target registry provider (for registry updates)	String	Same as the specified <code>queryManagerURL</code> value
<code>javax.xml.registry.semanticEquivalences</code> Specifies semantic equivalences of concepts as one or more tuples of the ID values of two equivalent concepts separated by a comma; the tuples are separated by vertical bars: <code>id1,id2 id3,id4</code>	String	None
<code>javax.xml.registry.security.authenticationMethod</code> Provides a hint to the JAXR provider on the authentication method to be used for authenticating with the registry provider	String	None; UDDI_GET_AUTHTOKEN is the only supported value
<code>javax.xml.registry.uddi.maxRows</code> The maximum number of rows to be returned by find operations. Specific to UDDI providers	Integer	None
<code>javax.xml.registry.postalAddressScheme</code> The ID of a ClassificationScheme to be used as the default postal address scheme.	String	None

Table 4-2 Sun ONE-specific JAXR Implementation Connection Properties

Property Name and Description	Data type	Default Value
<code>com.sun.xml.registry.http.proxyHost</code> Specifies the HTTP proxy host to be used for accessing external registries.	String	Proxy host value specified in <which file>?
<code>com.sun.xml.registry.http.proxyPort</code> Specifies the HTTP proxy port to be used for accessing external registries; usually 8080	String	Proxy port value specified in <which file>?
<code>com.sun.xml.registry.https.proxyHost</code> Specifies the HTTPS proxy host to be used for accessing external registries	String	Same as HTTP proxy host value

Table 4-2 Sun ONE-specific JAXR Implementation Connection Properties

Property Name and Description	Data type	Default Value
<code>com.sun.xml.registry.https.proxyPort</code> Specifies the HTTPS proxy port to be used for accessing external registries; usually 8080	String	Same as HTTP proxy port value
<code>com.sun.xml.registry.http.proxyUserName</code> Specifies the user name for the proxy host for HTTP proxy authentication, if one is required	String	None
<code>com.sun.xml.registry.http.proxyPassword</code> Specifies the password for the proxy host for HTTP proxy authentication, if one is required	String	None
<code>com.sun.xml.registry.useCache</code> Tells the JAXR implementation to look for registry objects in the cache first and then to look in the registry if not found	Boolean, passed in as String	True
<code>com.sun.xml.registry.useSOAP</code> Tells the JAXR implementation to use Apache SOAP rather than the Java API for XML Messaging; may be useful for debugging	Boolean, passed in as String	False

You can set these properties as shown in the code below:

```
String queryURL =
"http://www-3.ibm.com/services/uddi/v2beta/inquiryapi";

String publishURL=
"https://www-3.ibm.com/services/uddi/v2beta/protect/publishapi";

Properties props = new Properties();

props.setProperty("javax.xml.registry.queryManagerURL",
queryUrl);

props.setProperty("javax.xml.registry.lifeCycleManagerURL",
publishUrl);
```

Creating a Connection

A client first creates a set of properties that specify the URL or URLs of the registry or registries being accessed.

The client then sets the properties for the connection properties and creates the connection.

```
factory.setProperties(props);
Connection connection = factory.createConnection();
```

Obtaining the RegistryService and Managers

The client uses the connection to obtain a `RegistryService` object and then the interface or interfaces it will use. The following code illustrates how to obtain the registry service:

```
RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();
BusinessLifecycleManager blcm =
    rs.getBusinessLifecycleManager();
```

Setting Client Authentication Information

The following code illustrates how to set the client authorization information for privileged registry operations:

```
PasswordAuthentication passwdAuth = new
    PasswordAuthentication(username, password.toCharArray());
Set creds = new HashSet();
creds.add(passwdAuth);
```

Querying a Registry

The client uses the registry by querying it for information about the organization that have submitted data to it. The client can query the registry based on one or more of the following criterion:

- `findOrganizations`, which returns a list of organizations that meet the specified criteria - often a name pattern or a classification within a classification scheme.
- `findServiceBindings`, which returns the service bindings (information about how to access the service) that are supported by a specified service.
- `findService`, which returns a set of services offered by a specified organization.

This section describes the procedure to query a registry based on the following criterion:

- [Finding Organizations by Name](#)
- [Finding Organizations by Classification](#)
- [Finding Organizations by WSDL Descriptions](#)
- [Finding Services and Service Bindings](#)

Finding Organizations by Name

To find an organization by name, you use a combination of find qualifiers (which affect sorting and pattern matching) and name patterns (which specify the strings to be searched). The `findOrganizations` method takes a collection of `findQualifier` as its first argument and a collection of `namePattern` objects as its second argument.

The following code illustrates the use of `findOrganizations` method to search for an organization whose name begins with a specific string `qString`, and to sort them in alphabetical order:

```
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
namePatterns.add(qString);
```

The above code lines define the find qualifiers and name patterns.

To find an organization using the name, use the `findOrganizations()` method as shown in the code illustration below:

```
BulkResponse response = bqm.findOrganizations(findQualifiers,
namePatterns, null, null, null, null);

Collection orgs = response.getCollection();
```

Finding Organizations by Classification

To find an organization by classification, you need to establish a classification within a particular classification scheme and specify the classification as a parameter to the `findOrganizations()` method.

Let us assume that you are browsing the UDDI registry and wish to find an organization that provides services of the NAICS (North American Industry Classification System) type Computer Systems Design and Related Services in the United States. To perform this query with JAXR, invoke a `findOrganizations()` method with classification listed under the well-known taxonomies NAICS and ISO 3166 Geographic Code System (ISO 3166). As JAXR provides a taxonomy service for these classifications, the client can easily access the classification information needed to be passed as `findOrganization()` parameters.

```

ClassificationScheme cScheme =
    bqm.findClassificationSchemeByName (null, "ntis-gov:naics");

Classification classification =
    (Classification)blcm.createClassification(cScheme, "Snack and
    Nonalcoholic Beverage Bars", "722213");

Collection classifications = new ArrayList();

classifications.add(classification);

// make JAXR request

BulkResponse response = bqm.findOrganizations(null, null,
    classifications, null, null, null);

Collection orgs = response.getCollection();

```

Finding Organizations by WSDL Descriptions

You can find organizations based on technical specifications that take the form of WSDL documents. In JAXR, a concept is used as a proxy to hold the specification. The client must find the specification concepts first, then the organizations that use those concepts.

The following code illustrates finding an organization based on the WSDL specification instances used within a given registry.

```

String schemeName = "uddi-org:types";

ClassificationScheme uddiOrgTypes =
    bqm.findClassificationSchemeByName(null, schemeName);

/*
 * Create a classification, specifying the scheme
 * and the taxonomy name and value defined for WSDL
 * documents by the UDDI specification.
 */

Classification wsdlSpecClassification =
    blcm.createClassification(uddiOrgTypes, "wsdlSpec", "wsdlSpec");

ArrayList classifications = new ArrayList();
classifications.add(wsdlSpecClassification);

// Find concepts

BulkResponse br = bqm.findConcepts(null, null, classifications,
    null, null);

