Oracle9*i* Application Server

Oracle9*i*AS SOAP Developer's Guide

Release 1 (v1.0.2.2)

May 2001

Part No. A90297-01

ORACLE

Oracle9*i* Application Server Oracle9*i*AS SOAP Developer's Guide, Release 1 (v1.0.2.2)

Part No. A90297-01

Primary Author: Thomas Van Raalte

Contributors: Julie Basu, Diane Davison, Anish Karmarkar, Olivier LeDiouris, Steve Muench, Cyril Scott

Editor: Kay Kaufmann

Contents

2 Using Oracle SOAP with Java Services

3 SOAP Parameters and Encodings

6 Writing SOAP Providers

7 Writing Deployment Managers

8 SOAP Administration

A Apache Software License, Version 1.1

Index

Send Us Your Comments

Oracle9*i* Application Server Oracle9*i*AS SOAP Developer's Guide, Release 1 (v1.0.2.2)

Part No. A90297-01

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

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

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

- Electronic mail: iasdocs_us@oracle.com
- Postal service:
  Oracle Corporation
  Oracle9*i* Application Server Oracle9*i*AS SOAP Developer's Guide
  500 Oracle Parkway M/S 6op4
  Redwood Shores, CA 94065
  USA

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

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

Preface

The Simple Object Access Protocol (SOAP), is a lightweight, XML-based protocol for exchanging information in a decentralized, distributed environment. This guide describes Oracle SOAP. Oracle SOAP is an implementation of the Simple Object Access Protocol that is based on the Apache SOAP open source implementation.

This preface contains the following topics:

Audience

The *Oracle9i* Application Server Oracle9*i*AS SOAP Developer's Guide is intended for application programmers, system administrators, and other users who perform the following tasks:

- Configure software installed on the Oracle9*i* Application Server
- Create programs that implement SOAP services
- Create Java programs that run as SOAP clients

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

Organization

This document contains:

Chapter 1, "Simple Object Access Protocol Overview"

This chapter introduces the basic concepts for the Simple Object Access Protocol and provides a description of the SOAP architecture.

Chapter 2, "Using Oracle SOAP with Java Services"

This chapter provides an introduction to the procedures you use to write a SOAP Java service, to deploy the service, and to write a SOAP Java client that uses the service. The code examples in this chapter use the simple clock sample supplied with the Oracle SOAP installation.

Chapter 3, "SOAP Parameters and Encodings"

This chapter describes the procedures you use to write a SOAP Java service, to deploy the service, and to write a SOAP Java client for a service that uses arrays and other nonscalar types for parameters or return values. In addition, this chapter provides information on SOAP encodings.

Chapter 4, "SOAP Audit Logging"

This chapter describes the Oracle SOAP Audit Logging feature that monitors and records SOAP usage. Audit logging maintains records for postmortem analysis, accountability, and security. SOAP audit logging complements the audit logging capabilities available with the transport-specific server, the Apache HTTP Listener, that hosts the SOAP Request Handler Servlet (SOAP server).

Chapter 5, "SOAP Handlers"

The chapter describes Oracle SOAP handlers for the SOAP Request Handler Servlet. Handlers are configured in handler chains. Handlers are invoked to handle events associated with SOAP requests, responses, or errors.

Chapter 6, "Writing SOAP Providers"

This chapter describes the Oracle SOAP provider interface. The provider interface allows you to add your own service providers to Oracle SOAP.

Chapter 7, "Writing Deployment Managers"

This chapter describes advanced SOAP features, including the interfaces available for creating a Provider Manager and a Service Manager.

Chapter 8, "SOAP Administration"

This chapter describes configuration and administration details for Oracle SOAP.

Appendix A, "Apache Software License, Version 1.1"

This appendix contains the Apache software license.

Related Documentation

For more information, see the *Overview Guide* in the Oracle9*i* Application Server Documentation Library.

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

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

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

`http://www.oraclebookshop.com/`

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

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

`http://technet.oracle.com/membership/index.htm`

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

`http://technet.oracle.com/docs/index.htm`

For additional information, see:

- `http://www.w3.org/TR/SOAP` for information on the Simple Object Access Protocol (SOAP) 1.1 specification
- `http://www.w3.org/XML/Schema` for information on XML schema
- `http://www.w3.org/Addressing` for information on URIs, naming, and addressing
- `http://www.javasoft.com/products/javabeans/docs` for information on JavaBeans

Conventions

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

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
UPPERCASE monospace (fixed-width font)	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the datafiles and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JRepUtil class implements these methods.
lowercase monospace (fixed-width font) italic	Lowercase monospace italic font represents placeholders or variables.	You can specify the *parallel_clause*. Run U*old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate command-line statements. They are displayed in a monospace (fixed-width) font and are separated from normal text as shown in this example:

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

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (digits [ , precision ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS subquery;`
	■ That you can repeat a portion of the code	`SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

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

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

1

Simple Object Access Protocol Overview

This chapter provides an overview of the Simple Object Access Protocol (SOAP), and includes a description of the architecture of the Oracle SOAP implementation.

This chapter covers the following topics:

What Is the Simple Object Access Protocol?

The Simple Object Access Protocol (SOAP) is a lightweight, XML-based protocol for exchanging information in a decentralized, distributed environment. By combining SOAP-based requests and responses with a transport protocol, such as HTTP, the Internet becomes a medium for applications to publish database-backed **Web services**, such as:

- Restaurant listings: Which sushi bars are within five blocks of the Geary Theatre?
- Car dealer inquiries: What California dealers have a denim blue Audi TT coupe in stock?
- Financial information requests: What stocks in my portfolio are below their 50-day average?
- Ticket bookings: What are the two best seats available for Miss Saigon next week?

SOAP has a looser coupling between the client and the server than some similar distributed computing protocols, such as CORBA / IIOP, and it provides easier communication for a client and server that use different languages. SOAP exposes a standard way for processes to communicate, yet it leverages existing technologies.

SOAP requests are easy to generate, and a client can easily process the responses. One application can become a programmatic client of another application's services, with each exchanging rich, structured information. The ability to *aggregate* powerful, distributed Web services allows SOAP to provide a robust programming model that turns the Internet into an application development platform.

SOAP has the following features:

- Protocol independence
- Language independence
- Platform and operating system independence

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

How Does SOAP Work?

The SOAP specification describes a standard, XML-based way to encode requests and responses, including:

- Requests to invoke a method on a service, including *in parameters*
- Responses from a service method, including return value and *out parameters*
- Errors from a service

SOAP describes the structure and data types of message payloads by using the emerging W3C XML Schema standard issued by the World Wide Web Consortium (W3C). SOAP is a transport-agnostic messaging system; SOAP requests and responses *travel* using HTTP, HTTPS, or some other transport mechanism.

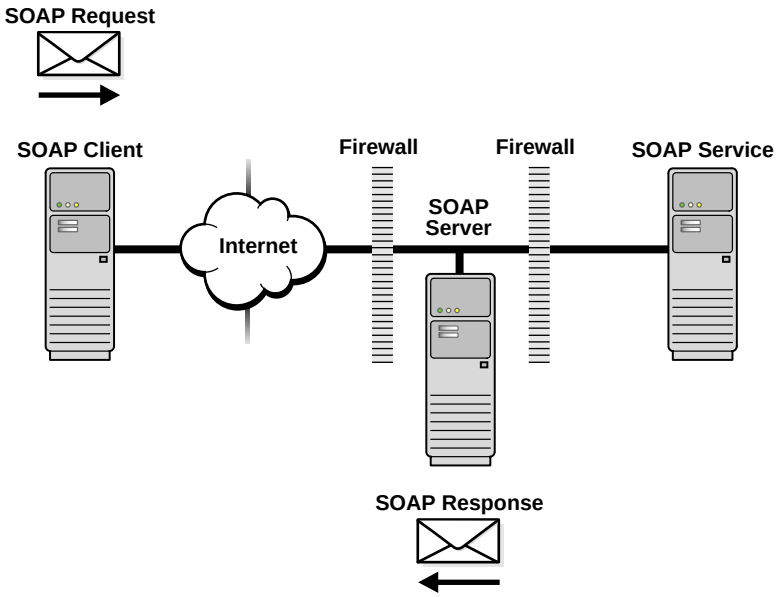
Figure 1–1 illustrates the components in the SOAP architecture. In general, a SOAP service remote procedure call (RPC) request / response sequence includes the following steps:

1. A *SOAP client* formulates a request for a service. This involves creating a conforming XML document, either explicitly or using Oracle SOAP client API.
2. A SOAP client sends the XML document to a *SOAP server*. This SOAP request is posted using HTTP or HTTPS to a SOAP Request Handler running as a servlet on a Web server. Example 1–1 shows the body of a SOAP message, an XML document, that represents a SOAP request for a service that provides an address from an address book.
3. The Web server receives the SOAP message, an XML document, using the SOAP Request Handler Servlet. The server then dispatches the message as a service invocation to an appropriate server-side application providing the requested service.
4. A response from the service is returned to the SOAP Request Handler Servlet and then to the caller using the standard SOAP XML payload format. Example 1–2 contains the body of a response to the request made in Example 1–1.

See Also:

- `http://www.w3.org/TR/SOAP`
- `http://www.w3.org/XML/Schema` for information on XML Schema

Figure 1–1 Components of the SOAP Architecture



The SOAP specification does not describe how the SOAP server should handle the content of the SOAP message body. The content of the body may be handed to a SOAP service, depending on the SOAP server implementation.

Example 1–1 SOAP Request for Address Book Listing Service

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getAddressFromName xmlns:ns1="urn:www-oracle-com:AddressBook"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<nameToLookup xsi:type="xsd:string" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
John B. Good
</nameToLookup>
</ns1:getAddressFromName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 1–2 SOAP Response from Address Book Service

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getAddressFromNameResponse xmlns:ns1="urn:www-oracle-com:AddressBook"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="urn:xml-soap-address-demo" xsi:type="ns2:address">
<city xsi:type="xsd:string">Anytown
</city>
<state xsi:type="xsd:string">NY
</state>
<phoneNumber xsi:type="ns2:phone">
<areaCode xsi:type="xsd:int">123
</areaCode>
<number xsi:type="xsd:string">7890
</number>
<exchange xsi:type="xsd:string">456
</exchange>
</phoneNumber>
<streetName xsi:type="xsd:string">Main Street
</streetName>
<zip xsi:type="xsd:int">12345</zip>
<streetNum xsi:type="xsd:int">123
</streetNum>
</return>
</ns1:getAddressFromNameResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Why Use SOAP?

Why do we need a standard like SOAP? By exchanging XML documents over HTTP, two programs can exchange rich, structured information without the introduction of an additional standard such as SOAP to explicitly describe a message *envelope* format and a way to encode structured content.

SOAP provides a standard so that developers do not have to invent a custom XML message format for every service they want to make available. Given the signature of the service method to be invoked, the SOAP specification prescribes an unambiguous XML message format. Any developer familiar with the SOAP specification, working in any programming language, can formulate a correct SOAP

XML request for a particular service and understand the response from the service by obtaining the following service details.

