BEA WebLogic Server™

Programming WebLogic Web Services

BEA WebLogic Server Version 6.1
Document Date: November 1, 2002

Copyright

Copyright © 2002 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA WebLogic Collaborate, BEA WebLogic Commerce Server, BEA WebLogic E-Business Platform, BEA WebLogic Enterprise, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Process Integrator, BEA WebLogic Server, E-Business Control Center, How Business Becomes E-Business, Liquid Data, Operating System for the Internet, and Portal FrameWork are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Programming WebLogic Web Services

Part Number	Document Date	Software Version
N/A	November 1, 2002	BEA WebLogic Server Version 6.1

Contents

2. Developing WebLogic Web Services

3. Invoking WebLogic Web Services

4. Administering WebLogic Web Services

5. Troubleshooting

About This Document

This document describes BEA WebLogic® Web Services and describes how to develop them and invoke them from a client application.

The document is organized as follows:

- Chapter 1, “Overview of WebLogic Web Services,” provides conceptual information about Web Services, the features of WebLogic Web services, and their architecture.
- Chapter 2, “Developing WebLogic Web Services,” describes how to develop WebLogic Web Services.
- Chapter 3, “Invoking WebLogic Web Services,” describes how to access WebLogic Web Services from client applications.
- Chapter 4, “Administering WebLogic Web Services,” describes how to administer Web Services using the Administration Console.
- Chapter 5, “Troubleshooting,” describes how to troubleshoot problems that might occur when creating client applications that invoke Web services.
- Appendix A, “Specifications Supported by WebLogic Web Services,” provides links to the specifications supported by WebLogic Web Services.
- Appendix B, “build.xml Elements and Attributes,” provides information about the `build.xml` Java build file that you use to assemble Web services into Enterprise Application archive (`*.ear`) files.
- Appendix C, “Manually Assembling the Web Services Archive File,” describes how to create a Web services archive file manually without using the **wsgen** Ant task.

- Appendix D, “Invoking Web Services Without Using the WSDL File,” describes how to create a client application that invokes a Web service without using its WSDL.

A glossary of relevant terms and an index follows the chapters.

Audience

This document is written for application developers who want to make EJBs that are currently running in WebLogic Server available to third-party clients as Web Services.

It is assumed that readers know Web technologies, XML, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`

Convention	Usage

|	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

. . .	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Overview of WebLogic Web Services

The following sections provide an overview of Web services, and how they are implemented in WebLogic Server:

What Are Web Services?

Web services are a type of service that can be shared by and used as components of distributed Web-based applications. They commonly interface with existing back-end applications, such as customer relationship management systems, order-processing systems, and so on.

Traditionally, software application architecture tended to fall into two categories: huge monolithic systems running on mainframes or client-server applications running on desktops. Although these architectures work well for the purpose the applications were built to address, they are relatively closed to the outside world and can not be easily accessed by the diverse users of the Web.

Thus the software industry is evolving toward loosely coupled service-oriented applications that dynamically interact over the Web. The applications break down the larger software system into smaller modular components, or shared services. These services can reside on different computers and can be implemented by vastly different technologies, but they are packaged and transported using standard Web protocols, such as XML and HTTP, thus making them easily accessible by any user on the Web.

The concept of services is not new—RMI, COM, and CORBA are all service-oriented technologies. However, applications based on these technologies require them to be written using that particular technology, often from a particular vendor. This requirement typically hinders widespread acceptance of an application on the Web. To solve this problem, Web services are defined to share the following properties that make them easily accessible from heterogeneous environments:

- Web services are accessed over the Web.
- Web services describe themselves using an XML-based description language.
- Web services communicate with clients (both end-user applications or other Web services) through XML messages that are transmitted by standard Internet protocols, such as HTTP.

Why Use Web Services?

The major reasons for using Web services are to gain:

- interoperability among distributed applications that span diverse hardware and software platforms.
- accessibility of applications through firewalls using Web protocols.
- a cross-platform, cross-language data model (XML) that facilitates developing heterogeneous distributed applications.

Because Web services are accessed using standard Web protocols, such as XML and HTTP, the diverse and heterogeneous applications on the Web (which typically already understand XML and HTTP) can automatically access Web services, solving the ever-present problem of how different systems communicate with each other.

These different systems might be Microsoft SOAP ToolKit clients, J2EE applications, legacy applications, and so on. These systems might be written in a variety of programming languages, such as Java, C++, or Perl. As long as the application that provides the functionality is packaged as a Web service each of these systems can communicate with any other.

Web Service Components

A Web service consists of the following components:

- An implementation hosted by a server on the Web.

  WebLogic Web Services are hosted by WebLogic Server, are implemented using standard J2EE components (such as Enterprise Java Beans and JMS), and are packaged as standard J2EE Enterprise Applications.

- A standardized way to transmit data and Web service invocation calls between the Web service and the user of the Web service.

  WebLogic Web Services use Simple Object Access Protocol (SOAP) 1.1 as the message format and HTTP as the connection protocol. For a description of SOAP, see “SOAP 1.1 with Attachments” on page 1-4.

- A standard way to describe the Web service to clients so they can invoke it.

  WebLogic Web Services use Web Services Description Language (WSDL) 1.1, an XML-based specification, to describe themselves. For more information on WSDL, see “WSDL 1.1” on page 1-5.

SOAP 1.1 with Attachments

SOAP (Simple Object Access Protocol) is a lightweight XML-based protocol used to exchange information in a decentralized, distributed environment. The protocol consists of:

- An envelope that describes the SOAP message. In particular, the envelope contains the body of the message, identifies who should process it, and describes how to process it.
- A set of encoding rules for expressing instances of application-specific data types.
- A convention for representing remote procedure calls and responses.

This information is embedded in a Multipurpose Internet Mail Extensions (MIME)-encoded package that can be transmitted over HTTP or other Web protocols. MIME is a specification for formatting non-ASCII messages so that they can be sent over the Internet.

The following example shows a SOAP request for stock trading information embedded inside an HTTP request:

```
POST /StockQuote HTTP/1.1
Host: www.sample.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <SOAP-ENV:Body>
          <m:GetLastStockQuote xmlns:m="Some-URI">
               <symbol>BEAS</symbol>
          </m:GetLastStockQuote>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

WSDL 1.1

Web Services Description Language (WSDL) is an XML-based specification used to describe a Web service. A WSDL document describes the methods provided by a Web service, the input and output parameters, and how to connect to it.

Developers of WebLogic Web Services do not need to create the WSDL files; these files can be generated automatically as part of the WebLogic Web Services development process.

The following example, for informational purposes only, shows a WSDL file that describes the stock trading Web service StockQuoteService that contains the method GetLastStockQuote:

```
<?xml version="1.0"?>
  <definitions name="StockQuote"
               targetNamespace="http://sample.com/stockquote.wsdl"
               xmlns:tns="http://sample.com/stockquote.wsdl"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:xsd1="http://sample.com/stockquote.xsd"
               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
               xmlns="http://schemas.xmlsoap.org/wsdl/">
      <message name="GetStockPriceInput">
          <part name="symbol" element="xsd:string"/>
      </message>
      <message name="GetStockPriceOutput">
          <part name="result" type="xsd:float"/>
      </message>
      <portType name="StockQuotePortType">
          <operation name="GetLastStockQuote">
             <input message="tns:GetStockPriceInput"/>
             <output message="tns:GetStockPriceOutput"/>
          </operation>
      </portType>
      <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
          <soap:binding style="rpc"
                        transport="http://schemas.xmlsoap.org/soap/http"/>
          <operation name="GetLastStockQuote">
             <soap:operation soapAction="http://sample.com/GetLastStockQuote"/>
             <input>
               <soap:body use="encoded" namespace="http://sample.com/stockquote"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
             </input>
             <output>
             <soap:body use="encoded" namespace="http://sample.com/stockquote"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

```
            </output>
          </operation>>
      </binding>
      <service name="StockQuoteService">
          <documentation>My first service</documentation>
          <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
             <soap:address location="http://sample.com/stockquote"/>
          </port>
      </service>
  </definitions>
```

WebLogic Web Service Features

This section discusses the features of the WebLogic Web Services subsystem:

- Web Services Programming Model
- SOAP 1.1 Implementation
- Web Services Run-time Component
- Standardized J2EE Web Services Assembly and Deployment
- Generation of the WSDL File
- Java Client to Invoke a WebLogic Web Service
- Examples of Creating and Invoking Web Services

Web Services Programming Model

The programming model describes how to implement, assemble, deploy, and invoke Web services that are hosted by a WebLogic Server. Apart from writing the Enterprise JavaBeans code that performs the actual work of the Web service, you develop most of the Web service itself by using a Java Ant task, called `wsgen`, that generates and packages the components of the Web service.

WebLogic Server supports two types of Web services: remote procedure call (RPC)-style and message-style.

RPC-Style Web Services

A remote procedure call (RPC)-style Web service is implemented using a stateless session EJB. It appears as a remote object to the client application.

The interaction between a client and an RPC-style Web service centers around a service-specific interface. When clients invoke the Web service, they send parameter values to the Web service, which executes the required methods, and then sends back the return values. Because of this back and forth conversation between the client and the Web service, RPC-style Web services are tightly coupled and resemble traditional distributed object paradigms, such as RMI or DCOM.

RPC-style Web services are synchronous, meaning that when a client sends a request, it waits for a response before doing anything else.

Message-Style Web Services

A message-style Web service is implemented using a JMS message listener, such as a message-driven bean, and must be associated with a JMS destination.

Message-style Web services are loosely coupled and document-driven rather than being associated with a service-specific interface. When a client invokes a message-style Web service, the client typically sends it an entire document, such as a purchase order, rather than a discrete set of parameters. The Web service accepts the entire document, processes it, and may or may not return a result message. Because no tightly-coupled request-response between the client and Web service occurs, message-style Web services promote a looser coupling between client and server.

Message-style Web services are asynchronous. A client that invokes the Web service does not wait for a response before it can do something else. The response from the Web service, if any, can appear hours or days later.

A client can either send or receive a document to or from a message-style Web service; the client can not do both using the same Web service.

SOAP 1.1 Implementation

WebLogic Server includes its own implementation of both the SOAP 1.1 and SOAP 1.1 With Attachments specifications that developers can use to create clients that invoke Web services.

RPC-style Web services use the SOAP 1.1 message format and message-style Web services use the SOAP 1.1 With Attachments message format.

Note: WebLogic Web Services currently ignore the actual attachment of a SOAP with attachments message.

Web Services Run-time Component

The WebLogic Web Services run-time component is a set of servlets and associated infrastructure needed to create a Web service. One element of the run-time is a set of servlets that handle SOAP requests from a client. You do not need to write these servlets; they are automatically included in the WebLogic Server distribution. Another element of the run-time is an Ant task that generates and assembles all the components of a WebLogic Web Service.

Standardized J2EE Web Services Assembly and Deployment

Web services developers use an Ant task, called `wsgen`, and the Administration Console to assemble and deploy Web services as standard J2EE Enterprise applications in an `*.ear` file. The `*.ear` file contains all the components of the Web service: for example, the EJBs, references to the SOAP servlets, the `web.xml` file, the `weblogic.xml` file, and so on.

Generation of the WSDL File

Developers that create clients that invoke a WebLogic Web Service need the WSDL that describes the Web service. WebLogic Server automatically generates the WSDL of a deployed Web service. You access the WSDL of a Web service through a special URL.

Java Client to Invoke a WebLogic Web Service

WebLogic Server can automatically generate a thin Java client that developers use to develop Java clients that invoke Web services. The Java client JAR file includes all the classes you need to invoke a Web service. These classes include the Java client API classes and interfaces, a parser to parse the SOAP requests and responses, the Java interface to the EJB, and so on. Client applications that use this Java client JAR file to invoke Web services do not need to include the full WebLogic Server JAR file on the client computer.

You download the Java client JAR file from the WebLogic Web Services Home Page. For detailed information on this Web page, see "Invoking the WebLogic Web Services Home Page" on page 3-4 in Chapter 3, "Invoking WebLogic Web Services."

Note: BEA does not currently license client functionality separately from the server functionality, so, if needed, you can redistribute this Java client JAR file to your own customers.

Examples of Creating and Invoking Web Services

WebLogic Server includes examples of creating both RPC-style and message-style Web services and examples of both Java and Microsoft VisualBasic client applications that invoke the Web services.

The examples are located in the ***BEA_HOME*/samples/examples/webservices** directory, where ***BEA_HOME*** refers to the main WebLogic Server installation directory. The RPC-style Web service example is in the **rpc** directory and the message-style Web service example is in the **message** directory.

For detailed instructions on how to build and run the examples, invoke the Web page *BEA_HOME*/samples/examples/webservices/package-summary.html in your browser.

WebLogic Web Services Architecture

When you develop a WebLogic Web Service, you use standard J2EE components, such as stateless session EJBs, message-driven beans, and JMS destinations. Because WebLogic Web Services are based entirely on the J2EE platform, they automatically inherit all the standard J2EE benefits, such as a simple and familiar component-based development model, easy scalability, support for transactions, automatic life-cycle management, easy access to existing enterprise systems through the use of J2EE APIs (such as JDBC and JTA), and a simple and unified security model.

WebLogic Server Web services are packaged as standard J2EE Enterprise applications that consist of the following specific components:

- A Web application that contains, at a minimum, a servlet that sends and receives SOAP messages to and from the client.

  Developers do not write this servlet themselves; rather, it is automatically included as part of the Web services development process.

- A stateless session EJB that implements an RPC-style Web service or a JMS listener (such as a message-driven bean) for a message-style Web service.

  In RPC-style Web service, the stateless session EJBs might do all the actual work of the Web service, or they may parcel out the work to other EJBs. The implementer of the Web service decides which EJBs do the real work. In message-style Web services, a J2EE object (typically a message-driven bean) gets the messages from the JMS destination and processes them.

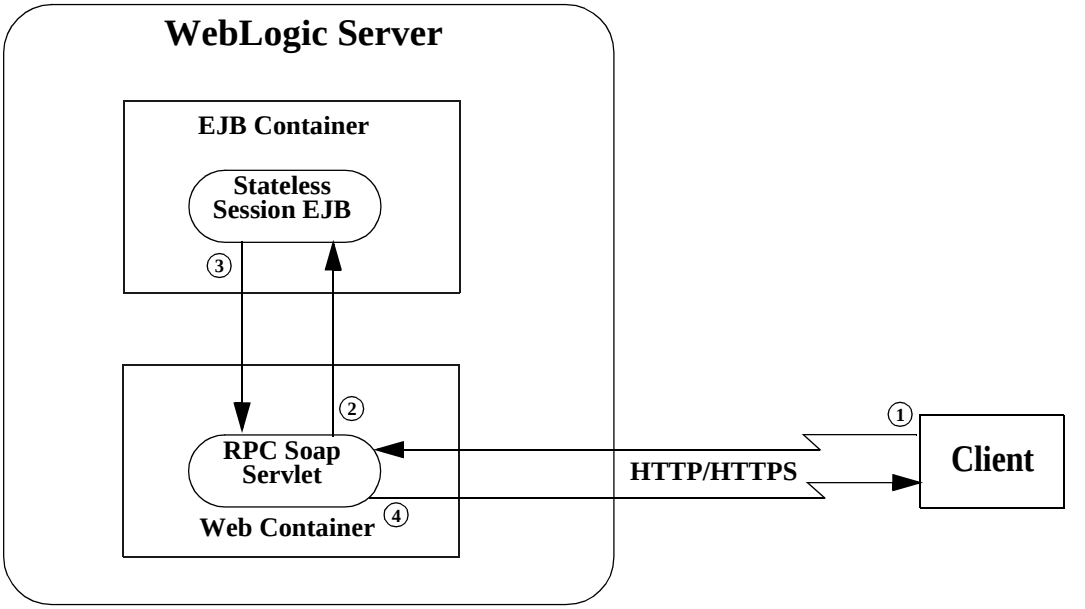
WebLogic Web Services are packaged as Enterprise archive (`*.ear`) files that contain the Web archive (`*.war`) files of the Web application and EJB archive (`*.jar`) files.

The following two sections describe the architecture of RPC-style and message-style Web services.

RPC-Style WebLogic Web Services Architecture

Figure 1-1 illustrates the architecture of RPC-style WebLogic Web Services.

Figure 1-1 RPC-Style WebLogic Web Services Architecture



Here's what happens when a client invokes an RPC-style WebLogic Web Service:

1. A client sends a SOAP message to WebLogic Server over HTTP/HTTPS. The SOAP message contains instructions, conforming to the WSDL of the Web service, to invoke an RPC-style Web service.
2. The SOAP servlet designed to handle RPC SOAP requests (which is part of the Web application invoked by the client) unwraps the SOAP message envelope and uses the unwrapped information to identify the appropriate stateless session EJB target. This servlet then unmarshals the parameters, binds them into the appropriate Java objects, invokes the target stateless session EJB, and passes it the parameters.

The stateless session EJB might perform all the work of the Web service, or it might parcel out some or all of the work to other EJBs.

3. The invoked stateless session EJB sends return values, if any, back to the RPC SOAP servlet.
4. The RPC SOAP servlet marshals the return values from the stateless session EJB into a SOAP message, and sends it back to the client over HTTP/HTTPS.

   If errors have occurred, the RPC SOAP servlet also sends a SOAP error message (called a SOAP *fault*) back to the client.

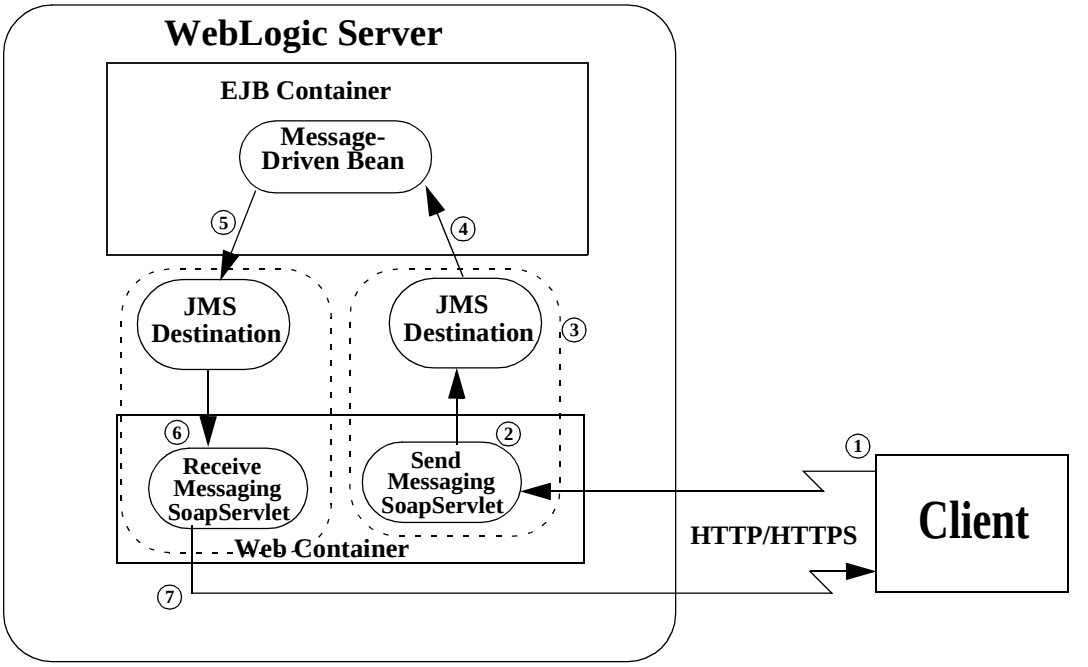
Message-Style WebLogic Web Services Architecture

Message-style Web services support a one-way communication; the client application either sends or receives a document to or from the Web service, but a single message-style Web service does not allow the client to do both. When you develop a message-style Web service, you specify whether the client sends or receives messages to or from the Web service. You can combine two message-style Web services, one for sending and one for receiving, in order to support round-trip communication. The same client can use both types, or either type, of service.

Figure 1-2 describes a possible architecture for both styles of message-style WebLogic Web Services working together.