```

Next, you must go through the concepts, find the WSDL documents they correspond to, and display the organizations that use each document:

```

// Display information about the concepts found
Collection specConcepts = br.getCollection();
Iterator iter = specConcepts.iterator();
if (!iter.hasNext()) {
    System.out.println("No WSDL specification concepts found");
} else {
    while (iter.hasNext()) {
        try{
            Concept concept = (Concept) iter.next();
            String name = getName(concept);
            Collection links = concept.getExternalLinks();
            System.out.println("\nSpecification Concept:\n Name: " +name
+ "\n Key: " + concept.getKey().getId() + "\n Description: " +
getDescription(concept));
            if (links.size() > 0) {
                ExternalLink link =
                    (ExternalLink) links.iterator().next();
                System.out.println("URL of WSDL document: '"
link.getExternalURI() + "'");
            }
        }
// Find organizations that use this concept
ArrayList specConcepts1 = new ArrayList();
specConcepts1.add(concept);

br = bpm.findOrganizations(null, null, null, specConcepts1, null,
null);

Collection orgs = br.getCollection();

// Display information about organizations
... }

```

Finding Services and Service Bindings

A JAXR client can find an organization's services and the service bindings associated with those using the `getService()` and the `getServiceBindings()` method respectively. The following code illustrates the use of the `getServices()` and `getServiceBindings()` method:

```

Iterator orgIter = orgs.iterator();

```

```

while (orgIter.hasNext()) {
    Organization org = (Organization) orgIter.next();
    Collection services = org.getServices();
    Iterator svcIter = services.iterator();
    while (svcIter.hasNext()) {
        Service svc = (Service) svcIter.next();
        Collection serviceBindings = svc.getServiceBindings();
        Iterator sbIter = serviceBindings.iterator();
        while (sbIter.hasNext()) {
            ServiceBinding sb =
                (ServiceBinding) sbIter.next();
        }
    }
}

```

Managing Registry Data

A JAXR client can submit data to a registry, modify the existing registry data, and remove data from the registry. A JAXR client must be authorized to manage the registry data. A client that has submitted data to the registry can only remove or modify it.

This section describes the following tasks:

- [Getting Authorization from the Registry](#)
- [Creating an Organization](#)
- [Adding Classifications](#)
- [Adding Services and Service Bindings to an Organization](#)

Getting Authorization from the Registry

The JAXR client sends its user name and password to the registry in a set of credentials on the connection. These credentials may be used by the provider to authenticate with the registry.

```
// Edit to provide your own username and password
String username = "";
String password = "";

// Get authorization from the registry

PasswordAuthentication passwdAuth = new
PasswordAuthentication(username, password.toCharArray());

Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
```

Creating an Organization

A JAXR client creates the organization and populates it with the data before saving it. The `Organization` object is used to create an organization. This object includes the following objects:

- A `Name` object
- A `Description` object
- A `Key` object, representing the ID by which the organization is identified to the registry. The key is created by the registry and not by the user.
- A `PrimaryContactObject` - is a user object that refers to an authorized user of the registry. This object includes the following information about the authorized user:
 - `PersonName`, `TelephoneNumber`, `EmailAddress`, and/or `PostalAddress`
 - A collection of classification objects
 - Service objects and their associated service bindings objects

The following code illustrates how you can create an organization using the `Organization` method:

```
// Create Organization in memory

Organization org = businessLifeCycleManager.createOrganization
("Sun Microsystems");

// Create User -- maps to Contact for UDDI

User user = businessLifeCycleManager.createUser();

PersonName personName =
businessLifeCycleManager.createPersonName("Bob");
```

```

    TelephoneNumber telephoneNumber =
    businessLifeCycleManager.createTelephoneNumber();

    telephoneNumber.setNumber("650-241-8979");

    telephoneNumber.setType("office");

    Collection numbers = new ArrayList();

    numbers.add(telephoneNumber);

    EmailAddress email =
    businessLifeCycleManager.createEmailAddress("bob@sun.com",
    "office");

    Collection emailAddresses = new ArrayList();
    emailAddresses.add(email);

    user.setPersonName(personName);

    Collection telephoneNumbers = new ArrayList();

    telephoneNumbers.add(telephoneNumber);

    user.setTelephoneNumbers(telephoneNumbers);

    user.setEmailAddresses(emailAddresses);

    org.setPrimaryContact(user);

```

Adding Classifications

Organizations commonly belong to one or more classifications within one or more classification schemes or taxonomies. To establish a classification for an organization within a taxonomy, the client first locates the taxonomy using the `BusinessQueryManager`. The `findClassificationSchemeByName` method takes a set of `FindQualifier` objects as its first argument, but this argument can be null.

```

// Set classification scheme to NAICS

ClassificationScheme cScheme =
bqm.findClassificationSchemeByName(null, "ntis-gov:naics");

```

The client then creates a classification. For example, the following code sets up a classification for the organization within the NAICS taxonomy.

```

// Create and add classification

Classification classification = (Classification)
blcm.createClassification(cScheme, "Snack and Nonalcoholic
Beverage Bars", "722213");

```

```
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);
```

Services also use classifications, so you can use similar code to add a classification to a `Service` object.

Using Taxonomies

A taxonomy is represented by a `ClassificationScheme` object. This section describes how to use the implementation of JAXR in:

- [Defining Taxonomies](#)
- [Specifying Postal Address](#)

Defining Taxonomies

The JAXR specification requires a JAXR provider to be able to add user-defined taxonomies that can be used by JAXR clients. The implementation of JAXR in Sun ONE Application Server uses a simple file-based approach to provide taxonomies to the JAXR client. These files are read at run time, when the JAXR provider starts up.

The taxonomy structure is defined by the JAXR Predefined Concepts DTD, which is declared both in the file `jaxrconcepts.dtd` and in XML schema form in the file `jaxrconcepts.xsd`. The file `jaxrconcepts.xml` contains the taxonomies for the implementation of JAXR. All these files are contained in the *install_dir/share/lib/jaxr-impl.jar* file.

To add a user-defined taxonomy, follow the procedure given below:

Publish the `JAXRClassificationScheme` element for the taxonomy as a `ClassificationScheme` object in the registry that you will be accessing. For example, you can publish the `ClassificationScheme` object to the UDDI Registry Server. In order to publish a `ClassificationScheme` object, you must set its name. You also give the scheme a classification within a known classification scheme such as `uddi-org:types`. In the following code line, the name is the first argument of the `LifeCycleManager.createClassificationScheme` method call.

```
ClassificationScheme cScheme =
blcm.createClassificationScheme("MyScheme", "A Classification
Scheme");

ClassificationScheme uddiOrgTypes =
bqm.findClassificationSchemeByName(null, "uddi-org:types");
```

```

if (uddiOrgTypes != null)
{
    Classification classification =
    blcm.createClassification(uddiOrgTypes,"postalAddress",
    "categorization" );

    postalScheme.addClassification(classification);

    ExternalLink externalLink =
    blcm.createExternalLink("http://www.mycom.com/myscheme.html","My
    Scheme");

    postalScheme.addExternalLink(externalLink);

    Collection schemes = new ArrayList();

    schemes.add(cScheme);

    BulkResponse br = blcm.saveClassificationSchemes(schemes);
}

//The BulkResponse object returned by the
saveClassificationSchemes method contains the key for the
classification scheme, which you need to retrieve

if (br.getStatus() == JAXRResponse.STATUS_SUCCESS) {
    System.out.println("Saved ClassificationScheme");

    Collection schemeKeys = br.getCollection();

    Iterator keysIter = schemeKeys.iterator();

    while (keysIter.hasNext())
    {
        javax.xml.registry.infomodel.Key key =
        (javax.xml.registry.infomodel.Key) keysIter.next();

        System.out.println("The postalScheme key is " + key.getId());

        System.out.println("Use this key as the scheme" + " uuid in the
        taxonomy file");
    }
}

```

In an XML file, define a taxonomy structure that is compliant with the JAXR Predefined Concepts DTD. Enter the `ClassificationScheme` element in your taxonomy XML file by specifying the returned key ID value as the `id` attribute and the name as the `name` attribute. For example, the opening tag for the `JAXRClassificationScheme` element looks something like this (all on one line):

```
<JAXRClassificationScheme id="uuid:nnnnnnnnn-nnnn-nnnn-nnnn-
nnnnnnnnnnnn" name="MyScheme">
```

The `ClassificationScheme id` must be a UUID.