- Service name
- Method names implemented by the service
- Method signature of each method
- Address of the service implementation (expressed as a URI)

Using SOAP streamlines the process for exposing an existing software component as a Web service since the method signature of the service identifies the XML document structure used for both the request and the response.

> **See Also:** `http://www.w3.org/Addressing` for information on URIs, naming, and addressing

What Is Oracle SOAP?

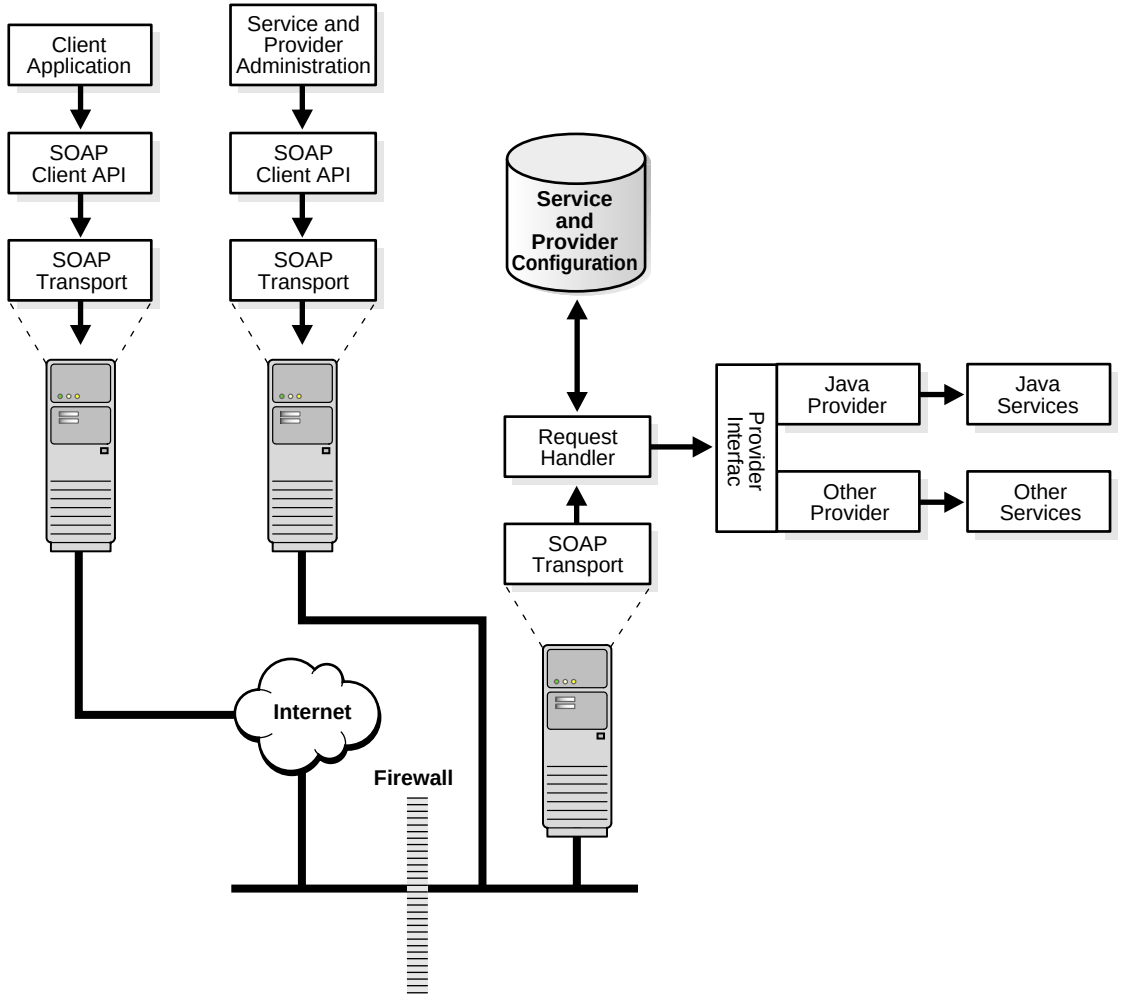
Oracle SOAP is an implementation of the Simple Object Access Protocol. Oracle SOAP is based on the SOAP open source implementation developed by the Apache Software Foundation.

This section describes the Oracle SOAP components shown in Figure 1–2. The topics covered include:

- Client Application and Client API
- SOAP Transports
- Administrative Clients
- SOAP Provider Interface and Providers
- SOAP Services

> **See Also:** `http://www.w3.org/TR/SOAP` and `http://xml.apache.org/soap`

Figure 1–2 Oracle SOAP Architecture



Client Application and Client API

A SOAP client application represents a user-written application that makes SOAP requests. A SOAP client application may include the following:

- SOAP Client API
- Oracle SOAP Security Features

SOAP Client API

SOAP clients generate the XML documents that compose a request for a SOAP service and handle the SOAP response. Oracle SOAP processes requests from any client that sends a valid SOAP request. To facilitate client development, Oracle SOAP includes a SOAP client API that provides a generic way to invoke a SOAP service. The SOAP client API supports a synchronous invocation model for requests and responses.

The SOAP client API makes it easier for you to write a Java client application to make a SOAP request. The SOAP client API encapsulates the creation of the SOAP request and the details of sending the request over the underlying transport protocol. The SOAP client API also supports a pluggable transport, allowing the client to easily change the transport (available transports include HTTP and HTTPS).

Oracle SOAP Security Features

Oracle SOAP uses the security capabilities in the transport to support secure access and to support other security features. For example, using HTTPS, Oracle SOAP provides confidentiality, authentication, and integrity over the Secure Sockets Layer (SSL). Other security features such as logging and authorization, are provided by the service provider.

SOAP Transports

SOAP transports are the protocols that *carry* SOAP messages. Oracle SOAP supports the following transports:

- HTTP: This protocol is the basic SOAP transport. The Oracle SOAP Request Handler Servlet manages HTTP requests and supplies responses directly over HTTP.
- HTTPS: The Oracle SOAP Request Handler Servlet manages HTTPS requests and supplies responses, with different security levels supported.

Administrative Clients

SOAP administrative clients include the Service Manager and the Provider Manager. These administrative clients are services that support dynamic deployment of new services and new providers.

SOAP Request Handler

The SOAP Request Handler is a Java servlet that receives SOAP requests, looks up the appropriate service provider, handles the service provider that invokes the requested method (service), and returns the SOAP response, if any.

SOAP Provider Interface and Providers

Oracle SOAP includes a provider implementation for Java classes. Other providers can be added.

Provider Interface

The provider interface allows the SOAP server to uniformly invoke service methods regardless of the type of provider (Java class, stored procedure, or some other provider type). There is one provider interface implementation for each type of service provider, and it encapsulates all provider-specific information. The provider interface makes SOAP implementation easily extensible to support new types of service providers.

Provider Deployment Administration

Oracle SOAP provides the provider deployment administration client to manage provider deployment information.

SOAP Services

SOAP application developers provide SOAP services. These services are made available using the supplied default Java class provider or custom providers. Oracle SOAP includes a service deployment administration client that runs as a service to manage services.

SOAP services, including Java services, represent user-written applications that are provided to remote SOAP clients.

2

Using Oracle SOAP with Java Services

This chapter provides an introduction to the procedures you use to write a SOAP Java service, to deploy the service, and to write a SOAP Java client that uses the service. Code examples in this chapter, use the simple clock sample supplied with the Oracle SOAP installation. Very little setup is required before you can use Oracle SOAP with Java clients. Using Oracle SOAP, you can easily make requests for SOAP services and generate replies from SOAP services.

A Java service runs on a SOAP server as part of the Oracle SOAP Java Provider. The Java service handles requests generated by a SOAP client.

This chapter covers the following topics:

- Writing a SOAP Java Service
- Deploying a SOAP Java Service
- Writing a SOAP Java Client
- SOAP Troubleshooting

Note: This chapter does not cover the SOAP message envelope, the HTTP protocol used for transport, or the XML that the Oracle SOAP Java Provider and the SOAP Request Handler Servlet pass from or to a SOAP client. Rather, the focus here is on the service request that the client generates using Java, and the return value that the service generates.

Writing a SOAP Java Service

Writing a SOAP Java service involves building a Java class that includes one or more methods that generate data used as responses to incoming calls. Normally a service is a method that can run independently of SOAP. There are very few restrictions on what actions a SOAP service can perform. At a minimum, most SOAP services generate some data or perform an action.

This section shows how to build a service that returns the current date and time. The clock service takes a SOAP request for a service and generates a response with a return value representing the date (a `String`).

The complete simple clock service is supplied with Oracle SOAP in the directory `$SOAP_HOME/samples/simpleclock` on UNIX or in `%SOAP_HOME%\samples\simpleclock` on Windows NT.

Developing a SOAP Java service consists of the following steps:

- Specifying a Package Name for the Service
- Defining Java Methods
- Returning a Result

Specifying a Package Name for the Service

Create a SOAP Java service by writing a class with methods that are deployed as a SOAP service. The Oracle SOAP Java Provider runs these methods in response to a request issued by the SOAP Request Handler Servlet. When looking at the simple clock service supplied with Oracle SOAP, note that the single jar file, `samples.jar`, contains a package, `samples`, that includes several samples. For the SOAP server installation the jar file is in the directory `$SOAP_HOME/webapps/soap/WEB-INF/lib` on UNIX or in `%SOAP_HOME%\webapps\soap\WEB-INF\lib` on Windows NT. For the SOAP client installation, the jar file is in the directory `$SOAP_HOME/lib` on UNIX or in `%SOAP_HOME%\lib` on Windows NT.

The class `MySimpleClockService` provides the simple clock service methods. If you want to place the Java service in a package, use the Java `package` specification to name the package. The first line of `MySimpleClockService.java` specifies the package name as follows:

```
package samples.simpleclock;
```

See Also: "Deploying a SOAP Java Service" on page 2-5

Defining Java Methods

The simple clock service is implemented with `SimpleClockService`, a public class. The service defines a single public method, `getDate()`, that supplies a date as a `String`. In general, a SOAP Java service defines one or more methods. As Example 2–1 shows, the `SimpleClockService` uses one public method, `getDate()`.

> **Note:** A Java implementation of a SOAP service must be a Java class that defines a public method for each SOAP method that is exposed as a SOAP service.

Example 2–1 Defining Simple Clock Service Methods

```
public class SimpleClockService extends Object
{
   public SimpleClockService()
   {
   }
   public static String getDate()
   {
   .
   .
   }
}
```

Serializing and Encoding Parameters and Results

The `getDate()` method returns a `String` value. Parameters and results sent between a client and a service go through the following steps:

1. Parameters are serialized and encoded in XML when sent from the client to a service.
2. Parameters are deserialized and decoded from XML when the SOAP Request Handler Servlet receives a service invocation request.
3. Parameters or results are serialized and encoded in XML when a request returns from the SOAP Request Handler Servlet to a SOAP client.
4. Parameters or results must be deserialized and decoded from XML when the client receives a reply.

Oracle SOAP supports a prepackaged implementation for handling these four steps for serialization and encoding, and deserialization and decoding, of scalar and user-defined types. Additionally, you can implement your own serialization and encoding mechanism.

