BEA WebLogic
Server™

**Programming WebLogic
Web Services**

Programming WebLogic Web Services

Part Number	Date	Software Version
N/A	January 9, 2004	BEA WebLogic Server Version 7.0

Contents

8. Invoking Web Services

9. Using Non-Built-In Data Types

10. Creating SOAP Message Handlers to Intercept the SOAP Message

11. Configuring Security

12. Creating JMS-Implemented WebLogic Web Services

13. Administering WebLogic Web Services

14. Publishing and Finding Web Services Using UDDI

15. Interoperability

16. Upgrading 6.1 WebLogic Web Services to 7.0

A. WebLogic Web Service Deployment Descriptor Elements

B. Web Service Ant Tasks and Command-Line Utilities

C. Customizing WebLogic Web Services

D. Specifications Supported by WebLogic Web Services

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 and the features of WebLogic Web services.

- Chapter 2, "Architectural Overview," provides an architectural overview of WebLogic Web services.

- Chapter 3, "Creating a WebLogic Web Service: A Simple Example," describes the end-to-end process of creating a simple WebLogic Web service based on a stateless session EJB.

- Chapter 4, "Designing WebLogic Web Services," describes the design issues you should consider before developing a WebLogic Web Service.

- Chapter 5, "Implementing WebLogic Web Services," describes how to create the back-end components that implement a Web service.

- Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks," describes how to use the WebLogic Web services Ant tasks to automatically generate the final parts of a Web service (such as the serialization information for non-built-in data types and client JAR file), package them all together into a deployable EAR file, and deploy the EAR file on WebLogic Server.

- Chapter 7, "Assembling a WebLogic Web Service Manually," describes how assemble a WebLogic Web service manually without using the WebLogic Web services Ant tasks.

- Chapter 8, "Invoking Web Services," describes how to write a client application that invokes WebLogic Web services.

- Chapter 9, "Using Non-Built-In Data Types," describes how to create the serializers and deserializers that convert user-defined data types between their XML and Java representations.

- Chapter 10, "Creating SOAP Message Handlers to Intercept the SOAP Message," describes how to create handlers that intercept a SOAP message for further processing.

- Chapter 11, "Configuring Security," describes how to configure security for WebLogic Web services.

- Chapter 12, "Creating JMS-Implemented WebLogic Web Services," describes how to create a WebLogic Web service that is implemented with a JMS message consumer or producer.

- Chapter 13, "Administering WebLogic Web Services," describes how to use the Administration Console to administer WebLogic Web services.

- Chapter 14, "Publishing and Finding Web Services Using UDDI," describes how to use the UDDI features included in WebLogic Server.

- Chapter 15, "Interoperability," describes what it means for Web services to interoperate with each other and provides tips for creating highly interoperable Web services.

- Chapter 16, "Upgrading 6.1 WebLogic Web Services to 7.0," describes how to upgrade Web services created in Version 6.1 of WebLogic Server to Version 7.0.

- Appendix A, "WebLogic Web Service Deployment Descriptor Elements," describes the elements in the Web services deployment descriptor file, web-services.xml.

- Appendix B, "Web Service Ant Tasks and Command-Line Utilities," describes the Ant tasks, along with their equivalent command-line utilities, used to assemble WebLogic Web services.

- Appendix C, "Customizing WebLogic Web Services," describes how to customize WebLogic Web services by updating the Web application's web.xml deployment descriptor file.

- Appendix D, "Specifications Supported by WebLogic Web Services," provides information about the specifications supported by WebLogic Web services, such as SOAP 1.1, WSDL 1.1, and so on.

Audience

This document is written for Java developers who want to create a Web service that runs on WebLogic Server.

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.

Convention	Usage
`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. *Example*s: 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]`
`|`	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

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

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 closed 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?

Major benefits of Web services include:

- Interoperability among distributed applications that span diverse hardware and software platforms

- Easy, widespread access to applications through firewalls using Web protocols

- A cross-platform, cross-language data model (XML) that facilitates developing heterogeneous distributed applications

Because you access Web services 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, and thus communicate with each other.

These different systems can be Microsoft SOAP ToolKit clients, J2EE applications, legacy applications, and so on. They are written in Java, C++, Perl, and other programming languages. Application interoperability is the goal of Web services and depends upon the service provider's adherence to published industry standards.

Web Service Standards

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 Java classes; 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 and 1.2 as the message format and HTTP as the connection protocol. See "SOAP 1.1 with Attachments" on page 1-4.

  Note: WebLogic Web services accept both SOAP 1.1 and 1.2 incoming requests, but produce only SOAP 1.1 outgoing responses.

■ 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. See "WSDL 1.1" on page 1-5.

■ A standard way for client applications to invoke a Web service.

  WebLogic Web services implement the Java API for XML-based RPC (JAX-RPC) as part of a client JAR that client applications can use to invoke WebLogic and non-WebLogic Web services. See "JAX-RPC" on page 1-6.

■ A standard way for client applications to find a registered Web service and to register a Web service.

WebLogic Web services implement the Universal Description, Discovery, and Integration (UDDI) specification. See "UDDI 2.0" on page 1-7.

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. WebLogic Server includes its own implementation of both the SOAP 1.1 and SOAP With Attachments specifications. The protocol consists of:

■ An envelope that describes the SOAP message. 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.

Note: WebLogic Web services accept both SOAP 1.1 and 1.2 incoming requests, but produce only SOAP 1.1 outgoing responses.

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 that describes a Web service. A WSDL document describes Web service operations, input and output parameters, and how to connect to the Web service.

Developers of WebLogic Web services do not need to create the WSDL files; you generate these files 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>
```

JAX-RPC

The Java API for XML-based RPC (JAX-RPC) is a Sun Microsystems specification that defines the client API for invoking a Web service.

The following table briefly describes the core JAX-RPC interfaces and classes.

Table 1-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Service	Main client interface. Used for both static and dynamic invocations.
ServiceFactory	Factory class for creating `Service` instances.
Stub	Represents the client proxy for invoking the operations of a Web service. Typically used for static invocation of a Web service.
Call	Used to dynamically invoke a Web service.
JAXRPCException	Exception thrown if an error occurs while invoking a Web service.

For detailed information on JAX-RPC, see the following Web site:
http://java.sun.com/xml/jaxrpc/index.html.

For a tutorial that describes how to use JAX-RPC to invoke Web services, see
http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXRPC.html.

UDDI 2.0

The Universal Description, Discovery and Integration (UDDI) specification defines a
standard way to describe a Web service; register a Web service in a well-known
registry; and discover other registered Web services.

See http://www.uddi.org.

WebLogic Web Service Features

The WebLogic Web services subsystem has the following features:

- **Standard Specifications**

  See "Web Service Standards" on page 1-3.

- **Support for Exposing Standard J2EE Components**

  WebLogic Web services support exposing standard J2EE components, such as
  stateless session EJBs and JMS consumers or producers.

- **Ant Tasks and Command Line Utilities**

  Ant tasks facilitate the implementation and packaging of Web services. See
  Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

- **UDDI Registry, Directory Explorer, and Client API**

  WebLogic Server includes a UDDI registry, a UDDI Directory Explorer, and an
  implementation of the UDDI client API. See Chapter 14, "Publishing and
  Finding Web Services Using UDDI."

- **Support for Both RPC-Oriented and Document-Oriented Operations**

WebLogic Web service operations can be either RPC-oriented (SOAP messages contain parameters and return values) or document-oriented (SOAP messages contain documents.) For details, see "RPC-Oriented or Document-Oriented Web Services?" on page 4-4.

■ **Support for User-Defined Data Types**

You can create a WebLogic Web service that uses non-built-in data types as its parameters and returns values. Non-built-in data types are defined as data types other than the supported built-in data types; examples of built-in data types include int and String. WebLogic Server Ant tasks can generate the components needed to use non-built-in data types; this feature is referred to as *autotyping*. You can also create these components manually. See Appendix B, "Web Service Ant Tasks and Command-Line Utilities," and Chapter 9, "Using Non-Built-In Data Types."

■ **SOAP Message Handlers to Access SOAP Messages**

A SOAP message handler accesses the SOAP message and its attachment in both the request and response of the Web service. You can create handlers in both the Web service itself and the client applications that invoke the Web service. See Chapter 10, "Creating SOAP Message Handlers to Intercept the SOAP Message."

■ **Java Client to Invoke a Web Service**

Developers can use an automatically generated thin Java client to create Java client applications that invoke WebLogic and non-WebLogic Web services. See Chapter 8, "Invoking Web Services."

Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

■ **The Web Services Home Web Page**

All deployed WebLogic Web services automatically have a Home Web Page that includes links to the WSDL of the Web service, the client JAR file that you can download for invoking the Web service, and a mechanism for testing the Web service to ensure that it is working as expected. See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22.

■ **Point-to-Point Security**

WebLogic Server supports connection oriented point-to-point security for WebLogic Web service operations, as well as authorization and authentication of Web service operations. See Chapter 11, "Configuring Security."

■ **Interoperability**

WebLogic Web services interoperate with major Web service platforms such as Microsoft .NET.

■ **Java 2 Platform Micro Edition (J2ME) Clients**

The WebLogic Server the `clientgen` Ant task can create a client JAR file that runs on J2ME. See Chapter 8, "Invoking Web Services."

Examples Of Creating and Invoking a Web Service

WebLogic Server includes the following examples of creating and invoking WebLogic Web services in the *WL_HOME*\samples\server\src\examples\webservices directory, where *WL_HOME* refers to the main WebLogic Platform directory:

■ `basic.statelessSession` : Uses a stateless session EJB backend component with built-in data types as its parameters and return value.

■ `basic.javaclass` : Uses a Java class backend component with built-in data types as its parameters and return value.

■ `complex.statelessSession` : Uses a stateless session EJB backend component with non-built-in data types as its parameters and return value.

■ `handler.log` : Uses both a handler chain and a stateless session EJB.

■ `handler.nocomponent` : Uses only a handler chain with *no* backend component.

■ `client.static` : Shows how to create a static client application that invokes a non-WebLogic Web service.

■ `client.static_out` : Shows how to create a static client application that invokes a non-WebLogic Web service that uses out parameters.

- `client.dynamic_wsdl` : Shows how to create a dynamic client application that uses WSDL to invoke a non-WebLogic Web service.

- `client.dynamic_no_wsdl` : Shows how to create a dynamic client application that does not use WSDL to invoke a non-WebLogic Web service.

- `message`: Shows how to create a JMS-implemented WebLogic Web service.

- `multicomponent`: Shows how to create a WebLogic Web services implemented with two Java classes.

- `r4client`: Shows how to run the WebLogic Web service client to exercise the SOAPBuilders Interoperability Lab Round 3 suite of tests.

For detailed instructions on how to build and run the examples, open the following Web page in your browser:

WL_HOME\samples\server\src\examples\webservices\package-summary.html

Additional examples of creating and invoking WebLogic Web services are listed on the Web services Web page on the Web Services dev2dev Download Page at http://webservice.bea.com.

Creating WebLogic Web Services: Main Steps

The following procedure describes the high-level steps to create a WebLogic Web service. Most steps are described in detail in later chapters.

Chapter 3, "Creating a WebLogic Web Service: A Simple Example," briefly describes an example of creating a Web service.

1. Design the WebLogic Web service.

   Decide on a synchronous or asynchronous Web service; the type of back-end components that implement the service; whether your service uses only built-in data types or custom data types; whether you need to intercept the incoming or outgoing SOAP message; and so on.

   See Chapter 4, "Designing WebLogic Web Services."

2. Implement the WebLogic Web service.

Write the Java code of the back-end components that make up the Web service; optionally create SOAP message handlers that intercept the SOAP messages; optionally create your own serialization class to convert data between XML and Java; and so on.

See Chapter 5, "Implementing WebLogic Web Services."

3. Assemble and package the WebLogic Web service.

   Gather all the implementation class files into an appropriate directory structure; create the Web service deployment descriptor file; create the supporting parts of the service (such as client JAR file); and package everything into a deployable EAR file.

   If your Web service is fairly simple, use the servicegen Ant task which performs all the assembly steps for you. If your Web service is more complicated, use additional Ant tasks or assemble the Web service manually.

   See Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

4. Create a client that accesses the Web service to test that your Web service is working as you expect. You can also use the Web Service Home Page to test your Web service.

   See Chapter 8, "Invoking Web Services."

5. Deploy the WebLogic Web service.

   Make the service 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.

   See *Deployment* at http://e-docs.bea.com/wls/docs70/deployment.html.

6. Optionally publish your Web service in a UDDI registry. See Chapter 14, "Publishing and Finding Web Services Using UDDI."

Differences Between 6.1 and 7.X WebLogic Web Services

WebLogic Web services changed between versions 6.1 and 7.X of WebLogic Server:

■ Due to changes in the runtime system between the two versions, you must upgrade WebLogic Web services created in version 6.1 of WebLogic Server to run on version 7.x. Do not deploy a 6.1 Web service on a 7.x WebLogic Server instance. See Appendix 16, "Upgrading 6.1 WebLogic Web Services to 7.0."

■ The WebLogic Web services client API included in version 6.1 of WebLogic Server has been deprecated. The 6.1 client API was packaged as `weblogic.soap.*`. Version 7.x includes a new client API, based on the Java API for XML based RPC (JAX-RPC). For details see Chapter 8, "Invoking Web Services."

■ You can implement 7.x Web services with multiple backend components, such as a stateless session EJB, a Java class, and a JMS listener. In 6.1, however, a single Web service was implemented with just one backend component. For this reason, 7.x does not use the terminology "rpc-style" and "message-style" to refer to an entire Web service.

Also in 7.x you can choose which methods of an EJB you expose to the public; in 6.1 you were forced to expose all public methods of an EJB.

See Chapter 2, "Architectural Overview."

■ 7.x Web services support non-built-in data types as parameters and return values, as well as out and in-out parameters. 6.1 WebLogic Web services were limited to a list of built-in data types that could be used as parameters and return values. See Chapter 9, "Using Non-Built-In Data Types."

■ Version 7.x introduces SOAP message handlers and handler chains that can intercept SOAP messages for further processing. 6.1 WebLogic Web services were unable to access the request or response SOAP message. See Chapter 10, "Creating SOAP Message Handlers to Intercept the SOAP Message."

■ In 7.x you can use SOAP message handlers to access the SOAP attachment for further processing; in 6.1 you could not access the attachment. See Chapter 10, "Creating SOAP Message Handlers to Intercept the SOAP Message."

- 7.x Web services now have their own deployment descriptor file, called web-services.xml. For reference information on the web-services.xml file, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements." For examples of the file, see Chapter 7, "Assembling a WebLogic Web Service Manually."

- WebLogic Server 7.x includes additional Ant task support for quickly implementing, assembling, and packaging WebLogic Web services. See Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

- In 7.x you can specify that a Web service be asynchronous one-way, which means the client application that invokes the Web service never receives a response, including any exceptions. In Version 6.1, all Web services were synchronous request-response, which meant that every time a client application invoked a service the client received a response. See Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

- The 7.x WebLogic Server Administration Console has a new **Web Services Components** deployment node for easy administration of Web services. See Chapter 13, "Administering WebLogic Web Services."

- Version 7.x of WebLogic Server includes UDDI server-side and client-side features. See Chapter 14, "Publishing and Finding Web Services Using UDDI."

- The 7.x Web Services Home Page has been enhanced from that of 6.1; the new page includes a facility to test your Web service. See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22.

Unsupported Features

WebLogic Server does not support the following WSDL and XML Schema features:

- Overloading operations in WSDL, due to a SOAP limitation

- XML Schema complex data type inheritance by restriction

- XML Schema union simple data types

Editing XML Files

When creating or invoking WebLogic Web services, you might need to edit XML files, such as the `web-services.xml` Web services deployment descriptor file, the EJB deployment descriptors, the Java Ant build files, and so on. You edit these files with the BEA XML Editor.

Note: This guide describes how to create or update the `web-services.xml` deployment descriptor manually so that programmers get a better understanding of the file and how the elements describe a Web service. You can also use the BEA XML Editor to update the file.

The BEA XML Editor is a simple, user-friendly Java-based 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 gives you two options for editing the XML document:

- The hierarchical tree view allows structured, constrained editing, 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.

The 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 online help.

You can download the BEA XML Editor from dev2dev Online at http://dev2dev.bea.com/resourcelibrary/utilitiestools.jsp.

2   Architectural Overview

The following sections provide an overview of WebLogic Web services architecture and three types of WebLogic Web service operations:

WebLogic Web Services Architecture

You develop a WebLogic Web service, by using standard J2EE components, such as stateless session EJBs, and Java classes. Because WebLogic Web services are based on the J2EE platform, they automatically inherit all the standard J2EE benefits, such as a simple and familiar component-based development model, scalability, support for transactions, 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.

A single WebLogic Web service consists of one or more operations; you can implement each operation using different backend components and SOAP message handlers. For example, an operation might be implemented with a single method of a stateless session EJB or with a combination of SOAP message handlers and a method of a stateless session EJB.

Backend Component-Only Operation

The following figure describes the architecture of a WebLogic Web service operation that is implemented with only a backend component, such as a method of a stateless session EJB.

Figure 2-1   WebLogic Web Service with Backend Component
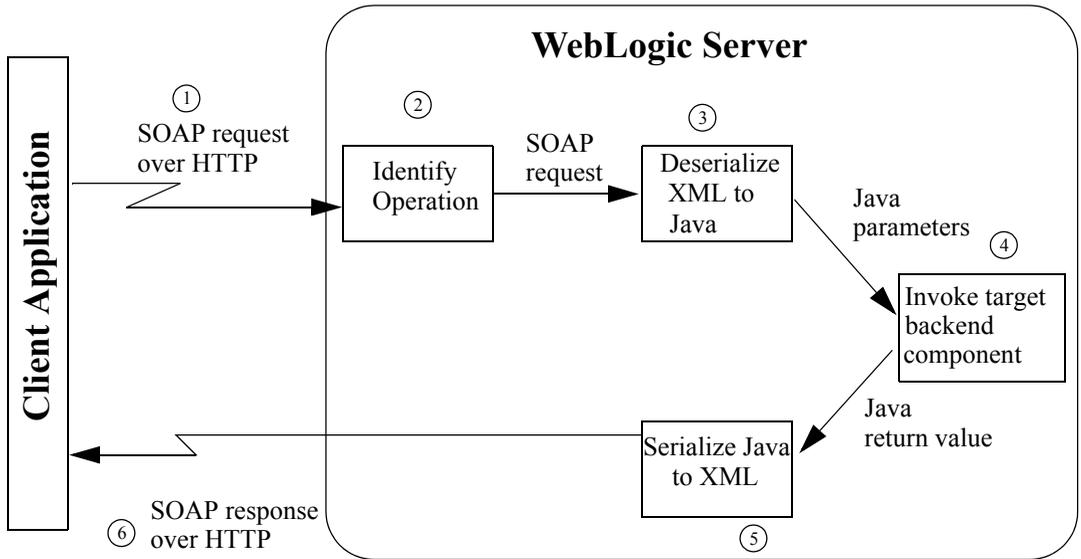
Here's what happens when a client application invokes this type of WebLogic Web service operation:

1. The client application sends a SOAP message request to WebLogic Server over HTTP. Based on the URI in the request, WebLogic Server identifies the Web service being invoked.

2. The Web service reads the SOAP message request and identifies the operation that it needs to run. The operation corresponds to an invoke of a method of a stateless session EJB or a Java class.

3. The Web service converts the operation's parameters in the SOAP message from their XML representation to their Java representation using the appropriate deserializer class. The deserializer class is either one provided by WebLogic Server for built-in data types or a user-created one for non-built-in data types.

4. The Web service invokes the appropriate backend component method, passing it the Java parameters.

5. The Web service converts the method's return value from Java to XML using the appropriate serializer class, and packages it into a SOAP message response.

6. The Web service sends the SOAP message response back to the client application that invoked the Web service.

Backend Component and SOAP Message Handler Chain Operation

The following figure describes a WebLogic Web service operation that is implemented with both a SOAP message handler chain and a backend component.

Here's what happens when a client application invokes this type of WebLogic Web service operation:

1. The client application sends a SOAP message request to WebLogic Server over HTTP. Based on the URI in the request, WebLogic Server identifies the Web service being invoked.

2. The Web service reads the SOAP message request and identifies the operation that it needs to run. The operation in this case corresponds to an invoke of a SOAP message handler chain followed by an invoke of a method of a stateless session EJB or a Java class.

3. The Web service invokes the appropriate handler chain. The handler chain accesses the contents of the SOAP message request, possibly changing it in some way.

4. The Web service converts the operation's parameters in the [possibly modified] SOAP message from their XML representation to their Java representation using the appropriate deserializer class. The deserializer class is either one provided by WebLogic Server for built-in data types or a user-created one for non-built-in data types.

5. The Web service invokes the appropriate backend component method, passing it the Java parameters.

6. The Web service converts the method's return value from Java to XML using the appropriate serializer class, and packages it into a SOAP message response.

7. The Web service invokes the appropriate SOAP message handler chain. The handler chain accesses the contents of the SOAP message response, possibly changing it in some way.

8. The Web service sends the [possibly modified] SOAP message response back to the client application that invoked the Web service.

SOAP Message Handler Chain-Only Operation

The following figure describes the architecture of a WebLogic Web service operation that is implemented with only a SOAP message handler chain.

**Figure 2-3   WebLogic Web Service Operation with SOAP Message Handler
Chain Only**
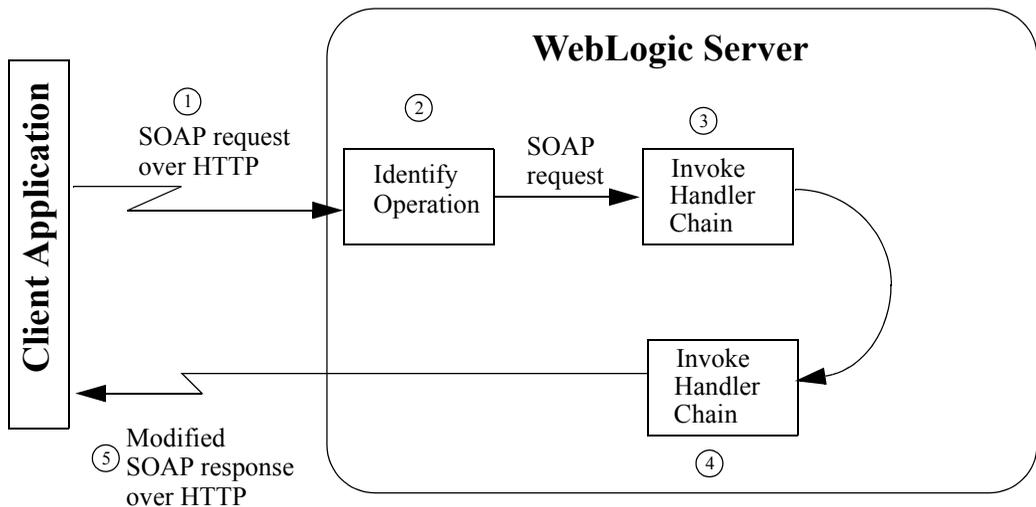
Here's what happens when a client application invokes this type of WebLogic Web
service operation:

1. The client application sends a SOAP message request to WebLogic Server over
   HTTP. Based on the URI in the request, WebLogic Server identifies the Web
   service being invoked.

2. The Web service reads the SOAP message request and identifies the operation
   that it needs to run. The operation in this case corresponds to an invoke of a
   SOAP message handler chain.

3. The Web service invokes the appropriate handler chain. The handler chain
   accesses the contents of the SOAP message request, possibly changing it in some
   way.

4. The Web service invokes the appropriate handler chain. The handler chain creates
   the SOAP message response.

5. The Web service sends the SOAP message response back to the client application
   that invoked the Web service.

3 Creating a WebLogic Web Service: A Simple Example

The following sections describe a simple example of creating a WebLogic Web service:

Description of the Example

This example describes the start-to-finish process of implementing, assembling, and deploying the WebLogic Web service provided as a product example in the directory `WL_HOME\samples\server\src\examples\webservices\complex\stateless Session`, where `WL_HOME` refers to the main WebLogic Platform directory.

The example shows how to create a WebLogic Web service based on a stateless session EJB. The example uses the `Trader` EJB, one of the EJB 2.0 examples located in the
`WL_HOME\samples\server\src\examples\ejb20\basic\statelessSession`
directory.

The `Trader` EJB defines two methods, `buy()` and `sell()`, that take as input a `String` stock symbol and an `int` number of shares to buy or sell. Both methods return a non-built-in data type called `TradeResult`.

When the `Trader` EJB is converted into a Web service, the two methods become public operations defined in the WSDL of the Web service. The `Client.java` application uses a JAX-RPC style client API to create SOAP messages that invoke the operations.

Example of Creating a WebLogic Web Service: Main Steps

To create the sample `Trader` WebLogic Web service, follow these steps:

1. Set up your environment.

   On Windows NT, execute the `setExamplesEnv.cmd` command, located in the directory `WL_HOME\samples\server\config\examples`, where `WL_HOME` is the main directory of your WebLogic Platform.

   On UNIX, execute the `setEnv.sh` command, located in the directory `WL_HOME/samples/server/config/examples`, where `WL_HOME` is the main directory of your WebLogic Platform.

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

   See "Writing the Java Code for the EJB" on page 3-4.

3. Write the Java code for the `TradeResult` non-built-in data type.

   See "Writing the Java Code for the Non-Built-In Data Type" on page 3-8.

4. Compile the Java code into class files.

5. Create the EJB deployment descriptors.

   See "Creating EJB Deployment Descriptors" on page 3-9.

6. Assemble the EJB class files and deployment descriptors into a `trader.jar` archive file.

   See "Assembling the EJB" on page 3-11.

7. Create the `build.xml` Ant build file. This file will execute the `servicegen` Ant task used to assemble the WebLogic Web service.

   See "Creating the build.xml Ant Build File" on page 3-11.

8. Create a staging directory.

9. Copy the EJB `trader.jar` file and the `build.xml` file into the staging directory.

10. Execute the Java Ant utility to assemble the `Trader` Web service into a `trader.ear` archive file:

    ```
    $ ant
    ```

11. Auto-deploy the `Trader` Web service to WebLogic Server for testing purposes by copying the `trader.ear` archive file to the *domain*/`applications` directory, where *domain* refers to the location of your domain.

12. View the Home Page of the `Trader` Web serivce by invoking the following URL in your browser:

    ```
    http://localhost:port/webservice/TraderService
    ```

    where

    - *localhost* refers to the machine on which WebLogic Server is running

    - *port* refers to port on which WebLogic Server is listening

    From the Web Service Home Page you can view the generated WSDL and test the Web service to make sure it's working correctly.

To invoke the `Trader` Web service from a Java client application, see the `Client.java` file in the `WL_HOME`\samples\server\src\examples\webservices\complex\stateless Session directory.

For instructions for building and running the client application, invoke the `WL_HOME`\samples\server\src\examples\webservices\complex\stateless Session\package-summary.html Web page in your browser.

Writing the Java Code for the EJB

The sample `Trader` stateless session EJB contains two public methods: `buy()` and `sell()`. The `Trader` EJB defines two methods, `buy()` and `sell()`, that take as input a `String` stock symbol and an `int` number of shares to buy or sell. Both methods return a non-built-in data type called `TraderResult`.

The following Java code is the public interface of the `Trader` EJB:

```
package examples.webservices.complex.statelessSession;

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

/**
 * The methods in this interface are the public face of TraderBean.
 * 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, TraderBeanE, implements this interface and
 * delegates to the bean.
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public interface Trader extends EJBObject {
  /**
   * Buys shares of a stock.
   *
   * @param stockSymbol      String Stock symbol
   * @param shares           int Number of shares to buy
   * @return                 TradeResult Trade Result
   * @exception              RemoteException if there is
   *                         a communications or systems failure
   */
  public  TradeResult buy (String stockSymbol, int shares)
    throws RemoteException;
  /**
   * Sells shares of a stock.
   *
   * @param stockSymbol      String Stock symbol
   * @param shares           int Number of shares to sell
   * @return                 TradeResult Trade Result
   * @exception              RemoteException if there is
   *                         a communications or systems failure
```

```
  */
  public TradeResult sell (String stockSymbol, int shares)
    throws RemoteException;
}
```

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

```
package examples.webservices.complex.statelessSession;

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

/**
 * TraderBean 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) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */
public class TraderBean 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
 * "TraderHome.java".
 * The parameter sets of the two methods are identical. When the client calls
 * <code>TraderHome.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.ejb11.basic.statelessSession.Trader
 */
public void ejbCreate () throws CreateException {
  log("ejbCreate called");
  try {
    InitialContext ic = new InitialContext();
    Integer tl = (Integer) ic.lookup("java:/comp/env/tradeLimit");
    tradeLimit = tl.intValue();
  } catch (NamingException ne) {
    throw new CreateException("Failed to find environment value "+ne);
  }
}

/**
 * Buys shares of a stock for a named customer.
```

```
     *
     * @param customerName        String Customer name
     * @param stockSymbol         String Stock symbol
     * @param shares              int Number of shares to buy
     * @return                    TradeResult Trade Result
     *                            if there is an error while buying the shares
     */
  public TradeResult buy(String stockSymbol, int shares) {
    if (shares > tradeLimit) {
      log("Attempt to buy "+shares+" is greater than limit of "+tradeLimit);
      shares = tradeLimit;
    }
    log("Buying "+shares+" shares of "+stockSymbol);
    return new TradeResult(shares, stockSymbol);
  }

  /**
   * Sells shares of a stock for a named customer.
   *
   * @param customerName        String Customer name
   * @param stockSymbol         String Stock symbol
   * @param shares              int Number of shares to buy
   * @return                    TradeResult Trade Result
   *                            if there is an error while selling the shares
   */
  public TradeResult sell(String stockSymbol, int shares) {
    if (shares > tradeLimit) {
      log("Attempt to sell "+shares+" is greater than limit of "+tradeLimit);
      shares = tradeLimit;
    }
    log("Selling "+shares+" shares of "+stockSymbol);
    return new TradeResult(shares, stockSymbol);
  }
}
```

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

```
package examples.webservices.complex.statelessSession;

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

/**
 * This interface is the home interface for the TraderBean.java,
 * which in WebLogic is implemented by the code-generated container
 * class TraderBeanC. A home interface may support one or more create
 * methods, which must correspond to methods named "ejbCreate" in the EJBean.
```

```
 *
 * @author Copyright (c) 1998-2002 by BEA Systems, Inc. All Rights Reserved.
 */
public interface TraderHome extends EJBHome {
  /**
   * This method corresponds to the ejbCreate method in the bean
   * "TraderBean.java".
   * The parameter sets of the two methods are identical. When the client calls
   * <code>TraderHome.create()</code>, the container
   * allocates an instance of the EJBean and calls <code>ejbCreate()</code>.
   *
   * @return               Trader
   * @exception            RemoteException if there is
   *                       a communications or systems failure
   * @exception            CreateException
   *                       if there is a problem creating the bean
   * @see                  examples.ejb11.basic.statelessSession.TraderBean
   */
  Trader create() throws CreateException, RemoteException;
}
```

Writing the Java Code for the Non-Built-In Data Type

The two methods of the EJB return a non-built-in data type called `TraderResult`. The following Java code describes this type:

```
package examples.webservices.complex.statelessSession;

import java.io.Serializable;

/**
 * This class reflects the results of a buy/sell transaction.
 *
 * @author Copyright (c) 1999-2002 by BEA Systems, Inc. All Rights Reserved.
 */
public final class TradeResult implements Serializable {

  // Number of shares really bought or sold.
  private int    numberTraded;

  private String stockSymbol;
```

```
  public TradeResult() {}

  public TradeResult(int nt, String ss) {
    numberTraded = nt;
    stockSymbol  = ss;
  }

  public int getNumberTraded() { return numberTraded; }

  public void setNumberTraded(int numberTraded) {
        this.numberTraded = numberTraded;
  }

  public String getStockSymbol() { return stockSymbol; }

  public void setStockSymbol(String stockSymbol) {
        this.stockSymbol = stockSymbol;
  }
}
```

Creating EJB Deployment Descriptors

See "Editing XML Files" on page 1-14 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 Trader 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>TraderService</ejb-name>
      <home>examples.webservices.complex.statelessSession.TraderHome</home>
      <remote>examples.webservices.complex.statelessSession.Trader</remote>

<ejb-class>examples.webservices.complex.statelessSession.TraderBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
```

```
          <env-entry-name>WEBL</env-entry-name>
          <env-entry-type>java.lang.Double </env-entry-type>
          <env-entry-value>10.0</env-entry-value>
      </env-entry>
      <env-entry>
          <env-entry-name>INTL</env-entry-name>
          <env-entry-type>java.lang.Double </env-entry-type>
          <env-entry-value>15.0</env-entry-value>
      </env-entry>
      <env-entry>
          <env-entry-name>tradeLimit</env-entry-name>
          <env-entry-type>java.lang.Integer </env-entry-type>
          <env-entry-value>500</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>TraderService</ejb-name>
        <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 Trader EJB:

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

<!DOCTYPE weblogic-ejb-jar PUBLIC
'-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN'
'http://www.bea.com/servers/wls700/dtd/weblogic700-ejb-jar.dtd'>

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>TraderService</ejb-name>
    <jndi-name>webservices-complex-statelesssession</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Assembling the EJB

To assemble the EJB class files and deployment descriptors into a `trader.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 `pre_trader.jar` archive file using the `jar` utility:

   ```
   jar cvf pre_trader.jar -C staging_dir .
   ```

6. Run the `weblogic.ejbc` utility to generate and compile EJB 2.0 and 1.1 container classes and create the final `trader.jar` file:

   ```
   java weblogic.ejbc pre_trader.jar trader.jar
   ```

Creating the build.xml Ant Build File

The Ant build file, `build.xml`, contains a call to the `servicegen` Ant task that introspects the `trader.jar` EJB file, generates all data type components (such as the serialization class), creates the `web-services.xml` deployment descriptor file, and packages it all up into a deployable `trader.ear` file.

The following `build.xml` file contains instructions that will build the EAR file into a temporary `build_dir` directory :

```
<project name="webServicesExample" default="build">
  <target name="build" >
    <delete dir="build_dir" />
    <mkdir dir="build_dir" />
    <copy todir="build_dir" file="trader.jar"/>
    <servicegen
      destEar="build_dir/trader.ear"
```