Enter each `JAXRConcept` element in your taxonomy XML file by specifying the following four attributes, in this order:

- a. `id` is the `JAXRClassificationScheme id` value, followed by a `/` separator, followed by the code of the `JAXRConcept` element
- b. `name` is the name of the `JAXRConcept` element
- c. `parent` is the immediate parent `id` (either the `ClassificationScheme id` or that of the parent `JAXRConcept`)
- d. `code` is the `JAXRConcept` element code value

The first `JAXRConcept` element in the `naics.xml` file looks like this (all on one line):

```
<JAXRConcept id="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2/11"
name="Agriculture, Forestry, Fishing and Hunting"
parent="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
code="11"></JAXRConcept>
```

To add the user-defined taxonomy structure to the JAXR provider, specify the system property `com.sun.xml.registry.userTaxonomyFileNames` when you run your client program. The command line (all on one line) would look like this. A vertical bar (`|`) is the file separator.

```
java myProgram
-DuserTaxonomyFileNames=c:\myfile\xxx.xml|c:\myfile\xxx2.xml
```

You can use a `<sysproperty>` tag to set this property in a `build.xml` file. Or, in your program, you can set the property as follows:

```
System.setProperty
("com.sun.xml.registry.userTaxonomyFileNames",
"c:\myfile\xxx.xml|c:\myfile\xxx2.xml");
```

Specifying Postal Address

The JAXR specification defines a postal address as a structured interface with attributes for street, city, country, and so on. The UDDI specification, on the other hand, defines a postal address as a free-form collection of address lines, each of which may also be assigned a meaning. To map the JAXR `PostalAddress` format to a known UDDI address format, you specify the UDDI format as a `ClassificationScheme` object and then specify the semantic equivalences between the concepts in the UDDI format classification scheme and the comments

in the JAXR `PostalAddress` classification scheme. The JAXR `PostalAddress` classification scheme is provided by the JAXR implementation of Sun ONE Application Server. A `PostalAddress` object has the fields `streetNumber`, `street`, `city`, `state`, `postalCode`, and `country`. These are predefined concepts in the `postalconcepts.xml` file, within the `ClassificationScheme` named `PostalAddressAttributes`.

To specify the mapping between the JAXR postal address format and another format, you need to set two connection properties:

- The `javax.xml.registry.postalAddressScheme` property, which specifies a postal address classification scheme for the connection.
- The `javax.xml.registry.semanticEquivalences` property, which specifies the semantic equivalences between the JAXR format and the other format.

First, you specify the postal address scheme using the id value from the `JAXRClassificationScheme` element (the UUID).

```
// Set properties for postal address mapping using my scheme
props.setProperty( " javax.xml.registry.postalAddressScheme",
    uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b" );
```

Next, you specify the mapping from the id of each `JAXRConcept` element in the default JAXR postal address scheme to the id of its counterpart in the IBM scheme:

```
props.setProperty( " javax.xml.registry.semanticEquivalences",
    urn:uuid:PostalAddressAttributes/StreetNumber, " +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/StreetAddressNumb
    er|" +
    "urn:uuid:PostalAddressAttributes/Street, " +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/StreetAddress|" +
    "urn:uuid:PostalAddressAttributes/City, " +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/City|" +
    "urn:uuid:PostalAddressAttributes/State, " +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/State|" +
    "urn:uuid:PostalAddressAttributes/PostalCode, " +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/ZipCode|" +
    "urn:uuid:PostalAddressAttributes/Country, " +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/Country" );
```

After you create the connection using these properties, you can create a postal address and assign it to the primary contact of the organization before you publish the organization.

For example,

```

String streetNumber = "99";
String street = "Imaginary Ave. Suite 33";
String city = "Imaginary City";
String state = "NY";
String country = "USA";
String postalCode = "00000";
String type = "";
PostalAddress postAddr =
blcm.createPostalAddress(streetNumber, street, city, state, country,
postalCode, type);
Collection postalAddresses = new ArrayList();
postalAddresses.add(postAddr);
primaryContact.setPostalAddresses(postalAddresses);

```

A JAXR query can then retrieve the postal address using `PostalAddress` methods, if the postal address scheme and semantic equivalences for the query are the same as those specified for the publication. To retrieve postal addresses when you do not know what postal address scheme was used to publish them, you can retrieve them as a collection of `Slot` objects. The `JAXRQueryPostal.java` sample program shows how to do this.

Adding Services and Service Bindings to an Organization

Many organizations add themselves to a registry in order to offer services, so the JAXR API has facilities to add services and service bindings to an organization.

Like an `Organization` object, a `Service` object has a name and a description. Also like an `Organization` object, it has a unique key that is generated by the registry when the service is registered. It may also have classifications associated with it.

A service also commonly has service bindings, which provide information about how to access the service. A `ServiceBinding` object normally has a description, an access URI, and a specification link, which provides the linkage between a service binding and a technical specification that describes how to use the service using the service binding.

The following code illustrates how to create a collection of services, add service bindings to a service, then add the services to the organization. It specifies an access URI but not a specification link. Because the access URI is not real and because JAXR by default checks for the validity of any published URI, the binding sets the `validateURI` property to false.

```
// Create services and service
```

```

Collection services = new ArrayList();
Service service = blcm.createService("My Service Name");
InternationalString is = blcm.createInternationalString("My
Service Description");
service.setDescription(is);

// Create service bindings

Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString("My Service Binding " +
"Description");
binding.setDescription(is);

binding.setValidateURI(false);
binding.setAccessURI("http://TheCoffeeBreak.com:8080/sb/");
serviceBindings.add(binding);

// Add service bindings to service
service.addServiceBindings(serviceBindings);

// Add service to services, then add services to organization
services.add(service);
org.addServices(services);

```

Publishing a Web Service to a UDDI Registry

JAXR provides the facility to publish your Web services to the UDDI registry. This section describes the steps to publish an existing Web service to the registry:

Publishing a service to a registry involves the following steps:

- Creating an Organization
- Creating its classification
- Creating services and service bindings
- Saving the information in the registry

To create a JAXR client that publishes a Web service to the registry, import the following required packages:

```

import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
import java.security.*;
import java.util.*;

```

Create a class that consists of a main method, a `makeConnection` method that establishes the connection to the registry, and an `executePublish` method, that publishes all the information about the service to the registry. The following code illustrates the creation of the main class `JAXRPublish`:

```
public class JAXRPublish {
    Connection connection = null;

    public JAXRPublish() {}

    public static void main(String[] args) {
        String queryURL =
            "http://www-3.ibm.com/services/uddi/v2beta/inquiryapi";

        String publishURL =
            "https://www-3.ibm.com/services/uddi/v2beta/protect/publishapi";

        String username = "";
        String password = "";

        JAXRPublish jp = new JAXRPublish();
        jp.makeConnection(queryURL, publishURL);
        jp.executePublish(username, password);
    }
}
```

The JAXR client must establish a connection to the UDDI registry and set the connection configuration properties. For detailed information, see [“Establishing a Connection” on page 96](#).