The prepackaged mechanism makes the four serialization and encoding steps easy both for SOAP client-side applications, and for implementation of SOAP Java services. Using the prepackaged mechanism, Oracle SOAP, as specified in the SOAP client, supports the following encoding mechanisms:

- Standard SOAP v.1.1 encoding: Using standard SOAP v1.1 encoding, the Java Provider handles serialization and encoding internally for supported **primitive types** supported by Oracle SOAP. Table 2–1 lists the primitive types. For serializing and encoding **complex types** and user-defined types not found in Table 2–1, the application programmer can use JavaBeans to support the supplied `BeanSerializer`.
- Literal XML encoding. Using Literal XML encoding, a Java client passes as a parameter, or a Java service returns a result, that is encoded as a conforming W3C Document Object Model (DOM) `Element`. When an `Element` passes as a parameter to a SOAP service, the service processes the `Element`. For return values sent from a SOAP service, the client parses and processes the element.

See Also:

- "Setting Up a Call to Request a Service" on page 2-10
- "Serializing Java Parameters and Results Using BeanSerializer" on page 3-3
- "Writing a SOAP Service Using Literal XML Encoding" on page 3-13

Table 2–1 Types Supported with SOAP Using Default Java Encoding

Type	Type
`String`	`long`
`int`	`short`
`boolean`	`arrays`
`double`	`bytes` (byte arrays)
`float`	`Hashtable`
`Vector`	`Enumeration`

Returning a Result

The `getDate()` code segment shown in Example 2–2 gets a date and returns a `String` value as a response. The Oracle SOAP Java Provider receives a request from the SOAP Request Handler Servlet and calls the `getDate()` method. After running, `getDate()` returns its result to the Oracle SOAP Java Provider. The Oracle SOAP provider processes the return value and produces a SOAP envelope. Finally, the SOAP Request Handler Servlet serializes the reply and sends a response to the SOAP client. Example 2–2 shows that the Java service writer only needs to return a `String` for the simple date service.

Example 2–2 Generate a Date and Return Result

```
public static String getDate()
{
   return (new java.util.Date()).toString();
}
```

When an error occurs while running a Java service, the service should throw an exception, and the SOAP Request Handler Servlet then returns a SOAP fault. The exception is sent to the log file when the logger is enabled and the `severity` value is set to `debug`.

See Also: "Setting Configuration Options for Debugging" on page 2-16

Deploying a SOAP Java Service

To deploy a SOAP service, you need to create a service deployment descriptor file and deploy the service using the Service Manager utility.

This section covers the following topics:

- Creating a Java Service Deployment Descriptor
- Adding Service Classes to the SOAP CLASSPATH
- Using the Service Manager to Deploy and Undeploy Java Services
- Using the Service Manager to Verify or Query Java Services

Creating a Java Service Deployment Descriptor

A service deployment descriptor file is an XML file that defines configuration information for the Java service. A service deployment descriptor file defines the following information:

- The service ID
- The service provider type (for example, Java)
- The available methods

Example 2–3 shows the `SimpleClock` service descriptor file `SimpleClockDescriptor.xml`. This descriptor file is included in the `samples/simpleclock` directory. The service descriptor file must conform to the service descriptor schema (the schema, `service.xsd`, is located in the directory `$SOAP_HOME/schemas` on UNIX or in `%SOAP_HOME%\schemas` on Windows NT).

The service descriptor file identifies methods associated with the service in the `isd:provider` element that uses the `methods` attribute. The `isd:java class` element identifies the Java class that implements the SOAP service, and provides an indication of whether the class is static.

Example 2–3 Java Service Descriptor File for Simple Clock Service

```
<isd:service xmlns:isd="http://xmlns.oracle.com/soap/2001/04/deploy/service"
             id="urn:jurassic-clock"
             type="rpc" >
  <isd:provider
      id="java-provider"
      methods="getDate"
      scope="Application" >
      <isd:java class="samples.simpleclock.SimpleClockService"/>
  </isd:provider>
  <!-- includes stack trace in fault -->
  <isd:faultListener class="org.apache.soap.server.DOMFaultListener"/>
</isd:service>
```

Note: The service descriptor file does not define the method signature for service methods. SOAP uses reflection to determine method signatures.

Adding Service Classes to the SOAP CLASSPATH

To deploy a SOAP Java service, the class that implements the service must be available when Oracle SOAP starts. To add the class to the `CLASSPATH`, modify the `CLASSPATH` for the SOAP Request Handler Servlet. Using the standard installation, modify the file j`servSoap.properties` in the directory `$ORACLE_HOME/Apache/Jserv/etc` on UNIX or in `%ORACLE_HOME%\Apache\Jserv\etc` on Windows NT.

Using the Service Manager to Deploy and Undeploy Java Services

The `ServiceManager` is an administrative utility that deploys and undeploys SOAP services.

To deploy the simple clock service, first set the SOAP environment, then use the `deploy` command to deploy the `SimpleClockService` service. On UNIX, the command is:

```
cd $SOAP_HOME/bin
source clientenv.csh
cd $SOAP_HOME/samples/simpleclock
ServiceManager.sh deploy SimpleClockDescriptor.xml
```

For Windows NT, the command is:

```
cd %SOAP_HOME%\bin
clientenv.bat
cd %SOAP_HOME%\samples\simpleclock
ServiceManager.bat deploy SimpleClockDescriptor.xml
```

When you are ready to undeploy a service, use the un`deploy` command with the registered service name as an argument. On UNIX, the command is:

```
ServiceManager.sh undeploy urn:jurassic-clock
```

For Windows NT, the command is:

```
ServiceManager.bat undeploy urn:jurassic-clock
```

Using the Service Manager to Verify or Query Java Services

The `ServiceManager` is an administrative utility that lists and queries SOAP services. To list the available services, first set the SOAP environment, then use the `list` command. One UNIX, the command is:

```
cd $SOAP_HOME/bin
source clientenv.csh
ServiceManager.sh list
```

On Windows NT, the command is:

```
cd %SOAP_HOME%\bin
clientenv.bat
ServiceManager.bat list
```

To query a service and obtain the descriptor parameters set in the service deployment descriptor file, use the `query` command. On UNIX, the command is:

```
ServiceManager.sh query urn:jurassic-clock
```

On Windows NT, the command is:

```
ServiceManager.bat query urn:jurassic-clock
```

Writing a SOAP Java Client

After creating and deploying one or more SOAP services, client-side applications can request service invocations. The example described in this section is a SOAP client-side application that requests a date, using the simple clock service, and the method `GetDate()`.

Developing a Java client-side SOAP application consists of the following steps:

- Specifying a Package Name Java Clients
- Importing for Java Clients
- Defining a Request
- Setting Up a Call to Request a Service
- Invoking a Call to Request a Service
- Waiting for a Response and Handling SOAP Faults
- Running a Client
- Using Security Features with a Client

Specifying a Package Name Java Clients

Use a SOAP Java service by making a SOAP request. The file `MySimpleClockClient.java` shows a SOAP client that makes a SOAP request. On UNIX, the directory `$SOAP_HOME/samples/simpleclock` contains this source file; on Windows NT, `%SOAP_HOME%\samples\simpleclock`.

The first line of `MySimpleClockClient.java` specifies the package name for the service that is added to `samples.jar`.

```
package samples.simpleclock;
```

Importing for Java Clients

The SOAP client-side application uses the following imports:

```
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Response;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.Constants;
import org.apache.soap.SOAPException;
import java.net.URL;
import java.net.MalformedURLException;
```

The `org.apache.soap.rpc` imports for the SOAP client-side are discussed in the following sections.

Defining a Request

The `MySimpleClockClient` class includes the SOAP request for the SOAP `getDate` Java service. Example 2–4 shows the start of the client's `main()` routine that processes the command line argument.

Example 2–4 SOAP Client-Side Call Main Routine

```
public static void main(String[] args)
{
   if (args.length == 0)
   {
     System.out.println(
       "Usage is java samples.simpleclock.MySimpleClockClient SOAP_server_url");
     System.exit(1);
   }
   else
   {
```

```
      try
      {
        URL url = new URL (args[0]);
        MySimpleClockClient soapClockClient = new MySimpleClockClient(url);
      }
      catch (MalformedURLException mue)
      {
        mue.printStackTrace();
      }
    }
}
```

Setting Up a Call to Request a Service

The package `org.apache.soap.rpc` contains the SOAP `Call` object. A SOAP client-side request uses a `Call` object to build a request for a SOAP service. After the SOAP request is created, it is invoked to enable the client API to pass the request on to a SOAP server. Example 2–5 shows the code that builds a request for a SOAP service invocation.

Example 2–5 Building a SOAP RPC Call

```
Call call = new Call();
call.setTargetObjectURI("urn:jurassic-clock");
call.setMethodName("getDate");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
```

This code performs the following functions:

- Creates the `Call` object.
- Sets the `Call` object's target URI which identifies the service.
- Sets the `Call` object's method name.
- Sets the `Call` object's encoding style.
- Defines the `Call` object's parameters. For this service, the `Call` object does not set any parameters to send with the service request.

In addition, the `Call` object's `setTimeout()` method sets the timeout, an integer representing seconds, for a SOAP call. The default `Call` timeout, implicitly set in this example specifies no timeout. Setting a timeout value of 0 also specifies no timeout.

Serializing and Encoding Java Parameters and Results

Example 2–5 shows the client-side method that sets the default encoding style for the service request `setEncodingStyleURI`. The simple clock service uses standard SOAP v1.1 encoding.

In a client-side application, the SOAP API uses the encoding style set for the call to serialize and encode any parameters that are sent to the SOAP service, unless a specific encoding style is set for the parameter. For primitive types, the SOAP Java client-side API handles serialization and encoding internally using the standard SOAP encoding. For complex Java types, not found in Table 2–1, the application programmer must supply serialization and encoding methods to the client-side API.

See Also:

- "Defining Java Methods Using Parameters with User-Defined Types" on page 3-2
- "Serializing Java Parameters and Results Using BeanSerializer" on page 3-3

Invoking a Call to Request a Service

Example 2–6 shows the `Call` object `invoke()` method that invokes the service at the specified SOAP URL. A `Response` object handles any response. The URL that you provide on the command line should be the URL for the SOAP Request Handler Servlet. This value depends on where SOAP is deployed. For example, the URL could be composed as follows:

`http://`*`machineName`*`:`*`port`*`/soap/servlet/soaprouter`

Example 2–6 Making the SOAP RPC Call Invocation

```
Response resp = call.invoke(url, "");
```

Note: The second argument to `invoke()`, the `actionURI` is empty because the Oracle SOAP Server currently does not use the argument.

Waiting for a Response and Handling SOAP Faults

During invocation, if the `Call` timeout is not reached, and the `invoke()` method returns, the `Response` object holds the SOAP return value or any fault generated. The return value is stored in a `Parameter` component of the `Response` object. Use the `getValue()` method to convert the response to a Java object. Example 2–7 shows how `GetDate` handles the response to either process a fault or show the date returned from the simple clock service's `getDate` method.