```
      warName="trader.war"
      contextURI="webservice">
      <service
         ejbJar="build_dir/trader.jar"
         targetNamespace="http://www.bea.com/examples/Trader"
         serviceName="TraderService"
         serviceURI="/TraderService"
         generateTypes="True"
         expandMethods="True" >
      </service>
   </servicegen>
  </target>
</project>
```

4 Designing WebLogic Web Services

The following sections discuss design issues you should consider before implementing WebLogic Web services:

- "Choosing Between Synchronous or Asynchronous Operations" on page 4-1
- "Choosing the Backend Components of Your Web Service" on page 4-2
- "RPC-Oriented or Document-Oriented Web Services?" on page 4-4
- "Data Types" on page 4-5
- "Using SOAP Message Handlers to Intercept the SOAP Message" on page 4-6
- "Stateful WebLogic Web Service" on page 4-7

Choosing Between Synchronous or Asynchronous Operations

WebLogic Web service operations can be either synchronous request-response or asynchronous one-way.

Synchronous request-response (the default behavior) means that every time a client application invokes a Web service operation, it receives a SOAP response, even if the method that implements the operation returns `void`. Asynchronous one-way means that the client never receives a SOAP response, even a fault or exception.

You specify this type of behavior with the `invocation-style` attribute of the `<operation>` element in the `web-services.xml` file.

Web service operations are typically synchronous request-response, mirroring typical RPC-style behavior. Sometimes, however, you might want to implement asynchronous behavior if your client application has no need for a response, even in the case of an error. When designing asynchronous one-way Web service operations, keep the following issues in mind:

- The backend component that implements the operation *must* explicitly return `void`.

- You cannot specify out or in-out parameters to the operation, you can only specify in parameters.

Choosing the Backend Components of Your Web Service

You implement a WebLogic Web service operation with one of the following types of backend component:

- a method of a stateless session EJB

- a method of a Java class

- a JMS message consumer or producer. For details, see Chapter 12, "Creating JMS-Implemented WebLogic Web Services."

EJB Backend Component

Web service operations implemented with a method of a stateless session EJB are interface driven, which means that the business methods of the underlying stateless session EJB determine how the Web service operation works. When clients invoke the Web service operation, they send parameter values to the method, which executes and sends back the return value.

Use a stateless session EJB backend component if your application has the following characteristics:

■ The behavior of the Web service can be expressed as an interface.

■ The Web service is process-oriented rather than data-oriented.

■ The Web service can benefit from the facilities of EJBs, such as persistence, security, transactions, and concurrency .

Examples of this type of Web service operation implementation 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.

Java Class Backend Component

Web service operations implemented with Java classes are similar to those implemented with an EJB method.   Creating a Java class, however, is often simpler and faster than creating an EJB. Use a Java class as a backend component when you do not need overhead of EJB facilities such as persistence, security, transactions, and concurrency.

There are limitations and restrictions to using a Java class as a backend component, however. For details, see "Implementing a Web Service By Writing a Java Class" on page 5-4.

RPC-Oriented or Document-Oriented Web Services?

The operations of a WebLogic Web service can be either RPC-oriented or document-oriented. As described in the WSDL 1.1 specification, an RPC-oriented operation is one in which the SOAP messages contain parameters and return values, and a document-oriented operation is one in which the SOAP messages contain XML documents.

The method that implements a document-oriented WebLogic Web service operation can have only *one* parameter, of any supported data type. There are no restrictions on the number of parameters of an RPC-oriented operation.

RPC-oriented WebLogic Web service operations use SOAP encoding. Document-oriented WebLogic Web service operations use literal encoding.

All operations in a single WebLogic Web service must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web service.

By default, the operations of a WebLogic Web service are RPC-oriented. If you want to specify that the operations are document-oriented, use the `style="document"` attribute of the `<service>` element when assembling a Web service using the `servicegen` Ant task. The generated `web-services.xml` deployment descriptor will contain a corresponding `style="document"` attribute for the appropriate `<web-service>` element.

For information on implementing document-oriented WebLogic Web services, see "Implementing a Document-Oriented Web Service" on page 5-6. For details on using the `servicegen` Ant task to assemble a document-oriented Web service, see "Assembling WebLogic Web Services Using the servicegen Ant task" on page 6-3 and "servicegen" on page B-19.

Data Types

WebLogic Web services support both built-in and non-built-in data types as parameters and return values to Web services operations. This means that WebLogic Web services can handle any type of data that can be represented using XML Schema.

Built-in data types are those specified by the JAX-RPC specification. If your Web service uses only built-in data types, the conversion of the data between its XML and Java representation is handled automatically by WebLogic Server. For the full list of built-in data types, see "Using Built-In Data Types" on page 5-12.

If, however, your Web service operation is more complex and uses a non-built-in data type as a parameter or return value, you must:

    a. Create the serialization class that convert the data between its XML and Java representation

    b. Describe the XML representation of the data type (using XML Schema notation) in the `web-services.xml` file

    c. Create the Java class file of the data type

    d. Describe the data type mapping in the `web-services.xml` file

WebLogic Server includes Ant tasks that perform these tasks for many common XML and Java data types; this feature is called *autotyping*. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13. For information on running these Ant tasks, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

Note: If you are using the autotyping Ant tasks to generate data type information for a Java class, your class must conform to the guidelines described in "Implementing Non-Built-In Data Types" on page 5-5.

If your data type is not either built-in or one of the supported non-built-in data types, then you must create the serialization class, and so on, manually. For details, see Chapter 9, "Using Non-Built-In Data Types."

Using SOAP Message Handlers to Intercept the SOAP Message

Some Web services need access to the SOAP message, for which you can create SOAP message handlers.

A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web service. You can create handlers in both the Web service itself and the client applications that invoke the Web service.

A simple example of using handlers is to encrypt and decrypt secure data in the body of a SOAP message. A client application uses a handler to encrypt the data before it sends the SOAP message request to the Web service. The Web service receives the request and uses a handler to decrypt the data before it sends the data to the back-end component that implements the Web service. The same steps happen in reverse for the response SOAP message.

Another example is accessing information in the header part of the SOAP message. You can use the SOAP header to store Web service specific information and then use handlers to manipulate it.

You can also use SOAP message handlers to improve the performance of your Web service. After your Web service has been deployed for a while, you might discover that many consumers invoke it with the same parameters. You could improve the performance of your Web service by caching the results of popular invokes of the Web service (assuming the results are static) and immediately returning these results when appropriate, without ever invoking the back-end components that implement the Web service. You implement this performance improvement by using handlers to check the request SOAP message to see if it contains the popular parameters.

Stateful WebLogic Web Service

You implement a WebLogic Web service operation using stateless session EJBs or Java classes, and thus a WebLogic Web service operation is not stateful, or one that can conduct a back and forth conversation beyond the standard request/response model.

You can, however, mimic a conversational Web service by using JDBC or entity beans. For example, you could design a Web service so that client applications that invoke it pass a unique ID to identify themselves to the stateless session EJB entry point. This EJB uses the ID to persist the conversation in some kind of persistent storage, using either entity beans or JDBC. The next time the same client application invokes the Web service, the stateless session EJB can recover the previous state of the conversation by selecting the persisted data using the unique ID.

For information on programming entity beans, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/index.html. For information on JDBC, see WebLogic jDrivers at http://e-docs.bea.com/wls/docs70/jdrivers.html.

5 Implementing WebLogic Web Services

The following sections describe how to implement WebLogic Web services:

Overview of Implementing a WebLogic Web Service

Implementing a WebLogic Web service refers to writing the Java code for the backend components that make up the Web service and optionally creating SOAP message handlers. Backend components include stateless session EJBs, Java classes, and JMS message consumers and producers. A Web service can be implemented with multiple combinations of these components.

A single WebLogic Web service consists of one or more operations; you can implement each operation using methods of different backend components and SOAP message handlers. For example, an operation might be implemented with a single method of a stateless session EJB or with a combination of SOAP message handlers and a method of a stateless session EJB.

If you are implementing a WebLogic Web service from an existing WSDL file, you can use the WebLogic Server `wsdl2Service` Ant task to generate much of the needed Java source code. You can use this generated file as a starting point, then add the rest of the Java code needed to make the Web service behave as you want.

It is assumed that you have read and understood the design issues discussed in Chapter 4, "Designing WebLogic Web Services," designed your Web service and that you know the types of components you need to create.

Implementing a WebLogic Web Service: Main Steps

The following procedure describes the high-level steps to implement a WebLogic Web service. Later parts of this document describe the steps in more detail. Although some of the steps are mandatory, others are optional, depending on the type of Web service you are implementing.

1. Write the Java code for the back-end components that make up the Web service.

   See "Writing the Java Code for the Components" on page 5-3.

2. If you need to process information in the SOAP request or response or access the SOAP attachments, create SOAP message handlers and handler chains.

   See Chapter 10, "Creating SOAP Message Handlers to Intercept the SOAP Message."

3. If your backend components use non-built-in data types as parameters or return values, generate or create the serialization class that converts the data between XML and Java.

   See "Implementing Non-Built-In Data Types" on page 5-5.

Writing the Java Code for the Components

When you implement a WebLogic Web service, you write Java code for one of these backend components:

- A stateless session EJB.

  See "Implementing a Web Service By Writing a Stateless Session EJB" on page 5-4 for information on writing the Java code. For an example, see "Writing the Java Code for the EJB" on page 3-4.

- A Java class.

  See "Implementing a Web Service By Writing a Java Class" on page 5-4 for information on writing the Java code.

- A JMS message consumer or producer, typically a message-driven bean.

  See Chapter 12, "Creating JMS-Implemented WebLogic Web Services."

If your Web service operations use non-built-in data types as parameters or return values, see "Implementing Non-Built-In Data Types" on page 5-5.

If you are implementing a Web service that uses document-oriented operations, rather than RPC-oriented, see "Implementing a Document-Oriented Web Service" on page 5-6.

If you are implementing a WebLogic Web service based on an existing WSDL file, and you want to implement the Web service with a Java class, use the WebLogic Server wsdl2Service Ant task to generate much of the needed Java code. For details about using this Ant task, see "Generating a Partial Implementation From a WSDL File" on page 5-7.

For information on throwing exceptions from your Web service implementation, see "Throwing SOAP Fault Exceptions" on page 5-11.

If you want your Web service operation to return multiple values, see "Implementing Multiple Return Values" on page 5-8.

Implementing a Web Service By Writing a Stateless Session EJB

Writing the Java code for the stateless session EJB for a Web service is no different from writing a stand-alone EJB, except for the following issues:

■ You can specify in the `web-services.xml` deployment descriptor that a Web service operation is *one-way*, which means that the client application that invokes the Web service does not wait for a response. When you write the Java code for the EJB method that implements this type of operation, you *must* specify that it return `void`.

For more information on specifying in the `web-services.xml` file that a Web service operation is one-way, see "operation" on page A-8.

■ If the data type of the parameters or return value of an EJB method are not part of the set of built-in data types, then you must generate or create the serialization class that converts these data types between their XML and Java representations. For the list of built-in data types, see "Using Built-In Data Types" on page 5-12

See "Implementing Non-Built-In Data Types" on page 5-5.

For an example of how to write a stateless session EJB, see "Writing the Java Code for the EJB" on page 3-4. For general information, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/index.html.

Implementing a Web Service By Writing a Java Class

You can implement a Web service operation using a Java class as long as you follow these rules:

■ Do not start any threads. This rule applies to all Java code that runs on WebLogic Server.

■ Define a default no-argument constructor.

■ Define as public the methods of the Java class that are going to be exposed as Web service operations.

- Write thread safe Java code, because WebLogic Server maintains just a single instance of a Java class that implements a Web service operation, and each invoke of the Web service uses this same instance.

For an example of implementing a WebLogic Web service operation with a Java class, go to the
`WL_HOME\samples\server\src\examples\webservices\basic\javaclass`
directory, where `WL_HOME` refers to the main directory of your WebLogic Server installation.

Implementing Non-Built-In Data Types

Stateless session EJBs or Java classes do not necessarily take built-in data types as parameters and return values, but rather, might use a Java data type that you create yourself. An example of a non-built-in data type is `TradeResult`, which has two fields: a String stock symbol and an integer number of shares traded. For the list of built-in data types, see "Using Built-In Data Types" on page 5-12.

If your backend components use non-built-in data types as parameters or return values, you must generate or create the serialization class that converts the data between XML and Java.You can do this in one of two ways:

- Use WebLogic Server `servicegen` or `autotype` Ant tasks to introspect your EJB and automatically generate the serialization class. These Ant tasks also create the corresponding XML Schema to represent your data in XML format and update your `web-services.xml` deployment descriptor file with the relevant data type mapping information. You will run these Ant tasks as part of assembling of the Web service, described in "Running the servicegen Ant Task" on page 6-5 and "Running the autotype Ant Task" on page 6-8.

  Warning: The serializer class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-17.

- Create the serialization class and XML and Java representations of your data type manually. This method is more complex and time-consuming than generating them using the Ant task. For details on handling non-built-in data types manually, see Chapter 9, "Using Non-Built-In Data Types."

If you are going to create the XML representation of your Java data type manually, along with the serialization class, you can code the Java class any way you want, because you will be writing all the conversion code yourself.

If you are going to use the `servicegen` or `autotype` Ant tasks to automatically generate the data type components, follow these requirements when writing the Java class for your data type:

- Define a default constructor, which is a constructor that takes no parameters.

- Define both `getXXX()` and `setXXX()` methods for each member variable which you want to expose.

- Make the data type of each exposed member variable one of the built-in data types, or a non-built-in data type that consists of built-in data types and has the corresponding serialization class and XML Schema representation.

The `servicegen` and `autotype` Ant tasks can generate data type components for most common XML and Java data types. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.

Implementing a Document-Oriented Web Service

When creating a WebLogic Web service, you can specify whether the Web service is document-oriented (the SOAP message contains a document) or RPC-oriented (the SOAP message contains parameters and return values).

If you create a document-oriented Web service:

- the methods that implement each operation of the Web service can have only *one* parameter. This single parameter can be of any supported data type; see "Data Types" on page 4-5 for more information.

- the methods that implement each operation cannot use out and in-out parameters.

Generating a Partial Implementation From a WSDL File

The `wsdl2Service` Ant task takes as input an existing WSDL file, and generates a Java source file that partially implements the Web service.

The Java source file contains a template for the full Java class-implemented WebLogic Web service. The template includes full method signatures that correspond to the operations in the WSDL file. You write the actual code for these methods so that they function as you want, following the guidelines in "Implementing a Web Service By Writing a Java Class" on page 5-4.

The `wsdl2Service` Ant task generates a partial implementation for only one Web service in a WSDL file (specified by the `<service>` element.) Use the `serviceName` attribute to specify a particular service; if you do not specify this attribute, the `wsdl2Service` Ant task generates a partial implementation for the first `<service>` element in the WSDL.

Running the wsdl2Service Ant Task

To run the `wsdl2Service` Ant task, follow these steps:

1. Create a file called `build.xml` that contains a call to the `wsdl2Service` Ant task. For details, see "Sample build.xml Files for the wsdl2Service Ant Task" on page 5-8.

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory `WL_HOME\server\bin`, where `WL_HOME` is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory `WL_HOME/server/bin`, where `WL_HOME` is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `wsdl2Service` Ant task, see "wsdl2Service" on page B-31.

Sample build.xml Files for the wsdl2Service Ant Task

The following example shows a simple `build.xml` file:

```
<project name="buildWebservice" default="generate-from-WSDL">
   <target name="generate-from-WSDL">
      <wsdl2service
            wsdl="c:\wsdls\myService.wsdl"
            destDir="c:\myService\implementation"
            typeMappingFile="c:\autotype\types.xml"
            packageName="example.ws2j.service" />
   </target>
</project>
```

In the example, the `wsdl2Service` Ant task generates a Java source file for the first `<service>` element it finds in the WSDL file `c:\wsdls\myService.wsdl`. It uses data type mapping information for any non-built-in data types from the `c:\autotype\types.xml` file; typically you have previously run the `autotype` Ant task to generate this file. The generated Java source file is in the package `example.ws2j.service`.

Implementing Multiple Return Values

WebLogic Web service operations typically return a single value: the return value of the EJB or Java class method that implements the Web service operation. If you want a Web service operation to return multiple values, you can:

- define the data type of the return value to be a complex type, such as an object with multiple parts or an array.

- specify that one or more of the parameters of the Web service operation be *out* or *in-out* parameters.

Out and in-out parameters are a mechanism whereby parameters to an operation can act as both standard in parameters *and* return values. The Out parameters are undefined when the operation is invoked but defined by the method that implements the operation when the operation completes; in-out parameters are defined when invoked and when completed. For example, assume a Web service operation contains one out parameter, and the operation is implemented with an EJB method. The EJB method sets the value of the out parameter and sends this value back to the client application that invoked it. The client application can then access the value of this out parameter as if it were a return value. An in-out parameter is one that acts as both a standard input parameter

for sending information to the method and an out parameter. This section discusses how to implement a Web service operation with an EJB or Java class method that uses out or in-out parameters.

The following example shows a method whose second parameter is an in-out parameter:

```
public String myMethod( String param1,
        javax.xml.rpc.holders.IntHolder intHolder ) {

System.out.println ("The input value is: " + intHolder.value );
intHolder.value = 20;  // the new value of the out parameter

return param1;
}
```

You invoke the method with two parameters, a String and an integer. The method returns two values: a String (the standard return value) and an integer (via the `IntHolder` holder parameter).

Out and in-out parameters must implement the `javax.xml.rpc.holders.Holder` interface. Use the `Holder.value` field to first access the input value of an in-out parameter and then set the value of out and in-out parameters.   In the preceding example, assume the method was invoked with a value of 40 as the second parameter; when the method completes, the value of `intHolder` is now 20.

Using Holder Classes to Implement Multiple Return Values

If the out or in-out parameter is a standard data type, you can use one of the JAX-RPC `Holder` classes, listed in the following table.

Table 5-1 Built-In Holder Classes Provided by WebLogic Server

Built-In Holder Class	Java Data Type That It Holds
javax.xml.rpc.holders.BooleanHolder	boolean
javax.xml.rpc.holders.ByteHolder	byte
javax.xml.rpc.holders.ShortHolder	short
javax.xml.rpc.holders.IntHolder	int
javax.xml.rpc.holders.LongHolder	long
javax.xml.rpc.holders.FloatHolder	float

Table 5-1 Built-In Holder Classes Provided by WebLogic Server

Built-In Holder Class	Java Data Type That It Holds
javax.xml.rpc.holders.DoubleHolder	double
javax.xml.rpc.holders.BigDecimalHolder	java.math.BigDecimal
javax.xml.rpc.holders.BigIntegerHolder	java.math.BigInteger
javax.xml.rpc.holders.ByteArrayHolder	byte[]
javax.xml.rpc.holders.CalendarHolder	java.util.Calendar
javax.xml.rpc.holders.QnameHolder	javax.xml.namespace.QName
javax.xml.rpc.holders.StringHolder	java.lang.String

If, however, the data type of the parameter is not provided, you must create your own implementation.

To create your own implementation of the `javax.xml.rpc.holders.Holder` interface, follow these guidelines:

- Name your implementation class *Type*Holder, where *Type* is the name of the complex type. For example, if your complex type is called Person, then your implementation class is called PersonHolder.

- The Holder implementation class should be packaged in a holders sub-package below the package of the class it is holding.

  For example, if your complex type Person is in the examples.webservices package, then the PersonHolder implementation class should be in the examples.webservices.holders package.

- Create a public field called value, whose data type is the same as that of the parameter.

- Create a default constructor that initializes the value field to a default value.

- Create a constructor that sets the value field to the value of the passed parameter.

The following example shows the outline of a PersonHolder implementation class:

```
package examples.webservices.holders;
```

```
public final class PersonHolder implements
           javax.xml.rpc.holders.Holder {

    public Person value;

    public PersonHolder() {

        // set the value variable to a default value

    }

    public PersonHolder (Person value) {

        // set the value variable to the passed in value

    }

}
```

Throwing SOAP Fault Exceptions

If you throw a `javax.xml.rpc.soap.SOAPFaultException` exception in your stateless session EJB or Java class, WebLogic Server maps it to a SOAP fault and sends it to the client application that invokes the operation.

The following excerpt describes the `SOAPFaultException` class:

```
public class SOAPFaultException extends java.lang.RuntimeException {
    public SOAPFaultException (QName faultcode,
                               String faultstring,
                               String faultactor,
                               javax.xml.soap.Detail detail ) {...}
    public Qname getFaultCode() {...}
    public String getFaultString() {...}
    public String getFaultActor() {...}
    public javax.xml.soap.Detail getDetail() {...}
}
```

If your EJB or Java class throws any other type of Java exception, WebLogic Server tries to map it to a SOAP fault as best it can. However, to ensure that the client application receives the best possible exception information, you should explicitly throw a `SOAPFaultException` exception or one that extends the exception.

Use the `weblogic.webservice.util.FaultUtil.newDetail()` WebLogic Web Service API to create the `javax.xml.soap.Detail` object, which is a JAX-RPC container for `DetailEntry` objects that provide detailed application-specific

information about the error. You can use the `javax.xml.soap.Detail.addDetailEntry()` method to add a `DetailEntry` to the `Detail` object.

The following excerpt shows an example of creating and throwing a `SOAPFaultException` from within the implementation of your Web Service:

```
  throw new SOAPFaultException(
      new QName( "http://schemas.xmlsoap.org/soap/envelope/", "Server" ),
      "info on the fault",
      "info on the actor",
      weblogic.webservice.util.FaultUtil.newDetail() );
}
```

Warning: If you create and throw your own exception (rather than use `SOAPFaultException`) and two or more of the properties of your exception class are of the same data type, then you must also create setter methods for these properties, even though the JAX-RPC specification does not require it. This is because when a WebLogic Web Service receives the exception in a SOAP message and converts the XML into the Java exception class, there is no way of knowing which XML element maps to which class property without the corresponding setter methods.

Using Built-In Data Types

The following sections describe the built-in data types supported by WebLogic Web services and the mapping between their XML and Java representations. As long as the data types of the parameters and return values of the backend components that implement your Web service are in the set of built-in data types, WebLogic Server automatically converts the data between XML and Java.

If, however, you use non-built-in data types, then you must create the serialization class to convert the data between XML and Java. WebLogic Server includes the `servicegen` and `autotype` Ant tasks that can generate the serialization class for most non-built-in data types. See "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13 for a list of supported XML and Java data types. For more information about using `servicegen` and `autotype`, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

If your data type is not supported, then you must create your serialization class manually. For details, see Chapter 9, "Using Non-Built-In Data Types."

XML Schema-to-Java Mapping for Built-In Data Types

The following table lists the defined mappings for all built-in data types defined by XML Schema (target namespace `http://www.w3.org/2001/XMLSchema`) and the corresponding SOAP data types (target namespace `http://schemas.xmlsoap.org/soap/encoding/`).

For a list of the supported non-built-in XML data types, see "Supported XML Non-Built-In Data Types" on page 6-14.

Table 5-2 XML Schema-to-Java Mapping for Built-In Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
boolean	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
integer	java.math.BigInteger
decimal	java.math.BigDecimal
string	java.lang.String
dateTime	java.util.Calendar
base64Binary	byte[]
hexBinary	byte[]

Table 5-2 XML Schema-to-Java Mapping for Built-In Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
duration	weblogic.xml.schema.binding.util.Duration
time	java.util.Calendar
date	java.util.Calendar
gYearMonth	java.util.Calendar The java.util.Calendar Java data type contains more fields than the gYearMonth data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gYear	java.util.Calendar The java.util.Calendar Java data type contains more fields than the gYearMonth data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gMonthDay	java.util.Calendar The java.util.Calendar Java data type contains more fields than the gYearMonth data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gDay	java.util.Calendar The java.util.Calendar Java data type contains more fields than the gYearMonth data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gMonth	java.util.Calendar The java.util.Calendar Java data type contains more fields than the gYearMonth data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
anyURI	java.lang.String
NOTATION	java.lang.String

Table 5-2 XML Schema-to-Java Mapping for Built-In Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
token	java.lang.String
normalizedString	java.lang.String
language	java.lang.String
Name	java.lang.String
NMTOKEN	java.lang.String
NCName	java.lang.String
NMTOKENS	java.lang.String[]
ID	java.lang.String
IDREF	java.lang.String
ENTITY	java.lang.String
IDREFS	java.lang.String[]
ENTITIES	java.lang.String[]
nonPositiveInteger	java.math.BigInteger
nonNegativeInteger	java.math.BigInteger
negativeInteger	java.math.BigInteger
unsignedLong	java.math.BigInteger
positiveInteger	java.math.BigInteger
unsignedInt	long
unsignedShort	int
unsignedByte	short
Qname	javax.xml.namespace.QName

Java-to-XML Mapping for Built-In Data Types

For a list of the supported non-built-in Java data types, see "Supported Java Non-Built-In Data Types" on page 6-16.

Table 5-3 Java-to-XML Mapping for Built-In Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Data Type
int	int
short	short
long	long
float	float
double	double
byte	byte
boolean	boolean
char	string (with facet of length=1)
java.lang.Integer	int
java.lang.Short	short
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double
java.lang.Byte	byte
java.lang.Boolean	boolean
java.lang.Character	string (with facet of length=1)
java.lang.String	string
java.math.BigInteger	integer
java.math.BigDecimal	decimal

Table 5-3 Java-to-XML Mapping for Built-In Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Data Type
java.lang.String	string
java.util.Calendar	dateTime
java.util.Date	dateTime
byte[]	base64Binary
weblogic.xml.schema.binding.util.Duration	duration
javax.xml.namespace.QName	Qname

6 Assembling WebLogic Web Services Using Ant Tasks

The following sections describe how to assemble and deploy WebLogic Web services using a variety of Ant tasks:

Overview of Assembling WebLogic Web Services Using Ant Tasks

Assembling a WebLogic Web service refers to gathering all the components of the service (such as the EJB JAR file, the SOAP message handler classes, and so on), generating the web-services.xml deployment descriptor file, and packaging everything into an Enterprise Application Archive (EAR) file that can be deployed on WebLogic Server.

There are two ways to assemble a WebLogic Web service using Ant tasks:

■ Using the servicegen Ant task, which performs all assembly steps for you.

The servicegen Ant takes as input an EJB JAR file (for EJB-implemented Web services) or a list of Java classes (for Java class-implemented Web services), and based on information after introspecting the Java code and the attributes of the Ant task, it automatically generates all the components that make up a WebLogic Web service and packages them into an EAR file.

For detailed information, see "Assembling WebLogic Web Services Using the servicegen Ant task" on page 6-3.

■ Using a variety of narrowly-focused Ant tasks, such as autotype, source2wsdd, and so on.

Typically, the servicegen Ant task is adequate for assembling most WebLogic Web services. If, however, you want more control over how your Web service is assembled, you can use a set of narrowly-focused Ant tasks instead. For example, you can use the source2wsdd to generate the web-services.xml file, and then you can update this file manually if you want to add more information.

For detailed information, see "Assembling WebLogic Web Services Using Other Ant Tasks" on page 6-6.

For detailed reference information on the Web services Ant tasks, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

Note: The Java Ant utility included in WebLogic Server uses the `ant` (UNIX) or `ant.bat` (Windows) configuration files in the *WL_HOME*\server\bin directory when setting the ANTCLASSPATH variable, where *WL_HOME* is the top-level directory of your WebLogic Platform installation   If you need to update the ANTCLASSPATH variable, make the relevant changes to the appropriate file for your operating system.

Assembling WebLogic Web Services Using the servicegen Ant task

The `servicegen` Ant task takes as input an EJB JAR file or list of Java classes and creates all the needed Web service components and packages them into a deployable EAR file.

What the servicegen Ant Task Does

In particular, the `servicegen` Ant task:

- Introspects the Java code, looking for public methods to convert into Web service operations and non-built-in data types used as parameters or return values of the methods.

- Creates a `web-services.xml` deployment descriptor file, based on the attributes of the `servicegen` Ant task and introspected EJB or Java class information.

- Optionally creates the serialization class that convert the non-built-in data between its XML and Java representations.   It also creates XML Schema representations of the Java objects and updates the `web-services.xml` file accordingly.   For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.

- Packages all the Web service components into a Web application WAR file, then packages the WAR and EJB JAR files into a deployable EAR file.

Assembling WebLogic Web Services Automatically: Main Steps

To assemble a Web service automatically using the servicegen Ant task:

1. Set your environment.

   On Windows NT, execute the setWLSEnv.cmd command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the setWLSEnv.sh command, located in the directory *WL_HOME*/server/bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

2. Create a staging directory to hold the components of your Web service.

3. If the Web service operations are implemented with EJBs, package them, along with any supporting EJBs, into an EJB JAR file. If the operations are implemented with Java classes, compile them into class files.

   For detailed information, refer to *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs70/programming/packaging.html.

4. Copy the EJB JAR file and/or Java class files to the staging directory.

5. In the staging directory, create the Ant build file (called build.xml by default) that contains a call to the servicegen Ant task.

   For details about specifying the servicegen Ant task, see "Running the servicegen Ant Task" on page 6-5.

   For general information about creating Ant build files, see http://jakarta.apache.org/ant/manual/.

6. Execute the Ant task or tasks specified in the build.xml file by typing ant in the staging directory, optionally passing the command a target argument:

   prompt> ant

   The Ant task generates the Web services EAR file in the staging directory which can then deploy on WebLogic Server.

Running the servicegen Ant Task

The following sample `build.xml`, file taken from the `examples.webservices.basic.statelessession` product example, specifies that you will run the `servicegen` Ant task:

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
      destEar="ws_basic_statelessSession.ear"
      contextURI="WebServices" >
      <service
        ejbJar="HelloWorldEJB.jar"
        targetNamespace="http://www.bea.com/webservices/basic/statelesSession"
        serviceName="HelloWorldEJB"
        serviceURI="/HelloWorldEJB"
        generateTypes="True"
        expandMethods="True"
        style="rpc" >
      </service>
    </servicegen>
  </target>
</project>
```

In the example, the `servicegen` Ant task creates one Web service called `HelloWorldEJB`. The URI to identify this Web service is `/HelloWorldEJB`; the full URL to access the Web service is

```
    http://host:port/WebServices/HelloWorldEJB
```

The `servicegen` Ant task packages the Web service in an EAR file called `ws_basic_statelessSession.ear`. The EAR file contains a WAR file called `web-services.war` (default name) that contains all the Web service components, such as the `web-services.xml` deployment descriptor file.

Because the `generateTypes` attribute is set to `True`, the WAR file also contains the serialization class for any non-built-in data types used as parameters or return values to the EJB methods. The Ant task introspects the EJBs contained in the `HelloWorldEJB.jar` file, looking for public operations and non-built-in data types, and updates the `web-services.xml` operation and data type mapping sections accordingly. Because the `expandMethods` attribute is also set to `True`, the Ant task lists each public EJB method as a separate operation in the `web-services.xml` file.

The `style="rpc"` attribute specifies that the operations in the Web service are all RPC-oriented. If the operations in your Web service are document-oriented, specify `style="document"`.

Assembling WebLogic Web Services Using Other Ant Tasks

Typically, the `servicegen` Ant task is adequate for assembling most WebLogic Web services. If, however, you want more control over how your Web service is assembled, you can use a set of narrowly-focused Ant tasks instead. For example, you can use the `source2wsdd` to generate the `web-services.xml` file, and then you can update this file manually if you want to add more information.

To assemble a WebLogic Web service using Ant tasks other than `servicegen`:

1. Package or compile the backend components that implement the Web service into their respective packages. For example, package stateless session EJBs into an EJB JAR file and Java classes into class files.

   For detailed instructions, see *WebLogic Server Application Packaging* at http://e-docs.bea.com/wls/docs70/programming/packaging.html.

2. Create the Web service deployment descriptor file (`web-services.xml`).

   If you implemented your Web service with a Java class, you can use the `source2wsdd` Ant task to generate a `web-services.xml` file. For details, see "Running the source2wsdd Ant Task" on page 6-7. If you used the `wsdl2Service` Ant task to generate a partial implementation of a Web service from an existing WSDL file, then the Ant task already generated a `web-services.xml` file for you.

   For all other cases, such as EJB-implemented Web services, you might have to create the `web-services.xml` file manually. See "Creating the web-services.xml File Manually: Main Steps" on page 7-4.

3. If your Web service uses non-built-in data types, create all the needed components, such as the serialization class, by using the `autotype` Ant task to generate these components automatically, as described in "Running the autotype Ant Task" on page 6-8.

4. Optionally create a client JAR file using the `clientgen` Ant task.

   See "Running the clientgen Ant Task" on page 6-10.

5. Package all components into a deployable EAR file by using the `wspackage` Ant task, as described in "Running the wspackage Ant task" on page 6-11.

Running the source2wsdd Ant Task

Use the `source2wsdd` Ant task to generate a `web-services.xml` deployment descriptor file from the Java source file that implements a Web service.

Note: You cannot use this Ant task to generate the `web-services.xml` file for an EJB-implemented Web service; you can only use it for Java class-implemented Web service.

To run the `source2wsdd` Ant task, follow these steps:

1. Create a file called `build.xml` that contains a call to the `source2wsdd` Ant task. See "Sample build.xml Files for the source2wsdd Ant Task."

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*/server/bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `source2wsdd` Ant task, see "source2wsdd" on page B-28.

Sample build.xml Files for the source2wsdd Ant Task

The following example shows a simple `build.xml` file:

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <source2wsdd
            javaSource="c:\source\MyService.java"
            typesInfo="c:\autotype\types.xml"
            ddFile="c:\ddfiles\web-services.xml"
            serviceURI="/MyService" />
</project>
```

In the example, the `source2wsdd` Ant task generates a `web-services.xml` file from the Java source file called `c:\source\MyService.java`. It uses non-built-in data type information from the `c:\autotype\types.xml` file; this information includes the XML Schema representation of non-built-in data types used as parameters or return values in your Web service, as well as data type mapping information that specifies the location of the serialization class, and so on. You typically generate this file using the `autotype` Ant task.

The `source2wsdd` Ant task outputs the generated deployment descriptor information into the file `c:\ddfiles\web-services.xml`. The URI of the Web service is `/MyService`, used in the full URL that invokes the Web service once it is deployed.

Running the autotype Ant Task

Use the `autotype` Ant task to generate non-built-in data type components, such as the serialization class. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.

To run the `autotype` Ant task, follow these steps:

1. Create a file called `build.xml` that contains a call to the `autotype` Ant task. For details, see "Sample build.xml Files for the Autotype Ant Task."

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*/`server/bin`, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

```
prompt> ant
```

For reference information about the `autotype` Ant task, see "autotype" on page B-7.

Sample build.xml Files for the Autotype Ant Task

The following example shows a simple `build.xml` file:

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
     <autotype  javatypes="mypackage.MyType"
                targetNamespace="http://www.foobar.com/autotyper"
                packageName="a.package.name"
                destDir="d:\output" />
   </target>
</project>
```

In the example, the `autotype` Ant task creates the non-built-in data type components for a Java class called `mypackage.MyType`. The package name used in the generated serialization class is `a.package.name`. The serialization Java class and XML schema inforamtion is generated and placed in the `d:\output` directory. The generated XML Schema and type-mapping information are in a file called `types.xml` in this output directory.