Create the connection, passing it the configuration properties. For detailed information, see [“Creating a Connection” on page 99](#).

Create an organization, its classification, its services, and save it to the registry.

For more information, see the following sections:

- [“Creating an Organization” on page 105](#)
- [“Adding Classifications” on page 106](#)
- [“Adding Services and Service Bindings to an Organization” on page 111](#)

The following code illustrates the steps to publish a Web service:

```
public void executePublish(String username, String password) {
    RegistryService rs = null;
    BusinessLifeCycleManager blcm = null;
    BusinessQueryManager bqm = null;
}
```

```

String orgName = "The Coffee Break";
String orgDesc = "Purveyor of the finest coffees. Established 1895";
String contactName = "Jane Doe";
String contactPhone = "(800) 555-1212";
String contactEmail = "jane.doe@TheCoffeeBreak.com";
String serviceName = "My Service Name";
String serviceDesc = "My Service Description";
String serviceBindingDesc = "My Service Binding Description";
String serviceBindingURI = "http://localhost:1024";
String scheme = "ntis-gov:naics";
String conceptName = "Snack and Nonalcoholic Beverage Bars";
String conceptCode = "722213";
try {

    java.io.BufferedInputStream bfInput = null;
    Properties propTemp = new Properties();
    bfInput = new java.io.BufferedInputStream (new
        java.io.FileInputStream("jaxr.properties"));
    propTemp.load(bfInput);
    bfInput.close(); orgName = propTemp.getProperty("org-name");
    orgDesc = propTemp.getProperty("org-desc");
    contactName = propTemp.getProperty("contact-name");
    contactPhone = propTemp.getProperty("contact-phone");
    contactEmail = propTemp.getProperty("contact-email");
    serviceName = propTemp.getProperty("service-name");
    serviceDesc = propTemp.getProperty("service-desc");
    serviceBindingDesc =
        propTemp.getProperty("service-binding-desc");
    serviceBindingURI = propTemp.getProperty("service-binding-uri");
    scheme = propTemp.getProperty("scheme");
    conceptName = propTemp.getProperty("concept");
    conceptCode = propTemp.getProperty("concept-code");
}
try {

    rs = connection.getRegistryService();
    blcm = rs.getBusinessLifeCycleManager();
    bqm = rs.getBusinessQueryManager();
    System.out.println("Got registry service, query " + "manager, and
        life cycle manager");

    // Get authorization from the registry

```

```

PasswordAuthentication passwdAuth = new
PasswordAuthentication(username,password.toCharArray());
Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
System.out.println("Established security credentials");

// Create organization name and description

Organization org = blcm.createOrganization(orgName);
InternationalString s = blcm.createInternationalString(orgDesc);
org.setDescription(s);

// Create primary contact, set name

User primaryContact = blcm.createUser();
PersonName pName = blcm.createPersonName(contactName);
primaryContact.setPersonName(pName);

// Set primary contact phone number

TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber(contactPhone);
Collection phoneNums = new ArrayList();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);

// Set primary contact email address

EmailAddress emailAddress =
blcm.createEmailAddress(contactEmail);
Collection emailAddresses = new ArrayList();
emailAddresses.add(emailAddress);
primaryContact.setEmailAddresses(emailAddresses);

// Set primary contact for organization

org.setPrimaryContact(primaryContact);

// Set classification scheme to NAICS

ClassificationScheme cScheme =
bqm.findClassificationSchemeByName(null,scheme);

if (cScheme != null) {

    // Create and add classification

    Classification classification = (Classification)
    blcm.createClassification(cScheme, conceptName, conceptCode);
    Collection classifications = new ArrayList();
    classifications.add(classification);
    org.addClassifications(classifications);
}

```

```

// Create services and service

Collection services = new ArrayList();
Service service = blcm.createService(serviceName);
InternationalString is =
blcm.createInternationalString(serviceDesc);
service.setDescription(is);

// Create service bindings

Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString(serviceBindingDesc);
binding.setDescription(is);

// allow us to publish a bogus URL without an error

binding.setValidateURI(false);
binding.setAccessURI(serviceBindingURI);
serviceBindings.add(binding);

// Add service bindings to service

service.addServiceBindings(serviceBindings);

// Add service to services, then add services to organization

services.add(service);
org.addServices(services);

// Add organization and submit to registry

// Retrieve key if successful

Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);
Collection exceptions = response.getExceptions();

if (exceptions == null) {

    System.out.println("Organization saved");
    Collection keys = response.getCollection();
    Iterator keyIter = keys.iterator();

    if (keyIter.hasNext()) {

```

```

        javax.xml.registry.infomodel.Key orgKey =
        (javax.xml.registry.infomodel.Key) keyIter.next();
        String id = orgKey.getId();
        System.out.println("Organization key is " + id);
        org.setKey(orgKey);
    }
}
}
}

```

Assembling and Deploying a JAXR Client

The following steps describe how you can assemble and deploy a JAXR client:

1. Execute the default target `core` to compile Java files and build the `.jar` file. The `.jar` file has the JAXR API classes and a wrapper client class.

```
asant core
```

2. Build Javadocs. For example:

Execute the following `asant` command under `install_dir/samples/webservices/jaxr/src/` to create javadocs:

```
asant javadoc
```

3. Deploy the client.
 - a. JAXR can be configured to access various registries. You can use either your own registry server or you can use public registry servers. If you choose to use a public registry server, make certain that you can publish to the registry server. Modify `jaxr.properties` with the correct parameters. This file contains the following parameters:
 - `query-url` - Fully qualified inquiry URI for the registry server.
 - `publish-url` - Fully qualified publish URI for the registry server.
 - `username` - Username to publish an organization to the registry server.
 - `password` - Password to publish an organization to the registry server.
 - `query-string` - The search string to search for in the registry.
 - `key-string` - The key of the organization to be deleted from the registry server.

- b.** If you wish to publish to the registry server, modify the publish organization info section in the `jaxr.properties`, if required.
- 4.** Run the client using the following command:

```
asant run
```

Sample JAXR Client

You can find various sample applications that demonstrate the use of JAXR API in Sun ONE Application Server environment at the following location:

install_dir/samples/webservices/jaxr/

XML Schema Definitions

XML Schema Definition (XSD) is a W3C standard for an XML-based type system known as XML Schema. The language used to define is an XML grammar known as XML Schema Definition Language. Web services use XML as the underlying format for representing messages and data. Thus, XSD is a natural choice as the Web service type system.