Note: The dotted lines encapsulate two different message-style Web services. You do not have to use message-driven beans to take messages off the JMS destinations, although this is typically the best way to go.

Figure 1-2 Message-Style WebLogic Web Services Architecture



Here's what happens when a client invokes message-style WebLogic Web Services:

1. A client sends a SOAP message to WebLogic Server over HTTP/HTTPS. The SOAP message contains instructions, conforming to the WSDL of the Web service, to invoke a message-style Web service.
2. The messaging SOAP servlet that is part of the Web application invoked by the client unwraps the SOAP envelope, decodes the body of the message, and puts the resulting object on the appropriate JMS destination.

   Note: In WebLogic Server 6.1 there is no support for accessing the contents of the attachments to the SOAP 1.1 With Attachments message.
3. The message sits in the JMS destination until the appropriate JMS listener (typically a message-driven bean) picks it up.

4. The message-driven bean picks up the message from the JMS destination. The message-driven bean might do all the work of the Web service, or it might parcel out some or all of the work to other EJBs.
5. The message-driven bean sends the resulting document to another JMS destination that is associated with a separate message-style Web service that is configured to allow clients to receive messages.
6. The messaging SOAP servlet associated with the second Web service picks up the message from the JMS destination.
7. The messaging SOAP servlet sends the document back to the client when the client invokes the second receive Web service.

This sample architecture shows two message-style Web services working together to get and send back information to the client. Note that the client has to invoke *two* message-style Web services.

SOAP and WSDL Features Not Supported by WebLogic Web Services

The following SOAP features are not supported by WebLogic Web Services:

- the `Header` element - this means that you cannot set or get SOAP Header elements using the WebLogic Web Services client API. Additionally, the internal WebLogic Web services runtime ignores the SOAP Header; it only handles the SOAP Body.
- the SOAP attachment

The following WSDL features are not supported by WebLogic Web Services:

- the `import` element
- the `element` attribute of the `part` element

Editing XML Files

When creating or invoking WebLogic Web services, you might need to edit XML files, such as the EJB deployment descriptors, the Java Ant build files, and so on. To edit these files, BEA provides the BEA XML Editor, an entirely Java-based XML stand-alone editor.

The BEA XML Editor is a simple, user-friendly tool for creating and editing XML files. It displays XML file contents both as a hierarchical XML tree structure and as raw XML code. This dual presentation of the document provides you with the following two methods of editing the XML document:

- The hierarchical tree view allows structured, limited constrained editing, providing you with a set of allowable functions at each point in the hierarchical XML tree structure. The allowable functions are syntactically dictated and in accordance with the XML document's DTD or schema, if one is specified.
- The raw XML code view allows free-form editing of the data.

BEA XML Editor can validate XML code according to a specified DTD or XML schema.

For detailed information about using the BEA XML Editor, see its on-line help.

You can download the BEA XML Editor from the BEA dev2dev at http://dev2dev.bea.com/resourcelibrary/utilitiestools/xml.jsp?highlight=utilitiestools.

CHAPTER

2 Developing WebLogic Web Services

The following sections describe how to develop WebLogic Web Services:

Developing WebLogic Web Services: Main Steps

Most of the following steps are described in detail in later sections:

1. Design the WebLogic Web Service.

   Decide whether the Web service will be RPC-style or message-style, which EJB should implement the service, and so on. The section "Designing a WebLogic Web Service" on page 2-3 discusses design considerations.

2. Implement the WebLogic Web Service.

Write the business logic Java code for the EJBs that make up most of the WebLogic Web Service. For detailed information, see "Implementing a WebLogic Web Service" on page 2-17.

3. Package the EJBs that implement the Web service (stateless session EJB for RPC-style Web services and a message-driven bean for message-style Web services), along with any supporting EJBs, into an EJB archive file (`*.jar`).

   For detailed information on this step, refer to *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs61/programming/packaging.html.

4. Assemble the WebLogic Web Service.

   Package all the components that make up the service (such as stateless session EJBs, the Web application that contains a reference to the SOAP servlet, and so on) into a J2EE Enterprise Application archive (`*.ear`) file so that it can be deployed on WebLogic Server. You use Java Ant to assemble WebLogic Web Services. Assembling also refers to setting up other J2EE components, such as JMS destinations for message-style Web services.

   For detailed information, see "Assembling a WebLogic Web Service" on page 2-19.

5. Deploy the WebLogic Web Service.

   Make the service available to remote clients. For more information, see "Deploying a WebLogic Web Service" on page 2-25.

6. Create a client that accesses the Web service to test that your Web service is working as you expect. For detailed information, see Chapter 3, "Invoking WebLogic Web Services."

WebLogic Server includes examples of creating both RPC-style and message-style Web services and examples of both Java and Microsoft VisualBasic client applications that invoke the Web services.

The examples are located in the *`BEA_HOME`*`/samples/examples/webservices` directory, where *`BEA_HOME`* refers to the main WebLogic Server installation directory. The RPC-style Web service example is in the `rpc` directory and the message-style Web service example is in the `message` directory.

For detailed instructions on how to build and run the examples, invoke the Web page *`BEA_HOME`*`/samples/examples/webservices/package-summary.html` in your browser.

Designing a WebLogic Web Service

The bulk of WebLogic Web Services are the EJBs that do the work in the background after the SOAP request has been received and processed.

The first design issue is whether you should create an RPC-style or message-style Web service. This topic is discussed in "Choosing Between an RPC-Style and a Message-Style Web Service" on page 2-3.

The following sections discuss RPC-style design issues:

- EJB That Implements an RPC-Style Web Service
- Converting an Existing EJB Application into an RPC-Style Web Service
- Avoiding Overloaded Methods in Stateless Session EJBs

The following sections discuss message-style design issues:

- Message-Style Web Services and JMS
- Converting an Existing JMS Application Into a Web Service

The following sections discuss issues common to both types of Web services:

- Supported Data Types for Parameters and Return Values of WebLogic Web Services
- XML-Java Conversion in WebLogic Web Services
- Security Issues

Choosing Between an RPC-Style and a Message-Style Web Service

This section describes when to use an RPC-style or message-style Web service.

When to Use RPC-Style Web Services

RPC-style Web services are interface driven, which means that the business methods of the underlying stateless session EJB determine how the Web service works. When clients invoke the Web service, they send parameter values to the Web service, which executes the corresponding methods and sends back the return values. The relationship is synchronous, which means that the client waits for a response from the Web service before it continues with the remainder of its application.

Create an RPC-style Web service if your application has the following characteristics:

- The client invoking the Web service needs an immediate response.
- The client and Web service work in a back-and-forth, conversational way.
- The behavior of the Web service can be expressed as an interface.
- The Web service is process-oriented rather than data-oriented.

Examples of RPC-style Web services include providing the current weather conditions in a particular location; returning the current price for a given stock; or checking the credit rating of a potential trading partner prior to the completion of a business transaction. In each case the information is returned immediately, implying a synchronous relationship between the client and the Web service.

When to Use Message-Style Web Services

You should create a message-style Web service if your application has the following characteristics:

- the client has an asynchronous relationship with the Web service, or in other words, the client does not expect an immediate response.
- the Web service is data-oriented rather than process-oriented.

Examples of message-style Web services include processing a purchase order; accepting a request for new DSL home service; or responding to a request for quote order from a customer. In each case, the client sends an entire document, such as purchase order, to the Web service and assumes that the Web service is processing it in some way, but the client does not require an answer right away or even at all. If your Web service will work in this asynchronous, document-driven manner, then you should consider designing it as a message-style Web service.

EJB That Implements an RPC-Style Web Service

You implement an RPC-style Web service using a single stateless session EJB that either does all the actual work of the Web service or it parcels out some or all of the work to other EJBs. This EJB is the one that defines the methods that a client executes when it invokes a WebLogic Web Service.

Design your EJB to minimize the data that travels between the client and the Web service. This conversation is synchronous and over the Web, thus the fewer the requests and responses, the faster the entire transaction.

The data types of the parameters and return values of the EJB are restricted to a list of supported Web service data types, described in "Supported Data Types for Parameters and Return Values of WebLogic Web Services" on page 2-9. This data type restriction facilitates interoperability with other Web service implementation, both Java and non-Java, such as Microsoft SOAP ToolKit.

Converting an Existing EJB Application into an RPC-Style Web Service

You might be able to convert an existing stateless session EJB into an RPC-style Web service, as long as the data types of its parameters and return values are included in the list of supported Web services data types, listed in "Supported Data Types for Parameters and Return Values of WebLogic Web Services" on page 2-9.

If you cannot convert an existing EJB, then you must create a new stateless session EJB that implements the Web service, sends and receives parameters and return values from the client using the supported data types, then converts these values into the correct data types and passes the values to the existing stateless session EJB.

Alternatively, you can reprogram the existing stateless session EJB to accept as parameters and return values only the supported data types.

Avoiding Overloaded Methods in Stateless Session EJBs

Due to limitations in the SOAP specification, SOAP messages are unable to differentiate unambiguously between methods of the same name that have different signatures (overloaded methods). For this reason, WebLogic Server does not support overloaded methods in the EJBs that make up RPC-style Web services. Rather, each method should have its own unique name.

For example, assume your stateless session EJB defines a method called `myMethod()`, which can take as a parameter either a String or an integer. Because the SOAP specification does not force you to declare the data types of parameters in a SOAP message, the WebLogic Web Service might not know whether to execute `myMethod(String)` or `myMethod(int)` when a client invokes it. To clear up the confusion, rename one of the overloaded methods.

Message-Style Web Services and JMS

Message-style Web services use JMS listeners (such as message-driven beans) rather than stateless session EJBs as their entry points. This section describes the relationship between JMS and WebLogic Web Services and design considerations for developing message-style Web services.

Choosing a Queue or Topic

JMS queues implement a point-to-point messaging model whereby a message is delivered to exactly one recipient. JMS topics implement a publish/subscribe messaging model whereby a message is delivered to multiple recipients.

When you implement a message-style Web service you must make the following two decisions:

- Whether you want to use a JMS queue or topic.
- Whether the client application that invokes the Web service sends or receives the document to or from the service. The same service cannot support both sending and receiving.

Retrieving and Processing Documents

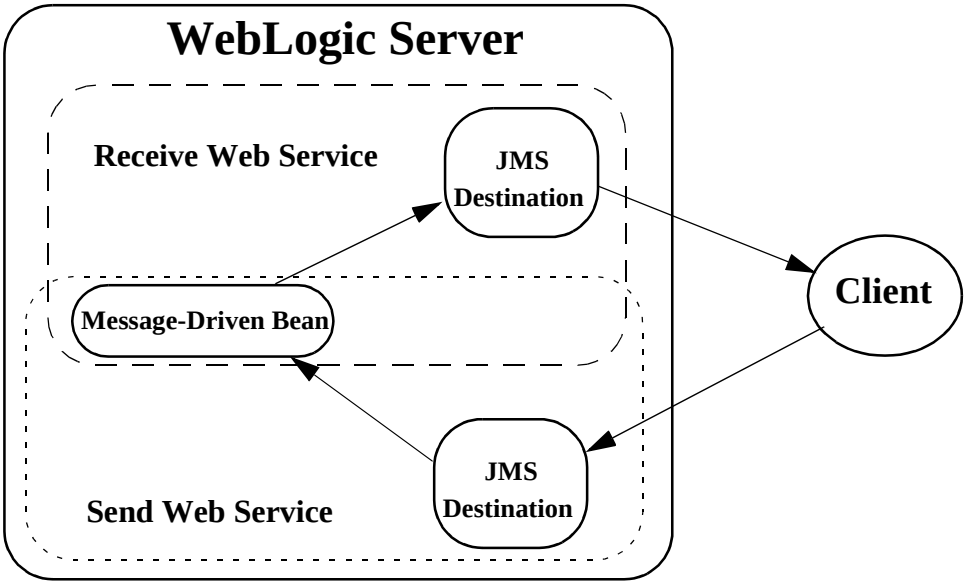
After you decide what type of JMS destination you are going to use, you must decide what type of J2EE component will retrieve the document from the JMS destination and process it. Typically this will be a message-driven bean. This message-driven bean can do all the document-processing work, or it can parcel out some or all of the work to other EJBs. Once the message-driven bean finishes processing the document, the execution of the Web service is complete.

This means that if you want the client that invokes the Web service by sending documents to receive some sort of response or data, you must create a second message-style Web service that the client subsequently invokes to retrieve a response. The second Web service is related to the original Web service because the original message-driven bean that processed the document puts the resulting information or response on the JMS destination corresponding to the second Web service. Again, you must decide whether the second JMS destination is a topic or a queue.

Example of Message-Style Web Services

As a simple example, Figure 2-1 shows two separate Web services, one for receiving a document from a client and one for sending a document back to the client. The two Web services have their own JMS destinations. The message-driven bean gets messages from the first JMS destination, processes the information, then puts a message back onto the second JMS destination. The client invokes the first Web service to send the document to WebLogic Server and then invokes the second Web service to receive a document back from WebLogic Server.

Figure 2-1 Data Flow Between Message-Style Web Services and JMS



Converting an Existing JMS Application Into a Web Service

You might be able to convert an existing JMS application into a message-style Web service, as long as the message-driven bean that gets messages from the JMS destination can handle the Java objects that end up on the JMS destination. For example, WebLogic Web Services convert standard XML documents from a client into `org.w3c.dom.Document` objects, as described in "XML-Java Conversion in WebLogic Web Services" on page 2-11.

If the message-driven bean in your existing JMS application expects some other type of document object, then you can do one of two things: either reprogram the message-driven bean to accept `org.w3c.dom.Document` objects, or create a new

message-driven bean that accepts `org.w3c.dom.Document` objects; converts them into the data type accepted by the original message-driven bean; and puts the new object on a JMS destination for the original message-driven bean to pick up.

Supported Data Types for Parameters and Return Values of WebLogic Web Services

To facilitate interoperability with other Web service implementations, both Java and non-Java, WebLogic limits the data types that can be used as parameters and return values to the Web service.

The following table lists the mapping between the supported Java data types and their XML equivalent.

Table 2-1 Java to XML Mapping

Java Data Type	Corresponding XML Data Type
int	int
boolean	boolean
float	float
long	long
short	short
double	double
java.lang.Integer	int
java.lang.Boolean	boolean
java.lang.Float	float
java.lang.Long	long
java.lang.Short	short
java.lang.Double	double

Table 2-1 Java to XML Mapping

Java Data Type	Corresponding XML Data Type
java.lang.String	string
java.math.BigDecimal	decimal
java.util.Date	dateTime
byte[]	base64Binary
java.lang.Object	anyType
JavaBeans whose properties are of the supported Java data types listed in this table or another JavaBean.	Compound struct whose members are of the supported XML data types listed in this table or another compound struct.
Arrays of supported Java data types listed in this table (except for the reference equivalents of primitive types, such as java.lang.Integer). Single-dimensional arrays only.	SOAP array of supported XML data types listed in this table. Single-dimensional arrays only.
org.w3c.dom.Document	No XML equivalent.
org.w3c.dom.DocumentFragment	No XML equivalent.
org.w3c.dom.Element	No XML equivalent.

The following table lists the mapping between the supported XML data types and their Java equivalent.

Table 2-2 XML to Java Mapping

XML Data Type	Corresponding Java Data Type
int	java.lang.Integer
boolean	java.lang.Boolean
float	java.lang.Float
long	java.lang.Long

Table 2-2 XML to Java Mapping

XML Data Type	Corresponding Java Data Type
short	java.lang.Short
double	java.lang.Double
decimal	java.math.BigDecimal
dateTime	java.util.Date
timeInstant	java.util.Date
byte	java.lang.Byte
base64Binary	byte[]
hexBinary	byte[]
Compound struct whose members are of the supported XML data types listed in this table or another compound struct.	JavaBeans whose properties are of the supported Java data types listed in this table or another JavaBean.
SOAP array of supported XML data types listed in this table. Single-dimensional arrays only.	Arrays of supported Java data types listed in this table (except for the reference equivalents of primitive types, such as java.lang.Integer). Single-dimensional arrays only.

XML-Java Conversion in WebLogic Web Services

WebLogic Web Services support the following two encoding styles:

- http://schemas.xmlsoap.org/soap/encoding/
- http://xml.apache.org/xml-soap/literalxml

Note: The preceding URIs are not “real” in the sense that you can actually invoke them in a browser. Rather, it is a standard convention to name encoding styles using URIs.

When a WebLogic Web Service receives data from a client, it uses the encoding style specified in the SOAP message to identify the data type of the parameter or message so that it can be converted to the correct Java object.

Note: If you create a Java client using WebLogic's generated Java client JAR file, you do not need to know about specific encoding styles, because the Java client JAR file contains code that handles it for you. This section is included for programmers who create non-Java clients that invoke WebLogic Web Services and need to know how they handle encoding styles.

If the SOAP packet specifies the SOAP encoding style, then the Web service tries to convert the XML data inside the body of the SOAP message into one of the Java data types listed in "Supported Data Types for Parameters and Return Values of WebLogic Web Services" on page 2-9.

If the conversion is unsuccessful (for example, if there is no corresponding Java data type defined for one of the parameters), then the Web service returns a SOAP fault to the client that invoked the Web service.

If the conversion from XML to Java is successful, then the different styles of Web services do different things:

- RPC-style Web services pass the resulting Java objects to the appropriate stateless session EJBs.
- Message-style Web services wrap the Java object into a JMS `javax.jms.ObjectMessage` data type and put the message on the appropriate JMS destination.

If the SOAP packet specifies the Literal XML encoding style, the Web service converts the XML data inside the body of the XML message into a `org.w3c.dom.Element` data type, and then either sends the document to a stateless session EJB or wraps the document in a `javax.jms.ObjectMessage` data type and puts the message on the appropriate JMS destination, depending on whether the Web service is RPC-style or message-style, respectively.

The reverse happens when WebLogic Web Services send data back to the client: `org.w3c.dom.Element` return values are encoded using the Literal XML encoding style before being sent back to the client, and other Java data types are encoded using the SOAP encoding style.

Security Issues

As previously discussed, WebLogic Web Services are packaged as standard J2EE Enterprise applications. Consequently, to secure access to the Web service, you secure access to some or all of the following standard J2EE components that make up the Web service:

- The SOAP servlets
- The stateless session EJB upon which an RPC-style Web service is based

You can use basic HTTP authentication or SSL to authenticate a client that is attempting to access a WebLogic Web Service. Because the preceding components are standard J2EE components, you secure them in using standard J2EE security procedures. For general information about basic HTTP authentication and SSL, see *Programming WebLogic Security* at http://e-docs.bea.com/wls/docs61/security/index.html.

For information about implementing 2-way SSL so that a client invoking a WebLogic Web Service is required to present its digital cerficate, see "Using 2-Way SSL When Invoking a WebLogic Web Service" on page 2-15.

Securing Message-Style Web Services

You secure a message-style Web service by securing the SOAP servlet that handles the SOAP messages between the client and the service.

Note: You can also use this method to secure an RPC-style Web service, although BEA recommends instead that you secure the EJB, as described in "Securing an RPC-Style Web service" on page 2-15.

When you assemble a WebLogic Web Service, either using the `wsgen` Ant task or manually, you reference SOAP servlets in the `web.xml` file of the Web application. These SOAP servlets handle the SOAP messages between WebLogic Server and client applications. They are always deployed on WebLogic Server, and are shared by all deployed WebLogic Web Services.

The particular SOAP servlet referenced by a Web service depends on its type (RPC-style or message-style). The following list describes each SOAP servlet:

- `weblogic.soap.server.servlet.DestinationSendAdapter`—handles SOAP messages in a message-style Web service that receives data from a client application to a JMS destination.
- `weblogic.soap.server.servlet.QueueReceiveAdapter`—handles SOAP messages in a message-style Web service that sends data from a JMS Queue to a client application.
- `weblogic.soap.server.servlet.TopicReceiveAdapter`—handles SOAP messages in a message-style Web service that sends data from a JMS Topic to a client application.
- `weblogic.soap.server.servlet.StatelessBeanAdapter`—handles SOAP messages between an RPC-style Web service and a client application.