Example 2–7 SOAP Request Response and Fault Handling

```
 try
 {
    System.out.println("Calling urn:jurassic-clock");
    Response resp = call.invoke(url, "");
    if (!resp.generatedFault())
    {
      Parameter result = resp.getReturnValue();
      System.out.println(result.getValue());
    }
    else
    {
       System.out.println("FAULT Returned");
       System.out.println(resp.getFault().getFaultString());
    }
}
catch (SOAPException soapE)
{
  soapE.printStackTrace();
}
catch (Exception e)
{
   e.printStackTrace();
}
```

Running a Client

After writing a SOAP client and setting the SOAP environment, run the client as you would any Java program. On UNIX, use the following commands:

```
cd $SOAP_HOME/bin
source clientenv.csh
java ${JAXP} samples.simpleclock.MySimpleClockClient ${SOAP_URL}
```

On Windows NT, use the following commands:

```
cd %SOAP_HOME%\bin
clientenv.bat
java %JAXP% samples.simpleclock.MySimpleClockClient %SOAP_URL%
```

Using Security Features with a Client

Oracle SOAP uses the security capabilities of the underlying transport that sends SOAP messages. Oracle SOAP supports the HTTP and HTTPS protocols for sending SOAP messages. HTTP and HTTPS support the following security features:

- HTTP proxies
- HTTP authentication (basic RFC 2617)
- Proxy authentication (basic RFC 2617)

Table 2–2 lists the client-side security properties that Oracle SOAP supports.

In a SOAP client-side application, you can set the security properties shown in Table 2–2 as system properties by using the `-D` flag at the Java command line. You can also set security properties in the Java program by adding these properties to the system properties (use `System.setProperties()` to add properties).

Example 2–8 shows how Oracle SOAP allows you to override the values specified for system properties using Oracle SOAP transport specific APIs. The `setProperties()` method in the class `OracleSOAPHTTPConnection` contains set properties specifically for the HTTP connection (this class is in the package `oracle.soap.transport.http`).

Example 2–8 Setting Security Properties for OracleSOAPHHTTPConnection

```
org.apache.soap.rpc.Call call = new org.apache.soap.rpc.Call();
oracle.soap.transport.http.OracleSOAPHTTPConnection conn =
(oracle.soap.transport.http.OracleSOAPHTTPConnection) call.getSOAPTransport();
java.util.Properties prop = new java.util.Properties();
// Use client code to set name-value pairs of properties in prop
.
.
.
conn.setProperties(prop);
```

Note: The property `java.protocol.handler.pkgs` must be set as a system property.

Table 2–2 SOAP HTTP Transport Security Properties

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

Table 2–2 (Cont.) SOAP HTTP Transport Security Properties

Property	Description
`oracle.soap.transport.allowUserInteraction`	Specifies the allows user interaction parameter. The case of the value specified is ignored. When this property is set to `true` and either of the following are true, the user is prompted for a username and password: 1. If any of properties `http.authType`, `http.username`, or `http.password` is not set, and a `401` HTTP status is returned by the HTTP server. 2. If either of properties `http.proxyAuthType`, `http.proxyUsername`, or `http.proxyPassword` is not set and a `407` HTTP response is returned by the HTTP proxy. Valid values: `true`, `false` Specifying any value other than `true` is considered as `false`.
`oracle.wallet.location`	Specifies the location of an exported Oracle wallet or exported trustpoints. Note: The value used is not a URL but a file location, for example: `/etc/ORACLE/Wallets/system1/exported_wallet` (on UNIX) `d:\oracle\system1\exported_wallet` (on Windows NT) This property must be set when HTTPS is used with SSL authentication, server or mutual, as the transport.
`oracle.wallet.password`	Specifies the password of an exported wallet. Setting this property is required when HTTPS is used with client, mutual authentication as the transport.

SOAP Troubleshooting

This section lists several techniques for troubleshooting Oracle SOAP, including:

- Tunneling Using the TcpTunnelGui Command
- Setting Configuration Options for Debugging
- Using DMS to Display Runtime Information

Tunneling Using the TcpTunnelGui Command

SOAP provides the `TcpTunnelGui` command to display messages sent between a SOAP client and a SOAP server. `TcpTunnelGui` listens on a TCP port, which is different than the SOAP server, and then forwards requests to the SOAP server.

Invoke `TcpTunnelGui` as follows:

```
java org.apache.soap.util.net.TcpTunnelGui TUNNEL-PORT SOAP-HOST SOAP-PORT
```

Table 2–3 lists the command line options for `TcpTunnelGui`.

Table 2–3 TcpTunnelGui Command Arguments

Argument	Description
TUNNEL-PORT	The port that `TcpTunnelGui` listens to on the same host as the client
SOAP-HOST	The host of the SOAP server
SOAP-PORT	The port of the SOAP server

For example, suppose the SOAP server is running as follows,

```
http://system1:8080/soap/servlet/soaprouter
```

You would then invoke `TcpTunnelGui` on port 8082 with this command:

```
java org.apache.soap.util.net.TcpTunnelGui 8082 system1 8080
```

To test a client and view the SOAP traffic, you would use the following SOAP URL in the client program:

```
http://system1:8082/soap/servlet/soaprouter
```

Setting Configuration Options for Debugging

To add debugging information to the SOAP Request Handler Servlet log files, change the value of the severity option for the value `debug` in the file `soapConfig.xml`. This file is placed in the directory `$SOAP_HOME/webapps/soap/WEB-INF/config` on UNIX or in `%SOAP_HOME%\webapps\soap\WEB-INF\config` on Windows NT.

For example, the following `soapConfig.xml` segment shows the value to set for `severity` to enable debugging:

```
<!-- severity can be: error, status, or debug -->
<osc:logger class="oracle.soap.server.impl.ServletLogger">
    <osc:option name="severity" value="debug" />
</osc:logger>
```

After modifying the `value` attribute for the `severity` option element in `soapConfig.xml`, perform the following steps to view debug information.

1. Stop SOAP by using the `stopSoapJServ` command.
2. Restart SOAP by using the `startSoapJServ` command. You do not need to perform this step if the SOAP Request Handler Servlet is running in auto start mode.

After stopping and restarting the SOAP Request Handler Servlet, you can view debug information in the file `jserv.log`. The file is in the directory `$ORACLE_HOME/Apache/Jserv/logs` on UNIX or in `%ORACLE_HOME%\Apache\Jserv\logs` on Windows NT.

See Also: "Using Auto Start Mode" on page 8-4

Using DMS to Display Runtime Information

Oracle SOAP is instrumented with DMS to gather information on the execution of the SOAP Request Handler Servlet, the Java Provider, and on individual services.

DMS information includes execution intervals from start to stop for the following:

- Total time spent in SOAP request and response (includes time in providers and services)
- Total time spent in the Java Provider (includes time in services)
- Total time executing services (`soap/java-provider/`*`service-URI`*)

To view the DMS information, go to the following site:

`http://`*`hostname`*`:`*`port`*`/soap/servlet/Spy`

3

SOAP Parameters and Encodings

This chapter describes the procedures you use to write a SOAP Java service, to deploy the service, and to write a SOAP Java client for a service that uses arrays and other nonscalar types for parameters or return values (Table 2–1 lists the scalar types). In addition, this chapter provides information on SOAP encodings. You should be familiar with the information in Chapter 2, "Using Oracle SOAP with Java Services" before reading this chapter.

This chapter covers the following topics:

- Writing a SOAP Java Service Using User-Defined Types
- Deploying a SOAP Java Service Using User-Defined Types
- Developing a SOAP Java Client Using Parameters
- Writing a SOAP Service Using Arrays as Parameters
- Writing a SOAP Service Using Literal XML Encoding

Writing a SOAP Java Service Using User-Defined Types

Developing a SOAP Java service involves building a Java class that includes one or more methods that generate responses to incoming calls. This section describes the procedures for developing an AddressBook service that provides methods to get an address, save a new address, and list all addresses in an AddressBook (the AddressBook data is stored in memory). These methods take arguments and expand on the basic SOAP Java service functionality shown in Chapter 2, "Using Oracle SOAP with Java Services".

The complete Address Book service is supplied in the directory `$SOAP_HOME/samples/addressbook` on UNIX or `%SOAP_HOME%\samples\addressbook` on Windows NT.

To develop a SOAP Java service that includes user-defined types as parameters or return values, you must complete the following steps:

- Specifying a Package Name for the Service
- Defining Java Methods Using Parameters with User-Defined Types
- Serializing Java Parameters and Results Using BeanSerializer
- Returning Results to the Request Handler Servlet

Specifying a Package Name for the Service

Create a SOAP Java service by writing a class containing methods that are deployed as a SOAP service. When looking at the AddressBook service supplied with Oracle SOAP, note that the single jar file, `samples.jar` contains a `samples` package, that includes several samples (the jar file is in the directory `$SOAP_HOME/webapps/soap/WEB-INF/lib`). The first line of `AddressBook.java` specifies the package name for the service, `addressbook`, which is added to `samples.jar`.

```
package samples.addressbook;
```

Defining Java Methods Using Parameters with User-Defined Types

The `Addressbook` service is implemented with a public class `AddressBook` and defines the following public methods:

```
public void addEntry(String name, Address address)
public Address getAddressFromName(String name)
public Element getAllListings()
public int putListings(Element el)
```

When SOAP Java methods take scalar arguments or return scalar types, and use the standard SOAP v1.1 encoding, the Oracle SOAP Request Handler Servlet performs serialization and encoding internally. For services implemented as Java methods that take user-defined types as parameters or return user-defined types as results, the application programmer needs to provide support classes to handle serialization. With the AddressBook service, a user supplied serialization routine is required for the `Address` type that `addEntry()` takes as a parameter and `getAddressFromName()` returns. The `Address` user-defined type contains a `PhoneNumber` type that also requires a user-supplied serialization routine.

Serializing Java Parameters and Results Using BeanSerializer

For types not found in Table 2–1, you can provide a JavaBean that supports serialization and deserialization, using the standard SOAP v1.1 encoding. The `BeanSerializer` class within Oracle SOAP uses JavaBeans to serialize and deserialize user-defined types with the SOAP-ENC encoding style. This technique enables handling of Java user-defined types for a SOAP service on both the client and server-side.

Perform the following steps to use a JavaBean and the `BeanSerializer`:

1. Write the JavaBean to support the user-defined types that the service uses. The following section, "Writing JavaBean Support Routines for User-Defined Types" on page 3-3 describes this step.
2. Add the compiled JavaBean class to the appropriate `CLASSPATH` (on the server-side and the client-side). The section, "Adding Compiled JavaBean Classes to the CLASSPATH" on page 3-4 describes this step.
3. Deploy the JavaBean on the SOAP server by specifying a map element in a service deployment descriptor. The section, "Deploying a SOAP Java Service Using User-Defined Types" on page 3-6 describes this step.

Writing JavaBean Support Routines for User-Defined Types

The AddressBook sample provides several JavaBeans that the SOAP `BeanSerializer` uses to support serialization in the AddressBook service. The files `Address.java` and `PhoneNumber.java` provide the JavaBean serialization support that allows SOAP to process and pass an address and a phone number between the SOAP client and the SOAP server. These routines are JavaBeans for the

Address and PhoneNumber types. JavaBeans conform to the JavaBeans specification and have the following characteristics:

- Public set and get methods for each member of the user-defined type. The set and get methods must have names conforming to the JavaBeans specification.
- A default constructor taking no arguments.