The following excerpt from a sample `build.xml` file shows another way to use the `autotype` task:

```
<autotype  wsdl="file:\wsdls\myWSDL"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

This example is similar to the first, except that instead of starting with a Java representation of a data type, the example starts with an XML Schema representation embedded within the WSDL of a Web service. In this case, the task generates the corresponding Java representation.

Running the clientgen Ant Task

To run the `clientgen` Ant task and automatically generate a client JAR file:

1. Create a file called `build.xml` that contains a call to the `clientgen` Ant task. For details, see "Sample build.xml File for the clientgen Ant Task."

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*/server/bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

   For reference information about the `clientgen` Ant task, see "clientgen" on page B-12.

Sample build.xml File for the clientgen Ant Task

The following example shows a simple `build.xml` file:

```
<project name="buildWebservice" default="generate-client">
   <target name="generate-client">
      <clientgen ear="c:/myapps/myapp.ear"
                 serviceName="myService"
                 packageName="myapp.myservice.client"
                 useServerTypes="True"
                 clientJar="c:/myapps/myService_client.jar" />
   </target>
</project>
```

In the example, the `clientgen` Ant task creates the `c:/myapps/myService_client.jar` client JAR file that contains the service-specific client interfaces and stubs and the serialization class used to invoke the WebLogic Web service called `myService` contained in the EAR file `c:/myapps/myapp.ear`. It packages the client interface and stub files into a package

called `myapp.myservice.client`. The `useServerTypes` attribute specifies that the `clientgen` Ant task should get the Java implementation of all non-built-in data types used in the Web service from the `c:/myapps/myapp.ear` file rather than generating Java code to implement the data types.

Running the wspackage Ant task

Use the `wspackage` Ant task to package the various components of a Web service into a deployable EAR file.

To run the `wspackage` Ant task, follow these steps:

1. Create a file called `build.xml` that contains a call to the `wspackage` Ant task. For details, see "Sample build.xml Files for the wspackage Ant Task."

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*`\server\bin`, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*`/server/bin`, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `wspackage` Ant task, see "wspackage" on page B-33.

Sample build.xml Files for the wspackage Ant Task

The following example shows a simple `build.xml` file for creating a deployable EAR file for a Java class-implemented Web service:

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <wspackage
           output="c:\myWebService.ear"
           contextURI="web_services"
```

```
           codecDir="c:\autotype"
           webAppClasses="example.ws2j.service.SimpleTest"
           ddFile="c:\ddfiles\web-services.xml" />
```

```
</project>
```

In the example, the `wspackage` Ant task creates an EAR file called
`c:\myWebService.ear`. The context URI of the Web service, used in the full URL
that invokes it, is `web_services`. The serializer class that contains the serializer class
for the non-built-in data types is located in the `c:\autotype` directory. The Java class
that implements the Web service is called `example.ws2j.service.SimpleTest` and
will be packaged in the `WEB-INF/classes` directory of the Web application. Finally,
the existing deployment descriptor file is `c:\ddfiles\web-services.xml`.

The Web Service EAR File Package

Web services are packaged into standard Enterprise Application EAR files that contain
a Web application WAR file along with the EJB JAR files.

The following graphic shows the hierarchy of a typical WebLogic Web services EAR file.

EAR file

└── META-INF (*Directory that contains standard application.xml file*)

    WAR file

        └── client.jar (*Downloadable client JAR file, one per Web service*)

        └── WEB-INF

            └── web-services.xml (*Web services deployment descriptor file*)

            └── classes   (*Directory that contains the serialization class, handler implementations, Java class components, and all other supporting classes.*)

            └── lib   (*Directory that contains JAR files of Java classes.*)

└── EJB JAR file   *(JAR file containing the EJBs that implement the Web service.)*

Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks

The tables in the following sections list the non-built-in XML and Java data types for which the `servicegen` and `autotype` Ant tasks can generate data type components, such as the serializer class, the Java or XML representation, and so on.

If your XML or Java data type is not listed in these tables, and it is not one of the built-in data types listed in "Using Built-In Data Types" on page 5-12, then you must create the non-built-in data type components manually. For details, see Chapter 9, "Using Non-Built-In Data Types."

Warning: The serializer class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-17.

For information on the ways that WebLogic Web Services are non-compliant with the JAX-RPC specification with respect to data types, see "Data Type Non-Compliance with JAX-RPC" on page 6-16.

Supported XML Non-Built-In Data Types

The following table lists the supported XML Schema non-built-in data types. If your XML data type is listed in the table, then the `servicegen` and `autotype` Ant tasks can generate the serializer class to convert the data between its XML and Java representations, as well as the Java representation and type mapping information for the `web-services.xml` deployment descriptor.

For details and examples of the data types, see the JAX-RPC specification.

Table 6-1 Supported Non-Built-In XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
Enumeration	Typesafe enumeration pattern. For details, see Section 4.2.4 of the JAX-RPC specification.
`<xsd:complexType>` with elements of both simple and complex types.	JavaBean
`<xsd:complexType>` with simple content.	JavaBean
`<xsd:attribute>` in `<xsd:complexType>`	Property of a JavaBean
Derivation of new simple types by restriction of an existing simple type.	Equivalent Java data type of simple type.

Table 6-1 Supported Non-Built-In XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
Facets used with restriction element. **Note:** The base primitive type must be one of the following: `string`, `decimal`, `float`, or `double`. Pattern facet is not enforced.	Restriction enforced during serialization and deserialization.
`<xsd:list>`	Array of the list data type.
Array derived from `soapenc:Array` by restriction using the `wsdl:arrayType` attribute.	Array of the Java equivalent of the `arrayType` data type.
Array derived from `soapenc:Array` by restriction.	Array of Java equivalent.
Derivation of a complex type from a simple type.	JavaBean with a property called `simpleContent` of type `String`.
`<xsd:anyType>`	`java.lang.Object`.
`<xsd:nil>` and `<xsd:nillable>` attribute	Java `null` value. If the XML data type is built-in and usually maps to a Java primitive data type (such as `int` or `short`), then the XML data type is actually mapped to the equivalent object wrapper type (such as `java.lang.Integer` or `java.lang.Short`).
Derivation of complex types by extension	Mapped using Java inheritance.
Abstract types	Abstract Java data type.

Supported Java Non-Built-In Data Types

The following table lists the supported Java non-built-in data types. If your Java data type is listed in the table, then the `servicegen` and `autotype` Ant tasks can generate the serializer class to convert the data between its Java and XML representations.

Table 6-2 Supported Non-Built-In Java Data Types

Java Data Type	Equivalent XML Schema Data Type
Array of any supported data type.	SOAP Array.
JavaBean whose properties are any supported data type.	<xsd:sequence>
java.util.List	SOAP Array.
java.util.ArrayList	SOAP Array.
java.util.LinkedList	SOAP Array.
java.util.Vector	SOAP Array.
java.util.Stack	SOAP Array.
java.lang.Object **Note:** The data type of the runtime object must be a known type: either a built-in data type or one that has type mapping information.	<xsd:anyType>
JAX-RPC-style enumeration class	<xsd:simpleType> with enumeration facets

Data Type Non-Compliance with JAX-RPC

The `autotype` Ant task does not comply with the JAX-RPC specification if the XML Schema data type (for which it is generating the Java representation) has all the following characteristics:

- The data type is a `complexType`.

- The `complexType` contains a single `sequence`.

- The `sequence` contains a single `element` with `maxOccurs` greater than 1 or unbounded.

The following example shows such an XML Schema data type:

```
<xsd:complexType name="Response">
   <xsd:sequence >
     <xsd:element name="code" type="xsd:string" maxOccurs="10" />
   </xsd:sequence>
</xsd:complexType>
```

The `autotype` Ant task maps this type of XML Schema data type directly to a Java array of the specified element. In the previous example, the `autotype` Ant task maps the `Response` XML Schema data type to a `java.lang.String[]` Java type. This is similar to the type of mapping that .NET does.

The JAX-RPC specification, in turn, states that this type of XML Schema data type should map to a Java array with a pair of setter and getter methods *in a JavaBean class*. WebLogic Web Services do not follow this last part of the specification.

Non-Roundtripping of Generated Data Type Components

When you use the `servicegen` or `autotype` Ant tasks to create the serializer class and Java or XML representation of non-built-in data types, it is very important to note that the process cannot be round-tripped. This means that if, for example, you use the `autotype` Ant task to generate the Java representation of an XML Schema data type, and then use `autotype` to create an XML Schema data type from the generated Java type, the original and generated XML Schema data type will not necessarily look the same, although they both describe the same XML data. This is also true if you start from Java, generate an XML Schema, then generate a new Java data type from the

generated XML Schema: the origianal and generated Java type will not necessarily look exactly the same. One possible difference, for example, is that the original and generated Java type might list the parameters of the constructor in a different order.

This behavior has a variety of repercussions. For example, assume you are developing a Web service from an existing stateless session EJB that uses non-built-in data types. You use the `autotype` Ant task to generate the serializer class and Java and XML representation of the data types and you use this generated code in your server-side code that implements your Web service. Later you use the `clientgen` Ant task to generate the Web service-specific client JAR file, which also includes a serializer class and the Java representation of the non-built-in data types. However, because `clientgen` by default generates these components from the WSDL of the Web service (and thus from an XML Schema), the `clientgen`-generated client-side Java representation might look different from the `autotype`-generated server-side Java code. This means that you might not necessarily be able to reuse any server-side code that handles the data type in your client application. If you want the `clientgen` Ant task to always use the generated serializer class and code from the WebLogic Web service EAR file, specify the `useServerTypes` attribute.

Deploying WebLogic Web Services

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 *WebLogic Server Deployment* at http://e-docs.bea.com/wls/docs70/programming/deploying.html.

7 Assembling a WebLogic Web Service Manually

The following sections provide information about assembling a WebLogic Web service manually:

Overview of Assembling a WebLogic Web Service Manually

Assembling a WebLogic Web service refers to gathering all the components of the service (such as the EJB JAR file, the SOAP message handler classes, and so on), generating the `web-services.xml` deployment descriptor file, and packaging everything into an Enterprise Application EAR file that can be deployed on WebLogic Server.

Typically you never assemble a WebLogic Web service manually, because the procedure is complex and time-consuming. Rather, use the WebLogic Ant tasks such as `servicegen`, `autotype`, `source2wsdd`, and so on to automatically generate all the needed components and package them into a deployable EAR file.

If, however, your Web service is so complex that the Ant tasks are not able to generate the needed components, or you want full control over all aspects of the Web service assembly, then use this chapter as a guide to assembling the Web service manually.

Assembling a WebLogic Web Service Manually: Main Steps

1. Package or compile the backend components that implement the Web service into their respective packages. For example, package stateless session EJBs into an EJB JAR file and Java classes into class files.

   For detailed instructions, see *WebLogic Server Application Packaging* at http://e-docs.bea.com/wls/docs70/programming/packaging.html.

2. Create the Web service deployment descriptor file (`web-services.xml`).

   For a description of the `web-services.xml` file, see "Overview of the web-services.xml File" on page 7-3. For detailed steps for creating the file manually, see "Creating the web-services.xml File Manually: Main Steps" on page 7-4.

3. If your Web service uses non-built-in data types, create all the needed components, such as the serialization class.

   For detailed information on creating these components manually, see Chapter 9, "Using Non-Built-In Data Types."

4. Package all components into a deployable EAR file.

   When packaging the EAR file manually, be sure to put the correct Web service components into a Web application WAR file. For details about the WAR and EAR file hierarchy, see "The Web Service EAR File Package" on page 6-12.

For instructions on creating WAR and EAR files, see *WebLogic Server Application Packaging* at http://e-docs.bea.com/wls/docs70/programming/packaging.html.

Overview of the web-services.xml File

The `web-services.xml` deployment descriptor file contains information that describes one or more WebLogic Web services, such as the backend components that implement the Web service; the non-built-in data types used as parameters and return values; the SOAP message handlers that intercept SOAP messages; and so on. As is true for all deployment descriptors, `web-services.xml` is an XML file.

Based on the contents of the `web-services.xml` deployment descriptor file, WebLogic Server dynamically generates the WSDL of a deployed WebLogic Web service. See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22 for details on getting the URL of the dynamically generated WSDL.

A single WebLogic Web service consists of one or more operations; you can implement each operation using methods of different backend components and SOAP message handlers. For example, an operation might be implemented with a single method of a stateless session EJB or with a combination of SOAP message handlers and a method of a stateless session EJB.

A single `web-services.xml` file contains a descritpion of at least one, and maybe more, WebLogic Web services.

If you are assembling a Web service manually (necessary, for example, is the service uses SOAP message handlers and handler chains), you need to create the `web-services.xml` file manually. If you assemble a WebLogic Web service with the `servicegen` Ant task, you do not need to create the `web-services.xml` file manually, because the Ant task generates one for you based on its introspection of the EJBs, the attributes of the Ant task, and so on.

Even if you need to manually assemble a Web service, you can use the `servicegen` Ant task to create a basic template, and then use this document to help you update the generated `web-services.xml` with the extra information that `servicegen` does not provide.

Creating the web-services.xml File Manually: Main Steps

The `web-services.xml` deployment descriptor file describes one or more WebLogic Web service. The file includes information about the operations that make up the Web services, the backend components that implement the operations, data type mapping information about non-built-in data types used as parameters and return values of the operations, and so on. See "Sample web-services.xml Files" on page 7-10 for complete examples of `web-services.xml` files that describe different kinds of WebLogic Web services. You can use any text editor to create the `web-services.xml` file.

For detailed descriptions of each element described in this section, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

The following example shows a simple `web-services.xml` file:

```
<web-services>
    <web-service name="stockquotes" targetNamespace="http://example.com"
              uri="/myStockQuoteService">
        <components>
            <stateless-ejb name="simpleStockQuoteBean">
                <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
            </stateless-ejb>
        </components>
        <operations>
            <operation method="getLastTradePrice"
                      component="simpleStockQuoteBean" />
        </operations>
    </web-service>
</web-services>
```

To create the `web-services.xml` file manually:

1. Create the root `<web-services>` element which contains all other elements:

    ```
    <web-services>
    ...
    </web-servives>
    ```

2. If one or more of your Web services include SOAP message handlers to intercept SOAP messages, create a `<handler-chains>` child element of the `<web-services>` root element and include all the relevant child elements to

describe the handlers in the handler chain, the order in which they should be invoked, and so on. For details, see "Updating the web-services.xml File with SOAP Message Handler Information" on page 10-16.

3. For each Web service you want to define, follow these steps:

    a. Create a `<web-service>` child element of the `<web-services>` element. Use the `name`, `targetNamespace`, and `uri` attributes to specify the name of the Web service, its target namespace, and the URI that clients will use to invoke the Web service, as shown in the following example:

    ```
    <web-service name="stockquote"
                 targetNamespace="http://example.com"
                 uri="myStockQuoteService">
    ...
    </web-service>
    ```

    To specify that the operations in your Web service are all document-oriented, use the `style="document"` attribute. The default value of the `style` attribute is `rpc`, which means the operations are all RPC-oriented.

    b. Create a `<components>` child element of the `<web-service>` element that lists the backend components that implement the operations of the Web service. For details, see "Creating the `<components>` Element" on page 7-6.

    c. If the operations in your Web service use non-built-in data types as parameters or return values, add data type mapping information by creating `<types>` and `<type-mapping>` child elements of the `<web-service>` element. For details, see "Creating the Data Type Mapping File" on page 9-11.

    Note: You do not have to perform this step if the operations of your Web service use only built-in data types as parameters or return values. See "Using Built-In Data Types" on page 5-12 for a list of the supported built-in data types.

    d. Create an `<operations>` child element of the `<web-service>` element that lists the operations that make up the Web service:

    ```
    <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ....
    </operations>
    ```

    e. Within the `<operations>` element, list the operations defined for the Web service. For details, see "Creating `<operation>` Elements" on page 7-7.

Creating the <components> Element

Use the `<components>` child element of the `<web-service>` element to list and describe the backend components that implement the operations of a Web service. Each backend component has a `name` attribute that you later use when describing the operation that the component implements.

Note: If you are creating a SOAP message handler-only type of Web service in which handlers and handler chains do all the work and never execute a backend component, you do not specify a `<components>` element in the `web-services.xml` file. For all other types of Web services you *must* declare a `<components>` element.

You can list one of the following types of backend components:

■ `<stateless-ejb>`

This element describes a stateless EJB backend component. Use either the `<ejb-link>` child element to specify the name of the EJB and the JAR file where it is located or the `<jndi-name>` child element to specify the JNDI name of the EJB, as shown in the following example:

```
<components>
  <stateless-ejb name="simpleStockQuoteBean">
     <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
  </stateless-ejb>
</components>
```

■ `<java-class>`

This element describes a Java class backend component. Use the `class-name` attribute to specify the fully qualified path name of the Java class, as shown in the following example:

```
<components>
   <java-class name="customClass"
               class-name="myclasses.MyOwnClass" />
</components>
```

Creating <operation> Elements

The <operation> element describes how the public operations of a WebLogic Web service are implemented. (The public operations are those that are listed in the Web service's WSDL and are executed by a client application that invokes the Web service.) The following example shows an <operation> declaration:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
   <params>
      <param name="in1" style="in" type="xsd:string" location="Header"/>
      <param name="in2" style="in" type="xsd:int"  location="Header"/>
      <return-param name="result" type="xsd:string"  location="Header"/>
   </params>
</operation>
```

Typically, every instance of an <operation> element in the web-services.xml file includes the name attribute which translates into the public name of the Web service operation. The only exception is when you use the method="*" attribute to specify all methods of an EJB or Java class in a single <operation> element; in this case, the public name of the operation is the name of the method.

Use the attributes of the <operation> element in combination to specify different kinds of operations. For details, see "Specifying the Type of Operation" on page 7-7.

Use the <params> element to optionally group together the parameters and return value of the operation. For details, see "Specifying the Parameters and Return Value of the Operation" on page 7-9.

Specifying the Type of Operation

Use the attributes of the <operation> element in different combination to identify the type of operation, the type of component that implements it, whether it is a one-way operation, and so on.

Note: For clarity, the examples in this section do not declare any parameters.

The following examples show how to declare a variety of different operations:

- To specify that an operation is implemented with just a method of a stateless session EJB, use the name, component, and method attributes, as shown in the following example:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
</operation>
```

- To specify with a single `<operation>` element that you want to include all the methods of an EJB or Java class, use the `method="*"` attribute; in this case, the public name of the operation is the name of the method:

```
<operation component="simpleStockQuoteBean"
           method="*">
</operation>
```

- To specify that an operation only receives data and does not return anything to the client application, add the `invocation-style` attribute:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote(java.lang.String)"
           invocation-style="one-way">
</operation>
```

The example also shows how to specify the full signature of a method with the `method` attribute. You only need to specify the full signature of a method if your EJB or Java class overloads the method and you thus need to unambiguously declare which method you are exposing as a Web service operation.

- To specify that an operation is implemented with a SOAP message handler chain and a method of a stateless session EJB, use the `name`, `component`, `method`, and `handler-chain` attributes:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote"
           handler-chain="myHandler">
</operation>
```

- To specify that an operation is implemented with just a SOAP message handler chain, use just the `name` and `handler-chain` attributes:

```
<operation name="justHandler"
           handler-chain="myHandler">

</operation>
```

Specifying the Parameters and Return Value of the Operation

Use the `<params>` element to explicitly declare the parameters and return values of the operation.

You do not have to explicitly list the parameters or return values of an operation. If an `<operation>` element does not have a `<params>` child element, WebLogic Server introspects the backend component that implements the operation to determine its parameters and return values. When generating the WSDL of the Web service, WebLogic Server uses the names of the corresponding method's parameters and return value.

You explicitly list an operation's parameters and return values when you need to:

- Make the name of the parameters and return values in the generated WSDL different from those of the method that implements the operation.

- Map a parameter to a name in the SOAP header request or response.

- Use out or in-out parameters.

Use the `<param>` child element of the `<params>` element to specify a single input parameter and the `<return-param>` child element to specify the return value. You must list the input parameters in the same order in which they are defined in the method that implements the operation. The number of `<param>` elements must match the number of parameters of the method. You can specify only one `<return-param>` element.

Use the attributes of the `<param>` and `<return-param>` elements to specify the part of the SOAP message where parameter is located (the body or header), the type of the parameter (in, out, or in-out), and so on. You must always specify the XML Schema data type of the parameter using the `type` attribute. The following examples show a variety of input and return parameters.

- To specify that a parameter is a standard input parameter, located in the header of the request SOAP message, use the `style` and `location` attributes as shown:

```
<param name="inparam" style="in"
       location = "Header" type="xsd:string" />
```

- Out and in-out parameters enable an operation to return more than one return value (in addition to using the standard `<return-value>` element.) The following sample `<param>` element shows how to specify that a parameter is an in-out parameter, which means that it acts as both an input and output parameter:

```
<param name="inoutparam" style="inout"
        type="xsd:int" />
```

Because the default value of the location attribute is Body, both the input and output parameter values are found in the body of the SOAP message.

- The following example shows how to specify a standard return value located in the header of the response SOAP message:

```
 <return-param name="result" location="Header"
                 type="xsd:string" />
```

Optionally use the <fault> child element of the <params> element to specify your own Java exception that is thrown if there is an error while invoking the operation. This exception will be thrown in addition to the java.rmi.RemoteException exception. For example:

```
<fault name="MyServiceException"
        class-name="my.exceptions.MyServiceException" />
```

Sample web-services.xml Files

The following sections describe sample web-services.xml files for the following types of WebLogic Web services:

- EJB Component Web Service With Built-In Data Types

- EJB Component Web Service With Non-Built-In Data Types

- EJB Component and SOAP Message Handler Chain Web Service

- SOAP Message Handler Chain-Only Web Service

EJB Component Web Service With Built-In Data Types

One kind of WebLogic Web service is implemented using a stateless session EJB whose parameters and return values are one of the built-in data types. The following Java interface is an example of such an EJB:

```
public interface SimpleStockQuoteService extends javax.ejb.EJBObject {
 public float getLastTradePrice(String ticker) throws java.rmi.RemoteException;
}
```

The `web-services.xml` deployment descriptor for a Web service implemented with this sample EJB can be as follows:

```xml
<web-services>
     <web-service name="stockquotes" targetNamespace="http://example.com"
                  uri="/myStockQuoteService">
         <components>
             <stateless-ejb name="simpleStockQuoteBean">
                 <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
             </stateless-ejb>
         </components>
         <operations>
             <operation method="getLastTradePrice"
                        component="simpleStockQuoteBean" />
         </operations>
     </web-service>
</web-services>
```

The example shows a Web service called `stockquotes`. The Web service is implemented with a stateless session EJB whose `<ejb-name>` in the `ejb-jar.xml` file is `StockQuoteBean` and is packaged in the EJB JAR file called `stockquoteapp.jar`. The internal name of this component is `simpleStockQuoteBean`. The Web service has one operation, called `getLastTradePrice`, the same as the EJB method name. The input and output parameters are inferred from the method signature and thus do not need to be explicitly specified in the `web-services.xml` file.

Note: The `servicegen` Ant task does not include the methods of `EJBObject` when generating the list of operations in the `web-services.xml` file.

The previous example shows how to explicitly list an operation of a Web service. You can, however, implicitly expose all the public methods of an EJB by including just one `<operation method="*">` element, as shown in the following example:

```xml
<operations>
     <operation method="*"
                component="simpleStockQuoteBean" />
</operations>
```

If your Web service supports only HTTPS, then use the `protocol` attribute of the `<web-service>` element, as shown in the following example:

```xml
<web-service name="stockquotes"
             targetNamespace="http://example.com"
```

```
                           uri="/myStockQuoteService"
                           protocol="https" >
...
</web-service>
```

EJB Component Web Service With Non-Built-In Data Types

A more complex type of Web service is one whose operations take non-built-in data types as parameters or return values. Because these non-built-in data types do not directly map to a XML/SOAP data type, you must describe the data type in the `web-services.xml` file.

For example, the following interface describes an EJB whose two methods return a `TradeResult` object:

```
public interface Trader extends EJBObject {
   public TradeResult buy (String stockSymbol, int shares)
      throws RemoteException;
   public TradeResult sell (String stockSymbol, int shares)
      throws RemoteException;
}
```

The `TradeResult` class looks like the following:

```
public class TradeResult implements Serializable {

  private int    numberTraded;
  private String stockSymbol;

  public TradeResult() {}

  public TradeResult(int nt, String ss) {
    numberTraded = nt;
    stockSymbol  = ss;
  }

  public int getNumberTraded() { return numberTraded; }
  public void setNumberTraded(int numberTraded) {
     this.numberTraded = numberTraded; }

  public String getStockSymbol() { return stockSymbol; }
  public void setStockSymbol(String stockSymbol) {
     this.stockSymbol = stockSymbol; }
}
```

The following `web-services.xml` file describes a Web service implemented with this EJB:

```
<web-services>

  <web-service name="TraderService"
               uri="/TraderService"
               targetNamespace="http://www.bea.com/examples/Trader">

    <types>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:stns="java:examples.webservices"
                  attributeFormDefault="qualified"
                  elementFormDefault="qualified"
                   targetNamespace="java:examples.webservices">
        <xsd:complexType name="TradeResult">
           <xsd:sequence><xsd:element maxOccurs="1" name="stockSymbol"
                                      type="xsd:string" minOccurs="1">
                     </xsd:element>
                     <xsd:element maxOccurs="1" name="numberTraded"
                                      type="xsd:int" minOccurs="1">
                      </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
      </xsd:schema>
     </types>

    <type-mapping>
     <type-mapping-entry
           deserializer="examples.webservices.TradeResultCodec"
           serializer="examples.webservices.TradeResultCodec"
           class-name="examples.webservices.TradeResult"
           xmlns:p1="java:examples.webservices"
           type="p1:TradeResult" >
     </type-mapping-entry>
    </type-mapping>

    <components>
      <stateless-ejb name="ejbcomp">
        <ejb-link path="trader.jar#TraderService" />
      </stateless-ejb>
    </components>

    <operations>
      <operation method="*" component="ejbcomp">
      </operation>
    </operations>

  </web-service>
```

```
</web-services>
```

In the example, the `<types>` element uses XML Schema notation to describe the XML representation of the `TradeResult` data type. The `<type-mapping>` element contains an entry for each data type described in the `<types>` element (in this case there is just one: `TradeResult`.) The `<type-mapping-entry>` lists the serialization class that converts the data between XML and Java, as well as the Java class file used to create the Java object.

EJB Component and SOAP Message Handler Chain Web Service

Another type of Web service is implemented with both a stateless session EJB backend component and a SOAP message handler chain that intercepts the request and response SOAP message. The following sample `web-services.xml` file describes such a Web service:

```
<web-services>
    <handler-chains>
        <handler-chain name="submitOrderCrypto">
            <handler class-name="com.example.security.EncryptDecrypt">
                <init-params>
                    <init-param name="elementToDecrypt" value="credit-info" />
                    <init-param name="elementToEncrypt" value="order-number" />
                </init-params>
            </handler>
        </handler-chain>
    </handler-chains>

    <web-service targetNamespace="http://example.com" name="myorderproc"
                uri="myOrderProcessingService">
        <components>
            <stateless-ejb name="orderbean">
                <ejb-link path="myEJB.jar#OrderBean" />
            </stateless-ejb>
        </components>
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
            <operation name="submitOrder" method="submit"
                        component="orderbean"
                        handler-chain="submitOrderCrypto" >
                <params>
                    <param name="purchase-order" style="in" type="xsd:anyType" />
                    <return-param name="order-number" type="xsd:string" />
```

```
            </params>
          </operation>
        </operations>
      </web-service>
</web-services>
```

The example shows a Web service that includes a SOAP message handler-chain called `submitOrderCrypto` used for decrypting and encrypting information in the SOAP request and response messages. The handler chain includes one handler, implemented with the `com.example.security.EncryptDecrypt` Java class. The handler takes two initialization parameters that specify the elements in the SOAP message that need to be decrypted and encrypted.

The Web service defines one stateless session EJB backend component called `orderbean`.

The `submitOrder` operation shows how to combine a handler-chain with a backend component by specifying the `method`, `component`, and `handler-chain` attributes in combination. When a client application invokes the `submitOrder` operation, the `submitOrderCrypto` handler chain first processes the SOAP request, decrypting the credit card information. The handler chain then invokes the `submit()` method of the `orderbean` EJB, passing it the modified parameters from the SOAP message, including the `purchase-order` input parameter. The `submit()` method then returns an `order-number`, which is encrypted by the handler chain, and the handler chain finally sends a SOAP response with the encrypted information to the client application that originally invoked the `submitOrder` operation.

SOAP Message Handler Chain-Only Web Service

You can also implement a WebLogic Web service with just a SOAP message handler chain and never invoke a backend component. This type of Web service might be useful, for example, as a front end to an existing workflow processing system. The handler chain simply takes the SOAP message request and hands it over to the workflow system, which performs all the further processing.

The following sample web-services.xml file describes such a Web service:

```
<web-services>
    <handler-chains>
        <handler-chain name="enterWorkflowChain">
            <handler class-name="com.example.WorkFlowEntry">
                <init-params>
```

```
                <init-param name="workflow-eng-jndi-name"
                            value="workflow.entry" />
            </init-params>
        </handler>
      </handler-chain>
    </handler-chains>

    <web-service targetNamespace="http://example.com"
                 name="myworkflow" uri="myWorkflowService">
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
            <operation name="enterWorkflow"
                       handler-chain="enterWorkflowChain"
                       invocation-style="one-way" />
        </operations>
    </web-service>
</web-services>
```

The example shows a Web service that includes one SOAP message handler chain, called `enterWorkflowChain`. This handler chain has one handler, implemented with the Java class `com.example.WorkFlowEntry`, that takes as an initialization parameter the JNDI name of the existing workflow system.

The Web service defines one operation called `enterWorkflow`. When a client application invokes this operation, the `enterWorkflowChain` handler chain takes the SOAP message request and passes it to the workflow system running on WebLogic Server whose JNDI name is `workflow.entry`. The operation is defined as asynchronous one-way, which means that the client application does not receive a SOAP response.

Note that because the `enterWorkflow` operation does *not* specify the `method` and `component` attributes, no backend component is ever invoked directly by the Web service. This also means that the `web-services.xml` file does not need to specify a `<components>` element.

8 Invoking Web Services

The following sections describe how to invoke Web services, both WebLogic and non-WebLogic, from client applications:

Overview of Invoking Web Services

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

> **Note:** This chapter uses the term *client application* to refer to both a standalone client that uses the WebLogic thin client to invoke a Web service, and a client that runs inside of an EJB running on WebLogic Server.

The sections that follow describe how to use BEA's implementation of the JAX-RPC specification to invoke a Web service from a Java client application. It is generally assumed that you are going to invoke *any* Web service rather than one running on WebLogic Server, except for those sections that describe the URLs needed to invoke a WebLogic Web service and its Home Page.

WebLogic Server provides optional Java client JAR files that include, for your convenience, all the classes, interfaces, and stubs you need to invoke a Web service. The client JAR files include the client runtime implementation of the JAX-RPC specification (called webserviceclient.jar and webserviceclient+ssl.jar) as well as Web service-specific implementations to minimize the amount of Java code needed to invoke a particular Web service.

JAX-RPC API

The Java API for XML based RPC (JAX-RPC) is a Sun Microsystems specification that defines the client API for invoking a Web service.

The following table briefly describes the core JAX-RPC interfaces and classes.

Table 8-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Service	Main client interface. Used for both static and dynamic invocations.
ServiceFactory	Factory class for creating Service instances.
Stub	Represents the client proxy for invoking the operations of a Web service. Typically used for static invocation of a Web service.
Call	Used to dynamically invoke a Web service.
JAXRPCException	Exception thrown if an error occurs while invoking a Web service.

WebLogic Server includes an implementation of the JAX-RPC specification.

For detailed information on JAX-RPC, see the following Web site:
http://java.sun.com/xml/jaxrpc/index.html.

For a tutorial that describes how to use JAX-RPC to invoke Web services, see
http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXRPC.html.

Examples of Clients That Invoke Web Services

WebLogic Server includes the following examples of creating and invoking WebLogic
Web services in the *WL_HOME*/samples/server/src/examples/webservices
directory, where *WL_HOME* refers to the main WebLogic Platform directory:

- `basic.statelessSession` : Uses a stateless session EJB backend component
  with built-in data types as its parameters and return value

- `basic.javaclass` : Uses a Java class backend component with built-in data
  types as its parameters and return value

- `complex.statelessSession` : Uses a stateless session EJB backend
  component with non-built-in data types as its parameters and return value

- `handler.log` : Uses both a handler chain and a stateless session EJB.

- `handler.nocomponent` : Uses only a handler chain with *no* backend
  component.

- `client.static` : Shows how to create a static client application that invokes a
  non-WebLogic Web service.

- `client.static_out` : Shows how to create a static client application that
  invokes a non-WebLogic Web service that uses out parameters.

- `client.dynamic_wsdl` : Shows how to create a dynamic client application that
  uses WSDL to invoke a non-WebLogic Web service.

- `client.dynamic_no_wsdl` : Shows how to create a dynamic client application
  that does not use WSDL to invoke a non-WebLogic Web service.

For detailed instructions on how to build and run the examples, open the following
Web page in your browser:

`WL_HOME/samples/server/src/examples/webservices/package-summary.html`

Additional examples of creating and invoking WebLogic Web services are listed on the Web services Web page on the Web Services dev2dev Download Page at http://webservice.bea.com.

Creating Java Client Applications to Invoke Web Services: Main Steps

To create a Java client application that invokes a Web service, follow these steps:

1. Get the Java client JAR files provided by WebLogic Server and add them to your CLASSPATH.

   If your client application is running on WebLogic Server, you can omit this step.

   Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

   For details, see "Getting the Java Client JAR Files" on page 8-4.

2. Write the Java client application code.

   For details on writing different kinds of client applications (static, dynamic, and so on), see "Writing the Java Client Application Code" on page 8-7 and "Writing a J2ME Client" on page 8-17.

3. Compile and run your Java client application.

Getting the Java Client JAR Files

WebLogic Server provides the following client JAR files:

- A runtime JAR file, called `webserviceclient.jar`, that contains the client runtime implementation of JAX-RPC. This JAR file is distributed as part of the WebLogic Server product.