For example, assume you have created a message-style Web service in which client applications send data to a JMS destination; the SOAP servlet that handles the SOAP messages is `weblogic.soap.server.servlet.DestinationSendAdapter`. The `wsgen` Ant task used to assemble the Web service adds the following elements to the `web.xml` deployment descriptor of the Web application:

```
<servlet>
  <servlet-name>sender</servlet-name>
  <servlet-class>
     weblogic.soap.server.servlet.DestinationSendAdapter
  </servlet-class>
  <init-param>
    <param-name>topic-resource-ref</param-name>
    <param-value>senderDestination</param-value>
  </init-param>
  <init-param>
    <param-name>connection-factory-resource-ref</param-name>
    <param-value>senderFactory</param-value>
  </init-param>
</servlet>
...

<servlet-mapping>
  <servlet-name>sender</servlet-name>
  <url-pattern>/sendMsg</url-pattern>
</servlet-mapping>
```

To restrict access to the `DestinationSendAdapter` SOAP servlet, you first define a role that is mapped to one or more principals in a security realm, then specify that the security constraint applies to this SOAP servlet by adding the following `url-pattern` element inside the `web-resources-collection` element to the `web.xml` deployment descriptor of the Web application:

```
<url-pattern>/sendMsg</url-pattern>
```

See Appendix C, "Manually Assembling the Web Services Archive File," for information on the structure of the Enterprise Application archive created by the `wsgen` Ant task.

For detailed procedural information about restricting access to servlets, see *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/security.html.

Securing an RPC-Style Web service

Restrict access to an RPC-style Web service by restricting access to the stateless session EJB that implements the Web service.

Thus client applications that invoke the RPC-style Web service always have access to the Web application and SOAP servlets, but might not be able to invoke the EJB. This type of security is useful if you want to closely monitor who has access to the business logic of the EJB but do not want to block access to the entire Web service.

For information about restricting access to EJBs, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html.

Using 2-Way SSL When Invoking a WebLogic Web Service

In 2-way SSL, client applications that invoke a WebLogic Web Service are required to present their digital certificates to WebLogic Server, which validates digital certificates against a list of trusted certificate authorities.

To use 2-way SSL when writing a Java client to invoke a WebLogic Web Service, follow these steps:

1. Configure WebLogic Server for 2-way SSL protocol (also called mutual authentication) and certificate authentication.

   For details, see *Configuring the SSL Protocol* at http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html#cnfgsec015 and

Configuring Mutual Authentication at http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html#cnfgsec020.

2. Add the following lines of Java code to your client application before you obtain the context you are using the look up your Web service::

```
System.out.println("********************** loading client certs");

InputStream certs[] = new InputStream[3];
certs[0]=new PEMInputStream(new FileInputStream("sample_key.pem"));
certs[1]=new PEMInputStream(new FileInputStream("sample_cert.pem"));
certs[2]=new PEMInputStream(new FileInputStream("sample_ca.pem"));

h.put(SoapContext.SSL_CLIENT_CERTIFICATE, certs);

String prov = "weblogic.net";

String s = System.getProperty("java.protocol.handler.pkgs");
if (s == null) {
    s = prov;
} else if (s.indexOf(prov) == -1) {
    s += "|" + prov;
}

System.setProperty("java.protocol.handler.pkgs", s);
```

In the preceding code excerpt:

- `sample_key.pem` is the name of the file that contains the client's private key associated with the certificate.
- `sample_cert.pem` is the name of the file that contains the client's certificate.
- `sample_ca.pem` is the name of the file that contains the certificate of the Certificate Authority that issued the client's certificate.

Note: When establishing an SSL connection, the subject DN of the digital certificate must match the host name of the server initiating the SSL connection. Otherwise, the SSL connection is dropped. If you use the demonstration certificates provided by WebLogic Server, the host names will not match.

To avoid this situation, use the `-Dweblogic.security.SSL.ignoreHostnameVerification=true` flag when running your client application, or even when starting WebLogic Server if you want this to be true all the time. This flag disables the Host Name

Verifier which compares the subject DNs and host names. This solution is recommended in development environments only. A more secure solution is to obtain a new digital certificate for the server making outbound SSL connections.

Implementing a WebLogic Web Service

Implementing a WebLogic Web Service refers to writing the Java code for the stateless session EJB (for RPC-style Web services) or a JMS listener (for message-style Web services) that is defined to be the entry point to the Web service. JMS listeners are typically message-driven beans. The stateless session EJB or JMS listener may contain all the Web service functionality, or it may call other EJBs to parcel out the work.

It is assumed that you have read and understood the design issues discussed in "Designing a WebLogic Web Service" on page 2-3, that you have designed your Web service, and that you essentially know the types of components you need to code.

Implementing an RPC-Style Web Service

To implement an RPC-style Web service, write the Java code for the stateless session EJB. Remember to use only the supported Java data types as the parameters and return value of the EJB, listed in "Supported Data Types for Parameters and Return Values of WebLogic Web Services" on page 2-9.

For detailed information about programming stateless session EJBs, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html.

Implementing Message-Style Web Services

There are two types of message-style Web services, as described in "Message-Style Web Services and JMS" on page 2-6: those that receive XML data from a client that invokes the Web service and those that send XML data to a client.

To implement a message-style Web service, follow these steps:

1. Use the Administration Console to configure the following JMS components of WebLogic Server:
   - The JMS destination (queue or topic) that will either receive the XML data from a client or send XML data to a client. Later, when you assemble the Web service as described in "Assembling a WebLogic Web Service" on page 2-19, you will use the name of this JMS destination.
   - The JMS Connection factory that the WebLogic Web Service uses to create JMS connections.

   See "Configuring JMS Components for Message-Style Web Services" on page 2-18 for details on this step. For general information about JMS, see the *WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/jms.html.
2. Write the Java code for the J2EE component (typically a message-driven bean) that will take messages off the JMS destination for message-style Web services that receive XML data from a client or will put messages on a JMS destination for message-style Web services that send XML data to a client.

   For detailed information about programming message-driven beans, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html.

Configuring JMS Components for Message-Style Web Services

This section assumes that you have already configured a JMS server. For information about configuring JMS servers, and general information about JMS, see the *WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/jms.html and *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs61/jms/index.html.

To configure a JMS destination (either queue or topic) and JMS Connection Factory, follow these steps:

1. Invoke the Administration Console in your browser. For details, see "Invoking the Administration Console" on page 4-1.
2. Click to expand the Services node in the left pane and expand the JMS node.

3. Right-click the Connection Factories node and choose Configure a new JMSConnectionFactory from the drop-down list.
4. Enter a name for the Connection Factory in the Name field.
5. Enter the JNDI name of the Connection Factory in the JNDIName field.
6. Click Create.
7. Click the Targets tab.
8. Move the name of the WebLogic Server hosting the service to the Chosen list box, if not already there.
9. Click Apply.
10. Click to expand the Servers node under the JMS node in the left pane.
11. Click to expand your JMS server node.
12. Right-click the Destinations node and choose either:
    - Configure a new JMSTopic from the drop-down list if you want to create a topic
    - Configure a new JMSQueue if you want to create a queue.
13. Enter the name of the JMS destination in the Name text field.
14. Enter the JNDI name of the destination in the JNDIName text field.
15. Click Create.

Assembling a WebLogic Web Service

This section describes how to assemble all the components of a Web service so it can be deployed on WebLogic Server and accessed by remote clients.

Assembling a WebLogic Web Service Using Java Ant Tasks

Assembling a WebLogic Web Service refers to packaging all the components of the Web service, such as the EJB that implements an RPC-style Web service, supporting EJBs, the Web application that contains the SOAP servlet, and so on, into an Enterprise Application archive (`*.ear`) so it can be deployed on WebLogic Server.

Developers use a Java Ant task, called `wsgen`, to assemble WebLogic Web Services. The `wsgen` Ant task generates most of the WebLogic Web Service components, such as the Web application that contains the SOAP servlet and the `application.xml` file that describes the Enterprise Application archive. The only components you need to have previously created are the EJB or message-driven beans that implement the Web service.

For general information about Ant, see http://jakarta.apache.org/ant/index.html.

Note: The Java Ant utility included in WebLogic Server uses the `ant` (UNIX) or `ant.bat` (Windows) configuration files in the *`BEA_HOME`*`\bin` directory when setting the ANTCLASSPATH variable, where *`BEA_HOME`* is the directory in which WebLogic Server is installed. If you need to update the ANTCLASSPATH variable, make the appropriate changes to these files.

For detailed procedures for assembling WebLogic Web Services manually, see Appendix C, "Manually Assembling the Web Services Archive File."

To assemble a WebLogic Web Service, follow these steps:

1. Create a temporary staging directory.
2. If you are assembling an RPC-style Web service, copy the EJB `*.jar` file that contains the EJB that implements the service, along with any supporting EJBs, to the staging directory.
3. Set up your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in the directory *`BEA_HOME`*`\config\`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

On UNIX, execute the `setEnv.sh` command, located in the directory *`BEA_HOME`*`/config/`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

4. Create a file called `build.xml` in the staging directory that contains the Ant task elements for assembling a WebLogic Web Service.

   For details on creating the `build.xml` file, refer to "Example of an Ant build.xml File" on page 2-21.

5. Change location to the staging directory and execute the Ant utility:

   ```
   $ ant
   ```

   The `wsgen` Ant task creates an `*.ear` file containing the service components in the staging directory. You are now ready to deploy this `*.ear` file on WebLogic Server.

Example of an Ant build.xml File

WebLogic Server includes the `wsgen` Ant task to help you quickly assemble the components of a WebLogic Web Service into an Enterprise archive file.

The following example shows a `build.xml` file that assembles three Web services: one RPC-style and two message-style (one for sending messages and one for receiving messages). Table 2-3 describes the file elements.

Listing 2-1 Example build.xml File for Assembling WebLogic Web Services

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
      <wsgen
             destpath="myWebService.ear"
             context="/myContext"
             protocol="http">
         <rpcservices path="myEJB.jar">
            <rpcservice
               bean="statelessSession"
               uri="/rpc_URI"/>
         </rpcservices>
         <messageservices>
           <messageservice
             name="sendMsgWS"
             action="send"
```

```
            destination="examples.soap.msgService.MsgSend"
            destinationtype="topic"
            uri="/sendMsg"
            connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
         <messageservice
            name="receiveMsgWS"
            action="receive"
            destination="examples.soap.msgService.MsgReceive"
            destinationtype="topic"
            uri="/receiveMsg"
            connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
      </messageservices>
      </wsgen>
   </target>
</project>
```

Table 2-3 Description of build.xml Example

Element or Attribute	Description
`wsgen` element	Specifies the `wsgen` Ant task used to assemble the Web service.
`destpath` attribute	Specifies that the resulting Enterprise archive will be called `myWebService.ear`.
`context` attribute	Specifies that the context root of the Web service is called `/myContext`. You will later use this context root in the URL used to view the generated WSDL for the Web service and to download the Java client JAR file.
`protocol` attribute	Specifies that clients use HTTP to invoke the service.
`rpcservices` element	Contains the single RPC-style Web service that is associated with the `/myContext` context.
`path` attribute	Specifies that the EJBs are archived in a JAR file called `myEJB.jar`.
`rpcservice` element	Specifies the properties of the RPC-style Web service.
`bean` attribute	Specifies that the name of the stateless session EJB that implements the RPC-style Web service is `statelessSession`. This name corresponds to the `ejb-name` element in the `ejb-jar.xml` file of the EJB archive in which the EJB is contained. The path to the EJB archive is specified in the parent `rpcservices` element using the `path` attribute.
`uri` attribute	Specifies that the URI of the service is `/rpc_URI`. This URI is used in the URL to access the WSDL of the Web service.

Element or Attribute	Description
`messageservices` element	Contains two message-style Web services that are associated with the /myContext context.
`messageservice` element	Specifies the properties of each message-style Web service.
`name` attribute	Assigns a unique name to each service: `sendMsgWS` and `receiveMsgWS`.
`action` attribute	Specifies whether a client that invokes the Web service sends or receives messages from the service. The first service specifies `send`, the second `receive`.
`destination` attribute	Specifies the JNDI name of the JMS destination that sends or receives messages. The first service specifies `examples.soap.msgService.MsgSend`, the second specifies `examples.soap.msgService.MsgReceive`.
`destinationtype` attribute	Specifies whether the JMS destination is a topic or a queue. Both services specify `topic`.
`uri` attribute	Specifies that the URIs of the services are `/sendMsg` and `/receiveMsg`, respectively. The URIs are combined to create the complete URL to the WSDL of the Web service
`connectionfactory` attribute	Specifies the JNDI name of the Connection Factory used to create a JMS connection. Both services use the same Factory: `examples.soap.msgService.MsgConnectionFactory`.

For a detailed description of the elements and attributes of the `build.xml` file, refer to Appendix B, "build.xml Elements and Attributes."

Creating the build.xml Ant Build File

The following procedure describes the Ant task elements you must include in your `build.xml` file to correctly assemble a WebLogic Web Service; use the example in the preceding section as a guide.

For detailed description of the elements and attributes of the `build.xml` file mentioned in the following procedure, as well as additional elements you can specify, refer to Appendix B, "build.xml Elements and Attributes."

See "Editing XML Files" on page 1-15 for information on using the BEA XML Editor to create and edit the `build.xml` file.

To create a `build.xml` Ant build file for assembling WebLogic Web Services:

1. Create an empty file called `build.xml` using your favorite text editor.
2. Add one `<project>` element with the following two attributes:
   - `name` - the name of your project.
   - `default` - set this attribute to `wsgen`.
3. Within the `<project>` element, add a `<target>` element with one attribute, `name`; set the `name` attribute to `wsgen`.
4. Within the `<target>` element, add a `<wsgen>` element with the following attributes:
   - `destpath`
   - `context`
   - `protocol`
5. If you are assembling one or more RPC-style Web services, add a single `<rpcservices>` element within the `<wsgen>` element with the following attributes:
   - `path`
6. Within the `<rpcservices>` element, add an `<rpcservice>` element for each RPC-style Web service you are assembling, with the following attributes:
   - `bean`
   - `uri`
7. If you are assembling one or more message-style Web services, add a single `<messageservices>` element within the `<wsgen>` element.
8. Within the `<messageservices>` element, add a `<messageservice>` element for each message-style Web service you are assembling, with the following attributes that describe the JMS destination and Connection factory that you previously set up for the message-style Web service:
   - `name`
   - `action`
   - `destination`

- `destinationtype`
- `uri`
- `connectionfactory`

Dynamic or Static WSDL?

WebLogic Web Services publish their WSDL files as JSPs. The WSDL JSP can either hard-code the host and port of a specific WebLogic Server, or it can dynamically generate the host and port based on the WebLogic Server that is hosting the service.

Typically, you want the WSDL of a WebLogic Web Service to dynamically generate the host and port, and you do this by *not* specifying the `host` and `port` attributes of the `wsgen` element in the `build.xml` Ant file used to assemble the Web service. If, however, you want the host and port to be hard-coded in the WSDL JSP, explicitly specify the `host` and `port` attributes.

Deploying a WebLogic Web Service

Deploying a WebLogic Web Service refers to making it available to remote clients. Because WebLogic Web Services are packaged as standard J2EE Enterprise applications, deploying a Web service is the same as deploying an Enterprise application.

For detailed information on deploying Enterprise applications, see *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html.

Developing a WebLogic Web Service: A Simple Example

This section describes the start-to-finish process of developing, assembling, and deploying the sample RPC-style WebLogic Web Service provided as a product example in the directory *BEA_HOME*/samples/examples/rpc.

To develop the sample Weather RPC-style WebLogic Web Service, follow these basic steps:

1. Set up your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in the directory *BEA_HOME*\config*domain*, where *BEA_HOME* is the directory in which WebLogic Server is installed and *domain* refers to the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in the directory *BEA_HOME*/config/*domain*, where *BEA_HOME* is the directory in which WebLogic Server is installed and *domain* refers to the name of your domain.

2. Write the Java interfaces and classes for the Weather stateless session EJB.

   See “Writing the Java Code for the EJB” on page 2-27 for details.

3. Compile the EJB Java code into class files.

4. Create the EJB deployment descriptors.

   See “Creating EJB Deployment Descriptors” on page 2-31 for details.

5. Assemble the EJB class files and deployment descriptors into a `weather.jar` archive file.

   See “Assembling the EJB” on page 2-32 for details.

6. Create the `build.xml` Java Ant build file used to assemble the WebLogic Web Service.

   See “Creating the build.xml File” on page 2-33 for details.

7. Create a staging directory.

8. Copy the EJB `weather.jar` file and the `build.xml` file into the staging directory.
9. Execute the Java Ant utility to assemble the Weather Web service into a `weather.ear` archive file:

   ```
   $ ant
   ```

10. Auto-deploy the Weather Web service for testing purposes by copying the `weather.ear` archive file to the *`BEA_HOME`*`/config/`*`domain`*`/applications` directory, where *`BEA_HOME`* refers to the main WebLogic Server installation directory and *`domain`* refers to the name of your domain.

To invoke the Weather Web service from both a Java and a Visual Basic client application, see the examples in *`BEA_HOME`*`/samples/examples/webservices/rpc/javaClient` and *`BEA_HOME`*`/samples/examples/webservices/rpc/vbClient`.

For instructions for building and running the client applications, invoke the *`BEA_HOME`*`/samples/examples/webservices/rpc/package-summary.html` Web page in your browser.

Writing the Java Code for the EJB

The sample Weather stateless session EJB contains one public method: `getTemp()`. The method takes a single argument, a zip code, and returns a float value of 77 if the zip code is 90210 and -273.15 otherwise.

Note: This method obviously simulates a real-world Web service that returns the *actual* temperature at a given zip code.

The following Java code is the public interface of the Weather EJB:

```
package examples.webservices.rpc.weatherEJB;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

/**
 * The methods in this interface are the public face of WeatherBean.
 * The signatures of the methods are identical to those of the EJBean, except
 * that these methods throw a java.rmi.RemoteException.
 * Note that the EJBean does not implement this interface. The corresponding
```

```
 * code-generated EJBObject, WeatherBean, implements this interface and
 * delegates to the bean.
 *
 * @author Copyright (c) 1998 by WebLogic, Inc. All Rights Reserved.
 * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
 */

public interface Weather extends EJBObject {
  /**
   * Gets the temperature of a given ZipCode.
   *
   * @param ZipCode           String Stock symbol
   * @return                  double Temperature
   * @exception               RemoteException if there is
   *                          a communications or systems failure
   */

  public float getTemp(String ZipCode) throws RemoteException;
}
```

The following Java code is the actual stateless session EJB class:

```
package examples.webservices.rpc.weatherEJB;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * WeatherBean is a stateless Session Bean. This bean illustrates:
 * <ul>
 * <li> No persistence of state between calls to the Session Bean
 * <li> Looking up values from the Environment
 * </ul>
 *
 * @author Copyright (c) 1998 by WebLogic, Inc. All Rights Reserved.
 * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
 */

public class WeatherBean implements SessionBean {
  private static final boolean VERBOSE = true;
  private SessionContext ctx;
  private int tradeLimit;
  // You might also consider using WebLogic's log service

  private void log(String s) {
    if (VERBOSE) System.out.println(s);
  }
```

```
/**
 * This method is required by the EJB Specification,
 * but is not used by this example.
 *
 */
public void ejbActivate() {
  log("ejbActivate called");
}
/**
 * This method is required by the EJB Specification,
 * but is not used by this example.
 *
 */
public void ejbRemove() {
  log("ejbRemove called");
}
/**
 * This method is required by the EJB Specification,
 * but is not used by this example.
 *
 */
public void ejbPassivate() {
  log("ejbPassivate called");
}
/**
 * Sets the session context.
 *
 * @param ctx               SessionContext Context for session
 */
public void setSessionContext(SessionContext ctx) {
  log("setSessionContext called");
  this.ctx = ctx;
}
/**
 * This method corresponds to the create method in the home interface
 * "WeatherHome.java".
 * The parameter sets of the two methods are identical. When the client calls
 * <code>WeatherHome.create()</code>, the container allocates an instance of
 * the EJBean and calls <code>ejbCreate()</code>.
 *
 * @exception               javax.ejb.CreateException if there is
 *                          a communications or systems failure
 * @see                     examples.ejb.basic.statelessSession.Weather
 */
public void ejbCreate () throws CreateException {
  log("ejbCreate called");
  try {
    InitialContext ic = new InitialContext();
  } catch (NamingException ne) {
```