> **See Also:**
>
> http://www.javasoft.com/products/javabeans/docs for information on JavaBeans

Example 3–1 shows the conforming setCity() set method and getCity() get method from the Address JavaBean that supports the AddressBook SOAP service (see Address.java for the complete JavaBean sample).

Example 3–1 Sample Set and Get Methods from the Address JavaBean

```
public void setCity(String city)
{
  this.city = city;
}

public String getCity()
{
  return city;
}
```

Adding Compiled JavaBean Classes to the CLASSPATH

After creating and compiling the JavaBean, add the generated support class to the CLASSPATH. Note, this code has to be made available to both the SOAP service on the server-side and to the SOAP client code on the client-side.

On the server-side add the class to the CLASSPATH for the SOAP Request Handler Servlet routine (the routine that invokes the Java service). There are two ways to add the service class to the CLASSPATH, depending on the mode the Oracle SOAP Request Handler Servlet is running in, either auto mode or non-auto mode.

- Auto Mode: In auto mode, modify the wrapper.classpath in the file jservSoap.properties in the directory $ORACLE_HOME/Apache/Jserv/etc.

- Non-Auto Mode: In non-auto mode, change the CLASSPATH in the routine `startSoapJServ.sh` on UNIX or `startSoapJServ.bat` on Windows NT.

After modifying the CLASSPATH, restart the SOAP Request Handler Servlet.

- Auto Mode: In auto mode, perform the following step to restart. On UNIX, the startup command is:

  ```
  % stopSoapJServ.sh
  ```

  On Windows NT, the command is:

  ```
  > stopSoapJServ.bat
  ```

- Non-Auto Mode: In non-auto mode, perform the following steps to restart on UNIX:

  ```
  % stopSoapJServ.sh
  % startSoapJServ.sh
  ```

  On Windows NT, the command is:

  ```
  > stopSoapJServ.bat
  > startSoapJServ.bat
  ```

See Also: "Using Auto Start Mode" on page 8-4

Returning Results to the Request Handler Servlet

The `getAddressFromName()` code segment shown in Example 3–2 gets the address from memory and returns an `Address`. The `getAddressFromName()` method returns its result to the Oracle SOAP Java Provider.

Example 3–2 Generate Data and Return Result for Date

```
public Address getAddressFromName(String name)
    throws IllegalArgumentException
{
  if (name == null)
  {
     throw new IllegalArgumentException("The name must not be " +  "null.");
  }
   return (Address)name2AddressTable.get(name);
 }
```

Encoding Java Parameters and Results

Oracle SOAP supports two types of XML encoding for user-defined parameters and results:

- Standard SOAP v.1.1 encoding
- Literal XML encoding

For Java service parameters and results, the `map` element of the service descriptor deployment file sets the encoding style, and the XML to Java and Java to XML serialization and deserialization routines for user-defined types.

See Also:

- "Deploying a SOAP Java Service Using User-Defined Types" on page 3-6
- "Handling Encoding, Serialization, and Mapping with Parameters" on page 3-9

Deploying a SOAP Java Service Using User-Defined Types

To deploy a SOAP service with user-defined types, you need to create a service deployment descriptor file and include a mappings element with `map` elements for each user-defined type. Example 3–3 shows the mappings element from the AddressBook service descriptor `ServiceDescriptor.xml`. This descriptor is included in the directory `$SOAP_HOME/samples/addressbook` on UNIX or `%SOAP_HOME%\samples\addressbook` on Windows NT. The service descriptor file should conform to the service descriptor schema defined in `service.xsd` in the directory `$SOAP_HOME/schemas` on UNIX or in `%SOAP_HOME%\schemas` on Windows NT. Table 3–1 describes the attributes of the `map` element.

Note: The SOAP Service Manager does not currently validate the deployment descriptor file using the schema `service.xsd`.

After creating the deployment descriptor file, deploy the service using the SOAP Service Manager utility.

Example 3–3 Java Service Descriptor File for AddressBook Service

```
<isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:x="urn:xml-soap-address-demo"
         qname="x:address"
         javaType="samples.addressbook.Address"
         java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
         xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:x="urn:xml-soap-address-demo"
         qname="x:phone"
         javaType="samples.addressbook.PhoneNumber"
         java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
         xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
</isd:mappings>
```

See Also: "Using the Service Manager to Deploy and Undeploy Java Services" on page 2-7

Table 3–1 Service Deployment Descriptor Map Element Attributes

Attribute	Description
`qname`	Specifies the qualified name of the serialized object. This is the XML type of the serialized object.
`javaType`	Establishes the relationship between `qname` and the corresponding Java type (the Java type that gets serialized or deserialized).
`java2XMLClassName`	Specifies the names of the classes that marshal user-defined parameters or return values, mentioned in the `javaType` attribute. The `BeanSerializer` class is delivered with the SOAP server, and uses the JavaBean for mapping. If the `javaType` specified is not a JavaBean, interfaces are supplied that allow you to write your own `java2XML` classes.
`xml2JavaClassName`	Specifies the name of the classes that unmarshal user-defined parameters or return values, mentioned in the `javaType` attribute. The `BeanSerializer` class is delivered with the SOAP server, and uses the JavaBean for mapping. If the `javaType` specified is not a JavaBean, interfaces are supplied that allow you to write your own `xml2java` classes.

Developing a SOAP Java Client Using Parameters

After creating and deploying one or more SOAP services, client-side applications request service invocations. For details on the basic components of a client-side application that makes a SOAP service request, see Chapter 2, "Using Oracle SOAP with Java Services". This section shows additional SOAP client-side techniques that allow you to use parameters to make service requests. The topics covered include:

- Creating Parameters to Pass to a Service
- Handling Encoding, Serialization, and Mapping with Parameters
- Setting Up a Call to Request a Service with Parameters
- Invoking a Call to Request a Service with Parameters
- Running a Client with Parameters

Creating Parameters to Pass to a Service

This section describes the procedure for using the `PutAddress` client to make an `addEntry()` method request of the AddressBook service. The AddressBook service is supplied with Oracle SOAP in `samples.jar`. For complete details on the code for this section, refer to `PutAddress.java` in the directory `$SOAP_HOME/samples/addressbook`.

The command line arguments specify the address and phone number for the `Address` parameter to pass to the SOAP method, `addEntry()`. Example 3–4 shows the instantiation for the `Address` object. The new `Address` is used as a parameter in the `addEntry()` service when the service runs on the SOAP server. `PutAddress` obtains the `args[]` values from the command line.

Example 3–4 Building an Address Object to Send as a SOAP Parameter

```
String nameToRegister = args[1];
PhoneNumber phoneNumber =
               new PhoneNumber( Integer.parseInt(args[7]), args[8],    args[9]);
Address address = new Address(Integer.parseInt(args[2]),
                                  args[3],
                                  args[4],
                                  args[5],
                                  Integer.parseInt(args[6]),
                                  phoneNumber);
```

Handling Encoding, Serialization, and Mapping with Parameters

Using the standard SOAP encoding, the SOAP Java client-side API handles serialization and encoding for primitive types internally. The `PutAddress` class makes a request for the SOAP `addEntry()` method from the AddressBook SOAP service. The `addEntry()` method takes an `Address` parameter, which is not a primitive type. To pass an address, the SOAP client needs to handle serialization and mapping for the `Address` and the `PhoneNumber` objects. Example 3–5 shows the `SOAPMappingRegistry`, the `BeanSerializer`, and the encoding style flags that allow the SOAP client to serialize these user-defined objects. Note that the `mapTypes()` method takes a `QName` argument. The `QName` is defined when a service is deployed on the SOAP server.

Example 3–5 SOAP Client-Side Call Serialization and Mapping

```
SOAPMappingRegistry smr = new SOAPMappingRegistry();
BeanSerializer beanSer = new BeanSerializer();

String  encodingStyleURI = Constants.NS_URI_SOAP_ENC;

// create the type mapping

smr.mapTypes(Constants.NS_URI_SOAP_ENC,
             new QName("urn:xml-soap-address-demo", "address"),
             Address.class, beanSer, beanSer);
smr.mapTypes(Constants.NS_URI_SOAP_ENC,
             new QName("urn:xml-soap-address-demo", "phone"),
             PhoneNumber.class, beanSer, beanSer);
```

See Also:

- "Deploying a SOAP Java Service Using User-Defined Types" on page 3-6
- "Writing a SOAP Service Using Literal XML Encoding" on page 3-13

Setting Up a Call to Request a Service with Parameters

The package `org.apache.soap.rpc` contains the SOAP `Call` object. A SOAP client-side request uses a `Call` object to build a request for a SOAP service. After the request is created, it is invoked to request that the client API pass the request on to the SOAP server. Example 3–6 shows the code that builds a request for a service invocation, including `name` and `address` that are sent to `addEntry()`.

Example 3–6 Building a SOAP RPC Call with Parameters

```
Call call = new Call();
call.setSOAPMappingRegistry(smr);
String serviceId = "urn:www-oracle-com:AddressBook";
call.setTargetObjectURI(serviceId);
call.setMethodName("addEntry");
call.setEncodingStyleURI(encodingStyleURI);
Vector params = new Vector();
params.addElement(new Parameter("name", String.class,
                                nameToRegister, null));
params.addElement(new Parameter("address", Address.class,
                                address, null));
call.setParams(params);
```

This code does the following:

- Creates the `Call` object
- Sets the `Call` object's target URI
- Sets the `Call` object's method name
- Sets the `Call` object's encoding style
- Defines the `Call` object's parameters with a `QName` to establish the mapping

In addition, the `setTimeout()` method sets the timeout, an integer representing seconds. The default `Call` timeout, implicitly set in this example, specifies no timeout. Setting a timeout value of 0 also specifies no timeout.

Invoking a Call to Request a Service with Parameters

In Example 3–7 the `Call` object's `invoke()` method makes a SOAP request and invokes the service at the specified URL. The `Response` object handles the response. The URL that you provide on the command line should be the URL for the SOAP Request Handler Servlet. This value depends on where SOAP is deployed. For example, the URL could be composed as follows:

`http://`*`machineName`*`:`*`port`*`/soap/servlet/soaprouter`

Example 3–7 Making the SOAP RPC Call Invocation

```
Response resp;
    try
    {
      resp = call.invoke(url, SOAPactionURI);
    }
    catch (SOAPException e)
    {
      System.err.println("Caught SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
      return;
    }
```

Note: The second argument to `invoke()`, the `actionURI` is not used. The Oracle SOAP server currently does not use this argument.

Running a Client with Parameters

After writing a SOAP client using parameters, and setting the SOAP environment, run the client as you would any Java program:

```
cd $SOAP_HOME/bin
source clientenv.csh
java ${JAXP} samples.addressbook.PutAddress ${SOAP_URL} "John Doe" 123 "Main
Street" AnyTown SS 12345 800 555 1212
```

Writing a SOAP Service Using Arrays as Parameters

Consider a SOAP Java service that returns the sum of the elements of an integer array. Example 3–8 shows the `SOAPArrayCalculator` service that takes an array parameter and returns the sum. The `getSum()` method returns the sum of the input elements of the integer array, `array1`.