For more information on XSD, visit the following URL:

<http://www.w3.org/2001/XMLSchema>

This appendix provides XSDs for the following files used in developing JAX-RPC Web services and clients:

- [XML Schema for wscompile Configuration File](#)
- [XML Schema for Deployment Descriptors](#)
- [XML Schema for Exported wscompile Model Files](#)
- [XML Schema for Runtime Descriptors](#)

XML Schema for wscompile Configuration File

The following code is the XML Schema used for creating wscompile configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://java.sun.com/xml/ns/jax-rpc/ri/config"
    targetNamespace="http://java.sun.com/xml/ns/jax-rpc/ri/config"
```

```
elementFormDefault="qualified"
attributeFormDefault="unqualified"
version="1.0">
```

```
<xsd:annotation>
```

```
  <xsd:documentation>
```

```
    This is the schema for wscompile configuration files.
```

```
    The only allowed top-level element is "configuration".
```

```
  </xsd:documentation>
```

```
</xsd:annotation>
```

```
<xsd:element name="configuration">
```

```
  <xsd:annotation>
```

```
    <xsd:documentation>
```

```
      The top-level element. It must contain one out of three possible
      elements, corresponding to three different ways to feed service
      information to the tool.
```

```
      Elements: (mutually exclusive)
```

```
      "service" - a service description based on a set of service endpoint
      interfaces;
```

```
      "wsdl" - a WSDL document to import and process;
```

```
      "modelfile" - a previously saved model file (-model option in
      wscompile).
```

```
    </xsd:documentation>
```

```
</xsd:annotation>
```

```
  <xsd:complexType>
```

```
    <xsd:sequence>
```

```
      <xsd:choice>
```

```
        <xsd:element name="service" type="tns:serviceType"/>
```

```
        <xsd:element name="wsdl" type="tns:wsdlType"/>
```

```
        <xsd:element name="modelfile" type="tns:modelfileType"/>
```

```
      </xsd:choice>
```

```
    </xsd:sequence>
```

```

        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="serviceType">
        <xsd:annotation>
            <xsd:documentation>
                A description of a service based on a set of Java interfaces (called
                "service endpoint interfaces" in the spec).

                Attributes:

                "name" - service name;

                "targetNamespace" - target namespace for the generated WSDL
                document;

                "typeNamespace" - target namespace for the XML Schema embedded in
                the generated WSDL document;

                "packageName" - name of the Java package to use by default.

                Elements:

                "interface"* - a sequence of service endpoint interface
                descriptions;

                "typeMappingRegistry"? - the type mapping registry to use for
                this service;

                "handlerChains"? - default handler chains for the endpoints in
                this service;

                "namespaceMappingRegistry"? - XML namespace to Java package
                mapping information.
            </xsd:documentation>
        </xsd:annotation>

        <xsd:sequence>

            <xsd:element name="interface" type="tns:interfaceType" minOccurs="0"
            maxOccurs="unbounded"/>

            <xsd:element name="typeMappingRegistry"
            type="tns:typeMappingRegistryType" minOccurs="0"/>

            <xsd:element name="handlerChains" type="tns:handlerChainsType"
            minOccurs="0"/>

```

```

        <xsd:element name="namespaceMappingRegistry"
            type="tns:namespaceMappingRegistryType" minOccurs="0"/>
    </xsd:sequence>

    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="typeNameSpace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="packageName" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="interfaceType">
    <xsd:annotation>
        <xsd:documentation>
            An endpoint definition based on a service endpoint interface.

            Attributes:

                "name" - name of the service endpoint interface (a Java
                    interface);

                "servantName" (optional) - name of the service endpoint
                    implementation class;

                "soapAction" (optional) - SOAPAction string to use for all
                    operations in the interface;

                "soapActionBase" (optional) - base URI for the SOAPAction string;
                    the SOAPAction for a given operation will be obtained by
                    appending the operation name to the value provided here; this
                    attribute is exclusive with the "soapAction" one.

            Elements:

                "handlerChains" - specifies the handler chains for this endpoint.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="handlerChains" type="tns:handlerChainsType"
            minOccurs="0"/>
    </xsd:sequence>

```

```

<xsd:attribute name="name" type="xsd:string" use="required"/>
<xsd:attribute name="servantName" type="xsd:string"/>
<xsd:attribute name="soapAction" type="xsd:string"/>
<xsd:attribute name="soapActionBase" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="wsdlType">
<xsd:annotation>
    <xsd:documentation>
        A description of a service based on an existing WSDL document.
        Attributes:
            "location" - URL of the WSDL document;
            "packageName" - name of the Java package to use by default.
        Elements:
            "typeMappingRegistry"? - the type mapping registry to use for
            this service;
            "handlerChains"? - default handler chains for the endpoints in
            this service;
            "namespaceMappingRegistry"? - XML namespace to Java package
            mapping information.
    </xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="typeMappingRegistry"
        type="tns:typeMappingRegistryType" minOccurs="0"/>
    <xsd:element name="handlerChains" type="tns:handlerChainsType"
        minOccurs="0"/>
    <xsd:element name="namespaceMappingRegistry"
        type="tns:namespaceMappingRegistryType" minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="location" type="xsd:anyURI" use="required"/>
<xsd:attribute name="packageName" type="xsd:string" use="required"/>
</xsd:complexType>

```

```

<xsd:complexType name="modelFileType">
<xsd:annotation>
    <xsd:documentation>
        A description of a service based on an existing model file.
        Attributes:
            "location" - URL of the model file (typically ending in .xml.gz);
    </xsd:documentation>
</xsd:annotation>
<xsd:sequence>
</xsd:sequence>
    <xsd:attribute name="location" type="xsd:anyURI" use="required"/>
</xsd:complexType>
<xsd:complexType name="handlerChainsType">
<xsd:annotation>
    <xsd:documentation>
        A set of handlerChains.
    </xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="chain" type="tns:chainType" minOccurs="0"
        maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="chainType">
<xsd:annotation>
    <xsd:documentation>
        A handler chain.
        Attributes:

```

```

        "runAt" - specifies whether the chain must run on the client or
        the server;

        "roles" - the SOAP roles for the chain.

    Elements:

        "handler"* - a sequence of handlers that form this chain.
    </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
    <xsd:element name="handler" type="tns:handlerType" minOccurs="0"
    maxOccurs="unbounded"/>
</xsd:sequence>

    <xsd:attribute name="runAt" type="tns:runAtType" use="required"/>
    <xsd:attribute name="roles" type="tns:roleListType"/>
</xsd:complexType>

<xsd:simpleType name="roleListType">
<xsd:annotation>
<xsd:documentation>
    A list of SOAP roles, i.e. a list of URIs.
</xsd:documentation>
    </xsd:annotation>
    <xsd:list itemType="xsd:anyURI"/>
</xsd:simpleType>

<xsd:complexType name="handlerType">
<xsd:annotation>
<xsd:documentation>
    A handler description.
    Attributes:
        "className" - the name of the handler's class;
        "headers" - the names of the headers consumed by this handler.
    Elements:

```

```

        "property"* - initialization properties for this handler.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="property" type="tns:propertyType" minOccurs="0"
        maxOccurs="unbounded"/>
</xsd:sequence>
    <xsd:attribute name="className" type="xsd:string" use="required"/>
    <xsd:attribute name="headers" type="tns:headerListType"/>
</xsd:complexType>

<xsd:simpleType name="headerListType">
    <xsd:annotation>
        <xsd:documentation>
            A list of header names, i.e. a list of QNames.
        </xsd:documentation>
    </xsd:annotation>
<xsd:list itemType="xsd:QName"/>
</xsd:simpleType>

<xsd:complexType name="propertyType">
    <xsd:annotation>
        <xsd:documentation>
            An initialization property for a handler.
            Attributes:
                "name" - the name of the property;
                "value" - its value.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
</xsd:sequence>

```

```

        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="value" type="xsd:string" use="required"/>
    </xsd:complexType>

<xsd:simpleType name="runAtType">
    <xsd:annotation>
    <xsd:documentation>
        The places a handler chain can run at, one of "client" or
        "server".
    </xsd:documentation>
    </xsd:annotation>