```
      throw new CreateException("Failed to find environment value "+ne);
    }
  }
  /**
   * Gets the temperature of a given ZipCode.
   *
   * @param ZipCode           String ZipCode
   * @return                  float Temperature
   * @exception               RemoteException if there is
   *                          a communications or systems failure
   */
  public float getTemp(String ZipCode) {
    log("getTemp called");
    Float result;
    if (ZipCode.equals("90210")) {
      result = new Float(77.0);
    } else {
      result = new Float(-273.15);
    }
    return result.floatValue();
  }
}
```

The following Java code is the Home interface of the Weather EJB:

```
package examples.webservices.rpc.weatherEJB;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;


/**
 * This interface is the home interface for the WeatherBean.java,
 * which in WebLogic is implemented by the code-generated container
 * class WeatherBeanC. A home interface may support one or more create
 * methods, which must correspond to methods named "ejbCreate" in the EJBean.
 *
 * @author Copyright (c) 1998 by WebLogic, Inc. All Rights Reserved.
 * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
 */
public interface WeatherHome extends EJBHome {
  /**
   * This method corresponds to the ejbCreate method in the bean
   * "WeatherBean.java".
   * The parameter sets of the two methods are identical. When the client calls
   * <code>WeatherHome.create()</code>, the container
   * allocates an instance of the EJBean and calls <code>ejbCreate()</code>.
   *
```

```
   * @return                  Weather
   * @exception               RemoteException if there is
   *                          a communications or systems failure
   * @exception               CreateException
   *                          if there is a problem creating the bean
   * @see                     examples.ejb.basic.statelessSession.WeatherBean
   */
  Weather create() throws CreateException, RemoteException;
}
```

Creating EJB Deployment Descriptors

See "Editing XML Files" on page 1-15 for information on using the BEA XML Editor to create and edit the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files.

The following example shows the `ejb-jar.xml` deployment descriptor that describes the Weather EJB:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar
     PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN'
            'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
        <enterprise-beans>
                <session>
                   <ejb-name>statelessSession</ejb-name>
                   <home>
                       examples.webservices.rpc.weatherEJB.WeatherHome
                   </home>
                   <remote>
                       examples.webservices.rpc.weatherEJB.Weather
                   </remote>
                   <ejb-class>
                       examples.webservices.rpc.weatherEJB.WeatherBean
                   </ejb-class>
                   <session-type>Stateless</session-type>
                   <transaction-type>Container</transaction-type>
                </session>
        </enterprise-beans>
        <assembly-descriptor>
                <container-transaction>
                   <method>
                      <ejb-name>statelessSession</ejb-name>
                      <method-intf>Remote</method-intf>
                      <method-name>*</method-name>
```

```
                    </method>
                    <trans-attribute>Required</trans-attribute>
                </container-transaction>
        </assembly-descriptor>
</ejb-jar>
```

The following example shows the `weblogic-ejb-jar.xml` deployment descriptor that describes the Weather EJB:

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar
           PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB//EN'
          'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd'>

<weblogic-ejb-jar>
        <weblogic-enterprise-bean>
        <ejb-name>statelessSession</ejb-name>
                <caching-descriptor>
                    <max-beans-in-free-pool>100</max-beans-in-free-pool>
                </caching-descriptor>
                <jndi-name>statelessSession.WeatherHome</jndi-name>
        </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Assembling the EJB

To assemble the EJB class files and deployment descriptors into a `weather.jar` archive file, follow these steps:

1. Create a temporary staging directory.
2. Copy the compiled Java EJB class files into the staging directory.
3. Create a `META-INF` subdirectory in the staging directory.
4. Copy the `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptors into the `META-INF` subdirectory.
5. Create the `weather.jar` archive file using the jar utility:

   ```
   jar cvf weather.jar -C staging_dir .
   ```

Creating the build.xml File

See "Editing XML Files" on page 1-15 for information on using the BEA XML Editor to create and edit the `build.xml` file.

The following `build.xml` file references the `wsgen` Java ant task that assembles the `weather.jar` archive file into a WebLogic Web Service `weather.ear` enterprise application archive file:

```
<project name="weather-webservice" default="wsgen">
    <target name="wsgen">
      <wsgen
         destpath="weather.ear"
         context="/weather">
        <rpcservices path="weather.jar">
                <rpcservice bean="statelessSession" uri="/weatheruri"/>
        </rpcservices>
      </wsgen>
    </target>
</project>
```

CHAPTER

3 Invoking WebLogic Web Services

The following sections describe how to invoke WebLogic Web Services from client applications:

Overview of Invoking WebLogic Web Services

Invoking a WebLogic Web Service refers to the actions that a client application performs to use the Web service. Client applications that invoke WebLogic Web Services can be written using any technology: Java, Microsoft SOAP Toolkit, and so on.

The client application assembles a SOAP message that describes the Web service it wants to invoke and includes all the necessary data in the body of the SOAP message. The client then sends the SOAP message over HTTP/HTTPS to WebLogic Server, which executes the Web service and sends a SOAP message back to the client over HTTP/HTTPS.

Note: If you write your client application in Java, WebLogic Server provides an optional Java client JAR file that includes, for your convenience, everything you need to invoke a WebLogic Web Service, such as the WebLogic Web Services Client API and WebLogic FastParser. Unlike other Java WebLogic Server clients, you do not need to include the `weblogic.jar` file, thus making for a very thin client. For details on downloading this JAR file, see "Downloading the Java Client JAR File from the Web Services Home Page" on page 3-6.

Each Web service has its own Home Page; Web services that share the same servlet context share this Web page. You use this Web page to get the WSDL and Java client JAR file for a Web service. See "Invoking the WebLogic Web Services Home Page" on page 3-4 for details on this Web page and how to invoke it in your browser.

WebLogic Web Services Client API

WebLogic Server includes a client-side Java SOAP API in a Java client JAR file that you can download from a deployed WebLogic Web Service. Use this API to create Java client applications that invoke WebLogic Web Services. The examples in this book, as well as the examples on the product, use this API.

Warning: A standard client-side Web Service API specification from the W3C or JavaSoft is not yet available. Because the WebLogic Web Services client API has not yet been standardized in the Java community process, BEA Systems reserves the right to change how it works from one release to another, and may not be able to make it backward compatible.

The examples in this chapter briefly describe the main classes, interfaces, and methods of the WebLogic Web Services client API. For detailed documentation on the API, see the *WebLogic Server API Reference* and search for the `weblogic.soap` package.

Client Modes Supported by the WebLogic Web Services Client API

The WebLogic Web Services client API supports the following two modes of Java client applications that invoke WebLogic Web Services:

- **Static**: Static client applications explicitly use the EJB and JavaBean interfaces and classes that make up the Web service. These types of client applications are the most type-safe of the two modes supported by WebLogic Server, and are thus the type recommended by BEA. Additionally, static client applications do not contain any WebLogic-specific Java code. For an example of a static Java client application, see "Writing a Static Java Client" on page 3-9.
- **Dynamic**: Dynamic client applications do not explicitly reference the EJB interface of the Web service. For an example of a dynamic client application, see "Writing a Dynamic Java Client" on page 3-11.

Both the static and dynamic client applications described in this chapter use the WSDL of the Web service. See Appendix D, "Invoking Web Services Without Using the WSDL File," for information on creating a client application that does not use the WSDL.

You can use both static and dynamic client applications to invoke either RPC-style or message-style Web services.

Examples of Clients That Invoke WebLogic Web Services

WebLogic Server includes examples of creating both RPC-style and message-style Web services and examples of both Java and Microsoft VisualBasic client applications that invoke the Web services.

The examples are located in the *BEA_HOME*/samples/examples/webservices directory, where *BEA_HOME* refers to the main WebLogic Server installation directory. The RPC-style Web service example is in the rpc directory and the message-style Web service example is in the message directory.

For detailed instructions on how to build and run the examples, invoke the Web page *BEA_HOME*/samples/examples/webservices/package-summary.html in your browser.

Invoking the WebLogic Web Services Home Page

The WebLogic Web Services Home Page lists the Web services defined for a particular servlet context along with the WSDL files and Java client JAR file associated with each Web service.

Use the following template URL to invoke the WebLogic Web Services Home Page in your browser:

[*protocol*]://[*host*]:[*port*]/[*context*]/index.html

where

- *protocol* refers to the protocol attribute of the <wsgen> element of the build.xml Ant file used to build the Web service. The two valid values are http (default) and https.
- *host* refers to the hostname of the computer which hosts the Web service.
- *port* refers to the port number of the WebLogic Server instance that hosts the Web service.

- *context* refers to the `context` attribute of the `<wsgen>` element of the `build.xml` Ant file used to build the Web service.

For example, assume that you built a Web service using the following `build.xml` file:

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
      <wsgen
             destpath="myWebService.ear"
             context="/myContext"
             protocol="http">
         <rpcservices path="myEJB.jar">
            <rpcservice
              bean="statelessSession"
              uri="/rpc_URI"/>
         </rpcservices>
         <messageservices>
           <messageservice
             name="sendMsgWS"
             action="send"
             destination="examples.soap.msgService.MsgSend"
             destinationtype="topic"
             uri="/sendMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
          <messageservice
             name="receiveMsgWS"
             action="receive"
             destination="examples.soap.msgService.MsgReceive"
             destinationtype="topic"
             uri="/receiveMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
      </messageservices>
      </wsgen>
   </target>
</project>
```

The URL to invoke the WebLogic Web Services Home Page for the `/myContext` context on the `myHost` host at the default port of `7001` is:

```
http://www.myHost.com:7001/myContext/index.html
```

Getting the WSDL from the Web Services Home Page

To get the WSDL of a Web service from the Web Services Home Page:

1. Invoke the Web Services Home Page for your context in your browser, as described in “Invoking the WebLogic Web Services Home Page” on page 3-4.
2. Click the name of the Web service.
3. Click the **WSDL File** link. The WSDL file for the specified Web service appears in your browser in plain text.

Downloading the Java Client JAR File from the Web Services Home Page

WebLogic Server provides a Java client JAR file that contains most of the Java code you need to create a Java client application that invokes a WebLogic Web Service. In particular, the JAR file includes the WebLogic implementation of a client-side SOAP API, which means that you do not have to write the low-level Java code to create and process SOAP messages.

The Java client JAR file contains the following objects:

- WebLogic FastParser (high-performance XML parser)
- WebLogic Web Services Client API
- Remote interface of the stateless session EJB that implements the RPC-style Web service
- Class files for any JavaBeans that are used as EJB parameters or return values
- Additional class files specified by the `clientjar` element of the `build.xml` Java Ant build file used to assemble the Web service

Note: BEA does not currently license client functionality separately from the server functionality, so, if needed, you can redistribute this Java client JAR file to your own customers.

To download the Java client JAR file to your computer:

1. Invoke the Web Services Home Page for a given context in your browser, as described in “Invoking the WebLogic Web Services Home Page” on page 3-4.
1. Click the name of the Web service.

2. Click the **Client JAR File** link.
3. Specify a directory on your local computer in which to store the Java client JAR file.
4. Save the JAR file to the specified directory.
5. Update your CLASSPATH to include the Java client JAR file.

URLs to Invoke WebLogic Web Services and Get the WSDL

WSDL is used by client applications to describe the Web services they invoke.

The full URL to directly access the WSDL of a WebLogic Web Service is:

`[protocol]://[host]:[port]/[context]/[WSname]/[WSname].wsdl`

where

- *protocol* refers to the `protocol` attribute of the `<wsgen>` element of the `build.xml` Ant file used to build the Web service. By default this value is `http`.
- *host* refers to the hostname of the computer which hosts the Web service.
- *port* refers to the port number of the WebLogic Server instance that hosts the Web service.
- *context* refers to the `context` attribute of the `<wsgen>` element of the `build.xml` Ant file used to build the Web service.
- *WSname* is the name of the Web service:
  - For RPC-style Web services, the JNDI name of the stateless session EJB that implements the Web service.

    For example, if the `bean` attribute in the `build.xml` file specifies `statelessSession`, and the `weblogic-ejb-jar.xml` contains the following entry:

```
<weblogic-enterprise-bean>
  <ejb-name>statelessSession</ejb-name>
  <jndi-name>statelessSession.WeatherHome</jndi-name>
</weblogic-enterprise-bean>
```

then the `WSname` value is `statelessSession.WeatherHome`.

- For message-style Web services, the name of the Web service is specified by the `name` attribute of the `messageservice` element that defines the Web service in the `build.xml` file.

For example, using the sample `build.xml` file listed in "Invoking the WebLogic Web Services Home Page" on page 3-4, the URL to access the WSDL for the RPC-style Web service is:

```
http://www.myHost.com:7001/myContext/statelessSession.WeatherHome/statelessSess
ion.WeatherHome.wsdl
```

Similarly, the URLs to access the WSDL for the two message-style Web services are:

```
http://www.myHost.com:7001/myContext/sendMsgWS/sendMsgWS.wsdl
http://www.myHost.com:7001/myContext/receiveMsgWS/receiveMsgWS.wsdl
```

Creating a Client to Invoke an RPC-Style WebLogic Web Service

This section describes how to invoke an RPC-style Web service from two types of clients: Java and Microsoft SOAP ToolKit.

The examples in this section invoke an RPC-style Web service that is based on the Trader stateless session EJB described in the `examples.ejb.basic.statelessSession` WebLogic Server example.

Writing a Java Client

Creating a Java client application to invoke a WebLogic Web Service is simple because almost all of the Java code you need is provided by WebLogic Server and packaged in a Java client JAR file that you can download onto your client computer.

This section describes two modes of client applications: static and dynamic. Use a static client if you have the Java interfaces of the EJB and JavaBean parameters and return types, and want to use them directly in your client Java code. Use a dynamic client if you do not have the interfaces.

Writing a Static Java Client

The following example shows a simple static Java client that invokes an RPC-style Web service based on the `examples.ejb.basic.statelessSession` EJB example in WebLogic Server.

The example uses the URL http://www.myhost.com:7001/myContext/statelessSession/statelessSession.wsdl to get the WSDL of the Web Service. For details on how to construct this URL and an example of the `build.xml` file used to create the RPC-style Web service, refer to "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7.

The procedure after the example discusses relevant sections of the example as part of the basic steps you must follow to create this client.

```
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

import examples.ejb.basic.statelessSession.Trader;
import examples.ejb.basic.statelessSession.TradeResult;

public class Client{

  public static void main( String[] arg ) throws Exception

    Properties h = new Properties();

    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");

    h.put("weblogic.soap.wsdl.interface",
         Trader.class.getName() );

    Context context = new InitialContext(h);

    Trader service = (Trader)context.lookup(
 "http://www.myHost.com:7001/myContext/statelessSession/statelessSession.wsdl"
);

    TradeResult result = (TradeResult)service.buy( "BEAS", 100 );
```

```
    System.out.print( result.getStockSymbol() );
    System.out.print( ":" );
    System.out.println( result.getNumberTraded() );
  }
}
```

The Java code to statically invoke a WebLogic Web Service is similar to remote method invocation (RMI) client code that invokes EJBs. The main differences are:

- You do not need to look up and invoke the Home interface of the service.
- The Web service client uses a SOAP-specific INITIAL_CONTEXT_FACTORY.
- The Web service client specifies the interface in the parameters to the INITIAL_CONTEXT_FACTORY.

Follow these steps to create a static Java client that invokes an RPC-style WebLogic Web Service:

1. Get the Java client JAR file from the WebLogic Server hosting the WebLogic Web Service.

   For detailed information on this step, refer to "Downloading the Java Client JAR File from the Web Services Home Page" on page 3-6.

2. Add the Java client JAR file to your CLASSPATH on your client computer.

3. Create the client Java program. The following steps point out the Web service-specific parts of the Java code:

   a. Within the `main` method of your client application, add the following Java code to initialize the client so it can interact with the Web service:

```
Properties h = new Properties();

h.put(Context.INITIAL_CONTEXT_FACTORY,
     "weblogic.soap.http.SoapInitialContextFactory");

h.put("weblogic.soap.wsdl.interface",
     Trader.class.getName() );

Context context = new InitialContext(h);

Trader service = (Trader)context.lookup(
"http://www.myHost.com:7001/myContext/statelessSession/state
lessSession.wsdl" );
```

In the example, `Trader` is the public interface to the EJB. Refer to "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7 for details on how to construct the URL used in the `context.lookup()` method.

b. Invoke a Web service operation by executing a public method of the EJB, as shown in the following example:

```
TradeResult result = (TradeResult)service.buy( "BEAS", 100 );
```

The client executes the `buy()` method of the `Trader` EJB. The returned value is a `TraderResult` JavaBean object. To find out the public methods of the `Trader` EJB, either examine the returned WSDL of the Web service, or un-JAR the downloaded Java client JAR file and use the **javap** utility to list the methods of the `Trader` interface.

c. Use the `get` methods of the returned `TraderResult` JavaBean to get the returned results. To find out the methods of the `TraderResult` class, unJAR the Java client jar file and use the **javap** utility to list the methods of the `TraderResult` class.

```
System.out.print( result.getStockSymbol() );
System.out.print( ":" );
System.out.println( result.getNumberTraded() );
```

4. Compile and run the client Java program as usual.

Writing a Dynamic Java Client

The following example shows a simple dynamic Java client that invokes an RPC-style Web service based on the `examples.ejb.basic.statelessSession` EJB example in WebLogic Server.

The example uses the URL http://www.myhost.com:7001/myContext/statelessSession/statelessSession.wsdl to get the WSDL of the Web Service. For details on how to construct this URL and an example of the `build.xml` file used to create the RPC-style Web service, refer to "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7.

The procedure after the example discusses relevant sections of the example as part of the basic steps you must follow to create this client.

```
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import examples.ejb.basic.statelessSession.TradeResult;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;

public class DynamicClient{

  public static void main( String[] arg ) throws Exception{

    Properties h = new Properties();

    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");

      Context context = new InitialContext(h);

    WebServiceProxy proxy = (WebServiceProxy)context.lookup(

"http://www.myHost.com:7001/myContext/statelessSession/statelessSession.wsdl" );

    SoapMethod method = proxy.getMethod( "buy" );

    TradeResult result = (TradeResult)method.invoke(
         new Object[]{ "BEAS", new Integer(100) } );

    System.out.print( result.getStockSymbol() );
    System.out.print( ":" );
    System.out.println( result.getNumberTraded() );
  }
}
```

Follow these steps to create a dynamic Java client that invokes an RPC-style WebLogic Web Service:

1. Get the Java client JAR file from the WebLogic Server hosting the WebLogic Web Service.

   For detailed information on this step, refer to "Downloading the Java Client JAR File from the Web Services Home Page" on page 3-6.

2. Add the Java client JAR file to your CLASSPATH on your client computer.

3. Create the client Java program. The following steps point out the Web service-specific parts of the Java code:

   a. Within the `main` method of your client application, add the following Java code to initialize the client so it can interact with the Web service:

      ```
      Properties h = new Properties();
      ```

```
h.put(Context.INITIAL_CONTEXT_FACTORY,
     "weblogic.soap.http.SoapInitialContextFactory");

Context context = new InitialContext(h);

WebServiceProxy proxy = (WebServiceProxy)context.lookup(
"http://www.myHost.com:7001/myContext/statelessSession/state
lessSession.wsdl" );
```

In the example, the `context.lookup()` method returns a generic `WebServiceProxy` object rather than a specific `Trader` object; this makes the example more dynamic because `WebServiceProxy` can represent any EJB object. Refer to "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7 for details on how to construct the URL used in the `context.lookup()` method.

b. Invoke the Web service operation by executing a public method of the EJB, as shown in the following example:

```
SoapMethod method = proxy.getMethod( "buy" );

TradeResult result = (TradeResult)method.invoke(
     new Object[]{ "BEAS", new Integer(100) } );
```

The client indirectly executes the `buy()` method of the `Trader` EJB using the `invoke()` method. The returned value is a `TraderResult` JavaBean object. To find out the public methods of the `Trader` EJB, either examine the returned WSDL of the Web service or unJAR the downloaded Java client jar and use the **javap** utility to list the methods of the `Trader` interface.

c. Use the `get` methods of the returned `TraderResult` JavaBean to get the returned results. To find out the methods of the `TraderResult` class, unJAR the Java client jar file and use the **javap** utility to list the methods of the `TraderResult` class.