This section covers the following:

- Server-Side Adjustments for Using Arrays as Parameters
- Client-Side Adjustments for Using Arrays as Parameters

Server-Side Adjustments for Using Arrays as Parameters

For a SOAP Java service, passing arrays on the server-side is no different than passing other parameters, as shown in Example 3–8.

Example 3–8 SOAP Java Service Using an Array Parameter

```
package SOAPServices;
public class SOAPArrayCalculator extends Object
{
  public SOAPArrayCalculator()
  {
  }
  public static int getSum(int[] array1)
  {
    int result = 0;
    for (int i=0; i<array1.length; i++)
      result += array1[i];
    return result;
  }
}
```

Client-Side Adjustments for Using Arrays as Parameters

On the client-side, making a request for a SOAP Java service and passing arrays is slightly different than passing other parameters. Example 3–9 shows that the Java client-side application must send an array of `Integer` for the corresponding array of `int` in the service on the server-side. SOAP handles the encoding and decoding of the array using a reflection mechanism implemented at the SOAP server level. This mechanism allows SOAP to know what to receive and how to stream it in the requests and responses.

Example 3–9 SOAP Request Using an Array Parameter

```
Integer[] array = getIntegerArray();
Vector prms = new Vector();
prms.addElement(new Parameter("integerArray", Integer[].class, array, null));
call.setParams(prms);
```

Note: From the client-side, all parameters sent as components of an array must use an `Object` type.

Writing a SOAP Service Using Literal XML Encoding

This section uses the `getAllListings()` method in the AddressBook service to describe the procedure for writing a SOAP service using literal XML encoding. The `getAllListings()` method returns a list of all AddressBook listings as a literal XML element. The SOAP client application `GetAllListings.java` is included in the directory `$SOAP_HOME/samples/addressbook`. This section describes the changes you need to make on both the client and server sides to support passing parameters or results using a literal XML encoding.

This section covers the following topics:

Server-Side Adjustments for Using Literal XML Encoding

When writing a Java SOAP service that uses a literal XML encoding for a parameter or for a return value, add the following imports to support working in XML:

```
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import oracle.soap.util.xml.XmlUtils;
```

Creating a Return Value with Literal XML Encoding

The SOAP Request Handler Servlet uses literal XML encoding for all parameters that specify this encoding in a client request, and for return values when the default encoding style set for the call specifies XML encoding. Use the `Call` method, `setEncodingStyleURI()` with a value of `Constants.NS_URI_LITERAL_XML` to set the default encoding style for a call to literal XML.

The Java service implementation provides either the appropriate literal XML parameters or a literal XML return value. For example, when literal XML encoding is specified as the default encoding in the SOAP request, the return value sent back to the client from the Java service should be a valid XML element, as defined by `org.w3c.dom.Element`. Thus, the client call specifies the default encoding that is used for a return value, and for all parameters that do not explicitly set an encoding style. The service provides a valid object that conforms to the encoding. The AddressBook method `getAllListings()` shows how a Java service could build an `Element` that SOAP could encode literally in XML. Oracle SOAP supplies `XmlUtils` methods that support working with XML documents.

Client-Side Adjustments for Using Literal XML Encoding

On the client-side, a SOAP service that passes a literal XML argument or returns a literal XML value is very similar to the services using the basic SOAP Java functionality shown in Chapter 2, "Using Oracle SOAP with Java Services".

Minor changes are required to support literal XML encoding, including:

- Specifying a Call with Literal XML Encoding
- Invoking a Call with Literal XML Encoding

Specifying a Call with Literal XML Encoding

To specify a call using literal XML encoding for all parameters and results, use the `setEncodingStyleURI()` method, with the value `Constants.NS_URI_LITERAL_XML`. Example 3–10 shows the Java that builds the call for `getAllListings()`.

Example 3–10 Making a SOAP Call Using Literal XML Encoding

```
Call call = new Call();
String serviceId = "urn:www-oracle-com:AddressBook";
call.setTargetObjectURI(serviceId);
call.setMethodName("getAllListings");
call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
```

Invoking a Call with Literal XML Encoding

After setting up the client to receive a literal XML encoded result, invoke the service on the SOAP server as you would any service. Using `GetAllListings`, the result is sent in the response return value. Example 3–11 shows how the return value is placed in a SOAP client API `Parameter` object, which is then cast to an `Element` using the `getValue()` method.

Example 3–11 Invoking and Processing with Literal XML Results

```
Response resp;

try
{
   resp = call.invoke(url, "");
}
catch (SOAPException e)
{
      System.err.println("Caught SOAPException (" +
```

```
                        e.getFaultCode() + "): " +
                        e.getMessage());
      return;
}
Check value    // check the response

if (!resp.generatedFault())
{
      Parameter ret = resp.getReturnValue();
      Element bookEl = (Element)ret.getValue();

      System.out.println(DOM2Writer.nodeToString(bookEl));
}
else
{
      Fault fault = resp.getFault();

      System.err.println("Generated fault: ");
      System.out.println ("  Fault Code   = " + fault.getFaultCode());
      System.out.println ("  Fault String = " + fault.getFaultString());
    }
}
```

4

SOAP Audit Logging

The Oracle SOAP audit logging feature monitors and records SOAP usage. Audit logging maintains records for postmortem analysis and accountability. The SOAP audit logging feature complements the audit logging capabilities available with the transport-specific server, the Apache HTTP listener, which hosts the SOAP Request Handler Servlet (SOAP server).

Oracle SOAP stores audit trails as XML documents. Using XML documents, Oracle SOAP creates portable audit trails and enables the transformation of complete audit trails or individual audit records to different formats.

By default, Oracle SOAP audit logging uses an audit logger class that implements the `Handler` interface (part of the `oracle.soap.server` package). The audit logger class is invoked conditionally to monitor events including service requests, service responses, and errors.

This chapter covers the following topics:

Audit Logging Information

Table 4–1 lists the audit logging elements available for each audit log record. Individual audit log records may not contain all these elements. In the log file, each audit log record is stored as a `SoapAuditRecord` element.

Table 4–1 Available Audit Record Elements

Audit Record Element	Description
`HostName`	Specifies the hostname of the client that sent the request.
`IpAddress`	Specifies the IP address of the client that sent the request.
`Method`	Specifies the method name for the SOAP request.
`Request`	Provides the complete SOAP request message.
`Response`	Provides the complete SOAP response message.
`ServiceURI`	Specifies the service URI for the SOAP request.
`SoapAuditRecord`	Contains an individual record. The `chainType` attribute indicates if the record is generated as part of a request or a response.
`TimeStamp`	Specifies the system time when the SOAP audit record was generated.
`User`	Specifies the username associated with the request. Note, this element is only provided when a user context is associated with the service request or service response.

Audit Logging Output

The XML schema for the generated audit log is provided in the file `SoapAuditTrail.xsd` in the directory `$SOAP_HOME/schema` on UNIX or `%SOAP_HOME%\schema` on Windows NT. Refer to the schema file for complete details on the format of a generated audit log record.

Auditable Events

The audit logger class is invoked when an auditable event occurs and the SOAP Request Handler Servlet is configured to enable auditing for the event. Auditable events include a service request or a service response.

Audit Logging Filters

An audit logging filter can be specified in the SOAP configuration file to limit the set of auditable events that are recorded to the audit log. The SOAP server applies event filters to request and response events. Table 4–2 shows the filter attributes available to select with an event filter specification. When applied, filters limit the number of records generated in the audit log. For example, when a filter is specified for a particular host, only the auditable events generated for the specified host are saved to the audit log.

The syntax for defining auditable events with a filter is derived from RFC 2254. Table 4–3 shows the filter syntax, and Example 4–1 provides several examples.

See Also: "Configuring the Audit Logger" on page 4-5

Table 4–2 Audit Trail Events Filter Attributes

Audit Event Filter Attributes	Description

`Host`	Specifies the hostname of the host for the service request or response. If this attribute is not specified in a filter, the hostname of the client is not used in filtering audit log records.
	
	Fully specify the hostname of the client or use wildcards ("*"). Wildcards embedded within the specified hostname are not supported (see the examples below showing valid and invalid uses of wildcards). If a wildcard is used then the wildcard must be the first character in the filter.
	
	Case is ignored for hostnames. Care should be used in setting this attribute. Depending on the DNS setup, the hostname returned could be fully qualified or nonqualified; for example, `explosives.acme.com` or `explosives`. For some IP addresses, the DNS may not be able to resolve the hostname.
	
	Legal values for a `Host` filter attribute include the following examples:
	
	`explosives.acme.com, *.acme.com, *.com`
	
	Illegal values for a `Host` filter attribute include the following examples:
	
	`*, explosives.acme.*, explosives.*, ex*s.acme.com, *ives.acme.com`

Table 4–2 (Cont.) Audit Trail Events Filter Attributes

Audit Event Filter Attributes	Description

`ip`	Specifies the IP address of the client for the service request or response.
	The IP address of the client has to be either fully specified, using all four bytes, in the dot separate decimal form, or specified using wildcards ("*"). Embedded wildcards are not supported. If a wildcard is used then the wildcard must be the last character in the filter.
	If this attribute is not used in a filter then the IP address of the client is not used in filtering.
	Legal values for an `ip` filter attribute include the following examples:
	`138.2.142.154, 138.2.142.*, 138.2.*, 138.*`
	Illegal values for an `ip` filter attribute include the following examples:
	`*, 138.2.*.154, *.2, 138.*.152, 138.2.142, 138.2, 138`

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

Table 4–3 Audit Log Filter Syntax

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

Table 4–3 (Cont.) Audit Log Filter Syntax

Filter Value	Description
value	1*(any octet except ASCII representation of ")" - 0x29). The character "*" has a special meaning. The "*" character is referred to as a wildcard and matches anything.

Example 4–1 Audit Log Filters

```
(ip=138.2.142.154)
(!(host=localhost))
(!(host=*.acme.com))
(&(host=*.acme.com)(username=daffy))
(&(ip=138.2.142.*)(|(urn=urn:www-oracle-com:AddressBook)(username=daffy)))
```

Configuring the Audit Logger

Configure the default SOAP audit logger supplied with Oracle SOAP by setting parameters in the SOAP configuration file, `soapConfig.xml`. To enable the default audit logger and turn on audit logging, do the following in the configuration file.

- Define the name and options for the audit log handler. The default SOAP audit logger is defined in the class `oracle.soap.handlers.audit.AuditLogger`. The default audit logger supports several options that you specify in the configuration file. Table 4–4 shows the available audit logger options.
- Add the name for the audit logger handler to the `requestHandler`, `responseHandler`, or `errorHandler` chain (or to all of the handler chains).

Example 4–2 shows a sample segment from a SOAP configuration file including the audit logging configuration options. Example 4–2 shows configuration options set to use all options. However, this configuration would produce an extremely large audit log, and is not recommended.

Note: When you audit errors using the audit logger, it is possible that the request or response message may not be included in the audit log record, even with `includeRequest` or `includeResponse` enabled.