- A runtime JAR file, called `webserviceclient+ssl.jar`, that contains the runtime implementation of SSL. This JAR file is distributed as part of the WebLogic Server product.

- A runtime JAR file, called `webserviceclient+ssl_pj.jar`, that contains the runtime implementation of SSL for the CDC profile of J2ME. This JAR file is distributed as part of the WebLogic Server product.

- A Web service-specific JAR file that you generate with the `clientgen` Ant task. This file contains the Web service-specific stubs, defined by the JAX-RPC specification, that client applications use to statically invoke a Web service (either WebLogic or non-WebLogic), such as `Stub` and `Service`. Almost *all* the code you need is automatically generated for you.

  Note: If you are creating dynamic client applications, you do not need to use this JAR file; BEA Systems provides the file as a convenience when you use static clients to invoke Web services.

For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

To get the client JAR files, follow these steps:

1. Copy the file *WL_HOME*\server\lib\webserviceclient.jar to your client application development computer, where *WL_HOME* refers to the top-level directory of WebLogic Platform. This client JAR file contains the client runtime implementation of JAX-RPC.

   Note: If you are using SSL to secure your Web service and you want to use the WebLogic Server-provided implementation of the SSL client classes, copy the file *WL_HOME*\server\lib\webserviceclient+ssl.jar to your client application development computer. In addition to the SSL implementation, this JAR file includes the same class files as in `webserviceclient.jar`.

   If you are writing a J2ME client that uses SSL, copy the file *WL_HOME*\server\lib\webserviceclient+ssl_pj.jar to your client application computer.

2. Generate the Web service-specific client JAR file by running the clientgen Ant task.

   Specify the wsdl attribute to create a client JAR file for *any* Web service (including non-WebLogic ones) or the ear attribute for WebLogic Web services packaged in EAR files.

   For details and examples of running the clientgen Ant task, see "Running the clientgen Ant Task" on page 8-6. For reference information, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

   Note: If you are creating a client application to invoke a WebLogic Web service, you can also download the client JAR file from the Home Page. See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22 for more information.

3. Put these client JAR files on your client computer and update your CLASSPATH environment variable to find them.

Running the clientgen Ant Task

To run the clientgen Ant task and automatically generate a client JAR file:

1. Create a file called build.xml that contains a call to the clientgen Ant task. For details, see "Sample build.xml File for the clientgen Ant Task."

2. Set your environment.

   On Windows NT, execute the setWLSEnv.cmd command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the setWLSEnv.sh command, located in the directory *WL_HOME*/server/bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the build.xml file by typing ant in the same directory as the build.xml file:

   ```
   prompt> ant
   ```

For reference information about the clientgen Ant task, see "clientgen" on page B-12.

Sample build.xml File for the clientgen Ant Task

The following example shows a simple `build.xml` file:

```
<project name="buildWebservice" default="generate-client">
   <target name="generate-client">
     <clientgen wsdl="http://example.com/myapp/myservice.wsdl"
                packageName="myapp.myservice.client"
                clientJar="c:/myapps/myService_client.jar"
/>

   </target>
</project>
```

In the example, the `clientgen` task creates a client JAR file (called `c:/myapps/myService_client.jar`) to invoke the Web service described in the `http://example.com/myapp/myservice.wsdl` WSDL file. It packages the interface and stub files in the `myapp.myservice.client` package.

Writing the Java Client Application Code

The following sections describe how to write different types of Java client applications for invoking Web services, from the simplest static client that requires almost no Java code to a more complex client that uses out parameters.

All examples use the JAX-RPC API and assume that you have the necessary BEA-provided client JAR files in your CLASSPATH.

Getting Information about a Web Service

You usually need to know the name of the Web service and the signature of its operations before you write your client code.

Look at the WSDL of the Web service. The name of the Web service is contained in the `<service>` element, as shown in the following excerpt of the `TraderService` WSDL:

```
<service name="TraderService">
  <port name="TraderServicePort"
      binding="tns:TraderServiceSoapBinding">
...
  </port>
</service>
```

The operations defined for this Web service are listed under the corresponding `<binding>` element. For example, the following WSDL excerpt shows that the `TraderService` Web service has two operations, `buy` and `sell` (for clarity, only relevant parts of the WSDL are shown):

```
<binding name="TraderServiceSoapBinding" ...>
  ...
  <operation name="sell">
  ...
  </operation>
  <operation name="buy">
  </operation>
</binding>
```

To find the full signature of the Web service operations, un-JAR the Web service-specific client JAR file (generated with the `clientgen` Ant task) and look at the actual `*.java` files. The file *ServiceName*`Port.java` contains the interface definition of your Web service, where *ServiceName* refers to the name of the Web service. For example, look at the `TraderServicePort.java` file for the signature of the `buy` and `sell` operations.

Maintaining the HTTP Session

You specify whether your client application will participate in an HTTP session with a Web service endpoint by setting the following property in your application:

```
javax.xml.rpc.Call.SESSION_MAINTAIN_PROPERTY
```

When a client application invokes a WebLogic Web service, an internal servlet first handles the request and creates an `HttpSession` object for each client. The lifetime of this `HttpSession` object follows the standard J2EE guideslines. For more information about `HttpSession` objects, see *Session Tracking from a Servlet* at http://e-docs.bea.com/wls/docs70/servlet/progtasks.html#session_tracking.

Handling Web Services That Crash

The first time you invoke a Web service from a client application that uses the WebLogic client JAR files, the client caches the IP address of the computer on which the Web service is running, and by default this cache is never refreshed with a new DNS lookup. This means that if you invoke a Web service, and later the computer on which the Web service is running crashes, but then another computer with a different IP address takes over for the crashed computer, a subsequent invoke of the Web service from the original client application will fail because the client application continues to think that the Web service is running on the computer with the old cached IP address. In other words, it does not try to re-resolve the IP address with a new DNS lookup, but rather uses the cached information from the original lookup.

To work around this problem, update your client application to set the JDK 1.4 system property `sun.net.inetaddr.ttl` to the number of seconds that you want the application to cache the IP address.

Warning: This workaround is valid only for standalone client applications; if your client application is running in WebLogic Server 7.0, you cannot set this property. This is because WebLogic Server 7.0 uses Version 1.3.1 of the JDK, and this system property exists only in JDK 1.4.

Writing a Simple Static Client

When you use a static client application to invoke a Web service, you use a strongly-typed Java interface, in contrast to a dynamic client where you indirectly reference the Web service operations and parameters. Using a dynamic client is analogous to looking up and invoking a method using the Java reflection APIs.

You must include the Web service-specific client JAR file in your CLASSPATH when statically invoking a Web service. This JAR file includes the following classes and interfaces:

- A Web service-specific implementation of the `Service` interface, which acts a stub factory. The stub factory class uses the value of the `wsdl` attribute of the `clientgen` Ant task used to generate the client JAR file in its default constructor.

- An interface and implementation of each SOAP port in the WSDL.

- Serialization class for non-built-in data types and their Java representations.

The following code shows an example of writing a client application that invokes the sample `TraderService` Web service; in the example, `TraderService` is the stub factory and `TraderServicePort` is the stub itself:

```
package examples.webservices.complex.statelessSession;

/**
 * This class illustrates how to use the JAX-RPC API to invoke the TraderService
 * Web service to perform the following tasks:
 * <ul>
 * <li> Buy 100 shares of some stocks
 * <li> Sell 100 shares of some stocks
 * </ul>
 *
 * The TraderService Web service is implemented using the Trader
 * stateless session EJB.
 *
 * @author Copyright (c) 1998-2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class Client {

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

    // Setup the global JAXM message factory
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    // Parse the argument list
    Client client = new Client();
    String wsdl = (args.length > 0? args[0] : null);
    client.example(wsdl);
  }

  public void example(String wsdlURI) throws Exception {

    TraderServicePort trader = null;
    if (wsdlURI == null) {
      trader = new TraderService_Impl().getTraderServicePort();
    } else {
      trader = new TraderService_Impl(wsdlURI).getTraderServicePort();
    }
    String [] stocks = {"BEAS", "MSFT", "AMZN", "HWP" };
```

```
    // execute some buys
  for (int i=0; i<stocks.length; i++) {
    int shares = (i+1) * 100;
    log("Buying "+shares+" shares of "+stocks[i]+".");
    TradeResult result = trader.buy(stocks[i], shares);
    log("Result traded "+result.getNumberTraded()
        +" shares of "+result.getStockSymbol());
  }
  // execute some sells
  for (int i=0; i<stocks.length; i++) {
    int shares = (i+1) * 100;
    log("Selling "+shares+" shares of "+stocks[i]+".");
    TradeResult result = trader.sell(stocks[i], shares);
    log("Result traded "+result.getNumberTraded()
        +" shares of "+result.getStockSymbol());

  }

}

private static void log(String s) {
  System.out.println(s);
}

}
```

The main points to notice about the example are as follows:

- The following code shows how to create a `TraderServicePort` stub:

  ```
  trader = new TraderService_Impl().getTraderServicePort();
  ```

  The `TraderService_Impl` stub implements the JAX-RPC `Service` interface. The default constructor of `TraderService_Impl` creates a stub based on the WSDL URI specified when using the `clientgen` Ant task to create the client JAR file. The `getTraderServicePort()` method implements the `Service.getPort()` method, used to return an instance of the `TraderService` stub implementation.

- The following code shows how to invoke the `buy` operation of the `TraderService` Web service:

  ```
   TradeResult result = trader.buy(stocks[i], shares);
  ```

  The `trader` Web service has two operations: `buy()` and `sell()`. Both operations return a non-built-in data type called `TradeResult`.

Writing a Dynamic Client That Uses WSDL

When you create a dynamic client that uses WSDL, you first create a service factory using the `ServiceFactory.newInstance()` method, then create a `Service` object from the factory and pass it the WSDL and the name of the Web service you are going to invoke. You then create a `Call` object from the `Service`, passing it the name of the port and the operation you want to execute, and finally use the `Call.invoke()` method to actually invoke the Web service operation.

When you write a dynamic client, you do not use the Web service-specific client JAR file generated with the `clientgen` Ant task, because this JAR file is used only for static clients. You do, however, need to include the JAR file that contains WebLogic's implementation of the JAX-RPC specification in your CLASSPATH. For more information about these JAR files, see "Getting the Java Client JAR Files" on page 8-4.

For example, assume you want to create a dynamic client application that uses WSDL to invoke the Web service found at the following URL:

```
http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
```

The following Java code shows one way to do this:

```java
/**
 * This class demonstrates a java client invoking a WebService.
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

import java.net.URL;

import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

public class Main {

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

    // Setup the global JAXM message factory
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");
```

```
    // create service factory
    ServiceFactory factory = ServiceFactory.newInstance();

    // define qnames
    String targetNamespace =
      "http://www.themindelectric.com/"
      + "wsdl/net.xmethods.services.stockquote.StockQuote/";

    QName serviceName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuoteService");

    QName portName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuotePort");

    QName operationName = new QName("urn:xmethods-delayed-quotes",
                                    "getQuote");

    URL wsdlLocation =
      new
URL("http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl");

    // create service
    Service service = factory.createService(wsdlLocation, serviceName);

    // create call
    Call call = service.createCall(portName, operationName);

    // invoke the remote web service
    Float result = (Float) call.invoke(new Object[] {
      "BEAS"
    });

    System.out.println("\n");
    System.out.println("This example shows how to create a dynamic client
                        application that invokes a non-WebLogic Web service.");
    System.out.println("The webservice used was:

http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl");
    System.out.println("The quote for BEAS is: ");
    System.out.println(result);
  }

}
```

> **Note:** When you use the `javax.xml.rpc.Call` API to create a dynamic client that uses WSDL, you cannot use the following methods in your client application:
>
> ■ `getParameterTypeByName()`

- `getReturnType()`

Additionally, if you want to execute the `getTargetEndpointAddress()` method, you must have previously executed the `setTargetEndpointAddress()` method, even if the targetEndPointAddress is available in the WSDL.

Writing a Dynamic Client That Does Not Use WSDL

Dynamic clients that do not use WSDL are similar to those that use WSDL except for having to explicitly set information that is found in the WSDL, such as the parameters to the operation, the target endpoint address, and so on.

The following example shows how to create a client application that invokes a Web service without specifying the WSDL in the client application:

```
/**
 * This class demonstrates a java client invoking a WebService.
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

public class Main {

  public static void main(String[] args) throws Exception {
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    // create service factory
    ServiceFactory factory = ServiceFactory.newInstance();

    // define qnames
    String targetNamespace =
      "http://www.themindelectric.com/"
      + "wsdl/net.xmethods.services.stockquote.StockQuote/";
```

```
    QName serviceName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuoteService");

    QName portName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuotePort");

    QName operationName = new QName("urn:xmethods-delayed-quotes",
                                    "getQuote");

    // create service
    Service service = factory.createService(serviceName);

    // create call
    Call call = service.createCall();

    // set port and operation name
    call.setPortTypeName(portName);
    call.setOperationName(operationName);
    // add parameters
    call.addParameter("symbol",
                      new QName("http://www.w3.org/2001/XMLSchema", "string"),
                      ParameterMode.IN);

    call.setReturnType(new QName( "http://www.w3.org/2001/XMLSchema","float") );

    // set end point address
    call.setTargetEndpointAddress("http://www.xmethods.com:9090/soap");

    // invoke the remote web service
    Float result = (Float) call.invoke(new Object[] {
      "BEAS"
    });

    System.out.println("\n");
    System.out.println("This example shows how to create a dynamic client
                        application that invokes a non-WebLogic Web service.");
    System.out.println("The webservice used was:

http://www.themindelectric.com/wsdl/net.xmethods.services.stockquote.StockQuote
");
    System.out.println("The quote for BEAS is:");
    System.out.println(result);
  }

}
```

Note: In dynamic clients that do not use WSDL, the `getPorts()` method always returns `null`. This behaviour is different from dynamic clients that do use WSDL in which the method actually returns the ports.

Writing a Client that Uses Out or In-Out Parameters

Web services can use out or in-out parameters as a way of returning multiple values.

When you write a client application that invokes a Web service that uses out or in-out parameters, the data type of the out or in-out parameter must implement the `javax.xml.rpc.holders.Holder` interface. After the client application invokes the Web service, the client can query the out or in-out parameters in the `Holder` object and treat them as if they were standard return values.

For example, the Web service described by the following WSDL has an operation called `echoStructAsSimpleTypes()` that takes one standard input parameter and three out parameters:

```
http://soap.4s4c.com/ilab/soap.asp?WSDL
```

The following static client application shows one way to invoke this Web service. The application assumes that you have included the Web service-specific client JAR file that contains the Stub classes, generated using the `clientgen` Ant task, in your CLASSPATH.

```
package websvc;

/**
 * This class demonstrates a java client invoking a WebService.
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

public class Main {

  public static void main(String[] args) throws Exception {
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    InteropLab_Impl test = new InteropLab_Impl();
    InteropTest2PortType soap = test.getinteropTest2PortType();

    org.tempuri.x4s4c.x1.x3.wsdl.types.SOAPStruct inputStruct =
      new org.tempuri.x4s4c.x1.x3.wsdl.types.SOAPStruct();
```

```
   inputStruct.setVarInt(10);
   inputStruct.setVarFloat(10.1f);
   inputStruct.setVarString("hi there");

   javax.xml.rpc.holders.StringHolder outputString =
     new javax.xml.rpc.holders.StringHolder();
   javax.xml.rpc.holders.IntHolder outputInteger =
     new javax.xml.rpc.holders.IntHolder();
   javax.xml.rpc.holders.FloatHolder outputFloat =
     new javax.xml.rpc.holders.FloatHolder();

   soap.echoStructAsSimpleTypes(inputStruct, outputString, outputInteger,
                                outputFloat);

   System.out.println("This example shows how to create a static client
                      application that invokes a non-WebLogic Web service.");
   System.out.println("The webservice used was:
                      http://soap.4s4c.com/ilab/soap.asp?WSDL");
   System.out.println("This webservice shows how to invoke an operation that
                      uses out parameters.  The set parameters are below:");
   System.out.println("outputString.value: " + outputString.value);
   System.out.println("outputInteger.value: " + outputInteger.value);
   System.out.println("outputFloat.value: " + outputFloat.value);
  }

}
```

Writing a J2ME Client

You can create a Java 2 Platform, Micro Edition (J2ME) Web service-specific client JAR file to use with client applications that run on J2ME.

Note: The specific J2ME environment that we support is the CDC and Foundation profile.

Creating a J2ME client application that invokes a Web service is almost the same as creating a non-J2ME client. For example, you use the same runtime client JAR file as non-J2ME client applications (*WL_HOME*\server\lib\webserviceclient.jar.)

To write a J2ME client application, follow the steps described in "Creating Java Client Applications to Invoke Web Services: Main Steps" on page 8-4 but with the following changes:

- When you run the `clientgen` Ant task to generate the Web service-specific client JAR file, be sure you specify the `j2me="True"` attribute, as shown in the following example:

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="c:/myapps/myService_clients.jar"
           j2me="True"
/>
```

  Note: The J2ME Web service-specific client JAR file generated by `clientgen` is not compliant with the JAX-RPC specification in the following ways:

  - The methods of the generated stubs do not throw `java.rmi.RemoteException`.

  - The generated stubs do not extend `java.rmi.Remote`.

- When you write, compile, and run your Java client application, be sure you use the J2ME virtual machine and APIs.

For more information about J2ME, see http://java.sun.com/j2me/.

Writing a J2ME Client that Uses SSL

WebLogic Server includes support for creating J2ME client applications that use SSL. If you are writing a J2ME client that uses SSL, follow these guidelines in addition to the guidelines specified in the preceding section:

- You must use the following additional class and package:

  - `java.math.BigInteger` (class)

  - `java.util.*` (entire package)

- Copy the file *WL_HOME*\server\lib\webserviceclient+ssl_pj.jar to your client application computer and add it to your CLASSPATH.

  Warning: Do not include the `weblogic.jar` file in your CLASSPATH.

- If your client application uses the WSDL file to invoke a Web service, you must use a local copy of the WSDL file stored on your client computer; you cannot access the WSDL file using a `URLConnection` object.

Creating and Using Portable Stubs

If you use the Web services client JAR files (both the ones distributed with the product and the Web service-specific one generated by the `clientgen` Ant task) as part of an application that runs in WebLogic Server, you might find that the Java classes in the JAR file collide with the classes of WebLogic Server itself. This problem is more apparent if the WebLogic Server in which the client JAR file is deployed is a different version from that which the client JAR file was generated. To solve this problem, use portable stubs.

Note: You need to use portable stubs only if your client application is deployed and running on WebLogic Server. If your client application is standalone, you do not need to use portable stubs.

To enable your client application to use portable stubs:

1. Use the WebLogic Server release-specific client JAR file called `wsclient70.jar` (distributed with WebLogic Server in the *WL_HOME*\server\lib directory) with your client application rather than the generic `webserviceclient.jar` client JAR file. The `wsclient70.jar` file contains the same class files as the standard client JAR file, but they are renamed `weblogic70.*`. Because these class files are version-specific, they will not collide with any `weblogic.*` WebLogic Server classes.

2. Run the Web-service specific client JAR file you generated with the `clientgen` Ant task, as well as any supporting client JAR files, through the `VersionMaker` utility. This utility makes the following changes to the classes in these client JAR files:

   - renames all `weblogic.*` classes to `weblogic70.*`.

   - all references to `weblogic.*` classes are changed to reference `weblogic70.*` instead.

   Use these new version-specific client JAR files with your client application.

   For details on using `VersionMaker`, see "Using the VersionMaker Utility" on page 8-20.

Using the VersionMaker Utility

The `weblogic.webservice.tools.versioning.VersionMaker` utility takes the following parameters:

- `destination_dir` : the destination directory that will contain the new version-specific client JAR files.

- `client_jar_file` : the client JAR file, generated by the `clientgen` Ant task, whose class files are named `weblogic.*` and should be renamed `weblogic70.*`.

- `other_jar_files` : supporting JAR files

Follow these steps to update your client JAR files to use version-specific WebLogic Server classes:

1. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*/server/bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

2. Execute the utility `weblogic.webservice.tools.versioning.VersionMaker`, as shown in the following example:

   ```
   java weblogic.webservice.tools.versioning.VersionMaker \
      new_directory myclient.jar supporting.jar
   ```

   In the example, the `weblogic.*` classes in the `myclient.jar` and `supporting.jar` client JAR files are renamed `weblogic70.*`, and all references to these classes updated accordingly. The new client JAR files are generated into the directory called `new_directory` under the current directory.

Using a Proxy Server with the WebLogic Web Services Client

You can use a proxy server to proxy requests from a WebLogic Web Services client application to a server (either WebLogic or non-WebLogic) that is hosting a Web Service. However, be sure to set *all* the following system properties in your client application:

- `http.proxyHost`

- `http.proxyPort`

- `weblogic.webservice.transport.http.proxy.host`

- `weblogic.webservice.transport.http.proxy.port`

Note: If you are using HTTPS as the transport when invoking the Web Service, replace the `http` in the preceding properties with `https`. For example, use `https.proxyHost` instead of `http.proxyHost`.

For more information on these, and other, WebLogic system properties you can set in your client application, see "WebLogic Web Services System Properties" on page 8-25.

Additionally, if you have set up your proxy server to use proxy authentication, then you must also set the property `weblogic.net.proxyAuthenticatorClassName` in your client application to the name of the Java class that implements the `weblogic.common.ProxyAuthentication` interface, as shown in the following excerpt from a client application:

```
System.setProperty("weblogic.net.proxyAuthenticatorClassName",
"my.ProxyAuthenticator");
```

In the example, `my.ProxyAuthenticator` is a class in the client application's CLASSPATH that implements the `weblogic.common.ProxyAuthentication` interface.

The `weblogic.common.ProxyAuthentication` interface allows a client application to provide user authentication information required when tunneling WebLogic HTTP and SSL protocols through a proxy server that requires user authentication. For details on implementing this interface, see the `weblogic.common.ProxyAuthentication` Javadocs.

The WebLogic Web Services Home Page and WSDL URLs

Every Web service deployed on WebLogic Server has a Home Page. From the Home page you can:

- View the WSDL that describes the service.

- Download the Web service-specific client JAR file that contains the interfaces, classes, and stubs needed to invoke the Web service from a client application.

  Note: A link to download this client JAR file appears on the Home page only if the name of the client JAR file is *WebServiceName*_client.jar, where *WebServiceName* refers to the name of the Web service, specified by the `name` attribute of the `<web-service>` element in the `web-services.xml` file. If this is not true for your Web service, you must use the `clientgen` Ant task to create the JAR file. For details, see "Running the clientgen Ant Task" on page 6-10.

- Test each operation to ensure that it is working correctly.

  As part of testing a Web service, you can edit the XML in the SOAP request that describes non-built-in data types to debug interoperability conflicts.

- View the SOAP request and response messages from a successful execution of an operation

The following URLs show first how to invoke the Web service Home page and then the WSDL in your browser:

```
[protocol]://[host]:[port]/[contextURI]/[serviceURI]
```

```
[protocol]://[host]:[port]/[contextURI]/[serviceURI]?WSDL
```

where:

- *protocol* refers to the protocol over which the service is invoked, either http or https. This value corresponds to the `protocol` attribute of the `<web-service>` element that describes the Web service in the `web-servicex.xml` file. If you used the `servicegen` Ant task to assemble your Web service, this value corresponds to the `protocol` attribute.

- *host* refers to the computer on which WebLogic Server is running.

- *port* refers to the port number on which WebLogic Server is listening (default value is `7001`).

- *contextURI* refers to the context root of the Web application, corresponding to the `<context-root>` element in the `application.xml` deployment descriptor of the EAR file. If you used the `servicegen` Ant task to assemble your Web service, this value corresponds to the `contextURI` attribute.

  If your `application.xml` file does not include the `<context-root>` element, then the value of `contextURI` is the name of the Web application archive file or exploded directory.

- *serviceURI* refers to the URI of the Web service. This value corresponds to the `uri` attribute of the `<web-service>` element in the `web-services.xml` file. If you used the `servicegen` Ant task to assemble your Web service, this value corresponds to the `serviceURI` attribute.

For example, assume you used the following `build.xml` file to assemble a WebLogic Web service using the `servicegen` Ant task:

```
<project name="buildWebservice" default="build-ear">
  <target name="build-ear">
      <servicegen
        destEar="myWebService.ear"
        warName="myWAR.war"
        contextURI="web_services">
        <service
            ejbJar="myEJB.jar"
            targetNamespace="http://www.bea.com/examples/Trader"
            serviceName="TraderService"
            serviceURI="/TraderService"
            generateTypes="True"
            expandMethods="True" >
        </service>
    </servicegen>
```

```
      </target>
</project>
```

The URL to invoke the Web service Home Page, assuming the service is running on a host called `ariel` at the default port number, is:

```
http://ariel:7001/web_services/TraderService
```

The URL to get the automatically generated WSDL of the Web service is:

```
http://ariel:7001/web_services/TraderService?WSDL
```

Debugging Errors While Invoking Web Services

If you encounter an error while trying to invoke a Web service (either WebLogic or non-WebLogic), it is useful to view the SOAP request and response messages that are generated because they often point to the problem.

To view the SOAP request and response messages, run your client application with the `-Dweblogic.webservice.verbose=true` flag, as shown in the following example that runs a client application called `runService`:

```
prompt> java -Dweblogic.webservice.verbose=true runService
```

The full SOAP request and response messages are printed in the command window from which you ran your client application.

You can also configure WebLogic Server to print the SOAP request and response messages each time a deployed WebLogic Web service is invoked by specifying the `-Dweblogic.webservice.verbose=true` flag when you start WebLogic Server. The SOAP messages are printed to the command window from which you started WebLogic Server.

Note: Because of possible decrease in performance due to the extra work of printing debugging messages to the command window, BEA recommends you set this WebLogic Server flag only during the development phase.

WebLogic Web Services System Properties

The following table lists the WebLogic Web services system properties you can set in client applications that invoke Web services. Use the `System.setProperty()` method to set the properties.

Table 8-2 WebLogic Web Services System Properties

System Property	Description	Data Type
weblogic.webservice.transport.http.full-url	Specifies that the full URL, rather than the relative URL, of the Web Service that the client application is invoking be specified in the `Request-URI` field of HTTP request. Valid values are `True` and `False`. Default value is `False`.	Boolean.
weblogic.webservice.transport.http.proxy.host	If you use a proxy server to make HTTP connections, use this system property to specify the host name of the proxy server in your client applications.	String.
weblogic.webservice.transport.http.proxy.port	If you use a proxy server to make HTTP connections, use this system property to specify the port of the proxy server in your client applications.	String.
weblogic.webservice.transport.https.proxy.host	If you use a proxy server to make HTTPS (HTTP over SSL) connections, use this system property to specify the host name of the proxy server in your client applications.	String.
weblogic.webservice.transport.https.proxy.port	If you use a proxy server to make HTTPS (HTTP over SSL) connections, use this system property to specify the port of the proxy server in your client applications.	String.

Table 8-2 WebLogic Web Services System Properties

System Property	Description	Data Type
weblogic.webservice.verbose	Enables verbose mode during Web service invocation so you can view the SOAP request and response messages. Valid values are True and False. Default value is False. For details, see "Debugging Errors While Invoking Web Services" on page 8-24.	Boolean.
weblogic.webservice.client.ssl.strictcertchecking	Enables or disables strict certificate validation when using the WebLogic-provided implementation of SSL. Set to True to enable strict certificate validation, and False to disable. Default value is False. For an example, see "Using the WebLogic Server-Provided SSL Implementation" on page 11-8.	Boolean.
weblogic.webservice.client.ssl.trustedcertfile	The name of the file (located on the client application computer) that contains the certificates of CA (certificate authority). The CAs are trusted to issue WebLogic Server certificates. The file can also contain certificates that you trust directly.	String.
weblogic.webservice.client.ssl.adapterclass	Fully qualified name of an adapter class you have implemented to use a third-party SSL implementation. For an example, see "Using a Third-Party SSL Implementation" on page 11-11.	String.
weblogic.http.KeepAliveTimeoutSeconds	Number of seconds to maintain HTTP keep-alive before timing out the request. If you do not want to use HTTP keep-alive, set this property to 0. Default value is 30 seconds.	Integer.

9 Using Non-Built-In Data Types

The following sections describe how to use non-built-in data types in WebLogic Web services:

- "Overview of Using Non-Built-In Data Types" on page 9-1

- "Creating Non-Built-In Data Types Manually: Main Steps" on page 9-2

Overview of Using Non-Built-In Data Types

You can create a WebLogic Web service that uses non-built-in data types as the Web service parameters and return value. Non-built-in data types are defined as data types other than the supported built-in data types, such as `int` and `String`. For the full list of built-in types, see "Using Built-In Data Types" on page 5-12.

WebLogic Server transparently handles the conversion of the built-in data types between their XML and Java representation. However, if your Web service operation uses non-built-in data types, you must provide the following information so that WebLogic Server can perform the conversion:

- Serialization class that converts between the XML and Java representation of the data.

- A Java class to contain the Java representation of the data type.

- An XML Schema representation of the data type.

■ Data type mapping information in the `web-services.xml` deployment descriptor file.

WebLogic Server includes the `servicegen` and `autotype` Ant tasks which automatically generate the preceding components by introspecting the stateless session EJB or Java class backend component for your Web service. These Ant tasks can handle many non-built-in data types, so most programmers will not ever have to create the components manually.

Sometimes, however, you may need to create the non-built-in data type components manually. Your data type may be so complex that the Ant task cannot correctly generate the components. Or maybe you want more control over how the data is converted between its XML and Java representations rather than relying on the default conversion procedure used by WebLogic Server.

For a full list of the supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.

For procedural instructions on using `servicegen` and `autotype`, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks." For reference information, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

Creating Non-Built-In Data Types Manually: Main Steps

The following procedure describes how to create non-built-in data types and use the `servicegen` Ant task to create a deployable Web service:

1. Write the XML Schema representation of your data type. See "Writing the XML Schema Data Type Representation" on page 9-4.

2. Write a Java class that represents your data type. See "Writing the Java Data Type Representation" on page 9-5.

3. Write a serialization class that converts the data between its XML and Java representations. See "Writing the Serialization Class" on page 9-6.

4. Compile your Java code into classes. Ensure that your CLASSPATH variable can locate the classes.

5. Create a text file that contains the data type mapping information about your non-built-in data type. See "Creating the Data Type Mapping File" on page 9-11.

6. Assemble your Web service using the servicegen Ant task as described in "Assembling WebLogic Web Services Using the servicegen Ant task" on page 6-3, with the following addition: when creating the build.xml file that calls the servicegen Ant task, be sure you specify the typeMappingFile attribute of servicegen, setting it equal to the name of the data type mapping file you created in the preceding step.

   BEA recommends that you create an exploded directory, rather than an EAR file, by specifying a value for the destEar attribute of servicegen that does *not* have an .ear suffix. You can later package the exploded directory into an EAR file when you are ready to deploy the Web service.

7. Update the web-services.xml file (which was generated by the servicegen Ant task), adding the XML Schema representation of your data type that you created in the first step of this procedure. See "Updating the web-services.xml File With XML Schema Information" on page 9-12.

8. Either deploy the exploded directory as your Web service, or package the directory into an EAR file and deploy it on WebLogic Server.

9. If you want to use the clientgen Ant task to generate a Java client, follow the procedure described in "Running the clientgen Ant Task" on page 6-10 with the following additions to the build.xml file that calls clientgen:

   ● Specify the ear attribute and set it to the full name of your Web service EAR file. Do *not* specify the wsdl attribute.

   ● Specify the useServerTypes attribute and set it to True.

Writing the XML Schema Data Type Representation

Web services use SOAP as the message format to transmit data between the service and the client application that invokes the service. Because SOAP is an XML-based protocol, you must use XML Schema notation to describe the structure of non-built-in data types used by Web service operations.

Warning: XML Schema is a powerful and complex data description language, and its use is not recommended for the faint of heart.

The following example shows the XML Schema that describes a non-built-in data type called `EmployBean`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:stns="java:examples.newTypes"
            attributeFormDefault="qualified"
            elementFormDefault="qualified"
            targetNamespace="java:examples.newTypes">
  <xsd:complexType name="EmployeeBean">
    <xsd:sequence>
      <xsd:element name="name"
                   type="xsd:string"
                   nillable="true"
                   minOccurs="1"
                   maxOccurs="1">
      </xsd:element>
      <xsd:element name="id"
                   type="xsd:int"
                   minOccurs="1"
                   maxOccurs="1">
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The following XML shows an instance of the `EmployeeBean` data type:

```
<EmployeeBean>
  <name>Beverley Talbott</name>
  <id>1234</id>
</EmployeeBean>
```

For detailed information about using XML Schema notation to describe your non-built-in data type, see the XML Schema specification at http://www.w3.org/TR/xmlschema-0/.

Writing the Java Data Type Representation

You use the Java representation of the non-built-in data type in your EJB or Java class that implements the Web service operation.

The following example shows one possible Java representation of the `EmployeeBean` data type whose XML representation is described in the preceding section:

```
package examples.newTypes;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */

public final class EmployeeBean {

  private String name = "John Doe";
  private int id = -1;

  public EmployeeBean() {
  }

  public EmployeeBean(String n, int i) {
    name = n;
    id = i;
  }

  public String getName() {
    return name;
  }
  public void setName(String  v) {
    this.name = v;
  }

  public int getId() {
    return id;
  }
  public void setId(int  v) {
    this.id = v;
  }

  public boolean equals(Object obj) {
    if (obj instanceof EmployeeBean) {
      EmployeeBean e = (EmployeeBean) obj;
      return (e.name.equals(name) && (e.id == id));
    }
    return false;
```

```
  }
}
```

Writing the Serialization Class

The serialization class performs the actual conversion of your data between its XML and Java representations. You write only one class that contains methods to serialize and deserialize your data. In the class you use the WebLogic XML Streaming API to process the XML data.

The WebLogic XML Streaming API provides an easy and intuitive way to consume and generate XML documents. It enables a procedural, stream-based handling of XML documents.

For detailed information on using the WebLogic XML Streaming API, see *Programming WebLogic XML* at http://e-docs.bea.com/wls/docs70/xml/xml_stream.html.

The following example shows a class that uses the XML Streaming API to serialize and deserialize the data type described in "Writing the XML Schema Data Type Representation" on page 9-4 and "Writing the Java Data Type Representation" on page 9-5; the procedure after the example lists the main steps to create such a class:

```java
package examples.newTypes;

import weblogic.webservice.encoding.AbstractCodec;

import weblogic.xml.schema.binding.DeserializationContext;
import weblogic.xml.schema.binding.DeserializationException;
import weblogic.xml.schema.binding.Deserializer;
import weblogic.xml.schema.binding.SerializationContext;
import weblogic.xml.schema.binding.SerializationException;
import weblogic.xml.schema.binding.Serializer;


import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.CharacterData;
import weblogic.xml.stream.ElementFactory;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLOutputStream;
```

```
import weblogic.xml.stream.XMLStreamException;


public final class EmployeeBeanCodec extends
  weblogic.webservice.encoding.AbstractCodec
{
  public void serialize(Object obj,
                        XMLName name,
                        XMLOutputStream writer,
                        SerializationContext context)
    throws SerializationException
  {
    EmployeeBean emp = (EmployeeBean) obj;

    try {

      //outer start element
      writer.add(ElementFactory.createStartElement(name));

      //employee name element
      writer.add(ElementFactory.createStartElement("name"));
      writer.add(ElementFactory.createCharacterData(emp.getName()));
      writer.add(ElementFactory.createEndElement("name"));

      //employee id element
      writer.add(ElementFactory.createStartElement("id"));
      String id_string = Integer.toString(emp.getId());
      writer.add(ElementFactory.createCharacterData(id_string));
      writer.add(ElementFactory.createEndElement("id"));

      //outer end element
      writer.add(ElementFactory.createEndElement(name));

    } catch(XMLStreamException xse) {
      throw new SerializationException("stream error", xse);
    }
  }

  public Object deserialize(XMLName name,
                            XMLInputStream reader,
                            DeserializationContext context)
    throws DeserializationException
  {
    // extract the desired information out of reader, consuming the
    // entire element representing the type,
    // construct your object, and return it.
    EmployeeBean employee = new EmployeeBean();

    try {
      if (reader.skip(name, XMLEvent.START_ELEMENT)) {
        StartElement top = (StartElement)reader.next();
```

```
      //next start element should be the employee name
      if (reader.skip(XMLEvent.START_ELEMENT)) {
        StartElement emp_name = (StartElement)reader.next();

        //assume that the next element is our name character data
        CharacterData cdata = (CharacterData) reader.next();
        employee.setName(cdata.getContent());
      } else {
        throw new DeserializationException("employee name not found");
      }

      //next start element should be the employee id
      if (reader.skip(XMLEvent.START_ELEMENT)) {
        StartElement emp_id = (StartElement)reader.next();

        //assume that the next element is our id character data
        CharacterData cdata = (CharacterData) reader.next();
        employee.setId(Integer.parseInt(cdata.getContent()));
      } else {
        throw new DeserializationException("employee id not found");
      }

      //we must consume our entire element to leave the stream in a
      //good state for any other deserializer
      if (reader.skip(name, XMLEvent.END_ELEMENT)) {
        XMLEvent end = reader.next();
      } else {
        throw new DeserializationException("expected end element not found");
      }
    } else {
      throw new DeserializationException("expected start element not found");
    }
  } catch (XMLStreamException xse) {
    throw new DeserializationException("stream error", xse);
  }
  return employee;
}

public Object deserialize(XMLName name,
                          Attribute att,
                          DeserializationContext context)
  throws DeserializationException
{
  //NOTE: not used in this example

  // extract the desired information out of att, consuming the
  // entire element representing the type,
  // construct your object, and return it.
  return new EmployeeBean();
```

```
    }
}
```

To create the serialization class using the WebLogic XML Streaming API, follow these steps:

1. Import the following classes, which are implemented by the abstract class that your serialization class will extend:

```
import weblogic.xml.schema.binding.DeserializationContext;
import weblogic.xml.schema.binding.DeserializationException;
import weblogic.xml.schema.binding.Deserializer;
import weblogic.xml.schema.binding.SerializationContext;
import weblogic.xml.schema.binding.SerializationException;
import weblogic.xml.schema.binding.Serializer;
```

2. Import the WebLogic XML Streaming API classes as needed. The preceding example imports the following classes:

```
import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.CharacterData;
import weblogic.xml.stream.ElementFactory;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLOutputStream;
import weblogic.xml.stream.XMLStreamException;
```

3. Write your Java class to extend the following abstract class:

```
weblogic.webservice.encoding.AbstractCodec
```

4. Implement the `serialize()` method, used to convert the data from Java to XML. The signature of this method is as follows:

```
  void serialize(Object obj,
                 XMLName name,
                 XMLOutputStream writer,
                 SerializationContext context)
    throws SerializationException;
```

Your Java object will be contained in the `Object` parameter. Use the XML Streaming API to write the Java object to the `XMLOutputStream` parameter. Use the `XMLName` parameter as the name of the resulting element.

> **Warning:** Do not update the `SerializationContext` parameter; it is used internally by WebLogic Server.

5. Implement the `deserialize()` method, used to convert the data from XML to Java. The signature of this method is as follows:

```
Object deserialize(XMLName name,
                   XMLInputStream reader,
                   DeserializationContext context)
  throws DeserializationException;
```

The XML that you want to deserialize is contained in the `XMLInputStream` parameter. Use the WebLogic XML Streaming API to parse the XML and convert it into the returned `Object`. The `XMLName` parameter contains the expected name of the XML element.

Call the `deserialize()` method recursively to build contained `Object`s.

When you use the XML Streaming API to read the stream of events that make up your XML document, be sure you always finish reading an element all the way up to and including the `EndElement` event, rather than finish reading once you have read all the actual data. If you finish before reaching an `EndElement` event, the deserialization of subsequent elements might fail.

> **Warning:** Do not update the `DeserializationContext` parameter; it is used internally by WebLogic Server.

6. If the data type for which you are creating a serialization class is used as an attribute value in your XML files, implement the following variation of the `deserialize()` method:

```
Object deserialize(XMLName name,
                   Attribute att,
                   DeserializationContext context)
  throws DeserializationException;
```

The `Attribute` parameter contains the attribute value to deserialize. The `XMLName` attribute contains the expected name of the XML element.

> **Warning:** Do not update the `DeserializationContext` parameter; it is used internally by WebLogic Server.

Creating the Data Type Mapping File

The data type mapping file is a subset of the `web-services.xml` deployment descriptor file. It centralizes some of the information about non-built-in data types, such as the name of the Java class that describes the Java representation of the data, the name of the serialization class that converts the data between XML and Java, and so on. The `servicegen` Ant task uses this data type mapping file when creating the `web-services.xml` deployment descriptor for the WebLogic Web service that uses the non-built-in data type.

To create the data type mapping file, follow these steps:

1. Create a text file with any name.

2. Within in the text file, add a `<type-mapping>` root element:

   ```
   <type-mapping>
   ...
   </type-mapping>
   ```

3. For each non-built-in data type for which you have created a serialization class, add a `<type-mapping-entry>` child element of the `<type-mapping>` element. Include the following attributes:

   - `xmlns:`*name*—Declares a namespace.

   - `class-name`—Specifies the fully qualified name of the Java class.

   - `type`—Specifies the name of XML Schema type for which this data type mapping entry applies.

   - `serializer`—The fully qualified name of the serialization class that converts the data from its Java to its XML representation. For details on creating this class, see "Writing the Serialization Class" on page 9-6.

   - `deserializer`—The fully qualified name of the serialization class that converts the data from its XML to its Java representation. For details on creating this class, see "Writing the Serialization Class" on page 9-6.

   The following example shows a possible data type mapping file with one `<type-mapping>` entry for the XML Schema data type shown in "Updating the web-services.xml File With XML Schema Information" on page 9-12:

```
<type-mapping>
    <type-mapping-entry
        xmlns:p2="java:examples.newTypes"
```

```
        class-name="examples.newTypes.EmployeeBean"
        type="p2:EmployeeBean"
        serializer="examples.newTypes.EmployeeBeanCodec">
        deserializer="examples.newTypes.EmployeeBeanCodec"
    </type-mapping-entry>
</type-mapping>
```

Updating the web-services.xml File With XML Schema Information

The `web-services.xml` file generated by `servicegen` will not have the XML Schema information for the non-built-in data type for which you have created your own custom serialization class. For this reason, you must manually add the XML Schema information to the deployment descriptor, as described in the following steps:

1. In the existing `web-services.xml` file generated by the `servicegen` Ant task, find the `<types>` child element of the `<web-service>` element:

   ```
   <types>
   ...
   </types>
   ```

2. Merge your XML Schema representation of your non-built-in data type that you created in "Writing the XML Schema Data Type Representation" on page 9-4 with the any existing information within the `<types>` element, as shown in the following example:

   ```
   <types>
     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:stns="java:examples.newTypes"
                 attributeFormDefault="qualified"
                 elementFormDefault="qualified"
                 targetNamespace="java:examples.newTypes">
       <xsd:complexType name="EmployeeBean">
         <xsd:sequence>
           <xsd:element name="name"
                        type="xsd:string"
                        nillable="true"
                        minOccurs="1"
                        maxOccurs="1">
           </xsd:element>
           <xsd:element name="id"
                        type="xsd:int"
                        minOccurs="1"
   ```

```
                              maxOccurs="1">
               </xsd:element>
             </xsd:sequence>
          </xsd:complexType>
       </xsd:schema>
    </types>
```

10 Creating SOAP Message Handlers to Intercept the SOAP Message

The following sections discuss how to use SOAP message handlers to intercept the request and response SOAP messages when developing a WebLogic Web service:

Overview of SOAP Message Handlers and Handler Chains

A SOAP message handler intercepts the SOAP message in both the request and response of the Web service. You can create handlers in both the Web service itself and the client applications that invoke the Web service. Refer to "Using SOAP Message Handlers to Intercept the SOAP Message" on page 4-6 for examples of when to use handlers.

The following table describes the main classes and interfaces of the `javax.xml.rpc.handler` API; later sections in this chapter describe how to use them to create handlers.

Table 10-1 JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
Handler	Main interface that you implement when creating a handler. Contains methods to handle the SOAP request, response, and faults.
HandlerInfo	Contains information about the handler, in particular the initialization parameters, specified in the web-services.xml file.
MessageContext	Abstracts the message context processed by the handler. The MessageContext properties allow the handlers in a handler chain to share processing state.
soap.SOAPMessageContext	Sub-interface of the MessageContext interface used to get at or update the SOAP message.
javax.xml.soap.SOAPMessage	Object that contains the actual request or response SOAP message, including its header, body, and attachment.

Creating SOAP Message Handlers: Main Steps

To create SOAP message handlers to intercept request and response SOAP messages when developing WebLogic Web services:

1. Design the handlers and handler chains. See "Designing the SOAP Message Handlers and Handler Chains" on page 10-3.

2. For each handler in the handler chain, create a Java class that implements the `javax.xml.rpc.handler.Handler` interface. See "Implementing the Handler Interface" on page 10-6.

   WebLogic Server includes an extension to the JAX-RPC handler API which you can use to simplify the coding of your handler class: an abstract class called `weblogic.webservice.GenericHandler`. See "Extending the GenericHandler Abstract Class" on page 10-13.

3. Compile the Java code into class files. You will later package these class files into a deployable Web services EAR file, as described in Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

4. Update the `web-services.xml` deployment descriptor file with the appropriate information. See "Updating the web-services.xml File with SOAP Message Handler Information" on page 10-16.

For information about creating client-side SOAP message handlers and handler chains, see "Using SOAP Message Handlers and Handler Chains in a Client Application" on page 10-18.

Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers, you must decide:

- The number of handlers needed to perform all the work

- The sequence of execution

- Whether to invoke a backend component or whether the Web service consists of only a handler chain.

Each handler in a handler chain has one method for handling the request SOAP message and another method for handling the response SOAP message. You specify the handlers in the `web-services.xml` deployment descriptor file. An ordered group of handlers is referred to as a *handler chain*.

When invoking a Web service, WebLogic Server executes handlers as follows:

1. The `handleRequest()` methods of the handlers in the handler chain are all executed, in the order specified in the `web-services.xml` file. Any of these `handleRequest()` methods might change the SOAP message request.

2. When the `handleRequest()` method of the last handler in the handler chain executes, WebLogic Server invokes the backend component that implements the Web service, passing it the final SOAP message request.

Note: This step only occurs if a backend component has actually been defined for the Web service; it is possible to develop a Web service that consists of only a handler chain.

3. When the backend component has finished executing, the `handleResponse()` methods of the handlers in the handler chain are executed in the *reverse* order specified in the `web-services.xml` file. Any of these `handleResponse()` methods might change the SOAP message response.

4. When the `handleResponse()` method of the first handler in the handler chain executes, WebLogic server returns the final SOAP message response to the client application that invoked the Web service.

For example, assume that you have specified a handler chain called `myChain` that contains three handlers in the `web-services.xml` deployment descriptor, as shown in the following excerpt:

```
<handler-chains>
   <handler-chain name="myChain">
     <handler class-name="myHandlers.handlerOne" />
     <handler class-name="myHandlers.handlerTwo" />
     <handler class-name="myHandlers.handlerThree" />
```

```
      </handler-chain>
</handler-chains>
```

The following graphic shows the order in which WebLogic Server executes the `handleRequest()` and `handleResponse()` methods of each handler:
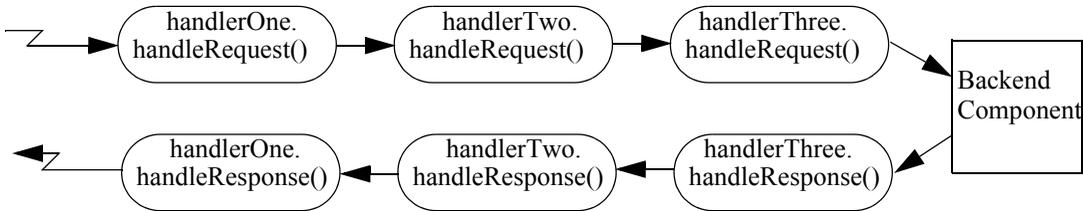
Each SOAP message handler has a separate method to process the request and response SOAP message because the same type of processing typically must happen in both places. For example, you might design an Encryption handler whose `handleRequest()` method decrypts secure data in the SOAP request and `handleResponse()` method encrypts the SOAP response.

You can, however, design a handler that process only the SOAP request and does no equivalent processing of the response.

You can also choose not to invoke the next handler in the handler chain and send an immediate response to the client application at any point. The way to do this is discussed in later sections.

Finally, you can design a Web service that contains only handlers in a handler chain, and no backend component at all. In this case, when the `handleRequest()` method in the last handler has executed, the chain of `handleResponse()` methods is automatically invoked. See "Updating the web-services.xml File with SOAP Message Handler Information" on page 10-16 for an example of using the `web-services.xml` file to specify that only a handler chain, and no backend component, implements a Web service.

Implementing the Handler Interface

Your SOAP message handler class must implement the `javax.rpc.xml.handler.Handler` interface, as shown in the following example. The example demonstrates a simple way to print out the SOAP request and response messages:

```java
package examples.webservices.handler.log;

import java.util.Map;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;

import weblogic.logging.NonCatalogLogger;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */
public final class LogHandler
  implements Handler
{
  private NonCatalogLogger log;

  private HandlerInfo handlerInfo;

  public void init(HandlerInfo hi) {
    log = new NonCatalogLogger("WebService-LogHandler");
    handlerInfo = hi;
  }

  public void destroy() {}

  public QName[] getHeaders() { return handlerInfo.getHeaders(); }

  public boolean handleRequest(MessageContext mc) {
    SOAPMessageContext messageContext = (SOAPMessageContext) mc;

    System.out.println("** Request: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;
  }
```

```
public boolean handleResponse(MessageContext mc) {

  SOAPMessageContext messageContext = (SOAPMessageContext) mc;

  System.out.println("** Response: "+messageContext.getMessage().toString());
  log.info(messageContext.getMessage().toString());
  return true;
}
public boolean handleFault(MessageContext mc) {
  SOAPMessageContext messageContext = (SOAPMessageContext) mc;

  System.out.println("** Fault: "+messageContext.getMessage().toString());
  log.info(messageContext.getMessage().toString());
}
}
```

The `javax.xml.rpc.handler.Handler` interface contains the following methods that you must implement:

- `init()`
- `destroy()`
- `getHeaders()`
- `handleRequest()`
- `handleResponse()`
- `handleFault()`

The following sections describe how to use each method to code your implementation.

Implementing the Handler.init() Method

The `Handler.init()` method is called to create an instance of a `Handler` object and to enable the instance to initialize itself. Its signature is:

```
public void init(HandlerInfo config) throws JAXRPCException {}
```

The `HandlerInfo` object contains information about the SOAP message handler, in particular the initialization parameters, specified in the `web-services.xml` file. Use the `HandlerInfo.getHandlerConfig()` method to get the parameters; the method returns a `Map` object that contains name-value pairs.

Implement the `init()` method if you need to process the initialization parameters or if you have other initialization tasks to perform.

Sample uses of initialization parameters are to turn debugging on or off, specify the name of a log file to which to write messages or errors, and so on.

Implementing the Handler.destroy() Method

The `Handler.destroy()` method is called to destroy an instance of a `Handler` object. Its signature is:

```
public void destroy() throws JAXRPCException {}
```

Implement the `destroy()` method to release any resources acquired throughout the handler's lifecycle.

Implementing the Handler.getHeaders() Method

The `Handler.getHeaders()` method gets the header blocks processed by this Handler instance. Its signature is:

```
public QName[] getHeaders() {}
```

Implementing the Handler.handleRequest() Method

The `Handler.handleRequest()` method is called to intercept a SOAP message request before it is processed by the back-end component. Its signature is:

```
public boolean handleRequest(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to decrypt data in the SOAP message before it is processed by the back-end component, to make sure that the request contains the correct number of parameters, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

 Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message request. The SOAP message request itself is stored in a `javax.xml.soap.SOAPMessage` object. For detailed information on this object, see "The javax.xml.soap.SOAPMessage Object" on page 10-12.

The `SOAPMessageContext` class defines two methods for processing the SOAP request:

- `SOAPMessageContext.getMessage()` returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message request.

- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)` updates the SOAP message request after you have made changes to it.

After you code all the processing of the SOAP request, do one of the following:

- Invoke the next handler on the handler request chain by returning `true`.

  The next handler on the request chain is specified as the next `<handler>` subelement of the `<handler-chain>` element in the `web-services.xml` deployment descriptor. If there are no more handlers in the chain, the method either invokes the backend-end component, passing it the final SOAP message request, or invokes the `handleResponse()` method of the last handler, depending on how you have configured your Web service.

- Block processing of the handler request chain by returning `false`.

  Blocking the handler request chain processing implies that the backend component does not get executed for this invoke of the Web service. You might want to do this if you have cached the results of certain invokes of the Web service, and the current invoke is on the list.

  Although the handler request chain does not continue processing, WebLogic Server does invoke the handler *response* chain, starting at the current handler. For example, assume that a handler chain consists of two handlers: handlerA and handlerB, where the `handleRequest()` method of handlerA is invoked before that of handlerB. If processing is blocked in handlerA (and thus the `handleRequest()` method of handlerB is *not* invoked), the handler response chain starts at handlerA and the `handleRequest()` method of handlerB is not invoked either.

- Throw the `javax.xml.rpc.soap.SOAPFaultException` to indicate a SOAP fault.

If the `handleRequest()` method throws a `SOAPFaultException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, and invokes the `handleFault()` method of this handler.

■ Throw a `JAXRPCException` for any handler specific runtime errors.

If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, and invokes the `handleFault()` method of this handler.

Implementing the Handler.handleResponse() Method

The `Handler.handleResponse()` method is called to intercept a SOAP message response after it has been processed by the backend component, but before it is sent back to the client application that invoked the Web service. Its signature is:

```
public boolean handleResponse(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to encrypt data in the SOAP message before it is sent back to the client application, to further process returned values, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message response.   The SOAP message response itself is stored in a `javax.xml.soap.SOAPMessage` object. See "The javax.xml.soap.SOAPMessage Object" on page 10-12.

The `SOAPMessageContext` class defines two methods for processing the SOAP response:

■ `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message response.

■ `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message response after you have made changes to it.

After you code all the processing of the SOAP response, do one of the following:

■ Invoke the next handler on the handler response chain by returning `true`.

The next response on the handler chain is specified as the preceding `<handler>` subelement of the `<handler-chain>` element in the `web-services.xml` deployment descriptor. (Remember that responses on the handler chain execute in the *reverse* order that they are specified in the `web-services.xml` file. See "Designing the SOAP Message Handlers and Handler Chains" on page 10-3 for more information.)

If there are no more handlers in the chain, the method sends the final SOAP message response to the client application that invoked the Web service.

■ Block processing of the handler response chain by returning `false`.

Blocking the handler response chain processing implies that the remaining handlers on the response chain do not get executed for this invoke of the Web service and the current SOAP message is sent back to the client application.

■ Throw a `JAXRPCException` for any handler specific runtime errors.

If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, and invokes the `handleFault()` method of this handler.

Implementing the Handler.handleFault() Method

The `Handler.handleFault()` method processes the SOAP faults based on the SOAP message processing model. Its signature is:

```
public boolean handleFault(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to handle processing of any SOAP faults generated by the `handleResponse()` and `handleRequest()` methods, as well as faults generated by the backend component.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message. The SOAP message itself is stored in a `javax.xml.soap.SOAPMessage` object. See "The javax.xml.soap.SOAPMessage Object" on page 10-12.

The `SOAPMessageContext` class defines the following two methods for processing the SOAP message:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message.

- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message after you have made changes to it.

After you code all the processing of the SOAP fault, do one of the following:

- Invoke the `handleFault()` method on the next handler in the handler chain by returning `true`.

- Block processing of the handler fault chain by returning `false`.

The javax.xml.soap.SOAPMessage Object

The `javax.xml.soap.SOAPMessage` abstract class is part of the Java API for XML Messaging (JAXM) specification. You use the class to manipulate request and response SOAP messages when creating SOAP message handlers. This section describes the basic structure of a `SOAPMessage` object and some of the methods you can use to view and update a SOAP message.

A `SOAPMessage` object consists of a `SOAPPart` object (which contains the actual SOAP XML document) and zero or more attachments.

Refer to the JAXM API Javadocs for the full description of the `SOAPMessage` class. For more information on JAXM, go to http://java.sun.com/xml/jaxm/index.html.

The SOAPPart Object

The `SOAPPart` object contains the XML SOAP document inside of a `SOAPEnvelope` object. You use this object to get the actual SOAP headers and body.

The following sample Java code shows how to retrieve the SOAP message from a `MessageContext` object, provided by the `Handler` class, and get at its parts:

```
SOAPMessage soapMessage =  messageContext.getRequest();
SOAPPart soapPart = soapMessage.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPBody soapBody = soapEnvelope.getBody();
SOAPHeader soapHeader = soapEnvelope.getHeader();
```

The AttachmentPart Object

The `AttachmentPart` object contains the optional attachments to the SOAP message. Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file.

Use the following methods of the `SOAPMessage` class to manipulate the attachments:

- `countAttachments()`: returns the number of attachments in this SOAP message.

- `getAttachments()`: retrieves all the attachments (as `AttachmentPart` objects) into an `Iterator` object.

- `createAttachmentPart()`: create an `AttachmentPart` object from another type of `Object`.

- `addAttachmentPart()`: adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.

Extending the GenericHandler Abstract Class

WebLogic Server includes an extension to the JAX-RPC handler API that you can use to simplify the Java code of your SOAP message handler class. This extension is the abstract class `weblogic.webservices.GenericHandler`. It implements the JAX-RPC `javax.xml.rpc.handler.Handler` interface.

Note: The `weblogic.webservices.GenericHandler` abstract class was originally developed for WebLogic Server when the JAX-RPC specification was not yet final and did not include this functionality. However, now that JAX-RPC includes its own `GenericHandler` class which is almost exactly

the same as the WebLogic Server class, BEA highly recommends that you use the standard JAX-RPC abstract class rather than the WebLogic-specific one. The documentation in this section is provided for compatibility reasons only.

For more information about the JAX-RPC `javax.xml.rpc.handler.GenericHandler` abstract class, see the JAX-RPC Javadocs.

Because `GenericHandler` is an abstract class, you need only implement the methods that will contain actual code, rather than having to implement every method of the `Handler` interface even if the method does nothing. For example, if your handler does not use initialization parameters and you do not need to allocate any additional resources, you do not need to implement the `init()` method.

The GenericHandler class is defined as follows:

```
package weblogic.webservice;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */
public abstract class GenericHandler
  implements Handler
{

  private HandlerInfo handlerInfo;

  public void init(HandlerInfo handlerInfo) {
    this.handlerInfo = handlerInfo;
  }

  protected HandlerInfo getHandlerInfo() { return handlerInfo; }

  public boolean handleRequest(MessageContext msg) {
    return true;
  }

  public boolean handleResponse(MessageContext msg) {
    return true;
  }

  public boolean handleFault(MessageContext msg) {}
```

```
  public void destroy() {}
  public QName[] getHeaders() { return handlerInfo.getHeaders(); }

}
```

The following sample code, taken from the `examples.webservices.handler.nocomponent` product example, shows how to use the `GenericHandler` abstract class to create your own handler. The example implements only the `handleRequest()` and `handleResponse()` methods. It does not implement (and thus does not include in the code) the `init()`, `destroy()`, `getHeaders()`, and `handleFault()` methods.

```
package examples.webservices.handler.nocomponent;

import java.util.Map;

import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.*;

import weblogic.webservice.GenericHandler;

import weblogic.utils.Debug;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */
public final class EchoStringHandler
  extends GenericHandler
{
  private int me = System.identityHashCode(this);

  public boolean handleRequest(MessageContext messageContext) {
    System.err.println("** handleRequest called in: "+me);
    return true;
  }

  public boolean handleResponse(MessageContext messageContext) {

    try {
      MessageFactory messageFactory = MessageFactory.newInstance();

      SOAPMessage m = messageFactory.createMessage();

      SOAPEnvelope env = m.getSOAPPart().getEnvelope();

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

```
    SOAPElement fResponse =
      body.addBodyElement(env.createName("echoResponse"));

    fResponse.addAttribute(env.createName("encodingStyle"),
      "http://schemas.xmlsoap.org/soap/encoding/");

    SOAPElement result =
      fResponse.addChildElement(env.createName("result"));

    result.addTextNode("Hello World");

    ((SOAPMessageContext)messageContext).setMessage(m);

    return true;

  } catch (SOAPException e) {
    e.printStackTrace();
    throw new JAXRPCException(e);
  }
 }
}
```

Updating the web-services.xml File with SOAP Message Handler Information

The `web-services.xml` deployment descriptor file describes the SOAP message handlers and handler chains defined for a Web service and the order in which they should be executed.

To update the `web-services.xml` file with handler information:

1. Create a `<handler-chains>` child element of the `<web-services>` root element that will contain a list of all handler chains defined for the Web service.

2. Create a `<handler-chain>` child element of the `<handler-chains>` element; within this element list all the handlers in the handler chain. For each handler, use the `class-name` attribute to specify the fully qualified name of the Java class that implements the handler. Use the `<init-params>` element to specify any initialization parameters of the handler.

   The following sample excerpt shows a handler chain called `myChain` that contains three handlers, the first of which has an initialization parameter:

```
<web-services>
  <handler-chains>
    <handler-chain name="myChain">
      <handler class-name="myHandlers.handlerOne" >
          <init-params>
             <init-param name="debug" value="on" />
          </init-params>
      </handler>
      <handler class-name="myHandlers.handlerTwo" />
      <handler class-name="myHandlers.handlerThree" />
    </handler-chain>
  </handler-chains>
...
</web-services>
```

3. Use the `<operation>` child element of the `<operations>` element (which itself is a child of the `<web-service>` element) to specify that the handler chain is an operation of the Web service. Follow one of the next two scenarios:

- The handler chain executes together with a backend component, such as a stateless session EJB.

  In this case use the `component`, `method`, and `handler-chain` attributes of the `<operation>` element, as shown in the following partial excerpt of a `web-services.xml` file:

  ```
  <web-service>
    <components>
      <stateless-ejb name="myEJB">
        ...
      </stateless-ejb>
    </components>
    <operations>
      <operation name="getQuote"
                 method="getQuote"
                 component="myEJB"
                 handler-chain="myChain" />
    </operations>
  </web-service>
  ```

  In the example, the request chain of the `myChain` handler chain executes first, then the `getQuote()` method of the `myEJB` stateless session EJB component, and finally the response chain of `myChain`.

- The handler chain executes on its own, *without* a backend component.

In this case use only the handler-chain attribute of the `<operation>` element and explicitly do not specify the component or method attributes, as shown in the following excerpt:

```
<web-service>
  <operations>
    <operation name="chainService"
               handler-chain="myChain" />
    </operations>
</web-service>
```

In the example, the Web service consists solely of the `myChain` handler chain.

Using SOAP Message Handlers and Handler Chains in a Client Application

Most of this chapter describes how to create SOAP message handlers in a handler chain that execute as part of the Web service running on WebLogic Server. You can also create handlers that execute in a client application. In the case of a client-side handler, the handler executes twice when a client application invokes a Web service:

- directly before the client appliation sends the SOAP request to the Web service

- directly after the client application receives the SOAP response from the Web service

You create a client-side handler in the same way you create a server-side handler: write a Java class that implements the `javax.rpc.xml.handler.Handler` interface. In many cases you can use the exact same handler class on both the Web service running on WebLogic Server *and* the client applications that invoke the Web service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client. For details about writing the handler Java class, see "Implementing the Handler Interface" on page 10-6.

After you have created your client-side handler class, the process for registering the handler on the client application is different from that of the server. Because client applications do not have deployment descriptors, you must register the handler programmatically using the `javax.xml.rpc.handler.HandlerInfo` and

javax.xml.rpc.handler.HandlerRegistry classes. The following sample client application shows how to do this, with relevant sections in bold discussed after the example:

```
import java.util.ArrayList;

import java.io.IOException;

import javax.xml.namespace.QName;

import javax.xml.rpc.ServiceException;

import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;

public class Main{

  public static void main( String[] args ){

    if( args.length == 1 ){
      new Main( args[0] );
    }else{
      throw new IllegalArgumentException( "URL of the service not specified" );
    }
  }

  public Main( String wsdlUrl ){
    try{

      HelloWorldService service = new HelloWorldService_Impl( wsdlUrl );
      HelloWorldServicePort port = service.getHelloWorldServicePort();

      QName portName = new QName( "http://tutorial/sample4/",
        "HelloWorldServicePort");

      HandlerRegistry registry = service.getHandlerRegistry();

      List handlerList = new ArrayList();
      handlerList.add( new HandlerInfo( ClientHandler.class, null, null ) );

      registry.setHandlerChain( portName, handlerList );

      System.out.println( port.helloWorld() );
    }catch( IOException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( ServiceException e ){
      System.out.println( "Failed to create web service client:" + e );
    }
  }
}
```

The main points to notice about the example are as follows:

- Import the JAX-RPC `HandlerInfo` and `HandlerRegistry` classes which will be used to register the client-side handler class:

```
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;
```

- Create a `QName` object that contains the qualified name of the Web service port:

```
 QName portName = new QName( "http://tutorial/sample4/",
        "HelloWorldServicePort");
```

Refer to the WSDL of the Web service you are invoking for the name of the port and its namespace.

- Create a `HandlerRegistry` object:

```
HandlerRegistry registry = service.getHandlerRegistry();
```

- Create a `List` object that contains a list of the handlers you want to register. This list becomes the client-side handler chain. Use the `HandlerInfo` class to specify the name of your Java handler class:

```
List handlerList = new ArrayList();
handlerList.add( new HandlerInfo( ClientHandler.class, null,
null ) );
```

In the example, the handler chain consists of just one handler: `ClientHandler.class`. You can, however, create a handler chain of as many handlers as you want.

> **Warning:** The order in which you add the handlers to the `List` object specifies the order in which the handlers are executed in the client application. For example, if you want `HandlerA.class` to execute first and then `HandlerB.class`, be sure you add `HandlerA.class` to the list before `HandlerB.class`.

- Register the handler chain with the client application using the `HandlerRegistry.setHandlerChain()` method:

```
registry.setHandlerChain( portName, handlerList );
```

11 Configuring Security

The following sections describe how to configure security for WebLogic Web services:

Overview of Configuring Security

Configuring security for WebLogic Web services is basically no different from securing any other type of application or component that runs on WebLogic Server. You can secure the entire Web service by restricting access to the URLs that invoke the Web service and its WSDL. When you secure the entire Web service, the components that make up the Web service are automatically secured. Or you can secure individual components of the Web service, such as the stateless session EJB, a selected list of its methods, the Web application that contains the `web-services.xml` file, and so on.

After you secure access to the Web service or some of its components, you configure client applications to use HTTP or SSL to authenticate themselves when they invoke the Web service.

For additional examples of securing WebLogic Web Services, see the Webservice Download Page at http://webservice.bea.com/index.html.

Configuring Security: Main Steps

To configure security for a WebLogic Web service and a client that invokes the service, follow these steps. Later sections describe the steps in detail.

1. Control access to either the entire Web service or some of its components by creating roles, mapping the roles to principals in your realm, then specifying which components are secured and accessible only by the principals in the role.

   See "Controlling Access to WebLogic Web Services" on page 11-3.

2. Optionally update the `web-services.xml` file to specify that the Web service can be accessed *only* by HTTPS.

   See "Specifying the HTTPS Protocol" on page 11-5.

3. If your client application will use SSL to authenticate itself, configure SSL for WebLogic Server.

   You can configure one-way SSL (the default) where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.

   For details about SSL, the difference between one-way and two-way, and procedures to configure both, see *Configuring SSL* at http://e-docs.bea.com/wls/docs70/secmanage/ssl.html.

   Warning: If you use 2-way SSL to secure the connection when invoking a WebLogic Web Service, WebLogic Server always asserts the identity of the certificate to ensure that it maps to a valid WebLogic Server user, even if the Web Service or the stateless EJB backend component does not require any special privileges. This is not true for 1-way SSL, because in that case the client application does not send its certificate.

4. Code your client to authenticate itself using HTTP or SSL when invoking a WebLogic Web service.

5. If your client application is using SSL, configure SSL on the client-side.

Controlling Access to WebLogic Web Services

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 Web Service URL

■ The stateless session EJB that implements the Web service

■ A subset of the methods of the stateless session EJB

■ The WSDL and Home Page of the Web Service

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 by using standard J2EE security procedures.

Note: If the backend component that implements your Web service is a Java class or a JMS listener, the only way to secure the Web service is by adding security constraints to the URL that invokes the Web service, as described in the next section.

For additional and detailed information about configuring, programming, and managing WebLogic security, see the security documentation at http://e-docs.bea.com/wls/docs70/security.html.

Securing Web Service URL

Client applications use a URL to access a Web service, as described in "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22. An example of such a URL is:

```
http://ariel:7001/web_services/TraderService
```

You can restrict access to the entire Web service by restricting access to its URL. To do this, update the web.xml and weblogic.xml deployment descriptor files (in the Web application that contains the web-services.xml file) with security information.

For detailed information about restricting access to URLs, see *Securing WebLogic Resources* at http://e-docs.bea.com/wls/docs70/secwlres/index.html.

Securing the Stateless Session EJB and Its Methods

If you secure the stateless session EJB that implements a Web Service, client applications that invoke the service have access to the Web application, the WSDL, and the Web Service Home Page, but might not be able to invoke the actual method that implements an operation. 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.

You can also use this type of security to decide at the method-level who has access to the various operations of the Web Service. For example, you can specify that any user can invoke a method that views information, but only a certain subset of users are allowed to update the information.

For more information and procedures about securing EJBs and individual methods of an EJB using the Adminstration Console, see *Securing WebLogic Resources* at http://e-docs.bea.com/wls/docs70/secwlres/intro.html.

Securing the WSDL and Home Page of the Web Service

You can restrict access to either the WSDL or Home Page of a WebLogic Web Service by updating the `web-services.xml` deployment descriptor that describes the service, as described in the following procedure:

1. Open the `web-services.xml` file in your favorite editor.

   The `web-services.xml` file is located in the `WEB-INF` directory of the Web application of the Web Services EAR file. See "The Web Service EAR File Package" on page 6-12 for more information on locating the file.

2. To restrict access to the WSDL, add the `exposeWSDL="False"` attribute to the `<web-service>` element that describes your Web Service. To restrict access to the Home page, add the `exposeHomePage="False"` attribute. The following excerpt shows an example:

```
<web-service
   name="stockquotes"
   uri="/myStockQuoteService"
   exposeWSDL="False"
   exposeHomePage="False" >
    ...
</web-service>
```

3. Re-deploy your Web Service for the change to take affect. The WSDL and Home Page of the Web Service will be inaccessible to all users.

Specifying the HTTPS Protocol

You make a Web service accessible only through HTTPS by updating the `protocol` attribute of the `<web-service>` element in the `web-services.xml` file that describes the Web service, as shown in the following excerpt:

```
<web-services>
  <web-service name="stockquotes"
               targetNamespace="http://example.com"
               uri="/myStockQuoteService"
               protocol="https" >
     ...
```

```
        </web-service>
      </web-services>
```

Note: If you configure SSL for WebLogic Server and you do not specify the HTTPS
protocol in the `web-services.xml` file, client applications can access the
Web service using *both* HTTP and HTTPS. However, if you specify HTTPS
access in the `web-services.xml` file, client applications cannot use HTTP to
access the Web service.

If you use the `servicegen` Ant task to assemble the Web service, use the `protocol`
attribute of the `<service>` element to specify the HTTPS protocol, as shown in the
following sample `build.xml` file:

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
       destEar="ws_basic_statelessSession.ear"
       contextURI="WebServices"
       <service
         ejbJar="HelloWorldEJB.jar"
         targetNamespace="http://www.bea.com/webservices/basic/statelesSession"
         serviceName="HelloWorldEJB"
         serviceURI="/HelloWorldEJB"
         protocol="https"
         generateTypes="True"
         expandMethods="True">
       </service>
    </servicegen>
  </target>
</project>
```

Coding a Client Application to Invoke a Secure Web Service

When you write a JAX-RPC client application that invokes a Web service, you use the
following two properties to send a user name and password to the service so that the
client can authenticate itself:

- `javax.xml.rpc.security.auth.username`
- `javax.xml.rpc.security.auth.password`

The following example, taken from the JAX-RPC specification, shows how to use these properties when using the `javax.xml.rpc.Stub` interfaces to invoke a secure Web service:

```
StockQuoteProviderStub sqp = // ... get the Stub;
sqp._setProperty ("javax.xml.rpc.security.auth.username", "juliet");
sqp._setProperty ("javax.xml.rpc.security.auth.password", "mypassword");
float quote sqp.getLastTradePrice("BEAS");
```

If you use the WebLogic-generated client JAR file to invoke a Web service, the Stub classes are already created for you, and you can pass the user name and password to the Service-specific implementation of the `getServicePort()` method, as shown in the following example taken from the JAX-RPC specification:

```
StockQuoteService sqs = // ... Get access to the service;
StockQuoteProvider sqp = sqs.getStockQuoteProviderPort ("juliet", "mypassword");
float quote = sqp.getLastTradePrice ("BEAS");
```

In this example, the implementation of the `getStockQuoteProvidePort()` method sets the two authentication properties.