    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="client"/>
        <xsd:enumeration value="server"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="typeMappingRegistryType">
    <xsd:annotation>
    <xsd:documentation>
        A type mapping registry.
        Elements:
        "import"? - a list of XML Schema documents that describe
        user-defined types.
        "typeMapping"* - a sequence of type mappings, one per encoding.
        "additionalTypes"? - a list of additional Java types that should
        be processed even if don't appear in the interfaces for the
        service.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>

```

```

        <xsd:element name="import" type="tns:importType" minOccurs="0"/>
        <xsd:element name="typeMapping" type="tns:typeMappingType"
minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="additionalTypes"
type="tns:additionalTypesType" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="importType">
    <xsd:annotation>
    <xsd:documentation>
        A list of schema documents to import, usually describing schema
        types used by pluggable serializers.
    Elements:
        "schema"* - a list of schema documents to import.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="schema" type="tns:schemaType" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="schemaType">
    <xsd:annotation>
    <xsd:documentation>
        One schema document to be imported.
    Attributes:
        "namespace" - the namespace that the document describes;
        "location" - a URL pointing to the schema document.
    </xsd:documentation>
    </xsd:annotation>

```

```

        <xsd:sequence>
        </xsd:sequence>
        <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
        <xsd:attribute name="location" type="xsd:anyURI" use="required"/>
    </xsd:complexType>

```

```

<xsd:complexType name="typeMappingType">
    <xsd:annotation>
    <xsd:documentation>
        A type mapping for a particular encoding.
    Attributes:
        "encodingStyle" - the URI denoting the encoding.
    Elements:
        "entry"* - a list of type mapping entries.
    </xsd:documentation>
    </xsd:annotation>

```

```

        <xsd:sequence>
            <xsd:element name="entry" type="tns:entryType" minOccurs="0"
                maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="encodingStyle" type="xsd:anyURI"
            use="required"/>
    </xsd:complexType>

```

```

<xsd:complexType name="entryType">
    <xsd:annotation>
    <xsd:documentation>
        An entry in a type mapping.
    Attributes:

```

```

        "schemaType" - the name of a schema type;
        "javaType" - the name of the corresponding Java class;
        "serializerFactory" - the name of the serializer factory class to
        use for this type;
        "deserializerFactory" - the name of the deserializer factory
        class to use for this type.
    </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
</xsd:sequence>

<xsd:attribute name="schemaType" type="xsd:QName" use="required"/>
<xsd:attribute name="javaType" type="xsd:string" use="required"/>
<xsd:attribute name="serializerFactory" type="xsd:string"
use="required"/>
<xsd:attribute name="deserializerFactory" type="xsd:string"
use="required"/>
</xsd:complexType>

<xsd:complexType name="additionalTypesType">
    <xsd:annotation>
    <xsd:documentation>
        A list of additional Java types to be processed by the tool..
    Elements:
        "class"* - a list of classes to be processed.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="class" type="tns:classType" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

```

```

<xsd:complexType name="classType">
    <xsd:annotation>
    <xsd:documentation>
        A Java class description.
    Attributes:
        "class" - the name of the class.
    </xsd:documentation>
</xsd:annotation>
<xsd:sequence> </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

```

```

<xsd:complexType name="namespaceMappingRegistryType">
    <xsd:annotation>
    <xsd:documentation>
        A registry mapping XML namespace to/from Java packages.
    Elements:
        "namespaceMapping"* - a list of mappings.
    </xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="namespaceMapping"
        type="tns:namespaceMappingType" minOccurs="0"
        maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>

```

```

<xsd:complexType name="namespaceMappingType">
    <xsd:annotation>
    <xsd:documentation>

```

One XML namespace to/from Java package mapping.

Attributes:

"namespace" - XML namespace name;

"packageName" - Java package name.

</xsd:documentation>

</xsd:annotation>

<xsd:sequence>

</xsd:sequence>

<xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>

<xsd:attribute name="packageName" type="xsd:string" use="required"/>

</xsd:complexType>

</xsd:schema>

XML Schema for Deployment Descriptors

The following code is the XML schema for creating Web service deployment descriptors:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsd:schema
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
  xmlns:tns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
```

```
  targetNamespace="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
```

```
  elementFormDefault="qualified"
```

```
  attributeFormDefault="unqualified"
```

```
  version="1.0">
```

```
  <xsd:annotation>
```

```
  <xsd:documentation>
```

```
    This is the schema for the deployment descriptors(jaxrpc-ri.xml).
```

```
    The top level element must be "webServices".
```

```
  </xsd:documentation>
```

```
  </xsd:annotation>
```

```

<xsd:element name="webServices">
    <xsd:annotation>
    <xsd:documentation>
        The top-level element.
    Attributes:
        "version" - version number;
        "targetNamespaceBase"? - base URI for the targetNamespace of the
        WSDL documents generated for the endpoints that don't have their
        own model file;
        "typeNameSpaceBase"? - same as "targetNamespaceBase", but used for
        the XML Schema documents embedded in the generated WSDL
        documents;
        "urlPatternBase"? - base URL pattern for all endpoints; it can be
        overridden by using and "endpointMapping" (see below).
    For all these base properties, the value used for a particular
    endpoint is given by the base value with the endpoint name appended
    to it.
    Elements:
        "endpoint"* - a sequence of endpoint descriptions;
        "endpointMapping"* - a sequence of endpoint mappings.
    </xsd:documentation>
    </xsd:annotation>

<xsd:complexType>
    <xsd:sequence>
        <xsd:element name="endpoint"
            type="tns:endpointType" minOccurs="0" maxOccurs="unbounded" />
        <xsd:element
            name="endpointMapping" type="tns:endpointMappingType"
            minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="version" type="xsd:string" use="required" />
    <xsd:attribute name="targetNamespaceBase" type="xsd:string" />
    <xsd:attribute name="typeNameSpaceBase" type="xsd:string" />

```

```

        <xsd:attribute name="urlPatternBase" type="xsd:string"/>
    </xsd:complexType>
</xsd:element>

<xsd:complexType name="endpointType">
    <xsd:annotation>
    <xsd:documentation>
        An endpoint description.
    Attributes:
        "name" - the name of the endpoint;
        "displayName"? - a human-readable name for the endpoint;
        "description"? - a description of the endpoint;
        "interface"? - the name of the service endpoint interface;
        "implementation"? - the name of the service implementation class;
        "model"? - a resource pointing to a model file describing the
        endpoint.
    Elements:
        "handlerChains"? - the handler chains for the endpoint.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="handlerChains" type="tns:handlerChainsType"
            minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="displayName" type="xsd:string"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="interface" type="xsd:string"/>
    <xsd:attribute name="implementation" type="xsd:string"/>
    <xsd:attribute name="model" type="xsd:anyURI"/>
</xsd:complexType>

```

```

<xsd:complexType name="endpointMappingType">
    <xsd:annotation>
    <xsd:documentation>
        An endpoint mapping entry, similar to servlet-mapping in web.xml.
    Attributes:
        "endpointName" - the name of the endpoint;
        "urlPattern" - the URL pattern for the endpoint.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
    </xsd:sequence>
    <xsd:attribute name="endpointName" type="xsd:string"
        use="required"/>
    <xsd:attribute name="urlPattern" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="handlerChainsType">
    <xsd:annotation>
    <xsd:documentation>
        A set of handlerChains. In a deployment descriptor, only
        server-side chains make sense.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="chain" type="tns:chainType" minOccurs="0"
            maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="chainType">
    <xsd:annotation>

```

```

xsd:documentation>
    A handler chain.
Attributes:
    "runAt" - specifies whether the chain must run on the client or
    the server;
    "roles" - the SOAP roles for the chain.
Elements:
    "handler"* - a sequence of handlers that form this chain.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="handler" type="tns:handlerType" minOccurs="0"
    maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="runAt" type="tns:runAtType" use="required"/>
<xsd:attribute name="roles" type="tns:roleListType"/>
</xsd:complexType>

<xsd:simpleType name="roleListType">
    <xsd:annotation>
    <xsd:documentation>
        A list of SOAP roles, i.e. a list of URIs.
    </xsd:documentation>
    </xsd:annotation>
    <xsd:list itemType="xsd:anyURI"/>
</xsd:simpleType>

<xsd:complexType name="handlerType">
    <xsd:annotation>
    <xsd:documentation>
        A handler description.