Example 4–2 Audit Logging Configuration

```
<osc:handlers>
   <osc:handler name="auditor"
      class="oracle.soap.handlers.audit.AuditLogger">
      <osc:option name="auditLogDirectory"
           value="/private1/oracle/app/product/tv02/soap/webapps/soap/WEB-INF"/>
      <osc:option name="filter" value="(!(host=localhost))"/>
      <osc:option name="includeRequest" value="true"/>
      <osc:option name="includeResponse" value="true"/>
   </osc:handler>
</osc:handlers>
<osc:requestHandlers names="auditor"/>
<osc:responseHandlers names="auditor"/>
<osc:errorHandlers names="auditor"/>
```

Table 4–4 Audit Logger Configuration Options

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

See Also: "Configuring the Request Handler Servlet" on page 8-2

5

SOAP Handlers

Oracle SOAP supports handlers for the SOAP Request Handler Servlet (SOAP server) that are configured in handler chains. Handlers are invoked at specific times for each SOAP request.

This chapter covers the following topics:

- Handler Overview
- Request Handlers
- Response Handlers
- Error Handlers
- Configuring Handlers

Handler Overview

A handler is a class that implements the `oracle.soap.server.Handler` interface. A handler can be configured as part of a chain in one of three contexts: request, response, or error. Note that handlers in a chain are invoked in the order they are specified in the configuration file.

See Also: "Configuring Handlers" on page 5-3

Request Handlers

Handlers in the request chain are invoked on every request that arrives, immediately after the SOAP Request Handler Servlet reads the SOAP Envelope and before performing the service lookup. If any handler in the request chain throws an exception, the processing of the chain is immediately terminated and the service is not invoked.

The error chain is invoked if any exception occurs during request chain invocation.

Response Handlers

Handlers in the response chain are invoked on every request immediately after the service completes. If any handler in the response chain throws an exception, processing of the chain is immediately terminated. The error chain is invoked if any exception occurs during response chain invocation.

Error Handlers

When an exception occurs during either request-chain invocation, service invocation, or response-chain invocation, the SOAP Request Handler Servlet invokes the handlers in the error chain. In contrast to the request and response chains, an exception from an error handler is logged and processing of the error chain continues. All handlers in the error chain are invoked, regardless of whether one of the error handlers throws an exception.

Configuring Handlers

Configure handlers and handler chains in the SOAP configuration file. Handlers can be invoked for each service request or response, or when an error occurs. Handlers are global in the sense that they apply to every SOAP request and cannot be configured on a subset of requests, such as all requests for a particular service.

Configure a handler by setting parameters in the SOAP configuration file, `soapConfig.xml`. Example 5–1 shows a sample segment from a SOAP configuration file showing the configuration for a handler.

Example 5–1 Handler Configuration

```
<osc:handlers>
   <osc:handler name="auditor"
      class="oracle.soap.handlers.audit.AuditLogger">
      <osc:option name="auditLogDirectory"
           value="/private1/oracle/app/product/tv02/soap/webapps/soap/WEB-INF"/>
      <osc:option name="filter" value="(!(host=localhost))"/>
   </osc:handler>
</osc:handlers>

<osc:requestHandlers names="auditor"/>
<osc:responseHandlers names="auditor"/>
<osc:errorHandlers names="auditor"/>
```

6

Writing SOAP Providers

This chapter describes the Oracle SOAP Provider Interface and includes information on the following topics:

- Provider Interface Overview
- Implementing a Provider Interface
- Handling Provider Deployment

Provider Interface Overview

Oracle SOAP includes a prepackaged provider implementation. In addition, Oracle SOAP includes a Provider Interface that allows you to write a custom provider to support services of types other than the prepackaged types. You can add a provider to support a service by implementing the Provider Interface.

Oracle SOAP includes provider implementations to support services for the following:

- Java classes

See Also:

- Chapter 1, "Simple Object Access Protocol Overview"
- Chapter 2, "Using Oracle SOAP with Java Services"

Implementing a Provider Interface

The Oracle SOAP Provider Interface is a Java interface that enables the SOAP Request Handler Servlet to work uniformly with different types of service providers. A SOAP Provider, also called a *service provider*, is an implementation of the Provider Interface that encapsulates the logic necessary to invoke methods on a specific type of service. Using a provider that is an implementation of the Provider Interface simplifies the SOAP Request Handler Servlet and allows you to add new types of service providers.

Note: When adding a new provider type, you only need to implement the Provider Interface once for each type of service. Thus, a SOAP server supports a small number of types of service providers and a potentially large number of services.

This section shows the basic steps for building a custom provider. For specific details, and a sample of the code required to build a custom provider, see the provider sample supplied with Oracle SOAP in the directory `$SOAP_HOME/samples/provider`.

Developing a custom SOAP Provider consists of the following steps:

- Implementing Provider Interface Methods
- Handling Provider Deployment

Implementing Provider Interface Methods

The provider writer implements the following methods that are included in the `Provider` Interface:

```
public void destroy();
public String getId();
public void init(ProviderDeploymentDescriptor pd, SOAPServerContext ssc);
public void invoke(RequestContext requestContext);
```

Working with the Provider init() Method

The `init()` method receives the provider configuration deployment descriptor options and initializes the provider using the supplied configuration information. Provider-specific options are passed to the `init()` method as attributes from the SOAP Request Handler Servlet. Use these attributes to initialize an instance of the provider. An example of provider-specific options are the Hostname and Port for a Database Server Connection.

The provider `init()` method handles any required initialization that is required as a one-time-only initialization for service providers. The `init()` method is invoked by the SOAP Request Handler Servlet exactly once before the handler makes any requests to services that the provider supports. This process allows the provider to set up any required provider-specific global context.

Working with the Provider invoke() Method

After provider initialization occurs, most of the work for the provider implementation occurs in the `invoke()` method. This method invokes the requested method in the specified SOAP service.

The `invoke()` method performs the following important actions:

- Supports RPC-based invocation
- Unmarshals request parameters destined for a service
- Marshals response parameters generated from a service invocation
- Runs in a thread-safe manner

The SOAP Request Handler Servlet passes service-specific options to the provider. These options are read from the service deployment descriptor when the service is deployed. Service-specific options describe a service deployed in a provider. An example of a service-specific option is the name of the Java class for a method that implements a Java service.

The `invoke()` parameter of type `RequestContext` completely describes a service request. Given the `RequestContext`, the `invoke()` method takes care of certain steps, as shown in the following procedure. For complete details on the code for these steps, see the provider sample supplied with Oracle SOAP in the directory `$SOAP_HOME/samples/provider`.

1. Get the service-specific options, and build the `SOAPMappingRegistry`. Use the `RequestContext` method `getServiceDeploymentDescriptor()` to get the service options.

```
UserContext ucontext = rc.getUserContext();
String      serviceId = rc.getServiceId();
ServiceDeploymentDescriptor sd = rc.getServiceDeploymentDescriptor();
SOAPMappingRegistry smr =
                ServiceDeploymentDescriptor.buildSOAPMappingRegistry(sd);
```

2. Unmarshall the parameters for the service method using a `Call` object, `Call.extractFromEnvelope()`, and the `RequestContext` method `getRequestEnvelope()`.

```
Call call = Call.extractFromEnvelope(rc.getRequestEnvelope(), smr);
rc.setRequestEncodingStyle(call.getEncodingStyleURI());
```

3. Build the arguments and determine response encoding style using `call.getEncodingStyleURI()` and `param.getEncodingStyleURI()`.

```
String respEncStyle = call.getEncodingStyleURI();
Vector params = call.getParams ();
Object[] args = null;
Class[] argTypes = null;
if (params != null)
{
   int paramsCount = params.size ();
   args = new Object[paramsCount];
   argTypes = new Class[paramsCount];
   for (int i = 0; i < paramsCount; i++)
   {
      Parameter param = (Parameter) params.elementAt (i);
      args[i] = param.getValue ();
      argTypes[i] = param.getType ();
      if (respEncStyle == null)
      {
         respEncStyle = param.getEncodingStyleURI ();
 }
 }
 }
```

4. Set a default encoding for the response.

```
if (respEncStyle == null)
{
   respEncStyle = Constants.NS_URI_SOAP_ENC;
}
rc.setRequestEncodingStyle(respEncStyle);
```

5. Invoke the service method, and obtain a result and a vector of parameters for any parameters that are returned (`outParams`).

```
Bean        result = null;
Vector      outParams = new Vector();
try
{
   m_log.log("Invoking method '" + methodName + "'", Logger.SEVERITY_DEBUG);
   result = new Bean(String.class, "this is the result");
   outParams.addElement(
               new Parameter("paramName", String.class, "this is a param",
                   null));
}
catch (Throwable t)
{
   throw new SOAPException(Constants.FAULT_CODE_SERVER,
}
```

6. Create a response, and save the result as a `Response` object. The SOAP Request Handler Servlet marshals the envelope before returning.

```
try
{
    if (sd.getServiceType() ==
            ServiceDeploymentDescriptor.SERVICE_TYPE_RPC)
    {
         Parameter ret = null;
        if (result != null && result.type != void.class)
        {
            ret = new Parameter (RPCConstants.ELEM_RETURN,
                result.type, result.value, null);
        }
        Response resp = new Response(rc.getServiceId(),
            call.getMethodName(), ret, outParams, null, respEncStyle);

         try
        {
            Envelope respEnvelope = resp.buildEnvelope();
```

```
                rc.setResponseEnvelope(respEnvelope);
                rc.setResponseMap(smr);
            }
            catch (Exception e)
            {
                throw new SOAPException (Constants.FAULT_CODE_SERVER,
                    "error building response envelope", e);
            }

        }
        else
        {
            throw new SOAPException (Constants.FAULT_CODE_SERVER,
                "invalid service type");
          }

    }
    catch (Exception e)
    {
        m_log.log("Error creating response: " + e, Logger.SEVERITY_DEBUG);
        throw new SOAPException (Constants.FAULT_CODE_SERVER,
            "error creating response", e);
    }
```

Working with the Provider destroy() Method

The `destroy()` method performs one-time service provider cleanup. This method is invoked by the SOAP Request Handler Servlet exactly once before the handler shuts down. This method gives the provider the opportunity to do global cleanup, such as closing connections.

Working with the Provider getId() Method

The `getId()` method supplies the provider ID. The provider ID, for a particular provider is unique within the SOAP Request Handler Servlet.

Handling Provider Deployment

After implementing a custom provider, if you are using the default Provider Manager and the default Service Manager, you should update the XML provider deployment descriptor schema and the XML service deployment descriptor schema to conform to the new requirements for the new provider. This sections covers the following:

- Updating the Provider Deployment Descriptor Schema
- Updating the Service Deployment Descriptor Schema

Note: Using the default Provider Manager and the default Service Manager, it is not required that you update the schemas. The Provider Manager and Service Manager process the deployment descriptor files and do not check the accuracy of the values found in the deployment descriptor files against the valid values specified in the schemas. Thus, the default Provider Manager and Service Manager support deployment descriptor values that are not in the corresponding schemas.