For additional information on writing a client application using JAX-RPC to invoke a secure Web service, see http://java.sun.com/xml/jaxrpc/index.html.

Configuring SSL for a Client Application

Configure SSL for your client application by using either:

- The WebLogic Server-provided SSL implementation

- A third-party SSL implementation

If you are using two-way SSL, your client application must also present its certificate to WebLogic Server. For details, see "Configuring Two-Way SSL For a Client Application" on page 11-13.

For additional detailed information about the APIs discussed in this section see the Web service security Javadocs at http://e-docs.bea.com/wls/docs70/javadocs/weblogic/webservice/client/package-summary.html.

Using the WebLogic Server-Provided SSL Implementation

WebLogic Server provides an implementation of SSL in the `webserviceclient+ssl.jar` client runtime JAR file. In addition to the SSL implementation, this client JAR file contains the standard client JAX-RPC runtime classes contained in `webservicesclient.jar`.

Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

To configure basic SSL support for your client application, follow these steps:

1. Copy the file *WL_HOME*\server\lib\webserviceclient+ssl.jar to your client application development computer, where *WL_HOME* refers to the top-level directory of WebLogic Platform. This client JAR file contains the client runtime implementation of JAX-RPC as well as the implementation of SSL.

2. Add the client JAR file to the client application's CLASSPATH variable.

3. Set the filename of the file containing trusted Certificate Authority (CA) certificates. Do this by either:

   - Setting the System property `trustedfile` to the name of the file that contains a collection of PEM-encoded certificates.

   - Executing the `BaseWLSSLAdapter.setTrustedCertificatesFile(String ca_filename)` method in your client application.

4. When you run your client application, set the following System properties on the command line:

   - `bea.home=license_file_directory`
   - `java.protocol.handler.pkgs=com.certicom.net.ssl`

   where *license_file_directory* refers to the directory that contains the BEA license file `license.bea`, as shown in the following example:

```
java -Dbea.home=c:\bea_home \
   -Djava.protocol.handler.pkgs=com.certicom.net.ssl  my_app
```

Note: If your client application is running on a computer different from the computer hosting WebLogic Server (which is typically the case), copy the BEA license file from the server computer to a directory on the client computer, and then point the `bea.home` System property to this client-side directory.

5. To disable strict certificate validation, either set the `weblogic.webservice.client.ssl.strictcertchecking` System property to `false` at the command line when you run the application, or programmatically use the `BaseWLSSLAdapter.setStrictCheckingDefault()` method.

For detailed information, see the Web service security Javadocs at http://e-docs.bea.com/wls/docs70/javadocs/weblogic/webservice/client/package-summary.html.

Configuring SSL Programatically

You can also configure the WebLogic Server-provided SSL implementation programatically by using the `weblogic.webservice.client.WLSSLAdapter` adapter class. This adapter class hold configuration information specific to WebLogic Server's SSL implementation and allows the configuration to be queried and modified.

The following excerpt shows an example of configuring the `WLSSLAdapter` class for a specific WebLogic Web service; the lines in bold are discussed after the example:

```
// instantiate an adapter...
WLSSLAdapter adapter  = new WLSSLAdapter();
adapter.setTrustedCertifcatesFile("mytrustedcerts.pem");

// optionally set the Adapter factory to use this
// instance always...
SSLAdapterFactory.getDefaultFactory().setDefaultAdapter(adapter);
SSLAdapterFactory.getDefaultFactory().setUseDefaultAdapter(true);

//create service factory
ServiceFactory factory = ServiceFactory.newInstance();

//create service
Service service = factory.createService( serviceName );

//create call
Call call = service.createCall();

call.setProperty("weblogic.webservice.client.ssladapter",
                 adapter);
```

```
try {

  //invoke the remote web service
  String result = (String) call.invoke( new Object[]{ "BEAS" } );
  System.out.println( "Result: " +result);
} catch (JAXRPCException jre) {
...
}
```

The example first shows how to instantiate an instance of the WebLogic Server-provided `WLSSLAdapter` class, which supports the SSL implementation contained in the `webserviceclient+ssl.jar` file. It then configures the adapter instance by setting the name of the file that contains the Certificate Authority certificates using the `setTrustedCertificatesFile(String)` method; in this case the file is called `mytrustedcerts.pem`.

The example then shows how to set `WLSSLAdapter` as the default adapter of the adapter factory and configures the factory to always return this default.

Note: This step is optional; it allows *all* Web services to share the same adapter class along with its associated configuration.

You can also set the adapter for a particular Web service port or call. The preceding example shows how to do this when using the `Call` class to invoke a Web service dynamically:

```
call.setProperty("weblogic.webservice.client.ssladapter", adapter);
```

Set the property to an object that implements the `weblogic.webservice.client.SSLAdapter` interface (which in this case is the WebLogic Server-provided `WLSSLAdapter` class.)

The following excerpt shows how to set the adapter when using the Stub interface to statically invoke a Web service:

```
((javax.xml.rpc.Stub)stubClass)._setProperty("weblogic.webservice.client.ssslada
pter", adapterInstance);
```

You can get the adapter for a specific instance of a Web service call or port by using the following method for dynamic invocations:

```
call.getProperty("weblogic.webservice.client.ssladapter");
```

Use the following method for static invocations:

```
((javax.xml.rpc.Stub)stubClass)._getProperty("weblogic.webservice.client.ssslada
pter");
```

For detailed information, see the Web service security Javadocs at http://e-docs.bea.com/wls/docs70/javadocs/weblogic/webservice/client/package-summary.html.

Using a Third-Party SSL Implementation

If you want to use a third-party SSL implementation, you must first implement your own adapter class. The following example shows a simple class that provides support for JSSE; the main steps to implementing your own class are discussed after the example:

```
import java.net.URL;
import java.net.Socket;
import java.net.URLConnection;
import java.io.IOException;

public class JSSEAdapter implements weblogic.webservice.client.SSLAdapter {

  javax.net.SocketFactory factory =
    javax.net.ssl.SSLSocketFactory.getDefault();

  // implements weblogic.webservice.client.SSLAdapter interface...

  public Socket createSocket(String host, int port) throws IOException  {
    return factory.createSocket(host, port);
  }

  public URLConnection openConnection(URL url) throws IOException {
    // assumes you have java.protocol.handler.pkgs properly set..
    return url.openConnection();
  }

  // the configuration interface...

  public void setSocketFactory(javax.net.ssl.SSLSocketFactory factory) {
    this.factory = factory;
  }

  public javax.net.ssl.SSLSocketFactory getSocketFactory() {
    return (javax.net.ssl.SSLSocketFactory) factory;
  }
}
```

To create your own adapter class, follow these steps:

1. Create a class that implements the following interface:

```
weblogic.webservice.client.SSLAdapter
```

2. Implement the `createSocket` method, whose signature is:

```
public Socket createSocket(String host, int port)
    throws IOException
```

This method returns an object that extends `java.net.Socket`. The object is connected to the designated hostname and port when a Web service is invoked.

3. Implement the `openConnection` method, whose signature is:

```
public URLConnection openConnection(URL url) throws IOException
```

This method returns an object that extends the `java.net.URLConnection` class. The object is configured to connect to the designated URL. These connections are used for infrequent network operations, such as downloading the Web service WSDL.

4. When you run your client application, set the following `System` property to the fully qualified name of your adapter class:

```
weblogic.webservice.client.ssl.adapterclass
```

The default `SSLAdapterFactory` class loads your adapter class and creates an instance of the class using the default no-argument constructor.

5. Configure your custom adapter class as shown in "Configuring SSL Programatically" on page 11-9, substituting your class for `WLSSLAdapter` and using the configuration methods defined for your adapter.

For detailed information, see the Web service security Javadocs at http://e-docs.bea.com/wls/docs70/javadocs/weblogic/webservice/client/package-summary.html.

Extending the **SSLAdapterFactory** Class

You can create your own custom SSL adapter factory class by extending the `SSLAdapterFactory` class, which is used to create instances of adapters. One reason for extending the factory class is to allow custom configuration of each adapter when it is created, prior to use.

To create a custom SSL adapter factory class, follow these steps:

1. Create a class that extends the following class:

```
weblogic.webservice.client.SSLAdapterFactory
```

2. Override the following method of the SSLAdapterFactory class:

```
public weblogic.webservice.client.SSLAdapter createSSLAdapter();
```

This method is called whenever a new SSLAdapter, or an adapter that implements this interface, is created by the adapter factory. By overriding this method, you can perform custom configuration of each new adapter before it is actually used.

3. In your client application, create an instance of your factory and set it as the default factory by executing the following method:

```
SSLAdapterFactory.setDefaultFactory(factoryInstance);
```

For detailed information, see the Web service security Javadocs at http://e-docs.bea.com/wls/docs70/javadocs/weblogic/webservice/client/package-summary.html.

Configuring Two-Way SSL For a Client Application

If you configured two-way SSL for WebLogic Server, the client application must present a certificate to WebLogic Server, in addition to WebLogic Server presenting a certificate to the client application as required by one-way SSL. The following sample Java code shows one way of doing this where the client application receives the client certificate file as an argument (relevant code in bold):

```
...

SSLAdapterFactory factory = SSLAdapterFactory.getDefaultFactory();
WLSSLAdapter adapter = (WLSSLAdapter) factory.getSSLAdapter();

if (argv.length > 1 ) {
  System.out.println("loading client certs from "+argv[1]);

FileInputStream clientCredentialFile  = new FileInputStream (argv[1]);
String pwd = "clientkey";

adapter.loadLocalIdentity(clientCredentialFile, pwd.toCharArray());

javax.security.cert.X509Certificate[] certChain = adapter.getIdentity("RSA",0);

factory.setDefaultAdapter(adapter);
```

```
factory.setUseDefaultAdapter(true);
```

```
...
```

For a complete example of using two-way SSL with WebLogic Web Services, see the Two-Way SSL Example at http://webservice.bea.com/index.html#qz25.

Using a Proxy Server

If your client application is running inside a firewall, for example, and needs to use a proxy server, set the host name and the port of the proxy server using the following two System properties:

- **weblogic.webservice.transport.https.proxy.host**

- **weblogic.webservice.transport.https.proxy.port**

For more information on these System properties, see "WebLogic Web Services System Properties" on page 8-25.

12 Creating JMS-Implemented WebLogic Web Services

The following sections describe how to create JMS-implemented WebLogic Web services:

- "Designing JMS-Implemented WebLogic Web Services" on page 12-3

- "Implementing JMS-Implemented WebLogic Web Services" on page 12-5

- "Assembling JMS-Implemented WebLogic Web Services Automatically" on page 12-7

- "Assembling JMS-Implemented WebLogic Web Services Manually" on page 12-9

- "Deploying JMS-Implemented WebLogic Web Services" on page 12-13

- "Invoking JMS-Implemented WebLogic Web Services" on page 12-13

Overview of JMS-Implemented WebLogic Web Services

In addition to implementing a Web service operation with a stateless session EJB or a Java class, you can use a JMS message consumer or producer, such as a message-driven bean.

There are three types of JMS-implemented operations:

- Operations that send data to a JMS destination.

  You implement this type of operation with a JMS message consumer. The message consumer consumes the message after a client that invokes the Web service operation sends data to the JMS destination.

- Operations that receive data from a JMS queue.

  You implement this type of operation with a JMS message producer. The message producer puts a message on the specified JMS queue and a client invoking this message-style Web service component polls and receives the message.

- Operations that receive data from a JMS topic.

  You implement this type of operation with a JMS message producer. The message producer publishes a message to the specified JMS topic, and a client invoking this message-style Web service component polls and receives the message.

  Note: Receiving data from a JMS topic is deprecated in this version of WebLogic Server. This means that although this feature currently works, future versions of WebLogic Server might not support it.

  If you are currently using this feature, update your application to use a stateless session EJB as the backend implementation of your Web service rather than a JMS topic. Program the EJB to use the JMS APIs to listen to the JMS topic in the same way that your client applications currently listen to the JMS implemented Web service. Then use the `servicegen` Ant task to reassemble your Web service. Finally, rewrite your client applications to poll the new EJB-implemented Web service for messages from the JMS topic.

When a client application sends data to a JMS-implemented Web service operation, WebLogic Server first converts the XML data to its Java representation using either the built-in or custom serializers, depending on whether the data type of the data is built-in or not. WebLogic Server then wraps the resulting Java object in a `javax.jms.ObjectMessage` object and puts it on the JMS destination. You can then write a JMS listener, such as a message-driven bean, to take the `ObjectMessage` and process it. Similar steps happen in reverse when a client application invokes a Web service to receive data from a JMS queue.

If you are using non-built-in data types, you must update the `web-services.xml` deployment descriptor file with the correct data type mapping information. If the Web service cannot find data type mapping information for the data, then it converts the data to a `javax.xml.soap.SOAPElement` data type, defined by the Java API for XML Messaging (JAXM).

Note: Input and return parameters to a Web service operation implemented with a JMS consumer or producer must implement the `java.io.Serializable` interface.

For detailed information about programming message-driven beans, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/index.html.

Designing JMS-Implemented WebLogic Web Services

This section describes the relationship between JMS and WebLogic Web services operations implemented with a JMS consumer or producer, and design considerations for developing these types of 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.

Before you implement a Web service operation with a JMS consumer or producer as the backend component, you must decide:

- Whether you want to use a JMS queue or topic.

- Whether the client application that invokes the Web service sends the message to or receives the message from the service. The same operation cannot support both sending and receiving.

Retrieving and Processing Messages

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

Because a single Web service operation cannot both send and receive data, you must create two Web service operations if you want a client application to be able to both send data and receive data. The sending Web service operation is related to the receiving one because the original message-driven bean that processed the message puts the response on the JMS destination corresponding to the receiving Web service operation.

Example of Using JMS Components

Figure 12-1 shows two separate Web service operations, one for receiving a message from a client and one for sending a message back to the client. The two Web service operations 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 operation to send the message to WebLogic Server and then invokes the second Web service operation to receive a message back from WebLogic Server.

Figure 12-1 Data Flow Between JMS-Implemented Web Service Operations and JMS
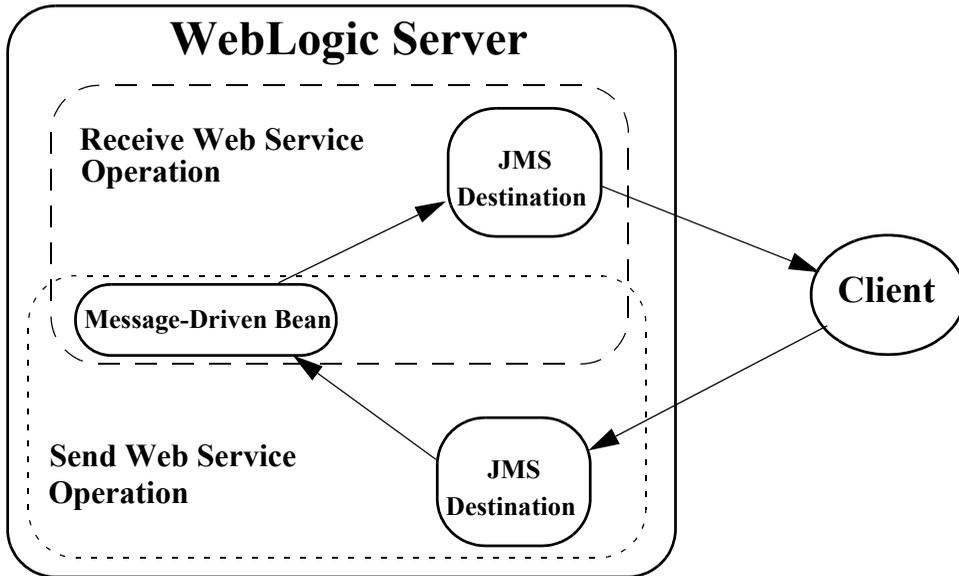
Implementing JMS-Implemented WebLogic Web Services

To implement a Web service implemented with a JMS message producer or consumer, follow these steps:

1. Write the Java code for the J2EE component (typically a message-driven bean) that will consume or produce the message from or on the JMS destination.

   For detailed information, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/index.html.

2. 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 Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks," you will use the name of this JMS destination.

   ● The JMS Connection factory that the WebLogic Web service uses to create JMS connections.

   For more information on this step, see "Configuring JMS Components for Message-Style Web Services" on page 12-6.

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/docs70/adminguide/jms.html and *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs70/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 "To invoke the Administration Console in your browser, enter the following URL:" on page 13-2.

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 JMS-Implemented WebLogic Web Services Automatically

You can use the `servicegen` Ant task to automatically assemble a JMS-implemented Web service. The Ant task creates a `web-services.xml` deployment descriptor file based on the attributes of the `build.xml` file, optionally creates the non-built-in data type components (such as serialization class), optionally creates a client JAR file used to invoke the Web service, and packages everything into a deployable EAR file.

To automatically assemble a Web service using the `servicegen` Ant task:

1. Create a staging directory to hold the components of your Web service.

2. Package your JMS message consumers and producers (such as message-driven beans) into a JAR file.

    For detailed information on this step, refer to *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs70/programming/packaging.html.

3. Copy the JAR file to the staging directory.

4. In the staging directory, create the Ant build file (called `build.xml` by default) that contains a call to the `servicegen` Ant task.

   For details about specifying the `servicegen` Ant task, see "Running the servicegen Ant Task" on page 12-8.

   For general information about creating Ant build files, see http://jakarta.apache.org/ant/manual/.

5. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*/server/bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

6. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   prompt> ant

   The Ant task generates the Web services EAR file in the staging directory which you can then deploy on WebLogic Server.

Running the servicegen Ant Task

The following sample `build.xml`, files shows how to run the `servicegen` Ant task:

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
      destEar="jms_send_queue.ear"
      contextURI="WebServices" >
      <service
          JMSDestination="jms.destination.queue1"
          JMSAction="send"
          JMSDestinationType="queue"
          JMSConnectionFactory="jms.connectionFactory.queue"
          JMSOperationName="sendName"
          JMSMessageType="types.myType"
```

```
            generateTypes="True"
            targetNamespace="http://tempuri.org"
            serviceName="jmsSendQueueService"
            serviceURI="/jmsSendQueue"
            expandMethods="True">
        </service>
      </servicegen>
   </target>
</project>
```

In the example, the `servicegen` Ant task creates a single Web service called `jmsSendQueueService`. The URI to identify this Web service is `/jmsSendQueue`; the full URL to access the Web service is

```
    http://host:port/WebServices/jmsSendQueue
```

The `servicegen` Ant task packages the Web service in an EAR file called `jms_send_queue.ear`. The EAR file contains a WAR file called `web-services.war` (default name) that contains all the Web service components, such as the `web-services.xml` deployment descriptor file.

The Web service exposes a single operation called `sendName`. Client applications that invoke this Web service operation send messages to a JMS queue whose JNDI name is `jms.destination.queue1`. The JMS `ConnectionFactory` used to create the connection to this queue is `jms.connectionFactory.queue`. The data type of the single parameter of the `sendName` operation is `types.myType`. Because the `generateTypes` attribute is set to `True`, the `servicegen` Ant task generates the non-built-in data type components for this data type, such as the serialization class.

Note: The `types.myType` Java class must be in `servicegen`'s CLASSPATH so that `servicegen` can generate appropriate components.

Assembling JMS-Implemented WebLogic Web Services Manually

If you want to assemble a JMS-implemented WebLogic Web service manually, follow these steps:

1. Read this section which describes JMS-specific information about assembling Web services.

2. Follow the steps described in "Assembling WebLogic Web Services Using Other Ant Tasks" on page 6-6, using the JMS-specific information where appropriate.

The following sections describe JMS-specific information about assembling Web services manually.

Packaging the JMS Message Consumers and Producers

Package your JMS message consumers and producers (such as message-driven beans) into a JAR file.

When you create the EAR file that contains the entire Web service, put this JAR file in the top-level directory, in the same location as EJB JAR files.

Updating the web-services.xml File With Component Information

Use the `<components>` child element of the `<web-service>` element to list and describe the JMS backend components that implement the operations of the Web service. Each backend component has a `name` attribute that you later use when describing the operation that the component implements.

See "Sample web-services.xml File for JMS Component Web Service" on page 12-11 for an example.

You can list the following types of backend components for JMS-implemented Web services:

■ `<jms-send-destination>`

This element describes a JMS backend component to which client applications send data. The component puts the sent data on to a JMS destination. Use the `connection-factory` attribute of this element to specify the JMS Connection factory that WebLogic Server uses to create a JMS Connection object. Use the `<jndi-name>` child element to specify the JNDI name of the destination, as shown in the following example:

```
<components>
   <jms-send-destination name="inqueue"
                         connection-factory="myapp.myqueueCF">
      <jndi-name path="myapp.myqueueIN" />
   </jms-send-destination>
</components>
```

- `<jms-receive-queue>`, `<jms-receive-topic>`

   These elements describe two JMS backend components in which client
   applications receive data, the first from a JMS queue and the second from a JMS
   topic. Use the `connection-factory` attribute to specify the JMS Connection
   factory that WebLogic Server users to create a JMS Connection object. Use the
   `<jndi-name>` child element to specify the JNDI name of either the queue or the topic,
   as shown in the following example:

```
<components>
   <jms-receive-queue name="outqueue"
                      connection-factory="myapp.myqueueCF">
      <jndi-name path="myapp.myqueueOUT" />
   </jms-receive-queue>
</components>
```

Sample web-services.xml File for JMS Component Web Service

The following sample `web-services.xml` file describes a Web service that is
implemented with a JMS message consumer or producer:

```
<web-services>
     <web-service targetNamespace="http://example.com"
                  name="myMessageService" uri="MessageService">

        <components>
          <jms-send-destination name="inqueue"
                                connection-factory="myapp.myqueuecf">
              <jndi-name path="myapp.myinputqueue" />
          </jms-send-destination>
          <jms-receive-queue name="outqueue"
                             connection-factory="myapp.myqueuecf">
              <jndi-name path="myapp.myoutputqueue" />
          </jms-receive-queue>
        </components>
```

```
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <operation invocation-style="one-way" name="enqueue"
                        component="inqueue" />
              <params>
                <param name="input_payload" style="in" type="xsd:anyType" />
              </params>
            </operation>
            <operation invocation-style="request-response" name="dequeue"
                        component="outqueue" >
              <params>
                <return-param name="output_payload" type="xsd:anyType"/>
              </params>
            </operation>
        </operations>
      </web-service>
</web-services>
```

The example shows two JMS backend components, one called `inqueue` in which a client application sends data to a JMS destination, and one called `outqueue` in which a client application receives data from a JMS queue.

Two corresponding Web service operations, `enqueue` and `dequeue`, are implemented with these backend components.

The `enqueue` operation is implemented with the `inqueue` component. This operation is defined to be asynchronous one-way, which means that the client application, after sending the data to the JMS destination, does not receive a SOAP response (not even an exception.) The data sent by the client is contained in the `input_payload` parameter.

The `dequeue` operation is implemented with the `outqueue` component. The `dequeue` operation is defined as synchronous request-response because the client application invokes the operation to receive data from the JMS queue. The response data is contained in the output parameter `output_payload`.

Deploying JMS-Implemented WebLogic Web Services

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 *WebLogic Server Deployment* at http://e-docs.bea.com/wls/docs70/programming/deploying.html.

Invoking JMS-Implemented WebLogic Web Services

This section shows two sample client applications that invoke JMS-implemented WebLogic Web services: one that sends data to a service operation, and one to receive data from another operation within the same Web service.   The first operation is implemented with a JMS destination, the second with a JMS queue, as shown in the following `web-services.xml` file that describes the Web service:

```
<web-services xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

  <web-service
    name="BounceService"
    targetNamespace="http://www.foobar.com/echo"
    uri="/BounceService">

    <components>

      <jms-send-destination name="inqueue"
                  connection-factory="weblogic.jms.ConnectionFactory">
          <jndi-name path="weblogic.jms.inqueue" />
      </jms-send-destination>
      <jms-receive-queue name="outqueue"
                  connection-factory="weblogic.jms.ConnectionFactory">
          <jndi-name path="weblogic.jms.outqueue" />
```

```
            </jms-receive-queue>
      </components>

      <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
         <operation invocation-style="one-way" name="submit" component="inqueue" >
         </operation>

         <operation invocation-style="request-response"
                    name="query" component="outqueue" >
            <params>
               <return-param name="output_payload" type="xsd:string"/>
            </params>
         </operation>
      </operations>

  </web-service>

</web-services>
```

Invoking an Asynchronous Web Service Operation to Send Data

The following example shows a dynamic client application that invokes the `submit` operation, described in the `web-services.xml` file in the preceding section. The `submit` operation sends data from the client application to the `weblogic.jms.inqueue` JMS destination. Because the operation is defined as `one-way`, it is asynchronous and does not return any value to the client application that invoked it.

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

/**
 * send2WS - this module sends to a specific Web service connected JMS queue
 * If the message is 'quit' then the module exits.
 *
 * @returns
```

```
 * @throws Exception
 */

public class send2WS{

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

  // Setup the global JAX-RPC service factory
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

  ServiceFactory factory = ServiceFactory.newInstance();

  //define qnames
  String targetNamespace = "http://www.foobar.com/echo";

  QName serviceName = new QName( targetNamespace, "BounceService" );
  QName portName = new QName( targetNamespace, "BounceServicePort" );

  //create service
  Service service = factory.createService( serviceName );

  //create outbound call
  Call Ws2JmsCall = service.createCall();

  QName operationName = new QName( targetNamespace, "submit" );

  //set port and operation name
  Ws2JmsCall.setPortTypeName( portName );
  Ws2JmsCall.setOperationName( operationName );

  //add parameters
  Ws2JmsCall.addParameter( "param",
      new QName( "http://www.w3.org/2001/XMLSchema", "string" ), ParameterMode.IN
);
  //set end point address
  Ws2JmsCall.setTargetEndpointAddress(
        "http://localhost:7001/BounceBean/BounceService" );

  // get message from user
  BufferedReader msgStream =
      new BufferedReader(new InputStreamReader(System.in));
  String line = null;
  boolean quit = false;
  while (!quit) {
    System.out.print("Enter message (\"quit\" to quit): ");
    line = msgStream.readLine();
    if (line != null && line.trim().length() != 0) {
          String result = (String)Ws2JmsCall.invoke( new Object[]{ line } );
      if(line.equalsIgnoreCase("quit")) {
```

```
            quit = true;
            System.out.print("Done!");
        }
      }
    }
  }
}
```

Invoking a Synchronous Web Service Operation to Send Data

The following example shows a dynamic client application that invokes the query
operation, described in the web-services.xml file in "Invoking JMS-Implemented
WebLogic Web Services" on page 12-13. The client application invoking the query
operation receives data from the weblogic.jms.outqueue JMS queue. Because the
operation is defined as request-response, it is synchronous and returns the data
from the JMS queue to the client application.

```
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

/**
 * fromWS - this module receives from a Web service associated JMS queue
 * If the message is 'quit' then the module exits.
 *
 * @returns
 * @throws Exception
 */

public class fromWS {

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

  boolean quit = false;
  // Setup the global JAX-RPC service factory
  System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

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

```
  //define qnames
  String targetNamespace = "http://www.foobar.com/echo";

  QName serviceName = new QName( targetNamespace, "BounceService" );
  QName portName = new QName( targetNamespace, "BounceServicePort" );

  //create service
  Service service = factory.createService( serviceName );

  //create outbound call
  Call Ws2JmsCall = service.createCall();

  QName operationName = new QName( targetNamespace, "query" );

  //set port and operation name
  Ws2JmsCall.setPortTypeName( portName );
  Ws2JmsCall.setOperationName( operationName );

  //add parameters
  Ws2JmsCall.addParameter( "output_payload",
        new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
                    ParameterMode.OUT );
  //set end point address
  Ws2JmsCall.setTargetEndpointAddress(
        "http://localhost:7001/BounceBean/BounceService" );