```

Attributes:

"className" - the name of the handler's class;

"headers" - the names of the headers consumed by this handler.

Elements:

"property"* - initialization properties for this handler.

```
</xsd:documentation>
```

```
</xsd:annotation>
```

```
<xsd:sequence>
```

```
  <xsd:element name="property" type="tns:propertyType"
    minOccurs="0" maxOccurs="unbounded"/>
```

```
</xsd:sequence>
```

```
<xsd:attribute name="className" type="xsd:string" use="required"/>
```

```
<xsd:attribute name="headers" type="tns:headerListType"/>
```

```
</xsd:complexType>
```

```
<xsd:simpleType name="headerListType">
```

```
  <xsd:annotation>
```

```
  <xsd:documentation>
```

A list of header names, i.e. a list of QNames.

```
</xsd:documentation>
```

```
</xsd:annotation>
```

```
  <xsd:list itemType="xsd:QName"/>
```

```
</xsd:simpleType>
```

```
<xsd:complexType name="propertyType">
```

```
  <xsd:annotation>
```

```
  <xsd:documentation>
```

An initialization property for a handler.

Attributes:

"name" - the name of the property;

"value" - its value.

```

        </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="value" type="xsd:string" use="required"/>
    </xsd:complexType>

    <xsd:simpleType name="runAtType">
        <xsd:annotation>
        <xsd:documentation>
            The places a handler chain can run at, one of "client" or
            "server".
        </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="client"/>
            <xsd:enumeration value="server"/>
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>

```

XML Schema for Exported wscompile Model Files

The following code is the XML schema for exported wscompile model files:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://java.sun.com/xml/ns/jax-rpc/ri/model"

```

```

targetNamespace="http://java.sun.com/xml/ns/jax-rpc/ri/model"
elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1.0">
<xsd:annotation>
<xsd:documentation>
    This is the schema for exported wscompile model files. An
    exported model is a graph of objects, some of which are
    "immediate" (e.g. integers, strings). Immediate objects have
    exactly one value.

    Non-immediate objects have a set of object-valued properties.
</xsd:documentation>
</xsd:annotation>

<xsd:element name="model" form="qualified">
    <xsd:annotation>
    <xsd:documentation>
        A model is a sequence of definitions, which can be of three
        kinds:

        "object" - object definition;
        "iobject" - immediate object definition;.
        "property" - property definition.

        In addition, a model has a version number ("version" attribute).
    </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:choice>
            <xsd:element name="object" type="tns:objectType"/>
            <xsd:element name="iobject" type="tns:iobjectType"/>
            <xsd:element name="property" type="tns:propertyType"/>
        </xsd:choice>
        </xsd:sequence>

```

```

        <xsd:attribute name="version" type="xsd:string" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:complexType name="objectType">
    <xsd:annotation>
        <xsd:documentation>
            Object definition. Contains a unique id as well as a type name
            for the object.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="iobjectType">
    <xsd:annotation>
        <xsd:documentation>
            Immediate object definition. In addition to id and type, it
            contains a value.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="propertyType">
    <xsd:annotation>

```

```

        <xsd:documentation>

        Property definition. It says that the "subject" (identified by
        id) has a property called "name" with the object of id "value" as
        its value.

        </xsd:documentation>

    </xsd:annotation>

</xsd:sequence>

<xsd:attribute name="name" type="xsd:string" use="required"/>
<xsd:attribute name="subject" type="xsd:string" use="required"/>
<xsd:attribute name="value" type="xsd:string" use="required"/>

</xsd:complexType>
</xsd:schema>

```

XML Schema for Runtime Descriptors

The following code is the XML schema for runtime descriptors:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema

    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://java.sun.com/xml/ns/jax-rpc/ri/runtime"
    targetNamespace="http://java.sun.com/xml/ns/jax-rpc/ri/runtime"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="1.0">

    <xsd:annotation>

        <xsd:documentation>

        This is the schema for the runtime descriptors
        (jaxrpc-ri-runtime.xml).

        The top-level element must be "endpoints".

        </xsd:documentation>

    </xsd:annotation>

```

```

<xsd:element name="endpoints">
  <xsd:annotation>
    <xsd:documentation>
      The top level element. It contains a "version" attribute and a
      sequence of endpoint definitions.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="endpoint" type="tns:endpointType"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="version" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="endpointType">
  <xsd:annotation>
    <xsd:documentation>
      An endpoint definition has several attributes:
      "name" - the endpoint name;
      "interface" - the name of the Java interface for the endpoint
      (called "service endpoint interface" in the spec);
      "implementation" - the name of the endpoint implementation class;
      "tie" - the name of the tie class for the endpoint;
      "model"? - the name of a resource corresponding to the model file
      for the endpoint;
      "wsdl"? - the name of a resource corresponding to the WSDL
      document for the endpoint;
      "service"? - the QName of the WSDL service that owns this
      endpoint;
      "port"? - the QName of the WSDL port for this endpoint;
      "urlpattern" - the URL pattern this endpoint is mapped to.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" use="required"/>
    <xsd:element name="interface" type="xsd:string" use="required"/>
    <xsd:element name="implementation" type="xsd:string" use="required"/>
    <xsd:element name="tie" type="xsd:string" use="required"/>
    <xsd:element name="model" type="xsd:string" use="optional"/>
    <xsd:element name="wsdl" type="xsd:string" use="optional"/>
    <xsd:element name="service" type="xsd:string" use="optional"/>
    <xsd:element name="port" type="xsd:string" use="optional"/>
    <xsd:element name="urlpattern" type="xsd:string" use="optional"/>
  </xsd:sequence>
</xsd:complexType>

```

```

</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
</xsd:sequence>
<xsd:attribute name="name" type="xsd:string" use="required"/>
<xsd:attribute name="interface" type="xsd:string" use="required"/>
<xsd:attribute name="implementation" type="xsd:string"
use="required"/>
<xsd:attribute name="tie" type="xsd:string" use="required"/>
<xsd:attribute name="model" type="xsd:string"/>
<xsd:attribute name="wsdl" type="xsd:anyURI"/>
<xsd:attribute name="service" type="xsd:anyURI"/>
<xsd:attribute name="port" type="xsd:anyURI"/>
<xsd:attribute name="urlpattern" type="xsd:anyURI" use="required"/>
</xsd:complexType>
</xsd:schema>

```


Index

A

- accessing elements [80](#)
- adding attachment to message [86](#)
 - accessing attachment part [89](#)
- ant build file
 - creating [29](#)
- ant tool [28](#)
 - using ant tasks [28](#)
- application classes [73](#)
- arrays [73](#)
- asadmin deploy command [42](#)
- assembling and deploying JAX-RPC web services [38](#)
 - war file [39](#)
- asynchronous message [77](#)