```
System.out.print( result.getStockSymbol() );
System.out.print( ":" );
System.out.println( result.getNumberTraded() );
```

4. Compile and run the client Java program as usual.

Writing a Microsoft SOAP Toolkit Client

You can invoke WebLogic Web Services from Microsoft Visual Basic applications by using the client-side components provided by the Microsoft SOAP ToolKit.

Note: WebLogic Server 6.1 supports only version 2.0sp2 of Microsoft SOAP ToolKit

The following sample Visual Basic code shows a simple example of invoking the WebLogic Web Service described by the `examples.webservices.rpc` example:

```
SET soapclient = CreateObject("MSSOAP.SoapClient")

Call soapclient.mssoapinit(

"http://myhost:7001/weather/statelessSession.WeatherHome/statelessSession.Weath
erHome.wsdl", "Weather", "WeatherPort")

wscript.echo soapclient.getTemp(94117)
```

To invoke a WebLogic Web Service from a Visual Basic application using the Microsoft SOAP ToolKit, follow these main steps:

1. Instantiate a `SoapClient` object in your Visual Basic application.
2. Initialize the `SoapClient` object by executing the `SoapClient.mssoapinit()` method, passing it the following parameters:
   - URL of the WSDL of the WebLogic Web Service. See "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7 for details on constructing this URL.
   - Name of the Web service, identified by the `name` attribute of the `service` element in the WSDL file.
   - Port of the Web service, identified by the `name` attribute of the `port` element in the WSDL file.

   After the `SoapClient` object is initialized, all the methods defined in the WSDL are dynamically bound to the `SoapClient` object.
3. Execute the WebLogic Web Service method.

Creating a Java Client to Invoke a Message-Style WebLogic Web Service

This section describes how to invoke message-style Web services from a Java client application.

Creating a Java client application to invoke a message-style WebLogic Web Service is simple because almost all of the Java code you need is provided by WebLogic Server and packaged in a Java client JAR file that you can download onto your client computer.

This section describes two types of Java clients: one that invokes a message-style Web service that sends data to WebLogic Server and one that invokes a message-style Web service that receives data. Both examples show how to create a dynamic Java client.

Note: The send and receive actions are from the perspective of the client application.

It is assumed that the two message-style Web services in the examples were assembled using the following `build.xml` file:

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
     <wsgen
       destpath="messageExample.ear"
       context="/msg"
       protocol="http" >
       <messageservices>
         <messageservice
           action="send"
           name="Sender"
           destination="examples.soap.msgService.MsgSend"
           destinationtype="topic"
           uri="/sendMsg"
           connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
         <messageservice
           action="receive"
           name="Receiver"
           destination="examples.soap.msgService.MsgReceive"
           destinationtype="topic"
           uri="/receiveMsg"
           connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
```

```
        </messageservices>
      </wsgen>
   </target>
</project>
```

The `build.xml` file shows two message-style Web services: one named `Sender` that client applications use to send data to a JMS topic with the JNDI name `examples.soap.msgService.MsgSend` and one named `Receiver` that client applications use to receive data from a JMS topic with the JNDI name `examples.soap.msgService.MsgReceive`. Both message-style Web services use the same ConnectionFactory to create the JMS connection: `examples.soap.msgService.MsgConnectionFactory`.

Sending Data to a Message-Style Web Service

This section describes how to create a dynamic Java client application that invokes a Web service to send data to WebLogic Server. For the sake of simplicity, the example sends a `String` data type that will contain the data.

Note: For a more complex example that shows how to send a `org.w3c.dom.Document`, `org.w3c.dom.DocumentFragment`, or `org.w3c.dom.Element` data type to the `send` method, see Appendix D, "Invoking Web Services Without Using the WSDL File."

Message-style Web services that send data to WebLogic Server define a single method called `send`; this is the only method you need to invoke from your Java client application. The `send` method takes a single parameter: the actual data. The data type can be anything you want: a `String` (used in the example), a DOM tree, an `InputStream`, etc. The data will eventually end up on the JMS destination you specify in the `build.xml` file used to assemble the Web service.

The example uses the URL `http://localhost:7001/msg/Sender/Sender.wsdl` to get the WSDL of the Web Service. For details on how to construct this URL, refer to "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7.

The procedure after the example discusses relevant sections of the example as part of the basic steps you follow to create this client.

```
package examples.soap;

import java.util.Properties;
import java.net.URL;
```

```
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;

public class ProducerClient{

  public static void main( String[] arg ) throws Exception{

    Properties h = new Properties();

    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");
    h.put("weblogic.soap.verbose", "true" );

    CodecFactory factory = CodecFactory.newInstance();
    factory.register( new SoapEncodingCodec() );
    h.put( "weblogic.soap.encoding.factory", factory );

    Context context = new InitialContext(h);

    WebServiceProxy proxy = (WebServiceProxy)context.lookup(
    "http://localhost:7001/msg/Sender/Sender.wsdl" );
    SoapMethod method = proxy.getMethod( "send" );

    String toSend = arg.length == 0 ? "No arg to send" : arg[0];
    Object result = method.invoke( new Object[]{ toSend } );

  }

}
```

Follow these steps to create a dynamic Java client that invokes a message-style WebLogic Web Service that sends data to WebLogic Server:

1. Get the Java client JAR file from the WebLogic Server hosting the WebLogic Web Service.

   For detailed information on this step, refer to “Downloading the Java Client JAR File from the Web Services Home Page” on page 3-6.

2. Add the Java client JAR file to your CLASSPATH on your client computer.

3. Create the client Java program. The following steps point out the Web service-specific parts of the Java code:

a. In the `main` method of your client application, create a `Properties` object and set some of the initial context properties:

```
Properties h = new Properties();

h.put(Context.INITIAL_CONTEXT_FACTORY,
     "weblogic.soap.http.SoapInitialContextFactory");
h.put("weblogic.soap.verbose", "true" );
```

b. Create a factory of encoding styles and register the SOAP encoding style:

```
CodecFactory factory = CodecFactory.newInstance();
factory.register( new SoapEncodingCodec() );
h.put( "weblogic.soap.encoding.factory", factory );
```

c. Create the initial context, use the WSDL to look up the Web service, then get the `send` method:

```
Context context = new InitialContext(h);

WebServiceProxy proxy = (WebServiceProxy)context.lookup(
"http://localhost:7001/msg/Sender/Sender.wsdl" );
SoapMethod method = proxy.getMethod( "send" );
```

d. Invoke the `send` method and send data to the Web service. In the example, the client application simply takes its first argument and sends it as a `String`; if the user does not specify an argument specified, then the client application sends the string `No arg to send`:

```
String toSend = arg.length == 0 ? "No arg to send" : arg[0];
Object result = method.invoke( new Object[]{ toSend } );
```

4. Compile and run the client Java program as usual.

Receiving Data From a Message-Style Web Service

This section describes how to create a dynamic Java client application that invokes a Web service to receive data from WebLogic Server.

Message-style Web services that receive data from WebLogic Server define a single method called `receive`; this is the only method you need to invoke from your Java client application. The `receive` method takes no input parameters. It returns a generic Java object that contains the data that the Web service got from the JMS destination you specify in the `build.xml` file used to assemble the Web service.

The example in this section uses the URL `http://localhost:7001/msg/Receiver/Receiver.wsdl` to get the WSDL of the Web Service. For details on how to construct this URL, refer to "URLs to Invoke WebLogic Web Services and Get the WSDL" on page 3-7.

The procedure after the example discusses relevant sections of the example as part of the basic steps you follow to create this client.

```
package examples.soap;

import java.util.Properties;
import java.net.URL;
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;

public class ConsumerClient{

  public static void main( String[] arg ) throws Exception{

    Properties h = new Properties();
    h.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.soap.http.SoapInitialContextFactory");
    h.put("weblogic.soap.verbose", "true" );

    CodecFactory factory = CodecFactory.newInstance();
    factory.register( new SoapEncodingCodec() );
    h.put( "weblogic.soap.encoding.factory", factory );

    Context context = new InitialContext(h);

    WebServiceProxy proxy = (WebServiceProxy)context.lookup(
    "http://localhost:7001/msg/Receiver/Receiver.wsdl" );
    SoapMethod method = proxy.getMethod( "receive" );

    while( true ){
      Object result = method.invoke( null );
      System.out.println( result );
    }
  }
}
```

Follow these steps to create a dynamic Java client that invokes a message-style WebLogic Web Service that receives data from WebLogic Server:

1. Get the Java client JAR file from the WebLogic Server hosting the WebLogic Web Service.

   For detailed information on this step, refer to “Downloading the Java Client JAR File from the Web Services Home Page” on page 3-6.

2. Add the Java client JAR file to your CLASSPATH on your client computer.

3. Create the client Java program. The following steps point out the Web service-specific parts of the Java code:

   a. In the `main` method of your client application, create a `Properties` object and set some of the initial context properties:

   ```
   Properties h = new Properties();
   h.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.soap.http.SoapInitialContextFactory");
   h.put("weblogic.soap.verbose", "true" );
   ```

   b. Create a factory of encoding styles and register the SOAP encoding style:

   ```
   CodecFactory factory = CodecFactory.newInstance();
   factory.register( new SoapEncodingCodec() );
   h.put( "weblogic.soap.encoding.factory", factory );
   ```

   c. Create the initial context, use the WSDL to look up the Web service , then get the `receive` method:

   ```
   Context context = new InitialContext(h);

   WebServiceProxy proxy = (WebServiceProxy)context.lookup(
   "http://localhost:7001/msg/Receiver/Receiver.wsdl" );
   SoapMethod method = proxy.getMethod( "receive" );
   ```

   d. Invoke the `receive` method to receive data from the Web service. In the example, the client application uses an infinite `while` loop to continuously invoke the `receive` method, in essence polling the JMS destination for messages. When the `receive` method returns data, the client application prints the result to the standard output:

   ```
   while( true ){
     Object result = method.invoke( null );
     System.out.println( result );
   ```

4. Compile and run the client Java program as usual.

Handling Exceptions from WebLogic Web Services

If an exception occurs while WebLogic Server is executing a Web service, the client application that invoked the Web service receives a run-time `weblogic.soap.SoapFault` exception that describes a standard SOAP fault.

The following types of exceptions in WebLogic Server could produce a run-time `SoapFault` exception in the client application:

- An exception from the stateless session EJB that implements an RPC-style Web service
- An exception from the SOAP servlets that handle the SOAP messages between the client application and WebLogic Web Services
- A JMS exception

If your client application receives a `SoapFault` exception, use the following methods of `weblogic.soap.SoapFault` to examine it:

- `getFaultCode()`—returns the SOAP faultcode.
- `getFaultString()`—returns the name of the class or interface that raised the exception in WebLogic Server. For example, if the stateless session EJB that comprises an RPC-style Web service raised an exception, the `getFaultString()` method returns the interface of this EJB.
- `printStackTrace()`—returns the stack trace of the exception.

The following excerpt from a Java client application shows an example of using `weblogic.soap.SoapFault` to examine any errors that occurred on WebLogic Server:

```
import weblogic.soap.SoapFault;

...

 try {
    TradeResult result = (TradeResult)method.invoke(
       new Object[]{ "BEAS", new Integer(100) } );
```

```
    System.out.print( result.getStockSymbol() );
    System.out.print( ":" );
    System.out.println( result.getNumberTraded() );
} catch (SoapFault fault){
    System.out.println( "Ooops, got a fault: " + fault );
    fault.printStackTrace();
  }
```

Initial Context Factory Properties for Invoking Web Services

The following table lists the Java properties you can set with the `Properties` object when you use the WebLogic-generated Java client JAR file in your Java client applications to invoke a WebLogic Web Service.

Note: The properties are passed to the initial context factory; these are not Java system properties.

Table 3-1 Initial Context Factory Properties for Invoking Web Services

Property	Description
`weblogic.soap.wsdl.interface`	Specifies the interface of the stateless session EJB upon which the Web service is based.
`weblogic.soap.verbose`	When set to `true`, the SOAP packet generated by the Java client to invoke a WebLogic Web Service is output to the client. Valid values are `true` and `false` (default).
`weblogic.soap.encoding.factory`	Specifies the CodecFactory that contains the encoders and decoders to convert between XML and Java data. Valid values are instances of `weblogic.soap.codec.CodecFactory`.
`java.naming.factory.initial`	Specifies the initial SOAP context factory. Valid values are instances of `weblogic.soap.http.SoapInitialContextFactory`

Table 3-1 Initial Context Factory Properties for Invoking Web Services

Property	Description
`java.naming.security.principal`	Specifies the user name when setting HTTP security.
`java.naming.security.credentials`	Specifies the user password when setting HTTP security.

Additional Classes Needed by Clients Invoking WebLogic Web Services

WebLogic Web Services support the following two encoding styles:

- http://schemas.xmlsoap.org/soap/encoding/
- http://xml.apache.org/xml-soap/literalxml

If your Java client application uses the SOAP encoding, then the Java client JAR file that you download from WebLogic Server includes all the classes you need to invoke a WebLogic Web Service.

However, if your client application uses the Literal XML encoding from Apache, then the Java client JAR file does not include all the files you need. The client JAR file is meant to be small, and adding all these classes to the JAR file would make it very large.

The following list shows some of the additional classes you might need to include:

- `weblogic.apache.Xerces.*`
- `weblogic.xml.jaxp.*`
- `org.w3c.dom.*`
- `org.w3c.sax.*`
- `javax.xml.parsers.*`

You can include these classes by either setting your CLASSPATH environment variable to their location when you run the client application or by using the `clientjar` element in the `build.xml` file when assembling the Web service using the `wsgen` Java Ant task.

To get the complete list of classes your client application needs, compile the application and then execute it with the `-verbose` flag, which will list all the classes it needs.

CHAPTER

4 Administering WebLogic Web Services

The following sections describe tasks for administering WebLogic Web Services:

Overview of Administering WebLogic Web Services

Once you have developed, assembled, and deployed a WebLogic Web Service, you can use the Administration Console to perform the following administrative task:

- View the Web services currently deployed on WebLogic Server.

Invoking the Administration Console

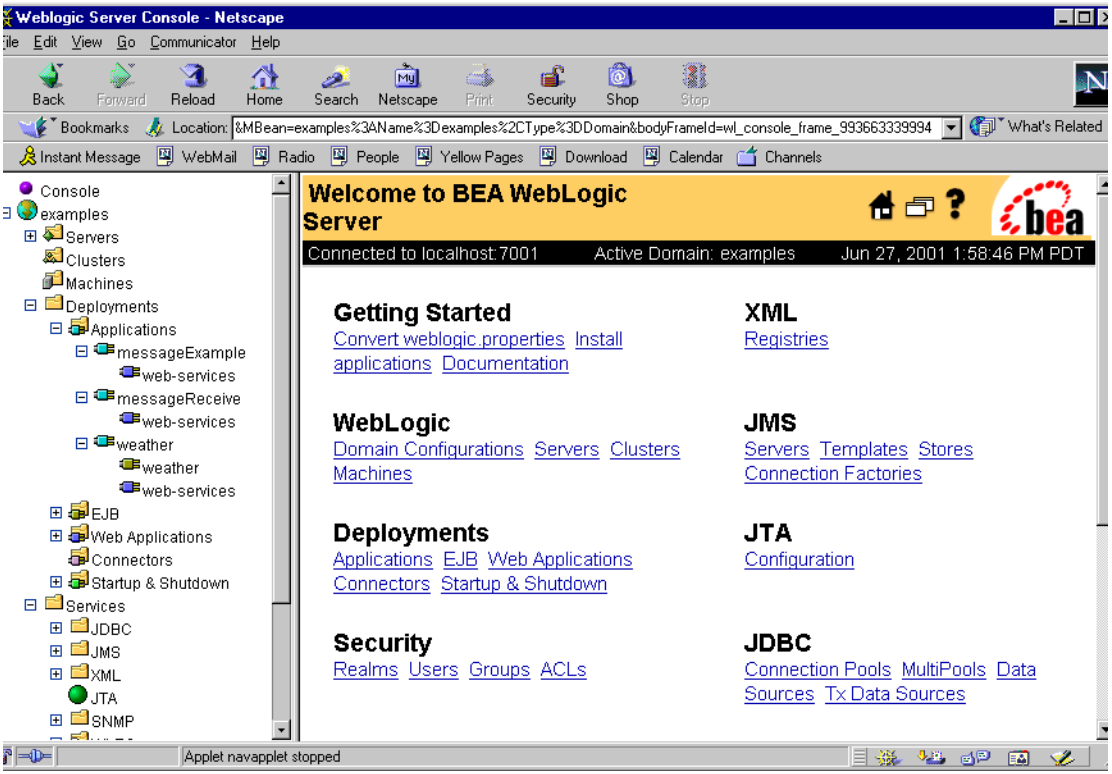
To invoke the Administration Console in your browser, enter the following URL:

```
http://host:port/console
```

where

- *host* refers to the computer on which WebLogic Administration server is running.
- *port* refers to the port number where WebLogic Administration server is listening for connection requests. The default port number for WebLogic Administration server is 7001.

The following figure shows the main Administration Console window.

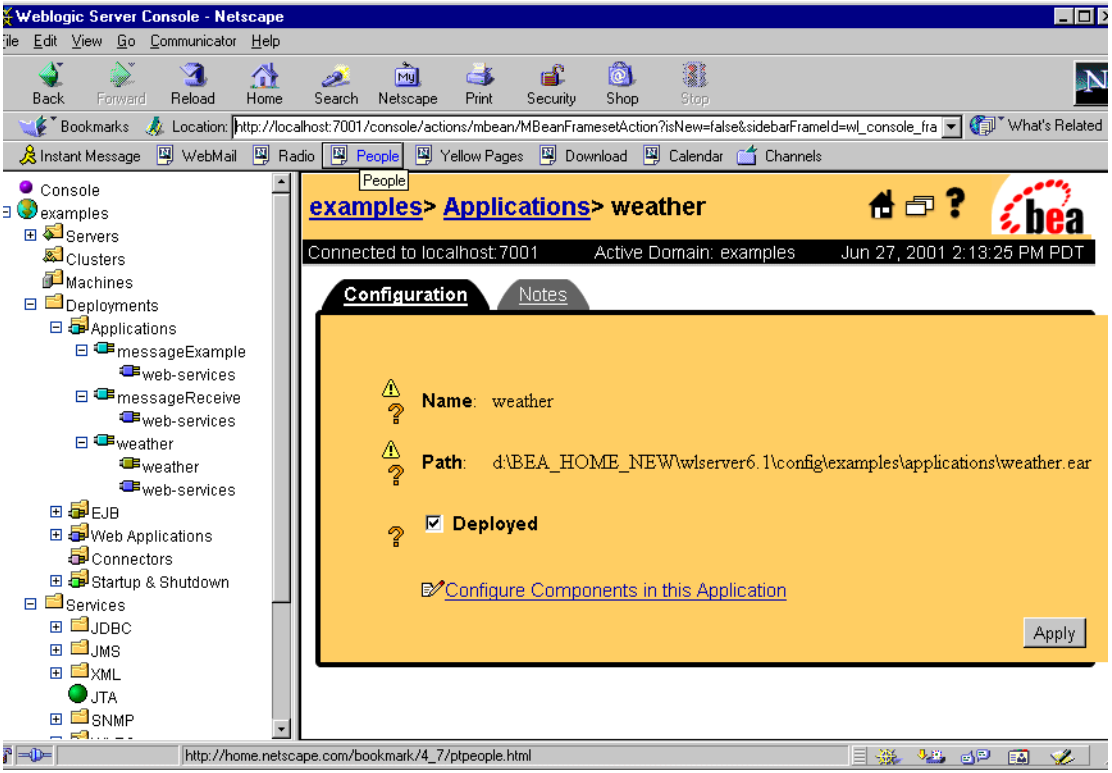


Viewing the Web Services Deployed on WebLogic Server

To view all the Web services that are deployed on WebLogic Server, and then view the properties of a particular Web service, follow these steps:

1. Start the WebLogic Administration server and invoke the Administration Console in your browser. See "Invoking the Administration Console" on page 4-1 for detailed information.
2. In the left pane, click to expand the Deployments node.
3. Click to expand the Applications node. A list of Enterprise applications appears below the node.
4. To determine which of the listed Enterprise applications is deployed as a Web service, follow these steps for each Enterprise application:
   a. Click to expand the Enterprise application. The list of components that make up the application, including Web applications and EJBs, appears below the name of the application.
   b. Look for a Web application component called `web-services`, which is the default name of the Web application that contains the SOAP servlets for Web services.