See Also: "Creating a Provider Manager" on page 7-1 and "Creating a Service Manager" on page 7-2

Updating the Provider Deployment Descriptor Schema

A service provider must be deployed to make its services available. To deploy a service provider, you can either implement a custom Provider Manager or use the default Provider Manager and deploy providers using the Provider Manager Client. The default Provider Manager reads an XML provider deployment descriptor that conforms to the XML provider deployment schema.

Example 6–1 shows a sample provider deployment descriptor file for a SOAP service using a custom Provider.

Example 6–1 Custom Provider Deployment Descriptor

```
<isd:provider xmlns:isd="http://xmlns.oracle.com/soap/2001/04/deploy/provider"
    id="company-provider"
    class="oracle.soap.providers.sp.SpProvider" >
  <isd:storedProcedure jdbc="jdbc:oracle:oci8"
           username="scott"
           password="tiger"
           service="YOUR-SERVICE-NAME" />
</isd:provider>
```

> **See Also:** `$SOAP_HOME/schema/provider.xsd` for the complete provider deployment descriptor schema

Updating the Service Deployment Descriptor Schema

A service provider encapsulates all of the logic necessary to invoke SOAP methods in services that are implemented as a specific type, such as a stored procedure or a Java class. A service provider must be deployed to make its services available. To deploy a service provider, implement a custom Service Manager, or use the default Service Manager and deploy providers with the Service Manager Client.

The default Server Manager reads an XML service deployment descriptor that conforms to the XML schema.

Example 6–2 shows a custom Service Deployment Descriptor file for a SOAP service using a custom Provider.

Example 6–2 Custom Service Deployment Descriptor

```
 <isd:service xmlns:isd="http://xmlns.oracle.com/soap/2001/04/deploy/service"
    id="urn:www-oracle-com:company"
    type="rpc" >
   <isd:provider
        id="company-provider"
        methods="ADDEMP GETEMP GETADDRESS GETEMPINFO CHANGESALARY REMOVEEMP"
        scope="Application" >

       <isd:storedProcedure schema="SCOTT" package="COMPANY" />

   </isd:provider>

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="urn:company-sample" qname="x:EMPLOYEE"
```

```
            javaType="samples.sp.company.Employee"
            sqlType="SCOTT.EMPLOYEE"
            java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"

xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:x="urn:company-sample" qname="x:ADDRESS"
            javaType="samples.sp.company.Address"
            sqlType="SCOTT.ADDRESS"
            java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"

xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>

  <isd:faultListener class="org.apache.soap.server.DOMFaultListener"/>

</isd:service>
```

See Also: `$SOAP_HOME/schema/service.xsd` for the complete service deployment descriptor schema.

7

Writing Deployment Managers

This chapter describes several deployment manager interfaces supporting SOAP administration. The topics covered include:

- Creating a Provider Manager
- Creating a Service Manager

Creating a Provider Manager

The `ProviderManager` interface defines the interface to manage providers. The SOAP Request Handler Servlet (SOAP server) uses a Provider Manager is to deploy providers, undeploy providers, and access provider deployment information. The `ProviderManager` interface is part of the `oracle.soap.server` package.

A `ProviderManager` implementation may cache deployment information. The `ProviderManager` must maintain the cache.

The HTTP server provides security for the Provider Manager. The Provider Manager can be configured with a specific URL. In order to be accepted, all requests must the be made to the specified URL. If a SOAP request for the Provider Manager is made to any other URL, the request is rejected. This URL should be an alias to the SOAP Request Handler Servlet, and HTTP Listener security may be set to control which users can post to the specified URL.

See Also:

- `$SOAP_HOME/docs/apiDocs/index.html` on UNIX or `%SOAP_HOME%\docs\apiDocs\index.html` on Windows NT
- "Setting Provider Manager and Service Manager Configuration Options" on page 8-3

Creating a Service Manager

The `ServiceManager` interface defines the interface to manage services. The SOAP Request Handler Servlet (SOAP server) uses a Service Manager is to deploy services, undeploy services, and access service deployment information. The `ServiceManager` interface is part of the `oracle.soap.server` package.

The Service Manager may cache deployment information and is responsible to maintain the cache.

The HTTP server provides security for the Service Manager. The Service Manager can be configured with a specific URL. In order to be accepted, all requests must be made to the specified URL. If a SOAP request for the Service Manager is made to any other URL, the request is rejected. This URL should be an alias to the SOAP Request Handler Servlet, and HTTP Listener security may be set to control which users can post to the specified URL.

See Also:

- `$SOAP_HOME/docs/apiDocs/index.html` on UNIX or `%SOAP_HOME%\docs\apiDocs\index.html` on Windows NT
- "Setting Provider Manager and Service Manager Configuration Options" on page 8-3

8

SOAP Administration

This chapter describes configuration and administration details for Oracle SOAP. This chapter covers the following topics:

- Configuring the Request Handler Servlet
- Using Auto Start Mode
- Setting Jserv Configuration and Security
- Changing the HTTP Listener Port Number
- Configuring Memory Options

Configuring the Request Handler Servlet

The Oracle SOAP Request Handler uses an XML configuration file to set required servlet parameters. By default, this file is named `soapConfig.xml` and is placed in the directory `$SOAP_HOME/webapps/soap/WEB-INF/config` on UNIX or `%SOAP_HOME\webapps\soap\WEB-INF\config` on Windows NT. The XML namespace for this file is:

```
http://xmlns.oracle.com/soap/2001/04/config
```

To use a different configuration file for SOAP installation, modify the path name specified for the `SoapConfig` parameter in the `soap.properties` file. For example, to change the configuration file from the default, `soapConfig.xml`, to `newConfig.xml`, modify the value set for `soapConfig` in `soap.properties`.

```
servlet.soaprouter.initArgs=soapConfig=soap_home/soap/webapps/soap/WEB-INF/config/newConfig.xml
```

Where *soap_home* is the full path to the SOAP installation on your system.

Table 8–1 lists the SOAP deployment parameters available for configuring the SOAP Request Handler Servlet.

Table 8–1 SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
`errorHandlers`	Specifies a list of handlers for the error handler chain.
`faultListeners`	Specifies a list of fault listeners.
`handlers`	Specifies the available handler names and the options for each handler.
`logger`	Error and informational messages are logged using the class defined in the logger element. The logger class must extend `oracle.soap.server.Logger`. Oracle SOAP includes the class `oracle.soap.server.impl.ServletLogger` that collects the servlet log methods so that SOAP messages are logged to the servlet log file. `ServletLogger` is the default logger. For the default logger, the severity option can be to any of the following values: `status`, `error`, `debug`.

Table 8–1 (Cont.) SOAP Request Handler Servlet Configuration File Parameters

Parameter	Description
`providerManager`	Defines how the server accesses provider deployment information. The `providerManager` class attribute specifies a Java class that implements `oracle.soap.server.ProviderManager`. Oracle SOAP includes the class `oracle.soap.server.impl.FileProviderManager` which stores provider deployment information in a file. Using FileProviderManager, the file name is specified with the `filename` option. See "Setting Provider Manager and Service Manager Configuration Options" on page 8-3 for more information.
`requestHandlers`	Specifies a list of handlers for the request handler chain
`responseHandlers`	Specifies a list of handlers for the response handler chain
`serviceManager`	Defines how the server accesses service deployment information. The `serviceManager` class attribute specifies a Java class that implements `oracle.soap.server.ServiceManager`. Oracle SOAP includes the class `oracle.soap.server.impl.FileServiceManager` which stores the service deployment information in a file. Using `FileServiceManager`, the file name is specified with the `filename` option. See "Setting Provider Manager and Service Manager Configuration Options" on page 8-3 for more information.

Setting Provider Manager and Service Manager Configuration Options

The SOAP `providerManager` and `serviceManager` can be configured as SOAP services. To configure these deployment administration routines as SOAP services, set the `serviceManager autoDeploy` option to the value `true` in the `soapConfig` configuration file. Set the value to `false` to disable this option.

When the administration routines are deployed as SOAP services, you can optionally specify a specific URL for these services (an administration URL). Specifying an administration URL helps to maintain security for SOAP service deployment, and is recommended. To set and control the administration URL, perform the following three steps.

1. Configure `ProviderManager` and `ServiceManager` by setting the `requiredRequestURI` option in `soapConfi.xml`. Set the `serviceManager` or `providerManager` option `requiredRequestURI` to specify a URL for administration.
2. Modify the Jserv mount points for the administration URL by setting configuration options in the file `jserv.conf` in the directory `$ORACLE_HOME/Apache/Jserv/etc`.

   For a manager that runs manually, with `autoDeploy` set to `false`, add a new path for a `ApJServMount`. For example,

   ```
   ApJServMount /servlets /soap/admin/servlet
   ```

   Run this with the URL,

   ```
   http://hostname:port/soap/admin/servlet/soaprouter
   ```

   For a manager that runs in auto mode, with `autoDeploy` set to `true`, add a new path for a `ApJServGroupMount`. For example,

   ```
   ApJServGroupMount /servlets /soap/admin/servlet
   ```

   Run this with the URL,

   ```
   http://hostname:port/soap/admin/servlet/soaprouter
   ```

3. Set security for the specified URL by setting configuration options in the file `httpd.conf` in the directory `$ORACLE_HOME/Apache/conf`.

   For example, if `/soap/admin/servlet` is set with secure access, you could set `requiredRequestURI` to `/soap/admin/servlet/soaprouter` for the `ProviderManager` and the `ServiceManager`.

Using Auto Start Mode

The SOAP Request Handler Servlet runs in auto mode or in non-auto mode. The mode determines how the servlet is started.

In auto mode, a process manager within Apache starts the SOAP Request Handler Servlet automatically and manages the process. In non-auto mode, the SOAP Request Handler Servlet needs to be manually started.

Oracle SOAP is installed using the default mode, `auto`. To change the mode, change the value for the `ApJServManual` configuration directive in the file `jserv.conf` in the directory `$ORACLE_HOME/Apache/Jserv/etc`.

Setting Jserv Configuration and Security

To set or update the list of IP addresses that is allowed to connect to Apache Jserv, modify the `security.allowedAddresses` parameter in the file `$ORACLE_HOME/Apache/Jserv/etc/jservSoap.properties`.

Changing the HTTP Listener Port Number

To set the port where the SOAP HTTP Listener runs, modify the Apache configuration file, `httpd.conf` found in the directory `$ORACLE_HOME/Apache/Apache/conf`.

You can view this file to determine the port where the Apache HTTP listener starts.

Configuring Memory Options

You can configure heap memory usage for the SOAP Request Handler Servlet in the file `jservSoap.properties`. This file resides in the directory `$ORACLE_HOME/Apache/Jserv/etc` on UNIX, or `%ORACLE_HOME%\Apache\Jserv\etc` on Windows NT. If you receive `java.lang.OutofMemory` errors from the SOAP Request Handler Servlet, increasing the heap size may solve this problem.

For example, to set the size of the SOAP Request Handler Servlet's heap memory to 32 megabytes, use the following settings in `jservSoap.properties`:

```
wrapper.bin.parameters=-Xmx32m
wrapper.bin.parameters--Xms32m
```

A

Apache Software License, Version 1.1

The Apache Software License, Version 1.1

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment:

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

   Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org/.

Portions of this software are based upon public domain software originally written at the National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign.

Index

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

X