B

- basic authentication over SSL [55](#)
- body object [80](#)
- build.xml [29](#)

C

- call interface method [49](#)

- classification scheme [106](#)
- Client Certificate Authentication [59](#)
- client environment [29](#)
 - using ant tasks [28](#)
- client jar files [29](#)
- client using a messaging provider [77](#)
- client without using a messaging provider [76](#)
- coffee break example [26](#)
- config.xml [66](#)
- configuration file [66](#)
 - web services attributes [40](#)
 - with a WSDL document [67](#)
 - with rmi interfaces [66](#)
- connection [83](#)
- creating a SOAP client [83](#)
 - accessing attachment part [89](#)
 - adding attachment to message [86](#)
 - adding contents to a message [86](#)
 - creating a message [85](#)
 - retrieve message content [88](#)
 - sending a message [88](#)

D

- developing a JAXR client [95](#)
- developing JAX-RPC clients
 - assemble the client [46](#)

- call interface method [49](#)
- compile the client [46](#)
- dynamic proxy method [48](#)
- generated stubs method [43](#)
- run the client [47](#)
- using a WSDL [50](#)

DII client

- call interface approach [49](#)
- dynamic proxy approach [48](#)

document-oriented model [16](#)

dynamic proxy method [48](#)

E

ebXML [18](#)

elements of a SOAP message [78](#)

establishing connection

- setting properties [97](#)

F

files

- build.xml [29](#)
- JAX-RPC runtime descriptor [51](#)
- jaxrpc-ri.xml [69](#)

G

generated stubs method [43](#)

I

implementing web services

- coffee break example [26](#)

invoking web services [42](#)

- creating a SOAP client [83](#)
- developing a JAXR client [95](#)

J

J2SE SDK classes [72](#)

Java APIs [21](#)

- JAXM [75](#)

- JAXR [95](#)

- SAAJ [75](#)

JavaBeans components [74](#)

javax.activation.DataHandler [88](#)

javax.xml.rpc.Service [48](#)

javax.xml.soap package [71](#)

javax.xml.soap.SoAPConnection.Call() [88](#)

JAXM [24](#), [75](#)

JAXMServlet class [90](#)

JAXP [22](#)

- DOM [22](#)

- SAX [22](#)

- XSLT [22](#)

- XSLTC [22](#)

JAXR [23](#), [95](#)

JAXR client

- assemble [117](#)

- authorization [100](#)

- creating connection [99](#)

- deploy [117](#)

- establishing connection [96](#)

- execute [117](#)

- getting access to a registry [96](#)

- obtaining registry service [100](#)

- publishing [117](#)

- querying a registry [100](#)

- querying registry [100](#)

JAXR connection properties [97](#)

jaxr.properties file [117](#)

JAX-RPC [23](#)

- JAX-RPC clients [43](#)

- types supported by JAX-RPC [72](#)

JAX-RPC clients

- assembling [51](#)

- deploying [51](#)

- invoking an EJB [52](#)

JAX-RPC sample applications [52](#)

JAX-RPC tools [63](#)

- wscompile tool [63](#)

- wsdeploy tool [68](#)

JAX-RPC web services [35](#)
 configuration file [40](#)

M

managing registry [104](#)
 adding classifications [106](#)
 adding service and bindings [111](#)
 creating organization [105](#)
 finding taxonomy [106](#)
 getting authorization [104](#)
 message endpoint [82](#)
 constructing an endpoint [82](#)
 message queue [20](#)
 message-oriented model [17](#)
 model file [66](#)
 mutual authentication over SSL [58](#)

N

namespace mappings [70](#)

O

onMessage method [90](#)
 override JAXP 1.1 implementation [31](#)

P

parts of a SOAP message [78](#)
 AttachmentPart object [79](#)
 body [80](#)
 SOAPEnvelope [79](#)
 SOAPHeader [80](#)
 SOAPMessage [79](#)
 SOAPPart [79](#)
 predefined faultcode [92](#)

pre-defined SOAP namespaces [81](#)
 primitive types [73](#)
 ProviderConnection object [83](#)

Q

query a registry [100](#)
 querying registry
 based on WSDL specification [102](#)
 find organizations by classification [101](#)
 find organizations by description [102](#)
 find organizations by name [101](#)
 find organizations by services and service
 bindings [103](#)

S

SAAJ [24, 75](#)
 sample JAXR Client [118](#)
 securing web services [54](#)
 basic authentication over SSL [55](#)
 mutual authentication over SSL [58](#)
 security properties [57](#)
 security properties [57](#)
 securing web services
 security properties [57](#)
 sending SOAP message [88](#)
 service bindings [111](#)
 SOAP [17](#)
 SOAP client
 client using a messaging provider [77](#)
 client without a messaging provider [76](#)
 samples [94](#)
 SOAP client messaging models
 client using a messaging provider [77](#)
 client without using a messaging provider [76](#)
 SOAP client scenarios [76](#)
 SOAP handlers [71](#)
 SOAP message fault handling [91](#)
 defining SOAP fault [93](#)
 SOAPFault object [91](#)

- SOAP serialization [81](#)
- SOAP service [90](#)
 - creating [90](#)
 - samples [94](#)
- SOAPConnection object [83](#)
- SOAPEnvelope object [79](#)
- SOAPFault subelements [92](#)
 - detail [92](#)
 - faultactor [92](#)
 - faultcode [92](#)
 - faultstring [92](#)
- SOAPHeader object [80](#)
- SOAPMessage object [79](#)
- SOAPPart object [79](#)
- specifying postal address [109](#)
- stubs [45](#)
- Sun customer support [13](#)
- Sun ONE web services features [20](#)
- synchronous message [76](#)
- syntax of asadmin deploy command [42](#)

T

- taxonomy [106](#)
 - defining [107](#)
- ties [45](#)
- type system [119](#)
- types supported by JAX-RPC [72](#)
 - application classes [73](#)
 - arrays [73](#)
 - J2SE SDK classes [72](#)
 - JavaBeans components [74](#)
 - primitives [73](#)

U

- UDDI [18](#)
- UDDI, registry [18](#)
- using ant tasks [28](#)
- using JAXP 1.2 implementation [31](#)

- using namespaces [81](#)
- using wscompile tool [63](#)
- using wsdeploy tool [68](#)

W

- war file [39](#)
- web service
 - endpoint [34](#)
- web service models
 - asynchronous [17](#)
 - synchronous [16](#)
- web services [15](#)
 - assembling and deploying JAX-RPC web services [38](#)
 - building clients [42](#)
 - invoking web services [42](#)
 - securing web services [54](#)
 - working of web services [19](#)
- web services standards [17](#)
 - ebXML [18](#)
 - SOAP [17](#)
 - UDDI [18](#)
 - WSDL [18](#)
- web.xml
 - adding security elements [56](#)
- working of web services [19](#)
- wscompile configuration file [66](#)
 - with rmi interfaces [66](#)
 - with wsdl [67](#)
- wscompile tool [63](#)
 - wscompile command options [64](#)
- wsdeploy tool [68](#)
 - jaxrpc-ri.xml file [69](#)
 - wsdeploy command options [68](#)
- WSDL [18](#)

X

- XML namespace [80](#)
 - using name spaces [81](#)

XML schema definition [119](#)

XSD [119](#)

 deployment descriptors [132](#)

 exported wscompile model files [138](#)

 runtime descriptors [141](#)

 wscompile configuration [119](#)