      The following figure shows three Enterprise applications: `messageExample`, `messageReceive`, and `weather`, each of which include a `web-services` Web application. This indicates that the three applications are deployed as Web services. The right pane displays information about the weather Web service.



c. If you find a Web application called `web-services`, right-click on it in the left pane and chose Edit Descriptor from the drop-down menu. The Deployment Descriptor Editor for the `web-services` Web application deployment descriptors appears in a new browser window.

d. In the left pane of the Deployment Descriptor Editor, see if the RPC Services node under the Web Services node contains an entry. If it does, then the Enterprise application is deployed as an RPC-style Web service. Similarly, if the Message Services node contains an entry, then the Enterprise application is deployed as a message-style Web service.

e. Click on the entry in either the Message Service or RPC Service node to view the properties of the Web service.

f. If you do not find a Web application called `web-services`, it is still possible that the Enterprise application is deployed as a Web service, but the Web application that contains the SOAP servlet has been named something other than the default `web-services`. In this case, you must check the deployment descriptors of each Web application contained in the Enterprise application to see if there are any entries under the Web services node, as described in Steps c through e of this procedure.

CHAPTER

5 Troubleshooting

The following sections describe troubleshooting topics related to WebLogic Web Services:

Turning on Verbose Mode

Use the `weblogic.soap.verbose` initial context factory property in your client application to print out the SOAP messages that pass between WebLogic Server and the client application, as well as any errors produced by WebLogic Server.

In the following example, a client application that invokes a WebLogic Web Service has the `weblogic.soap.verbose` initial context factory property set to true to enable verbose mode:

```
Properties h = new Properties();
h.put(Context.INITIAL_CONTEXT_FACTORY,
      "weblogic.soap.http.SoapInitialContextFactory");
h.put("weblogic.soap.verbose", "true" );
```

The output is printed to the shell from which you execute the client application. Use this output to troubleshoot problems you encounter while invoking a Web service.

java.io.FileNotFoundException

Problem

Your client application, while attempting to invoke a WebLogic Web Service, throws the `java.io.FileNotFoundException` exception.

Explanation

The problem could be caused by the following:

- The WebLogic Web Service is not currently deployed on WebLogic Server.
- The Web application that contains the SOAP servlets is not targeted for the correct instance of WebLogic Server.
- If you are invoking an RPC-style Web service, the stateless session EJB that implements the Web service is not targeted for the correct instance of WebLogic Server.
- If you are invoking a message-style Web service, the JMS Server or Connection Factory is not targeted for the correct instance of WebLogic Server.

The output from a `java.io.FileNotFoundException` error might look like the following:

```
Exception in thread "main" javax.naming.NamingException: i/o failed
java.io.FileNotFoundException:
http://localhost:7001/weather/statelessSession.WeatherHome/statelessSession.Wea
therHome.wsdl.
Root exception is java.io.FileNotFoundException:
http://localhost:7001/weather/statelessSession.WeatherHome/statelessSession.Wea
therHome.wsdl
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.ja
va:574)
        at weblogic.soap.WebServiceProxy.getXMLStream(WebServiceProxy.java:553)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:172)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
```

```
at javax.naming.InitialContext.lookup(InitialContext.java:350)
at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

Suggested Solution

If this error occurs when you attempt to invoke a WebLogic Web Service, follow these steps to ensure that the Web service and its components are correctly deployed and targeted:

1. Invoke the Administration Console in your browser. See "Invoking the Administration Console" on page 4-1 for details.
2. In the left pane, click to expand the Applications node under the Deployments node.
3. Click on the Enterprise Application that corresponds to the WebLogic Web Service that you are attempting to invoke.
4. In the right pane, if the Deployed check box is not selected, select it and click the Apply button.
5. In the left pane, under the Enterprise application that corresponds to your Web service, click the Web application that contains the SOAP servlets. The default name of this Web application is `web-services`.
6. In the right pane, select the Targets tab.
7. If it is not already there, move the name of the WebLogic Server instance on which the Web application should be running from the Available to the Chosen list box. Click Apply.
8. *If you are attempting to invoke an RPC-style WebLogic Web Service*, follow these steps:
   a. In the left pane, under the Enterprise application that corresponds to your Web service, click on the name of the EJB jar file.
   b. In the right pane, select the Targets tab.
   c. If it is not already there, move the name of the WebLogic Server instance on which the EJB should be running from the Available to the Chosen list box and click Apply.

If you are attempting to invoke a message-style Web service, follow these steps:

a. In the left pane, click to expand the JMS node under the Services node.

b. Click to expand the Connection Factories node.

c. In the right pane, click the name of the JMS Connection Factory that you configured for the message-style Web service that you are trying to invoke.

d. Select the Targets tab.

e. If it is not already there, move the name of the WebLogic Server instance for which the Connection Factory should be targeted from the Available to the Chosen list box. Click Apply.

f. In the right pane, click to expand the Servers node under the JMS node.

g. Click the name of the JMS server which your message-style Web service is using.

h. In the right pane, select the Targets tab.

i. If it is not already there, move the name of the WebLogic Server for which the JMS Server is targeted from the Available to the Chosen list box and click Apply.

Unable to Parse Exception

Problem

The client application receives an “Unable to Parse” exception.

Explanation

The client API used to invoke Web Services uses the WebLogic FastParser to parse the WSDL and SOAP messages from the invoked Web service. If the WSDL or SOAP message from the Web service is not well-formed, the client application might receive an Unable to Parse error.

For example, if a Web service's WSDL file is not well-formed because of an element specifying two attributes with the same name, the client application produces the following error:

```
Exception in thread "main" javax.naming.NamingException: unable to parse
org.xml.sax.SAXException: Attributes may not have the same name, more than
one xmlns:tns.
Root exception is org.xml.sax.SAXException: Attributes may not have the same name,
more than one xmlns:tns
        at
weblogic.xml.babel.baseparser.SAXElementFactory.createAttributes(SAXEleme
ntFactory.java:42)
        at
weblogic.xml.babel.baseparser.StreamElementFactory.createStartElementEven
t(StreamElementFactory.java:39)
        at
weblogic.xml.babel.parsers.StreamParser.streamParseSome(StreamParser.java:113)
        at
weblogic.xml.babel.parsers.BabelXMLEventStream.parseSome(BabelXMLEventStr
eam.java:46)
        at
weblogicx.xml.stream.XMLEventStreamBase.hasNext(XMLEventStreamBase.java:135)
        at
weblogicx.xml.stream.XMLEventStreamBase.hasStartElement(XMLEventStreamBase.java
:241)
        at
weblogicx.xml.stream.XMLEventStreamBase.startElement(XMLEventStreamBase.java:23
4)

        at weblogic.soap.wsdl.binding.Definition.parse(Definition.java:121)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:171)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
        at javax.naming.InitialContext.lookup(InitialContext.java:350)
        at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

Suggested Solution

Contact the Web service provider to ensure that the Web service produces well-formed WSDL and SOAP messages.

java.lang.NullPointerException

Problem

Your client application gets a `java.lang.NullPointerException` error in the methods in the `weblogic.soap.wsdl.binding.*` classes.

Explanation

One possible explanation is that the Web service's WSDL or SOAP messages, although possibly well-formed, are not valid.

For example, if the Web service's WSDL references an `inputs` element rather than the correct `input`, then the client application produces the following error:

```
was expecting 'input|output' but got:inputs
was expecting 'operation|input|output' but got:inputs
Exception in thread "main" java.lang.NullPointerException
        at weblogic.soap.wsdl.binding.Operation.getInputName(Operation.java:35)
        at
weblogic.soap.wsdl.binding.BindingOperation.populate(BindingOperation.jav
a:49)
        at weblogic.soap.wsdl.binding.Binding.populate(Binding.java:48)
        at weblogic.soap.wsdl.binding.Definition.populate(Definition.java:116)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:174)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
        at javax.naming.InitialContext.lookup(InitialContext.java:350)
        at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

Suggested Solution

Contact the Web service host and ensure that the Web service produces valid WSDL and SOAP messages.

java.net.ConnectException

Problem

Your client application gets a `java.net.ConnectException`.

Explanation

One possible explanation is that the Web service is unreachable. In particular:

- If the client application is attempting to invoke a WebLogic Web Service, the application receives a `Connection refused` error if WebLogic Server is not currently running.
- If the client application is attempting to invoke a non-WebLogic Web Service, the application receives an `Operation timed out` error after a few minutes if the host is unreachable for any reason.

For example, if the client application attempts to invoke a WebLogic Web Service from a WebLogic Server instance that is currently not running, the application receives the following error:

```
Exception in thread "main" javax.naming.NamingException: i/o failed
java.net.ConnectException: Connection refused: connect.
Root exception is java.net.ConnectException: Connection refused: connect
        at java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.net.PlainSocketImpl.doConnect(FancyJulietImpl.java:320)
        at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:133)
        at java.net.PlainSocketImpl.connect(FancySchmancyBeverleyImpl.java:120)
        at java.net.Socket.<init>(Socket.java:273)
        at java.net.Socket.<init>(Socket.java:100)
```

```
        at sun.net.NetworkClient.doConnect(NetworkClient.java:50)
        at sun.net.www.http.HttpClient.openServer(HttpClient.java:331)
        at sun.net.www.http.HttpClient.openServer(HttpClient.java:517)
        at sun.net.www.http.HttpClient.<init>(HttpClient.java:267)
        at sun.net.www.http.HttpClient.<init>(HttpClient.java:277)
        at sun.net.www.http.HttpClient.New(HttpClient.java:289)
        at
sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:408)
        at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.ja
va:501)
        at weblogic.soap.WebServiceProxy.getXMLStream(WebServiceProxy.java:553)
        at weblogic.soap.WebServiceProxy.getServiceAt(WebServiceProxy.java:172)
        at weblogic.soap.http.SoapContext.lookup(SoapContext.java:64)
        at javax.naming.InitialContext.lookup(InitialContext.java:350)
        at examples.webservices.rpc.javaClient.Client.main(Client.java:34)
```

Suggested Solution

Either restart WebLogic Server, or contact the Web service host and ensure that the Web service is reachable.

For information about starting WebLogic Server, see *WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/startstop.html.

CHAPTER

6 Interoperability

The following sections describe interoperability issues that surfaced when testing WebLogic Web services with other clients during the Round 2 SOAP Interoperability Tests :

.NET Client Interoperating With a 6.1 WebLogic Web Service

When a .NET client invokes the array-based methods of the Round 2 SOAP Interoperability Tests on a 6.1 WebLogic Web service, the returned data does not contain any of the array elements.

The array-based methods are:

- `echoStringArray`
- `echoIntegerArray`
- `echoFloatArray`
- `echoStructArray`

7.X WebLogic Client Interoperating with a 6.1 WebLogic Web Service

When a client application that uses the stubs created by the 7.0 `clientgen` Ant task invokes the `echoStructArray` method of the Round 2 SOAP Interoperability Tests running on a 6.1 WebLogic Web service, the returned data does not contain the correct array elements.

A Specifications Supported by WebLogic Web Services

The following sections describe the specifications supported by WebLogic Web Services:

- SOAP 1.1 Specification
- SOAP Messages With Attachments Specification
- Web Services Description Language (WSDL) 1.1 Specification

SOAP 1.1 Specification

Simple Object Access Protocol (SOAP) is a lightweight XML-based protocol for exchanging information in a decentralized, distributed environment. The protocol consists of three parts: an envelope that contains a message, a description of the message, and how to process it; a set of encoding rules for expressing instances of application-defined data types; and a convention for representing remote procedure calls and responses.

The SOAP 1.1 specification is available at http://www.w3.org/TR/SOAP.

SOAP Messages With Attachments Specification

A SOAP message may need to reference an attached file, often in binary format, such as an image or spreadsheet file. The SOAP Messages with Attachments specification describes a standard way to associate a SOAP message with one or more attachments in their native format in a multipart MIME structure for transport.

Note: WebLogic Web Services currently ignore the actual attachment of a SOAP with attachments message.

The SOAP Messages with Attachment specification is available at http://www.w3.org/TR/SOAP-attachments.

Web Services Description Language (WSDL) 1.1 Specification

WSDL is an XML-based language that describes Web services. WSDL defines Web services as a set of endpoints operating on messages; these message contain either message-style or RPC-style information. The operations and messages are described abstractly in WSDL, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow the description of endpoints and their associated messages regardless of what message formats or network protocols are used to communicate, however, the only bindings described in the specification describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST, and MIME.

Note: WebLogic Server supports only SOAP 1.1 bindings.

The WSDL 1.1 Specification is available at http://www.w3.org/TR/wsdl.

B build.xml Elements and Attributes

The `build.xml` file contains information that the `wsgen` Java Ant task uses to assemble Web services into Enterprise Application archive (`*.ear`) files.

The following sections provide an example `build.xml` file and describe its elements and attributes:

The `build.xml` file consists of a series of XML elements. Java Ant defines a variety of elements you can include in this file, such as `project` and `target`. This Appendix, however, describes only those elements that are part of the WebLogic-specific `wsgen` Java Ant task. For general information about Java Ant, see http://jakarta.apache.org/ant/index.html.

Note: The Java Ant utility included in WebLogic Server uses the `ant` (UNIX) or `ant.bat` (Windows) configuration files in the *`BEA_HOME`*`\bin` directory when setting the ANTCLASSPATH variable, where *`BEA_HOME`* is the directory in which WebLogic Server is installed. If you need to update the ANTCLASSPATH variable, make the appropriate changes to these files.

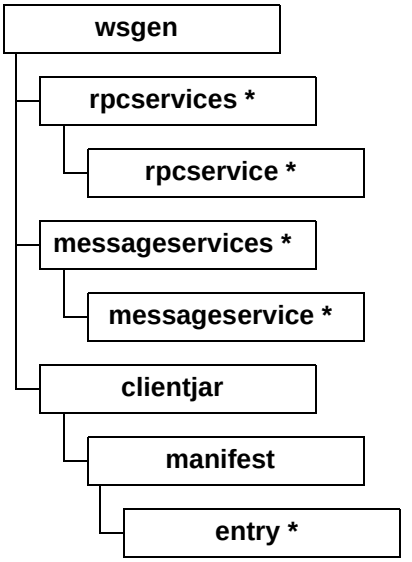
Example of a build.xml File

The following example shows a simple `build.xml` file used to assemble one RPC-style Web service and two message-style Web services:

```
<project name="myProject" default="wsgen">
   <target name="wsgen">
      <wsgen
             destpath="myWebService.ear"
             context="/myContext"
             protocol="http">
         <rpcservices path="myEJB.jar">
            <rpcservice
               bean="statelessSession"
               uri="/rpc_URI"/>
         </rpcservices>
         <messageservices>
           <messageservice
             name="sendMsgWS"
             action="send"
             destination="examples.soap.msgService.MsgSend"
             destinationtype="topic"
             uri="/sendMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
          <messageservice
             name="receiveMsgWS"
             action="receive"
             destination="examples.soap.msgService.MsgReceive"
             destinationtype="topic"
             uri="/receiveMsg"
             connectionfactory="examples.soap.msgService.MsgConnectionFactory"/>
      </messageservices>
      </wsgen>
   </target>
</project>
```

build.xml Hierarchy Diagram

The following diagram shows all the possible sub-elements of the `wsgen` element in the `build.xml` file, along with the element hierarchy. An asterisk (*) indicates that the element can be specified zero or more times.



Description of Elements and Attributes

The following sections describe `build.xml` elements and attributes.

wsgen

The wsgen element is the name of the Ant task in the build.xml file. Its attributes specify information that is common to all Web services described in the file.

This element contains the following attributes.

Table 6-1 wsgen Attributes

Attribute	Description	Required?
basepath	Location of the input Enterprise Application archive file (*.ear) or exploded directory that contains the EJB jar files for the EJB that implements the RPC-style Web services, as well as any supporting EJBs. Be sure to specify the full pathname of the file or directory if it is not located in the same directory as the build.xml file. Default value is null.	No.
destpath	Type and location of the output Enterprise Application archive. To create an actual Enterprise Application archive file (*.ear), specify the .ear suffix; to create an exploded Enterprise Application directory, specify a directory name. Specify the full pathname of the file or directory if you do not want the Ant task to create the archive in the local directory.	Yes.
context	Context root of the Web services. This value is part of the URL used to access the Web service.	Yes.
protocol	Protocol by which clients access the Web service. There are two possible values: http or https. The default value is http.	No.
host	Name of the host that is running the WebLogic Server instance that is hosting the Web service; for example, www.bea.com. If you do not specify this attribute, the host in the WSDL JSP is generated from the *hostname* section of the URL used to retrieve the WSDL.	No.

Table 6-1 wsgen Attributes (Continued)

`port`	Port number of WebLogic Server. Default value is 7001. If you do not specify this attribute, the port in the WSDL JSP is generated from the *port* section of the URL used to retrieve the WSDL.	No.
`webapp`	URI that specifies the path to a Web Application module used to expose a Web service. Default value is `web-services.war`.	No.
`classpath`	Semicolon-separated list of directories or JAR files that contain Java classes (such as utility classes) needed by the stateless session EJB that implements an RPC-style Web service.	No.

rpcservices

The `rpcservices` element specifies an EJB archive that contains the stateless session EJB that implements the RPC-style Web service, as well as any supporting EJBs.

This element can have any number of `rpcservice` sub-elements that describe each individual RPC-style Web service.

This element contains the following attributes.

Table 6-2 rpcservices Attributes

Attribute	Description	Required?
`module`	If the `basepath` attribute of the `wsgen` element is set, this attribute specifies the URI of the Enterprise Application module that corresponds to an EJB archive contained by the Enterprise Application archive.	Only if the `basepath` attribute of the `wsgen` element is set.
`path`	If the `basepath` attribute of the `wsgen` element is not set, this attribute specifies the location of an existing EJB archive that contains the EJBs, either archive as a `*.jar` file or as an exploded directory.	Only if the `basepath` attribute of the `wsgen` element is *not* set.

rpcservice

The `rpcservice` element specifies a specific RPC-style Web service.

This element does not have any sub-elements.

This element contains the following attributes.

Table 6-3 rpcservice Attributes

Attribute	Description	Required?
`bean`	Name of the stateless session EJB that implements the RPC-style Web service. This name corresponds to the `ejb-name` element in the `ejb-jar.xml` file of the EJB archive in which the EJB is contained. The path to the EJB archive is specified in the parent `rpcservices` element.	Yes.

`uri`	Part of the URL used by clients to invoke the Web service.
	The full URL to access the Web service is:
	`[`*`protocol`*`]://[`*`host`*`]:[`*`port`*`][`*`context`*`][`*`uri`*`]`
	where
	■ *`protocol`* refers to the `protocol` attribute of the `wsgen` element.
	■ *`host`* refers to the hostname of the computer upon which the WebLogic Server hosting the service is running.
	■ *port* refers to the port of WebLogic Server.
	■ *`context`* refers to the `context` attribute of the `wsgen` element.
	■ *`uri`* refers to this attribute.
	For example, the URL that accesses the RPC-style Web service in the example in "Example of a build.xml File" on page B-2 is:
	`http://www.myHost.com:7001/myContext/rpc_URI`
		Yes.

messageservices

The `messageservices` element is a container for any number of `messageservice` sub-elements.

This element does not have any attributes.

messageservice

The `messageservice` element describes a specific message-style Web service by specifying a JMS destination that will receive or send messages.

This element does not have any sub-elements.

This element contains the following attributes.

Table 6-4 messageservice Attributes

Attribute	Description	Required?
`name`	Name of the message-style Web service.	Yes.
`destination`	JNDI name of a JMS topic or queue.	Yes.
`destinationtype`	Type of JMS destination. Values: `topic` or `queue`.	Yes.
`action`	Specifies whether the client that invokes this message-style Web service sends or receives messages to or from the JMS destination. Values: `send` or `receive`. Specify `send` if the client sends messages to the JMS destination and `receive` if the client receives messages from the JMS destination.	Yes.
`connectionfactory`	JNDI name of the ConnectionFactory used to create a connection to the JMS destination.	Yes.

Table 6-4 messageservice Attributes (Continued)

`uri`	Part of the URL used by clients to invoke the Web service. The full URL to access the Web service is: [*protocol*]://[*host*]:[*port*][*context*][*uri*] where ■ *protocol* refers to the `protocol` attribute of the `wsgen` element. ■ *`host`* refers to the hostname of the computer upon which the WebLogic Server hosting the service is running. ■ *port* refers to the port of WebLogic Server. ■ *context* refers to the `context` attribute of the `wsgen` element. ■ *uri* refers to this attribute. For example, the URL that accesses the first message-style Web service in the example in "Example of a build.xml File" on page B-2 is: `http://www.myHost.com:7001/myContext/sendMsg`	Yes.

clientjar

Use the `clientjar` element to specify the name for the generated Java client jar file. You can also use it specify other arbitrary files that you want to add to the generated Java client jar file.

This element can have one sub-element: `manifest`, as well as many `filesets` and `zipfilesets` elements. The `filesets` and `zipfilesets` elements are generic Ant elements, rather than `wsgen`-specific elements; use them to specify additional files that should be included in the Java client JAR file.

This element contains the following attributes.

Table 6-5 clientjar Attributes

Attribute	Description	Required?
`path`	URI for the generated Java client JAR file that contains all the Java classes and interfaces needed to invoke the Web services. Default value is `client.jar`.	No.

manifest

The `manifest` element is a container for additional header entries to the manifest file (MANIFEST.MF) included in the generated Java client JAR file.

This element can have any number of `entry` sub-elements that describe the additional headers to the manifest file.

This element does not have any attributes.

entry

The `entry` element specifies the name and value of an additional header to the manifest file (MANIFEST.MF) included in the generated Java client JAR file.

This element does not have any sub-elements.

This element contains the following attributes.

Table 6-6 entry Attributes

Attribute	Description	Required?
`name`	Name of the additional header that will appear in the manifest file (MANIFEST.MF) of the generated Java client JAR file.	Yes.
`value`	Value of the additional header that will appear in the manifest file (MANIFEST.MF) of the generated Java client JAR file.	Yes.

C Manually Assembling the Web Services Archive File

The following sections describe how to assemble a Web service manually into an Enterprise Application `*.ear` archive file:

- Before You Begin
- Description of the Web Services Archive File
- Assembling an RPC-Style Web Service Archive File Manually
- Assembling a Message-Style Web Service Archive File Manually
- Creating the client.jar File Manually

Before You Begin

WebLogic Web Services are packaged as standard J2EE Enterprise application archive files (`*.ear`). Assembling a WebLogic Web Service archive file manually can be complicated. For this reason, BEA highly recommends that you use the `wsgen` Java Ant task to create an initial `*.ear` file. Then, if needed, you can customize the components contained within the archive file for your specific application. For details on using `wsgen`, see "Assembling a WebLogic Web Service" on page 2-19.

You might need to manually create or edit the Enterprise application archive file if:

- you want to integrate the archive with a J2EE deployment tool.
- you need to perform advanced configuration tasks on components of the archive that are not available through the `wsgen` Ant task. These tasks include securing the SOAP servlets, securing the EJB, and so on.
- you want to change the default naming conventions and directories that the `wsgen` Ant task uses.

The following procedures shows how to create an Enterprise application archive similar to the one that the `wsgen` Java Ant task creates. If you follow the naming conventions exactly, the instructions in other chapters of this guide that describe how to access the WSDL of a Web service, the client.jar file, etc, will continue to work correctly.

Description of the Web Services Archive File

The Enterprise application archive contains the following components:

- A Web application, packaged in a `*.war` file that contains, among other items:
  - An HTML Web page that corresponds to the Web Services Home Page that lists all the Web services packaged in this Enterprise application archive.
  - For each Web service, an HTML Web page that includes links to the WSDL and client JAR file of the Web service.
  - For each Web service, a WSDL JSP that returns the WSDL.
  - The `web.xml` and `weblogic.xml` deployment descriptor files that contain Web services-specific information, such as references to the SOAP servlets that process the SOAP requests from the client.
- The stateless session EJB `*.jar` file (for RPC-style Web services).
- Other supporting EJB `*.jar` files.

Assembling an RPC-Style Web Service Archive File Manually

This section describes how to assemble an RPC-style Web service manually into an Enterprise application `*.ear` file that can be deployed on WebLogic Server.

Note: It is assumed that you have already created the stateless session EJB which implements the RPC-style Web service and assembled it into a `*.jar` EJB archive file. For detailed information about programming and assembling stateless session EJBs, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html.

To assemble an RPC-style Web service archive file manually, follow these steps:

1. Create a temporary staging directory for assembling the Web application component. You can name this directory anything you want.
2. Set up your shell environment.

   On Windows NT, execute the `setEnv.cmd` command, located in the directory *`BEA_HOME`*`\config\`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in the directory *`BEA_HOME`*`/config/`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.
3. Execute the following command to automatically generate initial `web.xml` and `weblogic.xml` deployment descriptors in the `WEB-INF` subdirectory:

   ```
   java weblogic.ant.taskdefs.war.DDInit staging-dir
   ```

   where *`staging-dir`* refers to the staging directory.

   For more information on the Java-based `DDInit` utility for generating deployment descriptors, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs61/programming/packaging.html#pack004.
4. Edit the `WEB-INF/web.xml` file, adding WebLogic Web Services information, such as references to the SOAP servlets. For details, see "Updating the web.xml File for RPC-Style Web Services" in this appendix.

5. Edit the `WEB-INF/weblogic.xml` file, adding WebLogic Web Services information. For details, see "Updating the weblogic.xml File for RPC-Style Web Services" in this appendix.
6. In the main staging directory, create a sub-directory with the same name as the JNDI name of your stateless session EJB.

   The JNDI name of your EJB corresponds to the `jndi-name` element in the `weblogic-ejb-jar.xml` deployment descriptor file for your EJB.

   Note: Using the JNDI name is the `wsgen` Ant task naming convention, which you do not have to follow.
7. In the *`jndi-name`* subdirectory, create the WSDL JSP by running the following utility and redirecting the output to a file:

```
java weblogic.soap.wsdl.Remote2WSDL EJB_interface path -protocol protocol > 
wsdl.jsp
```

where

- *`EJB_interface`* refers to the fully qualified class name of the Remote interface of your stateless session EJB.
- *path* is either *context* or *context/jndi-name*, where *context* refers to the `context-root` element of the Web application in the `application.xml` file (to be created in a later step).
- *protocol* is either `http` or `https`.

Note: This generated WSDL JSP dynamically sets the host and port of the WebLogic Server upon which the Web service is currently running. This is typically the type of WSDL file you want in your Web service. If, however, you want to statically specify the host and port in the WSDL file, edit the `soap:address` element in the WSDL JSP, replacing the text `<%= request.getServerName() %>:<%= request.getServerPort() %>` with hard-coded host and port values.

8. In the *`jndi-name`* subdirectory, create an `index.html` file that contains links to the WSDL JSP you created in the preceding step and the client JAR file that you will create in a later step. The following example shows a simple `index.html` file:

```
<html>
<body>
<h3>jndi-name</h3>
<ul>
```

```
<li><a href='wsdl.jsp'>WSDL</a></li>
<li><a href='../client.jar'>client.jar</li>
</ul>
</body>
</html>
```

9. Create a `client.jar` file in the main staging directory. For details on creating this file, refer to "Creating the client.jar File Manually" in this appendix.

   Note: This step is optional. You only need to create a `client.jar` file if you are going to use a Java client application to invoke the Web service.

10. Create an `index.html` file in the main staging directory that lists the Web service in this Enterprise application archive and links to its `index.html` file that you created in a previous step. The following example shows a simple `index.html` file:

    ```
    <html>
    <body>
    <h3>RPC-Style Web Services</h3>
    <ul>
    <li><a href='/context/jndi-name/index.html'>jndi-name</a></li>
    </ul>
    </body>
    </html>
    ```

    In the example, *`context`* refers to the `context-root` element of the Web application in the `application.xml` file (to be created in a later step) and *`jndi-name`* refers to the name of sub-directory that contains the WSDL file you created in the previous step.

11. Create the Web application archive (`*.war` file) using a jar command such as:

    `jar cvf` *`web-app-name`*`.war -C` *`staging-dir`* `.`

    Note: The `wsgen` Java ant task assigns the default name `web-services.war` to the Web application `*.war` file. You do not have to follow this naming convention.

12. Create a second temporary staging directory for assembling the Enterprise application. You can name this directory anything you want.

13. Copy your stateless session EJB `*.jar` file into the second staging directory.

14. Copy the Web application archive `*.war` file you created in a previous step into the second staging directory.

15. Execute the following command to automatically generate an initial `application.xml` deployment descriptor in the `META-INF` subdirectory:

    ```
    java weblogic.ant.taskdefs.ear.DDInit staging-dir
    ```

    where *`staging-dir`* refers to the second staging directory.

    For more information on the Java-based `DDInit` utility for generating deployment descriptors, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs61/programming/packaging.html#pack004.

16. Edit the `META-INF/application.xml` file, adding WebLogic Web Services information. For details, see "Updating the application.xml File for RPC-Style Web Services" in this appendix.

17. Create the Enterprise Archive (`.ear` file) for the application, using a `jar` command such as:

    ```
    jar cvf application.ear -C staging-dir .
    ```

    You can deploy the resulting `.ear` file as a WebLogic Web Service through the Administration Console or the `weblogic.deploy` command-line utility.

Updating the web.xml File for RPC-Style Web Services

This section describes the elements you must update or add to the `web.xml` deployment descriptor for the Web application that references the SOAP servlets in an RPC-style WebLogic Web Services archive file. For the complete example of a `web.xml` deployment descriptor, see the last example in this section.

It is assumed that you have a basic understanding of Web applications and their deployment descriptors. For more information, see *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html.

To update a `web.xml` file for RPC-style Web services, add the following elements:

- A `<servlet>` element that references the SOAP servlet that delegates RPC-style SOAP requests to the EJB. Set the `<servlet-class>` element to `weblogic.soap.server.servlet.StatelessBeanAdapter`. The servlet takes one `<init-param>`: a reference to the stateless session EJB which comprises the RPC-style Web service. The following example shows a `<servlet>` entry for the SOAP servlet:

```
<servlet>
  <servlet-name>statelessSession.WeatherHome</servlet-name>
  <servlet-class>
       weblogic.soap.server.servlet.StatelessBeanAdapter
  </servlet-class>
  <init-param>
    <param-name>ejb-ref</param-name>
    <param-value>statelessSession.WeatherHome</param-value>
  </init-param>
</servlet>
```

- A `<servlet>` element that references the SOAP servlet that handles all SOAP faults. Set the `<servlet-class>` element to `weblogic.soap.server.servlet.FaultHandler`, as shown in the following example:

```
<servlet>
   <servlet-name>
     statelessSession.WeatherHomeFault
   </servlet-name>
    <servlet-class>
        weblogic.soap.server.servlet.FaultHandler
   </servlet-class>
</servlet>
```

- A `<servlet>` element that references the WSDL JSP, as shown in the following example:

```
<servlet>
    <servlet-name>
       statelessSession.WeatherHomeWSDL
    </servlet-name>
    <jsp-file>
        /statelessSession.WeatherHome/wsdl.jsp
    </jsp-file>
</servlet>
```

  The path to the JSP file, `<jsp-file>`, is the path in your Web application archive file to the WSDL JSP you created in "Assembling an RPC-Style Web Service Archive File Manually" in this appendix.

- For each of the preceding `<servlet>` elements, create a `<servlet-mapping>` element to map a URL to the servlet, as shown in the following example:

```
<servlet-mapping>
  <servlet-name>statelessSession.WeatherHome</servlet-name>
  <url-pattern>/weatheruri</url-pattern>
</servlet-mapping>
<servlet-mapping>
```

```
    <servlet-name>
         statelessSession.WeatherHomeFault
    </servlet-name>
    <url-pattern>/weblogic/webservice/fault</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>
          statelessSession.WeatherHomeWSDL
    </servlet-name>
    <url-pattern>
/statelessSession.WeatherHome/statelessSession.WeatherHome.wsdl
    </url-pattern>
  </servlet-mapping>
```

- An `<error-page>` element:

```
<error-page>
    <exception-type>
       weblogic.soap.FaultException
    </exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
```

- An `<ejb-ref>` element which references the stateless session EJB that implements the Web service, as shown in the following example:

```
<ejb-ref>
  <description>Web Service EJB</description>
  <ejb-ref-name>statelessSession.WeatherHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>examples.webservices.rpc.weatherEJB.WeatherHome</home>
  <remote>examples.webservices.rpc.weatherEJB.Weather</remote>
</ejb-ref>
```

The following complete sample `web.xml` deployment descriptor contains elements for the RPC-style Web service example `examples.webservices.rpc`:

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
           "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
  <servlet>
    <servlet-name>statelessSession.WeatherHome</servlet-name>
    <servlet-class>
         weblogic.soap.server.servlet.StatelessBeanAdapter
    </servlet-class>
    <init-param>
```

```
      <param-name>ejb-ref</param-name>
      <param-value>statelessSession.WeatherHome</param-value>
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>statelessSession.WeatherHomeFault</servlet-name>
    <servlet-class>weblogic.soap.server.servlet.FaultHandler</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>statelessSession.WeatherHomeWSDL</servlet-name>
    <jsp-file>
         /statelessSession.WeatherHome/wsdl.jsp
    </jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>statelessSession.WeatherHome</servlet-name>
    <url-pattern>/weatheruri</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>statelessSession.WeatherHomeFault</servlet-name>
    <url-pattern>/weblogic/webservice/fault</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>statelessSession.WeatherHomeWSDL</servlet-name>
    <url-pattern>
         /statelessSession.WeatherHome/statelessSession.WeatherHome.wsdl
    </url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>weblogic.soap.FaultException</exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
  <ejb-ref>
    <description>This bean is exported as a WebService</description>
    <ejb-ref-name>statelessSession.WeatherHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>examples.webservices.rpc.weatherEJB.WeatherHome</home>
    <remote>examples.webservices.rpc.weatherEJB.Weather</remote>
  </ejb-ref>
</web-app>
```

Updating the weblogic.xml File for RPC-Style Web Services

The `weblogic.xml` deployment descriptor for RPC-style Web services does not contain any Web services-specific elements. It contains standard references to the stateless session EJB that implements the Web service.

The following sample `weblogic.xml` deployment descriptor contains elements for the RPC-style Web service example `examples.webservices.rpc`:

```
<!DOCTYPE weblogic-web-app
         PUBLIC "-//BEA Systems, Inc.//DTD Web Application 6.0//EN"
        "http://www.beasys.com/j2ee/dtds/weblogic-web-jar.dtd">

<weblogic-web-app>
  <reference-descriptor>
    <ejb-reference-description>
      <ejb-ref-name>statelessSession.WeatherHome</ejb-ref-name>
      <jndi-name>statelessSession.WeatherHome</jndi-name>
    </ejb-reference-description>
  </reference-descriptor>
</weblogic-web-app>
```

For more information on the elements of the `weblogic.xml` deployment descriptor, see *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html.

Updating the application.xml File for RPC-Style Web Services

The `application.xml` deployment descriptor for RPC-style Web service contains standard references to the Web application that references the SOAP servlets and stateless session EJB that comprises the Web service.

The one Web services-related element is the `<context-root>` sub-element of the `<web>` element. The value of the `<context-root>` element is used in all URLs that access either the WSDL, the Home Page, or the Web service itself.

The following sample `application.xml` deployment descriptor contains elements for the RPC-style Web service example `examples.webservices.rpc`:

```
<!DOCTYPE application
        PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN'
        'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  <display-name>Web-services</display-name>
  <module>
    <web>
      <web-uri>web-services.war</web-uri>
      <context-root>/weather</context-root>
    </web>
  </module>
  <module>
    <ejb>weather.jar</ejb>
  </module>
</application>
```

See *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs61/programming/app_xml.html for descriptions of the elements in the `application.xml` file.

Assembling a Message-Style Web Service Archive File Manually

This section describes how to manually assemble a message-style Web service into an Enterprise application `*.ear` file that can be deployed on WebLogic Server.

It is assumed that you have used the Administration Console to set up the following JMS components:

- The JMS destination (queue or topic) which will either receive the message from a client or from which the message is sent to a client.
- The JMS Connection factory that the WebLogic Web Service uses to create JMS connections.

For detailed information about using the Administration Console to configure JMS components, see the *WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/jms.html.

To assemble a message-style Web service archive file manually, follow these steps:

1. Create a temporary staging directory for assembling the Web application component. You can name this directory anything you want.
2. Set up your shell environment.

   On Windows NT, execute the `setEnv.cmd` command, located in the directory *`BEA_HOME`*`\config\`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in the directory *`BEA_HOME`*`/config/`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.
3. Execute the following command to automatically generate initial `web.xml` and `weblogic.xml` deployment descriptors in the `WEB-INF` subdirectory:

   ```
   java weblogic.ant.taskdefs.war.DDInit staging-dir
   ```

   where *`staging-dir`* refers to the staging directory.

   For more information on the Java-based `DDInit` utility for generating deployment descriptors, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs61/programming/packaging.html#pack004.
4. Edit the `WEB-INF/web.xml` file, adding WebLogic Web Services information, such as references to the SOAP servlets. For details, see "Creating the Message-Style Web Service WSDL File" in this appendix.
5. Edit the `WEB-INF/weblogic.xml` file, adding WebLogic Web Services information. For details, see "Updating the weblogic.xml File for Message-Style Web Services" in this appendix.
6. In the main staging directory, create a sub-directory that will hold the WSDL JSP for the Web service. You can name this sub-directory anything you want. This name will become part of the URL used to invoke the Web service.

   For this procedure, assume the name of this directory is `wsdl_dir`.
7. In the `wsdl_dir` subdirectory, create the WSDL JSP. The `wsgen` Java utility names this JSP `wsdl.jsp` when generating it automatically; you can follow this naming convention, or follow a convention of your own.

   For details on creating this file, see "Creating the Message-Style Web Service WSDL File" in this appendix.

8. In the `wsdl_dir` subdirectory, create an `index.html` file that contains links to the WSDL JSP you created in the preceding step and the client JAR file that you will create in a later step. The following example shows a simple `index.html` file:

```
<html>
<body>
<h3>Web Service Name</h3>
<ul>
<li><a href='wsdl.jsp'>WSDL</a></li>
<li><a href='../client.jar'>client.jar</li>
</ul>
</body>
</html>
```

9. Create a `client.jar` file in the main staging directory. For details on creating this file, refer to "Creating the client.jar File Manually" in this appendix.

   Note: This step is optional. You only need to create a `client.jar` file if you are going to use a Java client application to invoke the Web service.

10. Create an `index.html` file in the main staging directory that lists the Web service in this Enterprise application archive and links to its `index.html` file that you created in a previous step. The following example shows a simple `index.html` file:

```
<html>
<body>
<h3>Message-Style Web Services</h3>
<ul>
<li><a href='/context/wsdl_dir/index.html'>wsdl_dir</a></li>
</ul>
</body>
</html>
```

    In the example, *context* refers to the `context-root` element of the Web application in the `application.xml` file (to be created in a later step) and *wsdl_dir* refers to the name of sub-directory that contains the WSDL file you created in the previous step.

11. Create the Web application archive (`*.war` file) using a jar command such as:

    ```
    jar cvf web-app-name.war -C staging-dir .
    ```

    Note: The `wsgen` Java ant task assigns the default name `web-services.war` to the Web application `*.war` file. You do not have to follow this naming convention.

12. Create a second temporary staging directory for assembling the Enterprise application. You can name this directory anything you want.
13. Copy the Web application archive `*.war` file you created in a previous step into the staging directory you created in step 12.
14. Execute the following command to automatically generate an initial `application.xml` deployment descriptor in the `META-INF` subdirectory:

    ```
    java weblogic.ant.taskdefs.ear.DDInit staging-dir
    ```

    where *`staging-dir`* refers to the staging directory you created in step 12.

    For more information on the Java-based `DDInit` utility for generating deployment descriptors, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs61/programming/packaging.html#pack004.
15. Edit the `META-INF/application.xml` file, adding WebLogic Web Services information. For details, see "Updating the application.xml File for Message-Style Web Services" in this appendix.
16. Create the Enterprise Archive (`.ear` file) for the application, using a `jar` command such as:

    ```
    jar cvf application.ear -C staging-dir .
    ```

    The resulting `.ear` file can be deployed as a WebLogic Web Service using the Administration Console or the `weblogic.deploy` command-line utility.

Creating the Message-Style Web Service WSDL File

The WSDL JSP files for all message-style WebLogic Web Services are very similar, because there are only two operations that these types of Web services ever perform: send or receive data to or from a client application.

To create the WSDL JSP for a message-style Web service, follow these steps:

1. Using your favorite text editor, create a file called `wsdl.jsp`.
2. Copy and paste the sample WSDL at the end of this section into the `wsdl.jsp` file, and edit it according to the following steps.

   In the sample WSDL, the sections that you must modify for your specific Web service are in bold.

3. Globally replace references to `myService` with the name of your Web service.
4. If your Web service is one in which client applications that invoke it receive messages from the service, globally replace the word `send` with the word `receive`.
5. Globally replace `url:local` with the unique namespace for your Web service.
6. Replace the URI used to invoke the Web service from `/msg/sendMsg` to the following URI:

   `/context-root/url-pattern`

   where *context-root* refers to the `<context-root>` element of the `application.xml` deployment descriptor and *url-pattern* refers to the `<url-pattern>` for the SOAP servlet in the `web.xml` deployment descriptor.
7. If you want the WSDL file to statically specify the host and port of the WebLogic server hosting your Web service, edit the `soap:address` element in the WSDL JSP, replacing the text `<%= request.getServerName() %>:<%= request.getServerPort() %>` with hard-coded host and port values.

Use the following sample WSDL JSP as a starting point for your WSDL JSP.

```
<?xml version="1.0"?>
<definitions
    targetNamespace="urn:local"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="urn:local"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" >

  <types>
    <schema targetNamespace='urn:local'
            xmlns='http://www.w3.org/1999/XMLSchema'>
    </schema>
  </types>

  <message name="sendRequest">
    <part name="message" type="xsd:anyType" />
  </message>
  <message name="sendResponse">
  </message>

  <portType name="myServicePortType">
    <operation name="send">
      <input message="tns:sendRequest"/>
```

```
      <output message="tns:sendResponse"/>
    </operation>
  </portType>

  <binding name="myServiceBinding" type="tns:myServicePortType">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http/"/>
    <operation name="send">
      <soap:operation soapAction="urn:send"/>
      <input>
        <soap:body use="encoded" namespace='urn:myService'
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded" namespace='urn:myService'
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>

  <service name="myService">
    <documentation>todo</documentation>
    <port name="myServicePort" binding="tns:myServiceBinding">
      <soap:address location="http://<%= request.getServerName() %>:<%=
request.getServerPort() %>/msg/sendMsg"/>
    </port>
  </service>

</definitions>
```

Updating the web.xml File for Message-Style Web Services

This section describes the elements you must update or add to the `web.xml` deployment descriptor for the Web application that references the SOAP servlets in a message-style WebLogic Web Services archive file. For the complete example of a `web.xml` deployment descriptor, see the end of this section.

It is assumed that you have a basic understanding of Web applications and their deployment descriptors. For more information, see *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html.

To update a `web.xml` file for message-style Web services, add the following elements:

- A `<servlet>` element that references the SOAP servlet that manages the SOAP messages between the message-style Web service and the client application. Set the `<servlet-class>` sub-element to one of the following servlet classes, depending on whether the JMS destination is a topic or queue and whether the client invoking the service sends or receives messages:
  - `weblogic.soap.server.servlet.DestinationSendAdapter`—handles SOAP messages between the service and a client application that sends messages to either a JMS topic or queue.
  - `weblogic.soap.server.servlet.QueueReceiveAdapter`—handles SOAP messages between the service and a client application that receives messages from a JMS queue.
  - `weblogic.soap.server.servlet.TopicReceiveAdapter`—handles SOAP messages between the service and a client application that receives messages from a JMS topic.

  This `<servlet>` element contains two `<init-params>` elements: one that references the JMS destination classes and another that references the JMS connection factory classes.

  The following example shows a `<servlet>` reference to a SOAP servlet:

```
<servlet>
  <servlet-name>myService</servlet-name>
  <servlet-class>
       weblogic.soap.server.servlet.DestinationSendAdapter
  </servlet-class>
  <init-param>
    <param-name>topic-resource-ref</param-name>
    <param-value>myServiceDestination</param-value>
  </init-param>
  <init-param>
    <param-name>connection-factory-resource-ref</param-name>
    <param-value>myServiceFactory</param-value>
  </init-param>
</servlet>
```

- A `<servlet>` element that references the SOAP servlet that handles all SOAP faults. Set the `<servlet-class>` element to `weblogic.soap.server.servlet.FaultHandler`, as shown in the following example:

```
<servlet>
  <servlet-name>myServiceFault</servlet-name>
  <servlet-class>
```

```
            weblogic.soap.server.servlet.FaultHandler
    </servlet-class>
  </servlet>
```

- A `<servlet>` element that references the WSDL JSP, as shown in the following example:

  ```
  <servlet>
    <servlet-name>myServiceWSDL</servlet-name>
    <jsp-file>/myService/wsdl.jsp</jsp-file>
  </servlet>
  ```

  The path to the JSP file, `<jsp-file>`, is the path in your Web application archive file to the WSDL JSP you created in "Assembling a Message-Style Web Service Archive File Manually" in this appendix.

- For each of the preceding `<servlet>` elements, create a `<servlet-mapping>` element to map a URL to the servlet, as shown in the following example:

  ```
  <servlet-mapping>
    <servlet-name>myServiceFault</servlet-name>
    <url-pattern>/weblogic/webservice/fault</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>myServiceWSDL</servlet-name>
    <url-pattern>/myService/myService.wsdl</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>myService</servlet-name>
    <url-pattern>/sendMsg</url-pattern>
  </servlet-mapping>
  ```

- An `<error-page>` element, exactly as shown:

  ```
  <error-page>
      <exception-type>
         weblogic.soap.FaultException
      </exception-type>
      <location>/weblogic/webservice/fault</location>
    </error-page>
  ```

- Two `<resource-ref>` elements to link the JMS destination and connection factory references in the first `<servlet>` element to a Java object in JNDI, as shown in the following example:

  ```
  <resource-ref>
    <res-ref-name>myServiceDestination</res-ref-name>
    <res-type>javax.jms.Destination</res-type>
    <res-auth>Container</res-auth>
  ```

```
    </resource-ref>
    <resource-ref>
      <res-ref-name>myServiceFactory</res-ref-name>
      <res-type>javax.jms.ConnectionFactory</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
```

The following complete sample `web.xml` deployment descriptor contains elements for the message-style Web service example `examples.webservices.message`:

```
<!DOCTYPE web-app
          PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
          "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>myService</servlet-name>
    <servlet-class>
         weblogic.soap.server.servlet.DestinationSendAdapter
    </servlet-class>
    <init-param>
      <param-name>topic-resource-ref</param-name>
      <param-value>myServiceDestination</param-value>
    </init-param>
    <init-param>
      <param-name>connection-factory-resource-ref</param-name>
      <param-value>myServiceFactory</param-value>
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>myServiceFault</servlet-name>
    <servlet-class>weblogic.soap.server.servlet.FaultHandler</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>myServiceWSDL</servlet-name>
    <jsp-file>/myService/wsdl.jsp</jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>myServiceFault</servlet-name>
    <url-pattern>/weblogic/webservice/fault</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>myServiceWSDL</servlet-name>
    <url-pattern>/myService/myService.wsdl</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>myService</servlet-name>
    <url-pattern>/sendMsg</url-pattern>
  </servlet-mapping>
```

```
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <error-page>
    <exception-type>weblogic.soap.FaultException</exception-type>
    <location>/weblogic/webservice/fault</location>
  </error-page>
  <resource-ref>
    <res-ref-name>myServiceDestination</res-ref-name>
    <res-type>javax.jms.Destination</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <resource-ref>
    <res-ref-name>myServiceFactory</res-ref-name>
    <res-type>javax.jms.ConnectionFactory</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

Updating the weblogic.xml File for Message-Style Web Services

The `weblogic.xml` deployment descriptor for message-style Web services does not contain any Web services-specific elements, but rather, contains standard references to the JMS Destination and JMS Connection Factories.

The following sample `weblogic.xml` deployment descriptor contains elements for the message-style Web service example `examples.webservices.message`:

```
<!DOCTYPE weblogic-web-app
        PUBLIC "-//BEA Systems, Inc.//DTD Web Application 6.0//EN"
        "http://www.beasys.com/j2ee/dtds/weblogic-web-jar.dtd">

<weblogic-web-app>
  <reference-descriptor>
    <resource-description>
      <res-ref-name>myServiceDestination</res-ref-name>
      <jndi-name>examples.soap.msgService.MsgSend</jndi-name>
    </resource-description>
    <resource-description>
      <res-ref-name>myServiceFactory</res-ref-name>
      <jndi-name>examples.soap.msgService.MsgConnectionFactory</jndi-name>
    </resource-description>
```

```
  </reference-descriptor>
</weblogic-web-app>
```

Updating the application.xml File for Message-Style Web Services

The `application.xml` deployment descriptor for message-style Web services contains the standard reference to the Web application that contains the SOAP servlets.

The one Web services-related element is the `<context-root>` sub-element of the `<web>` element. The value of the `<context-root>` element is used in all URLs that access either the WSDL, the Home Page, or the Web service itself.

The following sample `application.xml` deployment descriptor contains elements for the message-style Web service example `examples.webservices.message`:

```
<!DOCTYPE application
         PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN'
         'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  <display-name>Web-services</display-name>
  <module>
    <web>
      <web-uri>web-services.war</web-uri>
      <context-root>/msg</context-root>
    </web>
  </module>
</application>
```

Creating the client.jar File Manually

The Java client.jar file contains the following objects:

- WebLogic FastParser (high-performance XML parser).
- WebLogic Web Services Client API.

- Remote interface of the stateless session EJB that implements an RPC-style Web service. This object is optional and only needed if you are using a static client to invoke the service.
- Class files for JavaBeans that are used as EJB parameters or return values.

BEA recommends that you use the `wsgen` Java Ant task to create an initial `*.ear` file and then extract the Java client.jar file contained within the `*.ear` file and modify it for your specific Web service. For details on using `wsgen`, see "Assembling a WebLogic Web Service" on page 2-19.

D Invoking Web Services Without Using the WSDL File

This Appendix shows an example of a dynamic client application that does not use the WSDL file when it invokes a WebLogic Web Service. In particular, the example invokes a message-style Web service and sends data to WebLogic Server.

Dynamic client applications that do not use the WSDL of the Web service are dynamic in every way, because they can invoke a Web service without knowing either the interface of the Web service, or the JavaBean interface of return values and parameters, or even the number and signatures of the methods that make up the Web service.

The example uses the URL `http://www.myHost.com:7001/msg/sendMsg` to invoke the Web Service. Because the example shows a dynamic client application that does not use the WSDL of the Web service, the preceding URL is for the *Web service* itself, rather than the URL for the *WSDL* of the Web service.

The procedure after the example discusses relevant sections of the example as part of the basic steps you follow to create this client.

```
import java.util.Properties;
import java.net.URL;
import javax.naming.Context;
import javax.naming.InitialContext;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
import weblogic.soap.codec.CodecFactory;
```

```
import weblogic.soap.codec.SoapEncodingCodec;
import weblogic.soap.codec.LiteralCodec;

public class ProducerClient{

  public static void main( String[] arg ) throws Exception{

    CodecFactory factory = CodecFactory.newInstance();
    factory.register( new SoapEncodingCodec() );
    factory.register( new LiteralCodec() );

    WebServiceProxy proxy = WebServiceProxy.createService(
       new URL( "http://www.myHost.com:7001/msg/sendMsg" ) );
    proxy.setCodecFactory( factory );
    proxy.setVerbose( true );

    SoapType param = new SoapType( "message", String.class );
    proxy.addMethod( "send", null, new SoapType[]{ param } );
    SoapMethod method = proxy.getMethod( "send" );

    String toSend = arg.length == 0 ? "No arg to send" : arg[0];
    Object result = method.invoke( new Object[]{ toSend } );
  }
}
```

Follow these steps to create a dynamic Java client that does not use WSDL to invoke a message-style WebLogic Web Service that sends data to WebLogic Server:

1. Get the Java client JAR file from the WebLogic Server hosting the WebLogic Web Service.

   For detailed information on this step, refer to "Downloading the Java Client JAR File from the Web Services Home Page" on page 3-6.

2. Add the Java client JAR file to your CLASSPATH on your client computer.

3. Create the client Java program. The following steps describe the Web services-specific Java code:

   a. In the `main` method of your client application, create a factory of encoding styles and register the two that are supported by WebLogic Server (the SOAP encoding style and Apache's Literal XML encoding style):

      ```
      CodecFactory factory = CodecFactory.newInstance();
      factory.register( new SoapEncodingCodec() );
      factory.register( new LiteralCodec() );
      ```

   b. Add the following Java code to create the connection to the Web service and set the encoding style factory:

```
WebServiceProxy proxy = WebServiceProxy.createService(
      new URL( "http://www.myHost.com:7001/msg/sendMsg" ) );
proxy.setCodecFactory( factory );
proxy.setVerbose( true );
```

c. Add the following Java code to dynamically get the `send` method of the Web service:

```
SoapType param = new SoapType( "message", String.class );
proxy.addMethod( "send", null, new SoapType[]{ param } );
SoapMethod method = proxy.getMethod( "send" );
```

d. Invoke the `send` method and send data to the Web service. In the example, the client application simply takes its first argument and sends it as a `String`; if the user does not specify an argument specified, then the client application sends the string `No arg to send`:

```
String toSend = arg.length == 0 ? "No arg to send" : arg[0];
Object result = method.invoke( new Object[]{ toSend } );
```

4. Compile and run the client Java program as usual.

The following more complex example shows how to use a `send` method that accepts a `org.w3c.dom.Document`, `org.w3c.dom.DocumentFragment`, or `org.w3c.dom.Element` data type as its parameter. The example shows how to set literal encoding on this flavor of the `send` method.

```
import java.util.Properties;

import java.net.URL;
import java.io.File;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import weblogic.apache.xml.serialize.OutputFormat;
import weblogic.apache.xml.serialize.XMLSerializer;

import weblogic.apache.xerces.dom.DocumentImpl;

import weblogic.soap.WebServiceProxy;
import weblogic.soap.SoapMethod;
import weblogic.soap.SoapType;
```

```
import weblogic.soap.codec.CodecFactory;
import weblogic.soap.codec.SoapEncodingCodec;
import weblogic.soap.codec.LiteralCodec;

public class ProducerClient{
  public static void main(String[] args) throws Exception{
    String url = "http://localhost:7001";
    // Parse the arguments list
    if (args.length != 2) {
     System.out.println("Usage: java examples.webservices.message.ProducerClient
http://hostname:port \"message\"");
      return;
    } else if (args.length == 2) {
      url = args[0];
    }

    CodecFactory factory = CodecFactory.newInstance();
    factory.register(new SoapEncodingCodec());
    factory.register(new LiteralCodec());

    URL newURL = new URL(url + "/msg/sendMsg");

    WebServiceProxy proxy = WebServiceProxy.createService(newURL);
    proxy.setCodecFactory(factory);
    proxy.setVerbose(true);
    SoapType param = new SoapType( "message", Document.class );
    proxy.addMethod( "send", null, new SoapType[]{ param } );

    SoapMethod method = proxy.getMethod("send");

    // Print out proxy to make sure method signature looks good
    System.out.println("Proxy:"+proxy);

    DocumentBuilderFactory dbf =
                    DocumentBuilderFactory.newInstance();
    //Obtain an instance of a DocumentBuilder from the factory.
    DocumentBuilder db = dbf.newDocumentBuilder();
    //Parse the document.
    Document w3cDoc = db.parse(new File("/test/fdr_nodtd.xml"));

    //Class parserClass = Class.forName("org.jdom.adapters.XercesDOMAdapter");
    //DOMAdapter da = (DOMAdapter)parserClass.newInstance();
    //Document w3cDoc = da.getDocument(new File("/test/fdr_nodtd.xml"),false);

    // Print out XML just to make sure the document was read successfully
    OutputFormat of = new OutputFormat();
    of.setEncoding("UTF-8");
    of.setLineWidth(40);
    of.setIndent(4);
    XMLSerializer xs = new XMLSerializer(System.out,of);
    xs.serialize(w3cDoc);
```

```
    System.out.println("Before Invoke");
    Object result = method.invoke( new Object[]{w3cDoc} );
    System.out.println("Done");
  }
}
```

Glossary

Assembling a Web service

Packaging all the components of the Web service into an Enterprise Application archive file (*.ear). You use Java Ant tasks to assemble a WebLogic Web Service.

Deploying a Web service

Making the Web service available to remote clients. This is analogous, although not exactly the same, as deploying an EJB. You deploy a Web service after you have deployed the EJBs that make up the Web service. You use the Administration Console to deploy a WebLogic Web Service.

Implementing a Web service

Writing the Java code for the stateless session EJB (for RPC-style Web services) or a message-driven bean (for message-style Web services) that is defined to be the entry point to the Web service. The stateless session EJB or message-driven bean may contain all the Web service functionality, or it may call other EJBs to parcel out the work.

Invoking a Web service

The actions that a client application performs to use the Web service. The client first assembles a SOAP message that describes the Web service it wants to invoke and includes all the necessary data, either in the SOAP body or in an attachment. The client then sends the SOAP message over HTTP/HTTPS to the WebLogic Server, which executes the Web service and may or may not send a SOAP message back to the client over HTTP/HTTPS.

Java Ant

The Java utility that you use to assemble WebLogic Web Services into Enterprise Application archives.

Message-style Web services

A type of Web service that uses a JMS destination as its entry point. Message-style Web services are loosely coupled document-driven services; this means that clients typically use this type of Web service by sending entire documents that will be processed by the Web service rather than sending parameters and receiving return values.

Publishing a Web service

Registering the Web service in a well-known location so it can be found by anyone who wants to use it. This can be done by registering the Web service in a UDDI registry, emailing the URL that invokes the Web service to whoever wants it, and so on.

RPC-style Web service

A type of Web service that uses a stateless session EJB as its entry point. RPC-style Web services are tightly coupled interface-driven services; this means that clients typically use the Web service by sending it parameters and receiving return values rather than sending an entire document to be processed by the Web service.

SOAP

Simple Object Access Protocol. A lightweight XML-based protocol for exchanging information in a decentralized, distributed environment.

SOAP with attachments

A specification that describes a standard way to associate a SOAP message with one or more attachments in their native format in a multipart MIME structure for transport.

Web service

A shared application accessed by heterogeneous users over the Web that encapsulate a specific functionality.

Web Services Home Page

A Web page that lists the Web services defined for a particular context along with the WSDL files and Java client JAR file associated with each Web service.

WSDL

Web Services Description Language. An XML-based language used to describe Web services.

Index

T

U

V

W

X