  System.out.println("Setup complete.  Waiting for a message...");

  while (!quit) {
        String result = (String)Ws2JmsCall.invoke( new Object[] {} );
        if(result != null) {
          System.out.println("TextMessage:" + result);
      if (result.equalsIgnoreCase("quit")) {
            quit = true;
        System.out.println("Done!");
          }
        continue;
        }
    try {Thread.sleep(2000);} catch (Exception ignore) {}
  }
 }
}
```

13 Administering WebLogic Web Services

The following sections describe tasks for administering WebLogic Web services:

Overview of Administering WebLogic Web Services

Once you develop, assemble, and deploy a WebLogic Web service, you can use the Administration Console to view the Web services deployed on WebLogic Server. Additionally, you can use the Administration Console to perform standard WebLogic administration tasks on the deployed Web services.

A Web service is packaged as an EAR file. The EAR file consists of a WAR file that contains the `web-services.xml` file and optional Java classes (such as the Java classes that implement a Web service, handlers, and serialization classes for non-built-in data types) and a optional EJB JAR files that contain the stateless EJBs or JMS consumers and producers that implement the Web service operations.

The Administration Console identifies a Web service by the contents of the WAR file. In other words, if the WAR file contained in an EAR file contains a `web-services.xml` file, then the Administration Console lists the WAR file as a Web service.

For information on standard administration tasks for Web applications, see
Configuring WebLogic Server Web Components at
http://e-docs.bea.com/wls/docs70/adminguide/web_server.html.

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

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

where

- *host* refers to the computer on which the Administration Server is running.

- *port* refers to the port number where the Administration Server is listening for
  connection requests. The default port number for the Administration server is
  7001.

The following figure shows the main Administration Console window.

Figure 13-1   WebLogic Server Administration Console Main Window
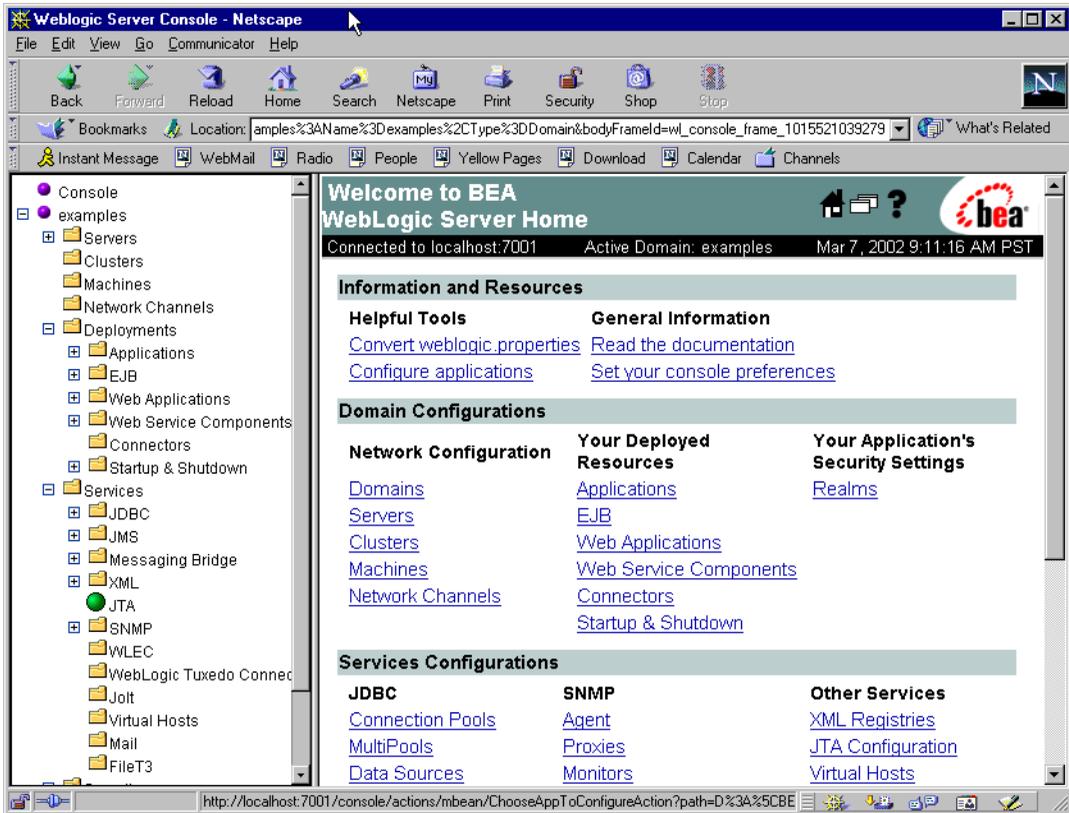
Viewing the Web Services Deployed on WebLogic Server

WebLogic Web services are packaged as EAR files that contain the following components:

- a Web application WAR file that contains the `web-services.xml` deployment descriptor, optional supporting class files (such as SOAP message handler

implementations and the serialization class), and so on. In the Administration Console, this WAR file is referred to as the *Web Service Component*.

■ optional EJB JAR file that contains the EJBs that implement the Web service operations.

To view the Web services deployed on WebLogic server:

1. Invoke the Administration Console in your browser. See "Overview of Administering WebLogic Web Services" on page 13-1.

2. Expand the Deployments->Web Service Components node in the left pane.

   A list of deployed Web services appears below the Web Service Components node, as well as in a table in the right pane.

   The list of Web services corresponds to the list of deployed Web applications that contain the `web-services.xml` deployment descriptor. Other than containing the `web-services.xml` file, a the Web service WAR file is exactly the same as a standard Web application WAR file and is treated the same way in the Administration Console.

   The Application column in the table in the right pane lists the Enterprise Application that contains the Web service component.

   The following figure shows one deployed Web service component, called `trader_service.war`, that is part of an Enterprise application called `_appsdir_webservices_trader_ear`.

Figure 13-2   Deployed Web Service
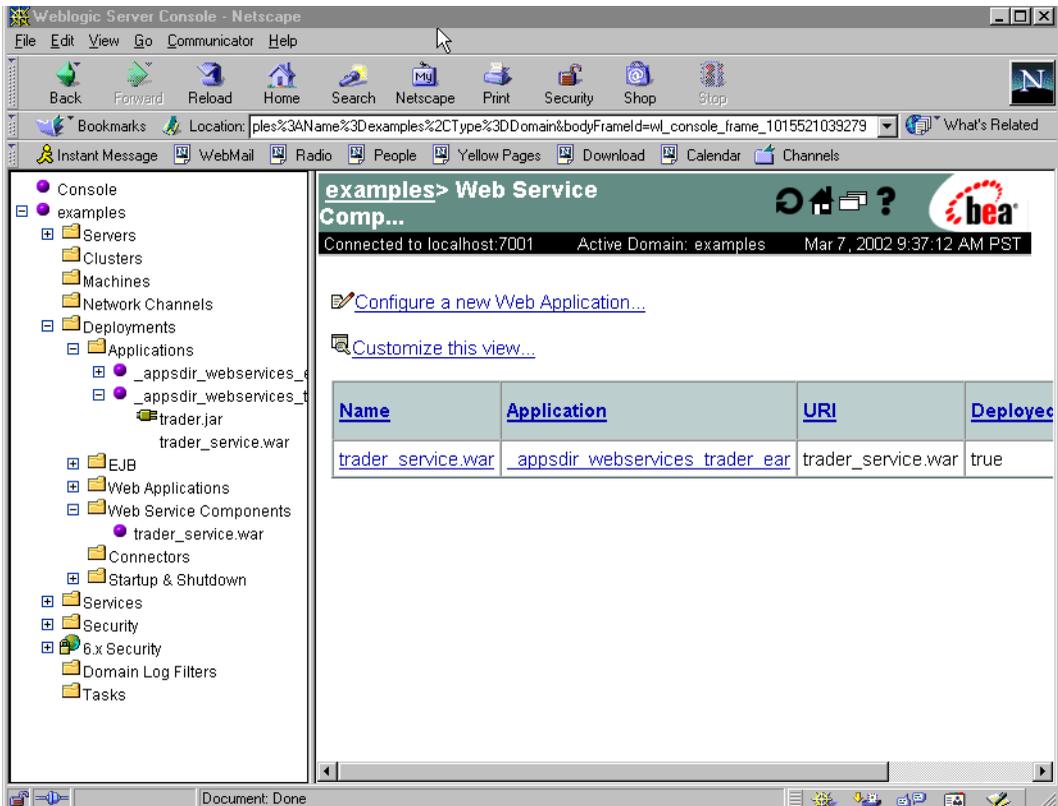
3. To view the other components that make up the Web service, such as the EJB JAR file, expand the Applications node under the Deployments node in the left pane.

4. Click on the name of the Enterprise application that contains the Web service component, such as `_appsdir_webservices_trader_ear` in our example. A list of all the components contained in the application appears under the name of the application in the left pane.

    The preceding figure shows that in addition to the `trader_services.war` component, the `_appsdir_webservices_trader_ear` Enterprise application also contains an EJB JAR file called `trader.jar`. This EJB JAR file contains the EJB backend components that implement some or all of the Web service operations.

14 Publishing and Finding Web Services Using UDDI

The following sections provide information about publishing and finding Web services using UDDI:

- "Introduction to UDDI" on page 14-1

- "WebLogic Server UDDI Features" on page 14-5

- "Invoking the UDDI Directory Explorer" on page 14-5

- "Using the UDDI Client API" on page 14-6

Introduction to UDDI

UDDI stands for Universal Description, Discovery and Integration. The UDDI Project is an industry initiative that is working to enable businesses to quickly, easily, and dynamically find and carry out transactions with one another.

A populated UDDI registry contains cataloged information about businesses, the services that they offer and communication standards and interfaces they use to conduct transactions.

Built on the Simple Object Access Protocol (SOAP) data communication standard, UDDI creates a global, platform-independent, open architecture space that will benefit businesses.

The UDDI registry can be broadly divided into two categories:

- UDDI and Web Services
- UDDI and Business Registry

For details about the UDDI data structure, see "UDDI Data Structure" on page 14-3.

UDDI and Web Services

The owners of Web Services publish them to the UDDI registry. Once published, the UDDI registry maintains pointers to the Web Service description and to the service.

The UDDI allows clients to search this registry, find the intended service and retrieve its details. These details include the service invocation point as well as other information to help identify the service and its functionality.

Web Service capabilities are exposed through a programming interface, and usually explained through Web Services Description Language (WSDL). In a typical publish-and-inquire scenario, the provider publishes its business, registers a service under it and defines a binding template with technical information on its Web Service. The binding template also holds reference to one or several *tModels*, which represent abstract interfaces implemented by this Web Service. The tModels might have been uniquely published by the provider, with information on the interfaces and URL references to the WSDL document.

A typical client inquiry may have one of two objectives:

1. To seek an implementation of a known interface.

   In other words, the client has a tModel ID and seeks binding templates referencing that tModel.

2. To seek the updated value of the invocation point (i.e., access point) of a known binding template ID.

UDDI and Business Registry

As a Business Registry solution, UDDI enables companies to advertise the business products and services they provide, as well as how they conduct business transactions

on the Web. This use of UDDI has the potential of fueling growth of business-to-business (B2B) electronic commerce.

The minimum required information to publish a business is a single business name. Once completed, a full description of a business entity may contain a wealth of information, all of which helps to advertise the business entity and its products and services in a precise and accessible manner.

A Business Registry may contain the following:

- **Business Identification**—Multiple names and descriptions of the business, comprehensive contact information and standard business identifiers such as a tax identifier.

- **Categories**—Standard categorization information (for example a D-U-N-S business category number).

- **Service Description**—Multiple names and descriptions of a service. As a container for service information, companies can advertise numerous services, while clearly displaying the ownership of services. The `bindingTemplate` information describes how to access the service.

- **Standards Compliance**—In some cases it is important to specify compliance with standards. These standards might display detailed technical requirements on how to use the service.

- **Custom Categories**—It is possible to publish proprietary specifications (tModels) that identify or categorize businesses or services.

UDDI Data Structure

The data structure within UDDI is comprised of four constructions: a `businessEntity` structure, a `businessService` structure, a `bindingTemplate` structure and a `tModel` structure.

The following table outlines the difference between these constructions when used for Web Service or Business Registry applications.

Table 14-1 UDDI Data Structure

Data Structure	Web Service	Business Registry
businessEntity	Represents a Web Service provider: • Company name • Contact detail • Other business information	Represents a company, a division or a department within a company: • Company name(s) • Contact details • Identifiers and Categories
businessService	A logical group of one or several Web Services. API(s) with a single name stored as a child element, contained by the business entity named above.	A group of services may reside in a single businessEntity. • Multiple names and descriptions • Categories • Indicators of compliancy with standards
bindingTemplate	A single Web Service. Information provided here gives client applications the technical information needed to bind and interact with the target Web Service. Contains access point (i.e., URI to invoke a Web Service).	Further instances of standards conformity. Access points for the service in form of URLs, phone numbers, email addresses, fax numbers or other similar address types.
tModel	Represents a technical specification; typically a specifications pointer, or metadata about a specification document, including a name and a URL pointing to the actual specifications. In the context of Web Services, the actual specifications document is presented in the form of a WSDL file.	Represents a standard or technical specification, either well established or registered by a user for specific use.

WebLogic Server UDDI Features

Weblogic Server provides the following UDDI features:

■ A UDDI Directory Explorer

■ A UDDI registry

■ An implementation of the client-side UDDI API so you can programmatically search for and publish Web services

The UDDI Directory Explorer allows authorized users to publish Web services in private WebLogic Server UDDI registries and to modify information for previously published Web services.

The UDDI Directory Explorer also enables you to search both public and private UDDI registries for Web services and information about the companies and departments that provide these Web services. The Directory Explorer also provides access to details about the Web services and associated WSDL files (if available.)

Invoking the UDDI Directory Explorer

To invoke the UDDI Directory Explorer in your browser, enter the following URL:

```
http://host:port/uddiexplorer
```

where

■ `host` refers to the computer on which WebLogic Server is running.

■ `port` refers to the port number where WebLogic Server is listening for connection requests. The default port number is 7001.

You can perform the following tasks with the UDDI Directory Explorer:

■ Search public registries

■ Search private registries

- Publish to a private registry

- Modify private registry details

- Setup UDDI directory explorer

For more information about using the UDDI Directory Explorer, click the Help link on the main page.

Using the UDDI Client API

Use the UDDI client API in a Java client application to search for and publish Web Services.

The two main classes of the UDDI client API are `Inquiry` and `Publish`. Use the `Inquiry` class to search for Web Services in a known UDDI registry and the `Publish` class to add your Web Service to a known registry.

WebLogic Server provides an implementation of the following client UDDI API packages:

- `weblogic.uddi.client.service`

- `weblogic.uddi.client.structures.datatypes`

- `weblogic.uddi.client.structures.exception`

- `weblogic.uddi.client.structures.request`

- `weblogic.uddi.client.structures.response`

For detailed information on using these packages, see the UDDI API Javadocs at http://e-docs.bea.com/wls/docs70/javadocs/index.html.

For examples of using the UDDI client API, go to the Web Services dev2dev Download Page at http://webservice.bea.com and scroll down until you find the following examples:

- UDDI Client API Example

- UDDI Publish Example

- UDDI Inquire Example

15 Interoperability

The following sections provide an overview of what it means for Web services to be interoperable and tips on creating Web services that interoperate with each other as much as possible:

- "Overview of Interoperability" on page 15-1

- "Avoid Using Vendor-Specific Extensions" on page 15-2

- "Stay Current With the Latest Interoperability Tests" on page 15-3

- "Understand the Data Models of Your Applications" on page 15-3

- "Understand the Interoperability of Various Data Types" on page 15-4

- "Results of SOAPBuilders Interoperability Lab Round 3 Tests" on page 15-5

- "Interoperating With .NET" on page 15-6

Overview of Interoperability

A fundamental characteristic of Web services is that they are interoperable. This means that a client can invoke a Web service regardless of the client's hardware or software. In particular, interoperability demands that the functionality of a Web service application be the same across differing:

- Application platforms, such as BEA WebLogic Server, IBM Websphere, or Microsoft .NET.

- Programming languages, such as Java, C++, C#, or Visual Basic.

- Hardware, such as mainframes, PCs, or peripheral devices.

- Operating systems, such as different flavors of UNIX or Windows.

- Application data models.

For example, an interoperable Web service running on WebLogic Server on a Sun Microsystems computer running Solaris can be invoked from a Microsoft .NET Web service client written in Visual Basic.

To ensure the maximum interoperability, WebLogic Server supports the following specifications and versions when generating your Web service:

- HTTP 1.1 for the transport protocol

- XML Schema to describe your data

- WSDL 1.1 to describe your Web service

- SOAP 1.1 for the message format

The following sections provide some useful interoperability tips and information when writing Web service applications.

Avoid Using Vendor-Specific Extensions

Avoid using vendor-specific implementation extensions to specifications (such as SOAP, WSDL, and HTTP) that are used by Web services. If your Web service relies on this extension, a client application that invokes it might not use the extension and the invoke might fail.

Stay Current With the Latest Interoperability Tests

Public interoperability tests provide information about how different vendor implementations of Web service specifications interoperate with each other. This information is very useful if you are creating a Web service on WebLogic Server that has to, for example, interoperate with Web services from other vendors, such as .NET.

Warning: BEA's participation in these interoperability tests does not imply that BEA officially certifies its Web Services implemenation against the other platforms participating in the tests.

The following Web sites include public interoperability tests:

■ Web Service Interoperability Organization at http://www.ws-i.org/

■ SoapBuilder Interoperability Lab at http://www.whitemesa.net/

You can also use the vendor implementations listed in these Web sites to exhaustively test your Web service for interoperability.

Understand the Data Models of Your Applications

A good use of Web services is to provide a cross-platform technology for integrating existing applications. These applications typically have very different data models which your Web service must reconcile.

For example, assume that you are creating a Web service application to integrate the two accounting systems in a large company.   Although the data models of each accounting system are probably similar, they most likely differ in at least some way, such as the name of a data field, the amount of information stored about each customer, and so on. It is up to the programmer of the Web service to understand each data model, and then create an intermediate data model to reconcile the two. Typically this

intermediate data model is expressed in XML using XML Schema. If you base your Web service application on only one of the data models, the two applications probably will not interoperate very well.

Understand the Interoperability of Various Data Types

The data types of the parameters and return values of your Web service operations have a great impact on the interoperability of your Web service. The following table describes how interoperable the various types of data types are.

Table 15-1 Interoperability of Various Types of Data Types

Data Type	Description
JAX-RPC built-in data types	Interoperate with no additional programming.
	The JAX-RPC specification defines a subset of the XML Schema built-in data types that any implementation of JAX-RPC must support. Because all of these data types map directly to a SOAP-ENC data type, they are interoperable.
Built-in WebLogic Server data types	Interoperate with no additional programming.
	WebLogic Server includes support for all the XML Schema built-in data types. Because all of these data types map directly to a SOAP-ENC data type, they are interoperable.
	For the full list of built-in WebLogic Server data types, see "Using Built-In Data Types" on page 5-12.

Table 15-1 Interoperability of Various Types of Data Types

Data Type	Description
Non-built-in data types	Interoperate with additional programming or tools support.
	If your Web service uses non-built-in data types, you must create the XML Schema that describes the XML representation of the data, the Java class that describes the Java representation, and the serialization class that converts the data between its XML and Java representation. WebLogic Server includes the `servicegen` and `autotype` Ant tasks that automatically generate these objects. Keep in mind, however, that these Ant tasks might generate an XML Schema that does not interoperate well with client applications or it might not be able to create an XML Schema at all if the Java data type is very complex. In these cases you might need to manually create the objects needed by non-built-in data types, as described in Chapter 9, "Using Non-Built-In Data Types."
	Additionally, you must ensure that client applications that invoke your Web service include the serialization class needed to convert the data between its XML representation and the language-specific representation of the client application. WebLogic Server can generate the serialization class for Weblogic client applications with the `clientgen` Ant task. If, however, the client applications that invoke your Web service are not written in Java, then you must create the serialization class manually.

Results of SOAPBuilders Interoperability Lab Round 3 Tests

For the results of WebLogic Web services' participation in the SOAPBuilders Interoperability Lab Round 3 tests, see http://webservice.bea.com:7001. The tests were run with version 7.0.0.1 of WebLogic Server.

For the test results, see http://webservice.bea.com/index.html#qz41; for the source code of the tests, see http://webservice.bea.com/index.html#qz40.

For more information on the SOAPBuilder Interoperability tests, see
http://www.whitemesa.net.

Warning: BEA's participation in these interoperability tests does not imply that
BEA officially certifies its Web Services implemenation against the other
platforms participating in the tests.

Interoperating With .NET

WebLogic Web Services interoperate seamlessly with .NET Web Services.

You invoke a .NET Web Service from a WebLogic Web Services client application
exactly as described in Chapter 8, "Invoking Web Services." When you execute the
clientgen Ant task to generate the Web Service-specific client JAR file, use the wsdl
attribute to specify the URL of the WSDL of the deployed .NET Web Service.

To invoke a deployed WebLogic Web Service from a .NET client application, use
Microsoft Visual Studio .NET to create an application, then add a Web Reference,
specifying the WSDL of the deployed WebLogic Web Service, as described in the
following example. In Microsoft Visual Studio, adding a Web Reference is equivalent
to executing the WebLogic clientgen Ant task.

Warning: The following example describes one way to invoke a WebLogic Web
Service from a .NET client application. For the most current and detailed
information about using Microsoft Visual Studio .NET to invoke
WebLogic (and other) Web Services, consult the Microsoft
documentation at
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro
7/html/vxconATourOfVisualStudio.asp.

1. Start and use Microsoft Visual Studio .NET to create your application as usual.

2. In the Solution Explorer in the right pane, right-click your application and chose
   Add Web Reference. The Solution Explorer Browser appears.

3. Enter the WSDL of the deployed WebLogic Web Service in the Solution
   Explorer Browser. As soon as the browser accepts the WSDL, the Add
   Reference button becomes active.

See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22 for information on getting the WSDL of a deployed WebLogic Web Service.

4. Click the Add Reference button. The WebLogic Web Service appears in the Solution Explorer.

5. In your application component that will be used to invoke the Web Service, such as a button, add Visual C# or Visual Basic code to invoke a particular operation of the Web Service. Visual Studio .NET uses statement completion to help you write this code. The following Visual C# code excerpt shows a simple example of invoking the `echoString` operation of the `SoapInteropBaseService` Web Service:

```
WebReference1.SoapInteropBaseService s = new SoapInteropBaseService();
string s = s.echoString("Hi there!");
```

In the example, `WebReference1` is the name of the Web Reference you added in preceding steps.

16 Upgrading 6.1 WebLogic Web Services to 7.0

The following sections describe how to upgrade a 6.1 WebLogic Web service to 7.0:

- "Overview of Upgrading 6.1 WebLogic Web Services" on page -1

- "Upgrading 6.1 WebLogic Web Services to 7.0 Automatically" on page -2

- "Upgrading 6.1 WebLogic Web Services to 7.0 Manually" on page -4

- "Converting a 6.1 build.xml file to 7.0" on page -5

- "Updating the URL Used to Access the Web Service" on page -7

Overview of Upgrading 6.1 WebLogic Web Services

Due to changes in the Web service runtime system between Versions 6.1 and 7.0 of WebLogic Server, you should upgrade Web services created in version 6.1 to run on version 7.0. This chapter describes the upgrade process.

There are two ways to upgrade a 6.1 Web service:

- Automatically, using the `wsgen` Ant task.   The Ant task uses default values for any required 7.0 `web-services.xml` attributes that have no corresponding 6.1 attribute.

  For details, see "Upgrading 6.1 WebLogic Web Services to 7.0 Automatically" on page -2.

- Manually, by rewriting the `build.xml` file you used to create a 6.1 Web service to now call the `servicegen` Ant task rather than the `wsgen` Ant task

  For details, see "Upgrading 6.1 WebLogic Web Services to 7.0 Manually" on page -4.

Note: You cannot deploy a 6.1 Web service on a 7.0 WebLogic Server instance.

The WebLogic Web services client API included in version 6.1 of WebLogic Server has been deprecated and you should not use it to invoke 7.0 Web services. Version 7.0 includes a new client API, based on the Java API for XML based RPC (JAX-RPC). You should rewrite client applications that used the 6.1 Web services client API to now use the JAX-RPC APIs. For details, see Chapter 8, "Invoking Web Services."

For more detailed information on the differences between 6.1 and 7.0 Web services, see Chapter 1, "Overview of WebLogic Web Services."

Upgrading 6.1 WebLogic Web Services to 7.0 Automatically

Use the `wsgen` Ant task to automatically upgrade a Web service from 6.1 to 7.0. It takes as input the `build.xml` file you used to assemble the 6.1 Web service, and based on the values in this file, the task:

- Generates a `web-services.xml` file that describes the Web service. The task uses default values for any required elements or attributes in the deployment descriptor that do not apply to 6.1 Web services. For details on these default values, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

■ Optionally creates a client JAR file that includes the Web service-specific classes, stubs, and interfaces needed by client applications that invoke the Web service. The classes, stubs, and interfaces are based on the JAX-RPC API.

■ Packages all the Web service components into a Web application WAR file, then packages the WAR and EJB JAR files into a deployable EAR file.

To upgrade a 6.1 WebLogic Web service to 7.0 automatically, follow these steps:

1. Create a temporary staging directory.

2. If you are upgrading a 6.1 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. Copy the 6.1 `build.xml` file that contains a call to the `wsgen` Ant task to the staging directory.

4. If, in your 6.1 `build.xml` file, you explicitly declared the `wsgen` Ant task using the `<taskdef>` element, as shown in the following example:

```
<taskdef name="wsgen" classname="weblogic.ant.taskdefs.ejb.WSGen"/>
```

remove this `<taskdef>` element from the `build.xml` file.

5. Optionally add the `targetNameSpace` and `packageName` attributes to the `<wsgen>` element of the `build.xml` file. For descriptions of these optional attributes, see "wsgen" on page B-36.

6. Set your WebLogic Server 7.0 environment.

On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*`\server\bin`, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

On UNIX, execute the `setWLSEnv.sh` command, located in the directory *WL_HOME*`/server/bin`, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

7. Execute the `wsgen` Ant task in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

```
prompt> ant
```

The Ant task generates an EAR file that you can deploy on WebLogic Server 7.0.

8. In your client application, update the URL you use to access the Web service or the WSDL of the Web service from that used in 6.1 to 7.0. For details, see "Updating the URL Used to Access the Web Service" on page -7.

Upgrading 6.1 WebLogic Web Services to 7.0 Manually

To upgrade a 6.1 WebLogic Web service to 7.0 manually, follow these steps:

1. Convert the `build.xml` Ant build file used to assemble 6.1 Web services with the `wsgen` Ant task to the 7.0 version that calls the `servicegen` Ant task.

   For details see "Converting a 6.1 build.xml file to 7.0" on page -5.

2. Un-jar the 6.1 Web services EAR file and extract the EJB JAR file that contains the stateless session EJBs (for 6.1 RPC-style Web services) or message-driven beans (for 6.1 message-style Web services), along with any supporting class files.

3. If your 6.1 Web service was RPC-style, see "Assembling WebLogic Web Services Using the servicegen Ant task" on page 6-3 for instruction on using the `servicegen` Ant task.

   If your 6.1 Web service was message-style, you must assemble the 7.0 Web service manually. For detailed instructions, see Chapter 12, "Creating JMS-Implemented WebLogic Web Services."

4. In your client application, update the URL you use to access the Web service or the WSDL of the Web service from that used in 6.1 to 7.0. For details, see "Updating the URL Used to Access the Web Service" on page -7.

Converting a 6.1 build.xml file to 7.0

The main difference between the 6.1 and 7.0 `build.xml` files used to assemble a Web service is the Ant task: in 6.1 the task was called `wsgen` and in 7.0 it is called `servicegen`. The `servicegen` Ant task uses many of the same elements and attributes of `wsgen`, although some do not apply anymore. The `servicegen` Ant task also includes additional configuration options. The table at the end of this section describes the mapping between the elements and attributes of the two Ant tasks.

Note: Although in 6.1 you were able to use the `wsgen` Ant task to assemble a message-style Web service, the `servicegen` Ant task does not have equivalent 7.0 functionality for JMS implemented backend components. Therefore, if you are upgrading a 6.1 message-style Web service, there is no need to convert the old `build.xml` file.

The following `build.xml` excerpt is from the 6.1 RPC-style Web services example:

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

The following example shows an equivalent 7.0 `build.xml` file:

```
<project name="myProject" default="servicegen">
    <target name="servicegen">
       <servicegen
           destEar="weather.ear"
           contextURI="weather" >
           <service
               ejbJar="weather.jar"
               serviceURI="/weatheruri"
               includeEJBs="statelessSession" >
           </service>
       </servicegen>
```

```
     </target>
</project>
```

For detailed information on the WebLogic Web service Ant tasks, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

The following table maps the 6.1 `wsgen` elements and attributes to their equivalent 7.0 `servicegen` elements and attributes.

Table 0-1 6.1 to 7.0 wsgen Ant Task Mapping

6.1 wsgen Element	Attribute	Equivalent 7.0 servicegen element	Attribute
wsgen	basepath	No equivalent.	No equivalent
	destpath	servicegen	destEar
	context	servicegen	contextURI
	protocol	servicegen.service	protocol
	host	No equivalent.	No equivalent
	port	No equivalent.	No equivalent
	webapp	servicegen	warName
	classpath	servicegen	classpath
rpcservices	module	No equivalent.	No equivalent
	path	servicegen.service	ejbJar
rpcservice	bean	servicegen.service	includeEJBS, excludeEJBs
	uri	servicegen.service	serviceURI
messageservices	N/A	No equivalent.	No equivalent
messageservice	All attributes.	No equivalent.	No equivalent
clientjar	path	servicegen.service.client	clientJarName

Updating the URL Used to Access the Web Service

The default URL used by client applications to access a WebLogic Web service and its WSDL has changed between versions 6.1 and 7.0 of WebLogic Server.

In Version 6.1, the default URL was:

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

as described in *URLs to Invoke WebLogic Web Services and Get the WSDL* at http://e-docs.bea.com/wls/docs61/webServices/client.html#client008.

For example, the URL to invoke a 6.1 Web service built with the `build.xml` file shown in "Converting a 6.1 build.xml file to 7.0" on page -5, is:

```
http://host:port/weather/statelessSession.WeatherHome/statelessSession.WeatherH
ome.wsdl
```

In 7.0, the default URL is:

    [*protocol*]://[*host*]:[*port*]/[*contextURI*]/[*serviceURI*]?WSDL

as described in "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22.

For example, the URL to invoke the equivalent 7.0 Web service after converting the 6.1 `build.xml` file shown in "Converting a 6.1 build.xml file to 7.0" on page -5 and running `wsgen` is:

```
http://host:port/weather/weatheruri?WSDL
```

A WebLogic Web Service Deployment Descriptor Elements

The `web-services.xml` deployment descriptor file contains information that describes one or more WebLogic Web services. This information includes details about the backend components that implement the operations of a Web service, the non-built-in data types used as parameters and return values, the SOAP message handlers that intercept SOAP messages, and so on. As is true for all deployment descriptors, `web-services.xml` is an XML file.

The following sections describe the `web-services.xml` file using different formats:

- "Graphical Representation" on page A-1

- "Element Reference" on page A-3

Graphical Representation

The following graphic describes the `web-services.xml` element hierarchy.

(Continued)
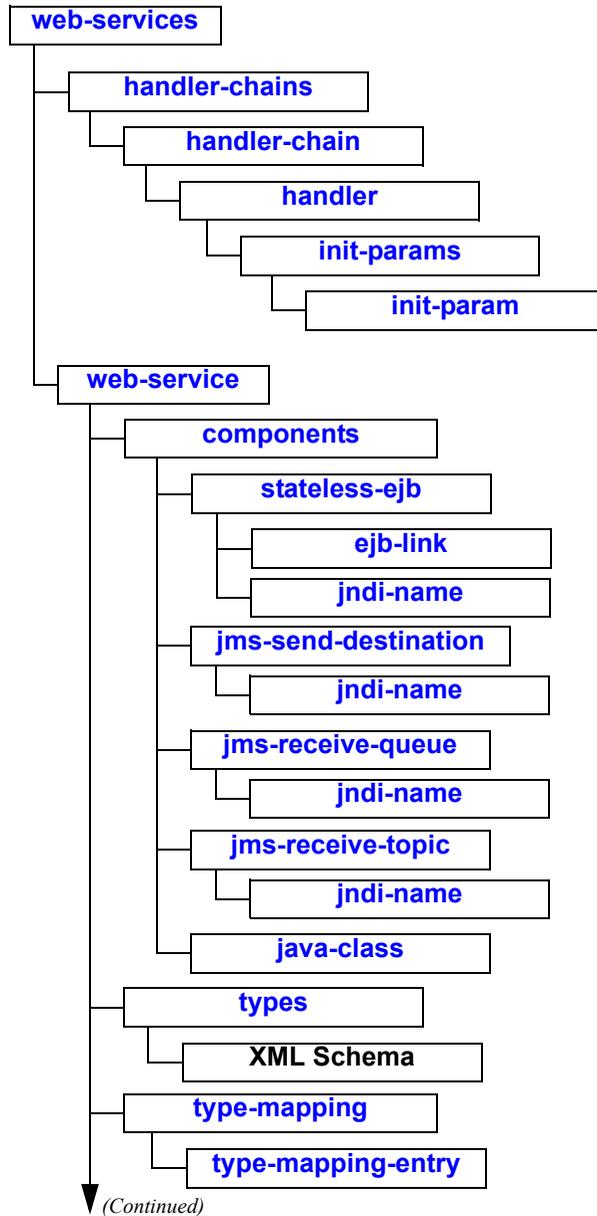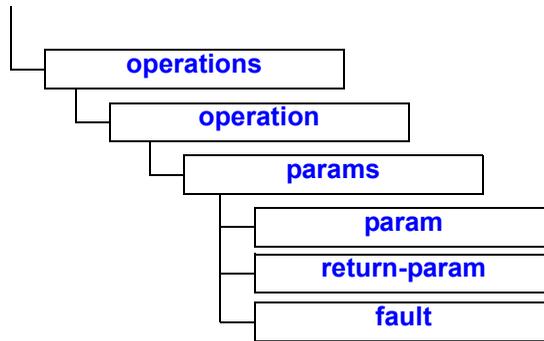
Element Reference

The following sectins, arranged alphabetically, describe each element in the `web-services.xml` file.

See for sample Web services deployment descriptor files for a variety of different types of WebLogic Web services.

components

Defines the backend components that implement the Web service.

A WebLogic Web service can be implemented using one or more of the following components:

- Stateless session EJB

- JMS destination

- A Java class

This element has no attributes.

ejb-link

Identifies which EJB in an EJB JAR file is used to implement the stateless session EJB backend component.

Attribute	Description	Datatype	Required?
path	Name of the EJB in the form of: *jar-name*#*ejb-name* *jar-name* refers to the name of the JAR file, contained within the Web service EAR file, that contains the stateless session EJB. The name should include pathnames relative to the top level of the EAR file. *ejb-name* refers to the name of the stateless session EJB, corresponding to the <ejb-name> element in the ejb-jar.xml deployment descriptor file in the EJB JAR file. Example: myapp.jar#StockQuoteBean	String	Yes

fault

Specifies the SOAP fault that should be thrown if there is an error invoking this operation.

This element is not required.

Attribute	Description	Datatype	Required?
name	Name of the fault.	String	Yes
class-name	Fully qualified Java class that implements the SOAP fault.	String	Yes

handler

Describes a SOAP message handler in a handler chain. A single handler chain can consist of one or more handlers.

If the Java class that implements the handler expects initialization parameters, specify them using the optional `<init-params>` child element of the `<handler>` element.

Attribute	Description	Datatype	Required?
class-name	Fully qualified Java class that implements the SOAP message handler.	String	Yes

handler-chain

Lists the SOAP message handlers that make up a particular handler chain. A single WebLogic Web service can define zero or more handler chains.

The order in which the handlers (defined by the `<handler>` child element) are listed is important. By default, the `handleRequest()` methods of the handlers execute in the order that they are listed as child elements of the `<handler-chain>` element. The `handleResponse()` methods of the handlers execute in the *reverse* order that they are listed.

Attribute	Description	Datatype	Required?
name	Name of this handler chain.	String	Yes

handler-chains

Contains a list of `<handler-chain>` elements that describe the SOAP message handler chains used in the Web service described by this `web-services.xml` file. A single WebLogic Web service can define zero or more handler chains.

This element does not have any attributes.

init-param

Specifies a name-value pair that represents one of the initialization parameters of a handler.

Attribute	Description	Datatype	Required?
name	Name of the parameter.	String	Yes
value	Value of the parameter.	String	Yes

init-params

Contains the list of initialization parameters that are passed to the Java class that implements a handler.

This element does not have any attributes.

java-class

Describes the Java class component that implements one or more operations of a Web service.

Attribute	Description	Datatype	Required
name	Name of this component.	String	Yes
class-name	Fully qualified name of the Java class that implements this component.	String	Yes

jms-receive-queue

Specifies that one of the operations in the Web service is mapped to a JMS queue. Use this element to describe a Web service operation that receives data from a JMS queue.

Typically, a message producer puts a message on the specified JMS queue, and a client invoking this Web service operation polls and receives the message.

Attribute	Description	Datatype	Required?
name	Name of this component.	String	Yes
connection-factory	JNDI name of the JMS Connection factory that WebLogic Server uses to create a JMS Connection object.	String	Yes
provider-url	URL used to connect to a non-WebLogic Server JMS implementation.	String	No
initial-context-factory	Context factory for a non-WebLogic Server JMS implementation.	String	No

jms-receive-topic

Specifies that one of the operations in the Web service is mapped to a JMS topic. Use this element to describe a Web service operation that receives data from a JMS topic.

Typically, a message producer puts a message on the specified JMS topic, and a client invoking this Web service component polls and receives the message.

Attribute	Description	Datatype	Required?
name	Name of this component.	String	Yes
connection-factory	JNDI name of the JMS Connection factory that WebLogic Server uses to create a JMS Connection object.	String	Yes
provider-url	URL used to connect to a non-WebLogic Server JMS implementation.	String	No
initial-context-factory	Context factory for a non-WebLogic Server JMS implementation.	String	No

jms-send-destination

Specifies that one of the operations in the Web service is mapped to a JMS destination (either a queue or a topic). Use this element to describe a Web service operation that sends data to a JMS destination.

Typically, a message consumer (such as a message-driven bean) consumes the message after it is sent to the JMS destination.

Attribute	Description	Datatype	Required?
name	Name of this component.	String	Yes
connection-factory	JNDI name of the JMS Connection factory that WebLogic Server uses to create a JMS Connection object.	String	Yes
provider-url	URL used to connect to a non-WebLogic Server JMS implementation.	String	No
initial-context-factory	Context factory for a non-WebLogic Server JMS implementation.	String	No

jndi-name

Specifies a reference to an object bound into a JNDI tree. The reference can be to a stateless session EJB or to a JMS destination.

Attribute	Description	Datatype	Required?
path	Path name to the object from the JNDI context root.	String	Yes

operation

Configures a single operation of a Web service. Depending on the value and combination of attributes for this element, you can configure the following types of operations:

- An invoke of a method of a stateless session EJB or Java class. Specify this type of operation by setting the `component` attribute to the name of the stateless session EJB or Java class component and the `method` attribute to the name of the method.

- An invoke of a JMS backend component. Specify this type of operation by setting the `component` attribute to the name of the JMS component.

- The sequential invoke of the SOAP message handlers on a handler chain together with the invoke of a backend component. Specify this type of operation by setting the `component` attribute to the name of the component, and the `handler-chain` attribute to the name of the handler chain you want to invoke.

- The sequential invoke of the SOAP message handlers on a handler chain, but with *no* backend component. Specify this type of operation by just setting the `handler-chain` attribute to the name of the handler chain you want to invoke and *not* setting the `component` and `method` attributes.

Use the `<params>` child element to explicitly specify the parameters and return values of the operation.

Attribute	Description	Datatype	Required?
name	Name of the operation that will be used in the generated WSDL. If you do not specify this attribute, the name of the operation defaults to either the name of the method or the name of the SOAP message handler chain.	String	No
component	Name of the component that implements this operation. The value of this attribute corresponds to the `name` attribute of the appropriate `<component>` element.	String	No

Attribute	Description	Datatype	Required?

method	Name of the method of the EJB or Java class that implements the operation if you specify with the `component` attribute that the operation is implemented with a stateless session EJB or Java class.
	
	You can specify all the methods with the asterisk (*) character.
	
	If your EJB or Java class does *not* overload the method, you need only specify the name of the method, such as:
	
	`method="sell"`
	
	If, however, the EJB or Java class overloads the method, then specify the full signature, such as:
	
	`method="sell(int)"`
		String	No

handler-chain	Name of the SOAP message handler chain that implements the operation. The value of this attribute corresponds to the `name` attribute of the appropriate `<handler-chain>` element.	String	No
invocation-style	Specifies whether the operation both receives a SOAP request and sends a SOAP response, or whether the operation only receives a SOAP request but does *not* send back a SOAP response. This attribute accepts only two values: `request-response` (default value) or `one-way`. **Note:** If the backend component that implements this operation is a method of a stateless session EJB or Java class and you set this attribute to `one-way`, the method must return `void`	String	No

Attribute	Description	Datatype	Required?
portTypeName	Port type in the WSDL file to which this operation belongs. You can include this operation in multiple port types by specifying a comma-separated list of port types. When the WSDL for this Web service is generated, a separate `<portType>` element is created for each specified port type. The default value is the value of the `portType` attribute of the `<web-service>` element.	String	No

operations

The `<operations>` element groups together the explicitly declared operations of this Web service.

This element does not have any attributes.

param

The `<param>` element specifies a single parameter of an operation.

You must list the parameters in the same order in which they are defined in the method that implements the operation. The number of `<param>` elements must match the number of parameters of the method.

Attribute	Description	Datatype	Required?
name	Name of the input parameter that will be used in the generated WSDL. If you do not specify this attribute, the parameter names are based on the data type of the parameter, such as `intvalue1`, `intvalue2`, `traderesult`, and so on.	String	No.

Attribute	Description	Datatype	Required?
location	Part of the request SOAP message (either the header, the body, or the attachment) that contains the value of the input parameter. Valid values for this attribute are `Body`, `Header`, or `attachment`. The default value is `Body`. If you specify `Body`, the value of the parameter is extracted from the SOAP Body, according to regular SOAP rules for RPC operation invocation. If you specify `Header`, the value is extracted from a SOAP Header element whose name is the value of the `type` attribute. If you specify `attachment`, the value of the parameter is extracted from the SOAP Attachment rather than the SOAP envelope. As specified by the JAX-RPC specification, only the following Java data types can be extracted from the SOAP Attachment: ■ `java.awt.Image` ■ `java.lang.String` ■ `javax.mail.internet.MimeMultiport` ■ `javax.xml.transform.Source` ■ `javax.activation.DataHandler`	String	No.
style	Style of the input parameter, either a standard input parameter, an out parameter used as a return value, or an in-out parameter for both inputting and outputting values. Valid values for this attribute are `in`, `out`, and `in-out`. If you specify a parameter as `out` or `in-out`, the Java class of the parameter in the backend component's method must implement the `javax.xml.rpc.holders.Holder` interface.	String	Yes.
type	XML Schema data type of the parameter.	NMTOKEN	Yes.

Attribute	Description	Datatype	Required?
class-name	Java class name of the Java representation of the data type of the parameter. If you do not specify this attribute, WebLogic Server introspects the backend component that implements the operation for the Java class of the parameter. You are required to specify this attribute only if you want the mapping between the XML and Java representations of the parameter to be different than the default. For example, xsd:int maps to the Java primitive int type by default, so use this attribute to map it to java.lang.Integer instead.	NMTOKEN	Maybe. See the description of the attribute.

params

The <params> element groups together the explicitly declared parameters and return values of an operation.

You do not have to explicitly list the parameters or return values of an operation. If an <operation> element does not have a <params> child element, WebLogic Server introspects the backend component that implements the operation to determine its parameters and return values. When generating the WSDL file of the Web service, WebLogic Server uses the names of the corresponding method's parameters and return value.

You explicitly list an operation's parameters and return values when you want:

- The name of the parameters and return values in the generated WSDL to be different from those of the method that implements the operation.

- To map a parameter to a name in the SOAP header request or response.

- To use out or in-out parameters.

Use the <param> child element to specify the parameters of the operation.

Use the <return-param> child element to specify the return value of the operation.

The `<params>` element does not have any attributes.

return-param

The `<return-param>` element specifies the return value of the Web service operation.

You can specify only one `<return-param>` element for a given operation.

Attribute	Description	Datatype	Required?
name	Name of the return parameter that will be used in the generated WSDL file. If you do not specify this attribute, the return parameter is called `result`.	String	No.
location	Part of the response SOAP message (either the header or the body) that contains the value of the return parameter. Valid values for this attribute are `Body` or `Header`. The default value is `Body`. If you specify `Body`, the value of the return parameter will be added to the SOAP Body. If you specify `Header`, the value will added as a SOAP Header element whose name is the value of the `type` attribute.	String	No.
type	XML Schema data type of the return parameter.	NMTOKEN	Yes.

Attribute	Description	Datatype	Required?
class-name	Java class name of the Java representation of the data type of the return parameter. If you do not specify this attribute, WebLogic Server introspects the backend component that implements the operation for the Java class of the return parameter. You are required to specify this attribute if: ■ The backend component that implements the operation is either `<jms-receive-queue>` or `<jms-receive-topic>`. ■ The mapping between the XML and Java representations of the return parameter is ambiguous, such as mapping `xsd:int` to either the `int` Java primitive type or `java.lang.Integer`.	NMTOKEN	Maybe. See the description of the attribute.

stateless-ejb

Describes the stateless session EJB component that implements one or more operations of a Web service.

Attribute	Description	Datatype	Required?
name	Name of the stateless EJB component. **Note:** The name is internal to the `web-services.xml` file; it does not refer to the name of the EJB in the `ejb-jar.xml` file.	String	Yes.

type-mapping

The `<type-mapping>` element contains the list of mappings between the XML data types defined in the `<types>` element and their Java representations.

For each data type in the `<types>` element, there is a corresponding `<type-mapping-entry>` element that lists the Java class that implements the data type, how to serialize and deserialize the data, and so on.

This element has no attributes.

type-mapping-entry

Describes the mapping between a single XML data type in the `<types>` element and its Java representation.

Attribute	Description	Datatype	Required?
class-name	Fully qualified name of the Java class that maps to its corresponding XML data type.	String	Yes.
element	Name of the XML data type that maps to the Java data type. Specify only if the XML Schema definition of the data type uses the `<element>` element.	NMTOKEN	One, but not both, of either `element` or `type` is required.
type	Name of the XML data type that maps to the Java data type. Specify only if the XML Schema definition of the data type uses the `<type>` element.	NMTOKEN	One, but not both, of either `element` or `type` is required.
serializer	Fully qualified name of the Java class that converts the data from Java to XML.	String	Only required if the data type is *not* one of the built-in data dates supported by the WebLogic Web services runtime, listed in "Using Built-In Data Types" on page 5-12.

Attribute	Description	Datatype	Required?
deserializer	Fully qualified name of the Java class that converts the data from XML to Java.	String	Only required if the data type is *not* one of the built-in data dates supported by the WebLogic Web services runtime, listed in "Using Built-In Data Types" on page 5-12.

types

Describes, using XML Schema notation, the non-built-in data types used as parameters or return types of the Web service operations.

For details on using XML Schema to describe the XML representation of a non-built-in data type, see http://www.w3.org/TR/xmlschema-0/.

The following example shows an XML Schema declaration of a data type called TradeResult that contains two elements: stockSymbol, a string data type, and numberTraded, an integer.

```
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:stns="java:examples.webservices"
             attributeFormDefault="qualified"
             elementFormDefault="qualified"
             targetNamespace="java:examples.webservices">
    <xsd:complexType name="TradeResult">
        <xsd:sequence>
            <xsd:element maxOccurs="1"
                         name="stockSymbol"
                         type="xsd:string" minOccurs="1">
            </xsd:element>
            <xsd:element maxOccurs="1"
                         name="numberTraded"
                         type="xsd:int"
                         minOccurs="1">
            </xsd:element>
        </xsd:sequence>
```

```
              </xsd:complexType>
          </xsd:schema>
      </types>
```

web-service

Defines a single Web service.

The Web service is defined by the following:

- Backend components that implement an operation, such as a stateless session EJB, a Java class, or a JMS consumer or producer..

- An optional set of data type declarations for non-built-in data types used as parameters or return values to the Web service operations.

- An optional set of XML to Java data type mappings that specify the serialization class and Java classes for the non-built-in data types.

- A declaration of the operations supported by the Web service.

Attribute	Description	Datatype	Required?
name	Name of the Web service.	String	Yes.
targetNamespace	Namespace of this Web service.	String	Yes.
uri	URI of the Web service, used subsequently in the URL that invokes the Web service. **Note:** Be sure to specify the leading "/", such as `/TraderService`.	String	Yes.
protocol	Protocol over which the service is invoked. Valid values are `http` or `https`. Default is `http`.	String	No.
exposeHomePage	Specifies whether to publicly expose the Home Page of the Web Service. Valid values for this attribute are `True` and `False`. The default value is `True`. This means that by default the Home Page is publicly accessible.	Boolean	No.

Attribute	Description	Datatype	Required?
exposeWSDL	Specifies whether to publicly expose the automatically generated WSDL of the Web Service. Valid values for this attribute are `True` and `False`. The default value is `True`. This means that by default the WSDL is publicly accessible.	Boolean	No.
style	Specifies whether the Web service has RPC-oriented or document-oriented operations. RPC-oriented WebLogic Web service operations use SOAP encoding. Document-oriented WebLogic Web service operations use literal encoding. Valid values are `rpc` and `document`. Default value is `rpc`. **Warning:** If you specify `document` for this attribute, all the methods that implement the operations of the Web service must have only *one* parameter. **Note:** Because the `style` attribute applies to an entire Web service, all operations specified in a single `<web-service>` element must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web service.	String	No.
portName	Name of the `<port>` child element of the `<service>` element of the dynamically generated WSDL of this Web service. The default value is the `name` attribute of this element with `Port` appended. For example, if the name of this Web service is `TraderService`, the port name will be `TraderServicePort`.	String	No

Attribute	Description	Datatype	Required?
portTypeName	Name of the default `<portType>` element in the dynamically generated WSDL of this Web service. The default value is the `name` attribute of this element with `Port` appended. For example, if the name of this Web service is `TraderService`, the portType name will be `TraderServicePort`.	String	No.
ignoreAuthHeader	Specifies that the Web Service ignore the `Authorization` HTTP header in the SOAP request. **Note:** Be careful using this attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it. Valid values are `True` and `False`. Default value is `False`.	Boolean.	No.

web-services

The root element of the `web-services.xml` deployment descriptor.

This element does not have any attributes.

B Web Service Ant Tasks and Command-Line Utilities

The following sections describe WebLogic Web service Ant tasks and the command-line utilities based on these Ant tasks:

- "Overview of WebLogic Web Services Ant Tasks and Command-Line Utilities" on page B-2

- "autotype" on page B-7

- "clientgen" on page B-12

- "servicegen" on page B-19

- "source2wsdd" on page B-28

- "wsdl2Service" on page B-31

- "wspackage" on page B-33

- "wsgen" on page B-36

Overview of WebLogic Web Services Ant Tasks and Command-Line Utilities

Ant is a Java-based build tool, similar to the `make` command but much more powerful. Ant uses XML-based configuration files (called `build.xml` by default) to execute tasks written in Java.

BEA provides a number of Ant tasks that help you generate important parts of a Web service (such as the serialization class, a client JAR file, and the `web-services.xml` file) and to package all the pieces of a WebLogic Web service into a deployable EAR file.

The Apache Web site provides other useful Ant tasks for packaging EAR, WAR, and EJB JAR files. For more information, see http://jakarta.apache.org/ant/manual/.

You can also run some of the Ant tasks as a command-line utility, using flags rather than attributes to specify how the utility works. The description of the flags is exactly the same as the description of its corresponding attribute.

Warning: Not all the attributes of the Ant tasks are available as flags to the equivalent command-line utility. See the sections that describe each Ant task for a list of the supported flags when using the command-line equivalent.

For further examples and explanations of using these Ant tasks, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

List of Web Services Ant Tasks and Command-Line Utilities

The following table provides an overview of the Web service Ant tasks provided by BEA and the name of the corresponding command-line utility.

Table B-1 WebLogic Web Services Ant Tasks

Ant Task	Corresponding Command-Line Utility	Description
autotype	Not available.	Generates the serialization class, Java representation, XML Schema representation, and data type mapping information for non-built-in data types used as parameters or return values to a WebLogic Web service.
clientgen	weblogic.webservice.clientgen	Generates a client JAR file that contains a thin Java client used to invoke a Web service.
servicegen	weblogic.webservice.servicegen	Main Ant task that performs all the steps needed to assemble a Web service. These steps include: ■ Creating the Web service deployment descriptor (`web-services.xml`). ■ Introspecting EJBs and Java classes and generating any needed non-built-in data type supporting components. ■ Generating the client JAR file. ■ Packaging all the pieces into a deployable EAR file.
source2wsdd	Not available	Generates a `web-services.xml` deployment descriptor file from the Java source file for a Java class-implemented WebLogic Web service.
wsdl2Service	Not available.	Generates the components of a WebLogic Web service from a WSDL file. The components include the `web-services.xml` deployment descriptor file and a Java source file that you can use as a starting point to implement the Web service.
wspackage	Not available.	Packages the components of a WebLogic Web service into a deployable EAR file.

Table B-1 WebLogic Web Services Ant Tasks

Ant Task	Corresponding Command-Line Utility	Description
wsgen	Not available.	Upgrades a 6.1 WebLogic Web service to Version 7.0 of WebLogic Server.

Using the Web Services Ant Tasks

To use the Ant tasks, follow these steps:

1. Create a file called `build.xml` that contains a call to the Web services Ant tasks.

   The following example shows a simple `build.xml` file (with details of the Web services Ant tasks `servicegen` and `clientgen` omitted for clarity):

```
<project name="buildWebservice" default="build-ear">
  <target name="build-ear">
    <servicegen attributes go here...>
      ...
    </servicegen>
  </target>
  <target name="build-client" depends="build-ear">
    <clientgen attributes go here .../>
  </target>
  <target name="clean">
    <delete>
      <fileset dir="."
               includes="example.ear,client.jar" />
    </delete>
  </target>
</project>
```

   Later sections provide examples of specifying the Ant task in the `build.xml` file.

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory *WL_HOME*\server\bin, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

On UNIX, execute the `setWLSEnv.sh` command, located in the directory `WL_HOME`/server/bin, where `WL_HOME` is the top-level directory of your WebLogic Platform installation.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

```
prompt> ant
```

Setting the Classpath for the WebLogic Ant Tasks

Each WebLogic Ant task accepts a `classpath` attribute or element so that you can add new directories or JAR files to your current CLASSPATH environment variable.

The following example shows how to use the `classpath` attribute of the `servicegen` Ant task to add to the CLASSPATH variable:

```
<servicegen destEar="myEJB.ear"
            classpath="${java.class.path};d:\my_fab_directory"
   ...
</servicegen>
```

The following example shows how to add to the CLASSPATH by using the `<classpath>` element:

```
<servicegen ...>
   <classpath>
       <pathelement path="${java.class.path}" />
       <pathelement path="d:\my_fab_directory" />
   </classpath>
...
</servicegen>
```

The following example shows how you can build your CLASSPATH variable outside of the WebLogic Web service Ant task declarations, then specify the variable from within the task using the `<classpath>` element:

```
<path id="myid">
   <pathelement path="${java.class.path}"/>
   <pathelement path="${additional.path1}"/>
   <pathelement path="${additional.path2}"/>
</path>

<servicegen ....>
   <classpath refid="myid" />
```

```
...
</servicegen>
```

Note: The Java Ant utility included in WebLogic Server uses the `ant` (UNIX) or
`ant.bat` (Windows) configuration files in the `WL_HOME`\server\bin
directory when setting the ANTCLASSPATH variable, where `WL_HOME` is the
top-level directory of your WebLogic Platform installation   If you need to
update the ANTCLASSPATH variable, make the relevant changes to the
appropriate file for your operating system.

Differences in Operating System Case Sensitivity When Manipulating WSDL and XML Schema Files

Many of the WebLogic Web Service Ant tasks have attributes that you can use to
specify an operating system file, such as a WSDL or an XML Schema file. For
example, you can use the `wsdl` attribute of the `clientgen` Ant task to create the Web
Services-specific client JAR file from an existing WSDL file that describes a Web
Service.

The Ant tasks process these files in a case-sensitive way. This means that if, for
example, the XML Schema file specifies two complex types whose names differ only
in their capilatization (for example, `MyReturnType` and `MYRETURNTYPE`), the
`clientgen` Ant task correctly generates two separate sets of Java source files for the
Java represenation of the complex data type: `MyReturnType.java` and
`MYRETURNTYPE.java`.

However, compiling these source files into their respective class files might cause a
problem if you are running the Ant task on Microsoft Windows, because Windows is
a case insensitive operating system. This means that Windows considers the files
`MyReturnType.java` and `MYRETURNTYPE.java` to have the same name. So when you
compile the files on Windows, the second class file overwrites the first, and you end
up with only one class file. The Ant tasks, however, expect that two classes were
compiled, thus resulting in an error similar to the following:

```
c:\src\com\bea\order\MyReturnType.java:14:
class MYRETURNTYPE is public, should be declared in a file named
MYRETURNTYPE.java
public class MYRETURNTYPE
       ^
```

To work around this problem rewrite the XML Schema so that this type of naming conflict does not occur, or if that is not possible, run the Ant task on a case sensitive operating system, such as Unix.

Using the Web Services Command-Line Utilities

To use the command-line utility equivalents of the Ant tasks, follow these steps:

1. Open a command shell window.

2. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory `WL_HOME\server\bin`, where `WL_HOME` is the top-level directory of your WebLogic Platform installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory `WL_HOME/server/bin`, where `WL_HOME` is the top-level directory of your WebLogic Platform installation.

3. Execute the utility using the `java` command, as shown in the following example:

```
prompt> java weblogic.webservice.clientgen \
            -ear c:\myapps\myapp.ear \
            -serviceName myService \
            -packageName myservice.client \
             -clientJar c:/myapps/myService_client.jar
```

Run the command with no arguments to get a usage message.

autotype

The `autotype` Ant task generates the following components for non-built-in data types that used as parameters or return values of your Web service operation:

■ Serialization class that converts between the XML and Java representation of the data.

- Given an XML Schema or WSDL file, a Java class to contain the Java representation of the data type.

- Given a Java class that represents the non-built-in data type, an XML Schema representation of the data type.

- Data type mapping information to be included in the web-services.xml deployment descriptor file.

For the list of non-built-in data types for which autotype can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.

You can specify one of the following types of input to the autotype Ant task:

- A Java class file that represents your non-built-in data types by specifying the javaTypes attribute. The autotype Ant task generates the corresponding XML Schemas, the serializer classes, and the data type mapping information for the web-services.xml file.

- A Java class file that contains a backend component, such as a stateless session EJB, by specifying the javaComponents attribute. The autotype Ant task looks for non-built-in data types used in the component, then generates the corresponding XML Schemas, the serializer classes, and the data type mapping information for the web-services.xml file.

- An XML Schema file that represents your non-built-in data type by specifying the schemaFile attribute. The autotype Ant task generates the corresponding Java representations, the serializer classes, and the data type mapping information for the web-services.xml file.

- A URL to a WSDL file that contains a description of your non-built-in data type by specifying the wsdlURI attribute. The autotype Ant task generates the corresponding Java representations, the serializer classes, and the data type mapping information for the web-services.xml file.

Use the destDir attribute to specify the name of a directory that contains the generated components. The generated XML Schema and data type mapping information are generated in a file called types.xml. You can use this file to manually update an existing web-services.xml file with non-built-in data type mapping information, or use it in conjunction with the typeMappingFile attribute of the servicegen or clientgen Ant tasks, or the typesInfo attribute of the source2wsdd Ant task.

Warning: The serializer class and Java and XML representations generated by the
autotype, servicegen, and clientgen Ant tasks cannot be
round-tripped. For more information, see "Non-Roundtripping of
Generated Data Type Components" on page 6-17.

Note: The fully qualified name for the autotype Ant task is
weblogic.ant.taskdefs.webservices.javaschema.JavaSchema.

Example

The following example shows how to create non-built-in data type components for the
Java class mypackage.MyType:

```
<autotype  javaTypes="mypackage.MyType"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

The following example is similar to the preceding one, except it creates non-built-in
data type components for an *array* of mypackage.MyType Java data types:

```
<autotype  javaTypes="[Lmypackage.MyType;"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

Note: The [L*classname*; syntax follows the Java class naming conventions as
outlined in the java.lang.Class.getName() method documentation.

The following example shows how to use the autotype Ant task against a WSDL file:

```
<autotype  wsdl="file:\wsdls\myWSDL"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="d:\output" />
```

Attributes

The following table describes the attributes of the `autotype` Ant task.

Attribute	Description	Required?
schemaFile	Name of a file that contains the XML Schema representation of your non-built-in data types.	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.
wsdl	Full path name or URI of the WSDL that contains the XML Schema description of your non-built-in data type.	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.
javaTypes	Comma-separated list of Java class names that represent your non-built-in data types. The Java classes must be compiled and in your CLASSPATH. For example: `javaTypes="my.class1,my.class2"` **Note:** Use the syntax `[Lclassname;` to specify an array of the Java data type. For an example, see "Example" on page B-9.	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.
javaComponents	Comma-separated list of Java class names that implement the Web service operation. The Java classes must be compiled and in your CLASSPATH. For example: `javaComponents="my.class1,my.class2"` The `autotype` Ant task introspects the Java classes to automatically generate the components for all non-built-in data types it finds.	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.

Attribute	Description	Required?
destDir	Full pathname of the directory that will contain the generated components. The generated XML Schema and data type mapping information are generated in a file called `types.xml`.	Yes.
typeMappingFile	File that contains data type mapping information for non-built-in data types for which have already generated needed components. The format of the information is the same as the data type mapping information in the `<type-mapping>` element of the `web-services.xml` file. The `autotype` Ant task does not generate non-built-in data type components for any data types listed in this file.	No.
packageBase	Base package name of the generated Java classes for any non-built-in data types used as a return value or parameter in a Web service. This means that each generated Java class will be part of the same package name, although the `autotype` Ant task generates its own specific name for each Java class which it appends to the specified package base name. If you do not specify this attribute, the `autotype` Ant task generates a base package name for you. **Note:** BEA recommends you not use this attribute, but rather, specify the full package name using the `packageName` attribute. The `packageBase` attribute is available for JAX-RPC compliance.	No. If you specify this attribute, you cannot also specify `packageName`.

Attribute	Description	Required?
packageName	Full package name of the generated Java classes for any non-built-in data types used as a return value or parameter in a Web service. If you do not specify this attribute, the `autotype` Ant task generates a package name for you. **Note:** Although not required, BEA recommends you specify this attribute in most cases. Currently, the only situation in which you should *not* specify this attribute is if you use the `javaTypes` attribute to specify a list of Java data types whose class names are the same, but their package names are different. In this case, if you also specify the `packageName` attribute, the `autotype` Ant task generates a serialization class for only the last class.	No. If you specify this attribute, you cannot also specify `packageBase`.
targetNamespace	Namespace URI of the Web service.	Yes.

clientgen

The `clientgen` Ant task generates a Web service-specific client JAR file that client applications can use to invoke both WebLogic and non-WebLogic Web services. Typically, you use the `clientgen` Ant task to generate a client JAR file from an existing WSDL file; you can also use it with an EAR file that contains the implementation of a WebLogic Web service.

The contents of the client JAR file includes:

- Client interface and stub files (conforming to the JAX-RPC specification) used to invoke a Web service in static mode.

- Optional serialization class for converting non-built-in data between its XML and Java representation.

- Optional client-side copy of the Web service WSDL file

You can use the `clientgen` Ant task to generate a client JAR file from the WSDL file of an existing Web service (not necessarily running on WebLogic Server) or from an EAR file that contains a Weblogic Web service implementation.

The WebLogic Server distribution includes a client runtime JAR file that contains the client side classes needed to support the WebLogic Web services runtime component. For more information, see "Getting the Java Client JAR Files" on page 8-4.

Warning: The serializer class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-17.

Warning: The `clientgen` Ant task does not support solicit-response or notification WSDL operations. This means that if you attempt to create a client JAR file from a WSDL file that contains these types of operations, the Ant task ignores the operations.

Note: The fully qualified name of the `clientgen` Ant task is `weblogic.ant.taskdefs.webservices.clientgen.ClientGenTask`.

Example

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="c:/myapps/myService_client.jar"
/>
```

Attributes

The following table describes the attributes of the clientgen Ant task.

Attribute	Description	Required?
wsdl	Full path name or URL of the WSDL that describes a Web service (either WebLogic or non-WebLogic) for which a client JAR file should be generated. The generated stub factory classes in the client JAR file use the value of this attribute in the default constructor.	Either wsdl or ear must be specified.
ear	Name of an EAR file or exploded directory that contains the WebLogic Web service implementation for which a client JAR file should be generated. **Note:** If the saveWSDL attribute of clientgen is set to True (the default value), the clientgen Ant task generates a WSDL file from the information in the EAR file, and stores it in the generated client JAR file. Because clientgen does not know the host name or port number of the WebLogic Server instance which will host the Web service, clientgen uses the following endpoint address in the generated WSDL: http://localhost:7001/*contextURI*/*serviceURI* where *contextURI* and *serviceURI* are the same values as described in "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22. If this endpoint address is not correct, and your client application uses the WSDL file stored in the client JAR file, you must manually update the WSDL file with the correct endpoint address.	Either wsdl or ear must be specified.
warName	Name of the WAR file which contains the Web service(s). The default value is web-services.war.	No. You can specify this attribute only in combination with the ear attribute.

Attribute	Description	Required?
serviceName	Web service name for which a corresponding client JAR file should be generated. If you specify the `wsdl` attribute, the Web service name corresponds to the `<service>` elements in the WSDL file. If you specify the `ear` attribute, the Web service name corresponds to the `<web-service>` element in the `web-services.xml` deployment descriptor file. If you do not specify the `serviceName` attribute, the `clientgen` task generates client classes for the first service name found in the WSDL or `web-services.xml` file.	No.
typeMappingFile	File that contains data type mapping information, used by the `clientgen` task when generating the JAX-RPC stubs. The format of the information is the same as the data type mapping information in the `<type-mapping>` element of the `web-services.xml` file. If you specified the `ear` attribute, the information in this file overrides the data type mapping information found in the `web-services.xml` file.	No.
packageName	Package name into which the generated JAX-RPC client interfaces and stub files should be packaged.	Yes.
autotype	Specifies whether the `clientgen` task should generate and include in the client JAR file the serialization class for any non-built-in data types used as parameters or return values to the Web service operations. Valid values are `True` and `False`. Default value is `True`.	No.
clientJar	Name of a JAR file or exploded directory into which the `clientgen` task puts the generated client interface classes, stub classes, optional serialization class, and so on. To create or update a JAR file, use a `.jar` suffix when specifying the JAR file, such as `myclientjar.jar`. If the attribute value does not have a `.jar` suffix, then the `clientgen` task assumes you are referring to a directory name. If you specify a JAR file or directory that does not exist, the `clientgen` task creates a new JAR file or directory.	Yes.
overwrite	Specifies whether to overwrite an existing client JAR file. Valid values are `True` and `False`. Default value is `True`.	No.

Attribute	Description	Required?
useServerTypes	Specifies where the clientgen task gets the implementation of any non-built-in Java data types used in a Web service: either the task generates the Java code or the task gets it from the EAR file that contains the full implementation of the Web service. Valid values are True (use the Java code in the EAR file) and False. Default value is False. For the list of non-built-in data types for which clientgen can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.	No. Use only in combination with the ear attribute.
saveWSDL	When set to True, specifies that the WSDL of the Web service be saved in the generated client JAR file. This means that client applications do not need to download the WSDL every time they create a stub to the Web service, possibly improving performance of the client because of reduced network usage. Valid values are True and False. Default value is True.	No.
j2me	Specifies whether the clientgen Ant task should create a J2ME/CDC-compliant client JAR file. **Note:** The generated client code is not JAX-RPC compliant. Valid values are True and False. Default value is False.	No.
useLowerCaseMethodNames	When set to true, specifies that the method names in the generated stubs have a lower-case first character. Otherwise, all method names will the same as the operation names in the WSDL file. Valid values are True and False. Default value is True.	No.

Attribute	Description	Required?
typePackageName	Specifies the full package name of the generated Java class for any non-built-in data types used as a return value or parameter in a Web service. If you specify this attribute, you cannot also specify `typePackageBase`. If you do not specify this attribute *and* the XML Schema in the WSDL file defines a target namespace, then the `clientgen` Ant task generates a package name for you based on the target namespace. This means that if your XML Schema does *not* define a target namespace, then you must specify either the `typePackageName` (preferred) or `typePackageBase` attributes of the `clientgen` Ant task. **Note:** Although not required, BEA recommends you specify this attribute.	Required *only* if you specified the `wsdl` attribute and the XML Schema in the WSDL file does not define a target namespace.
typePackageBase	Specifies the base package name of the generated Java class for any non-built-in data types used as a return value or parameter in a Web service. This means that each generated Java class will be part of the same package name, although the `clientgen` Ant task generates its own specific name for each Java class which it appends to the specified package base name. If you specify this attribute, you cannot also specify `typePackageName`. If you do not specify this attribute *and* the XML Schema in the WSDL file defines a target namespace, then the `clientgen` Ant task generates a package name for you based on the target namespace. This means that if your XML Schema does *not* define a target namespace, then you must specify either the `typePackageName` (preferred) or `typePackageBase` attributes of the `clientgen` Ant task. **Note:** Rather than using this attribute, BEA recommends that you specify the full package name with the `typePackageName` attribute. The `typePackageBase` attribute is available for JAX-RPC compliance.	Required *only* if you specified the `wsdl` attribute and the XML Schema in the WSDL file does not define a target namespace.

Attribute	Description	Required?
usePortNameAsMethodName	Specifies where the `clientgen` Ant task should get the names of the operations when generating a client from a WSDL file. If this value is set to true, then operations take the name specified by the `name` attribute of the `<port>` element in the WSDL file (where `<port>` is the child element of the `<service>` element). If `usePortNameAsMethodName` is set to false, then operations take the name specified by the `name` attribute of the `<portType>` element in the WSDL file (where `<portType>` is the child element of the `<definitions>` element). Valid values are `True` and `False`. Default value is `False`.	No.

Equivalent Command-Line Utility

The equivalent command-line utility of the `clientgen` Ant task is called `weblogic.webservice.clientgen`. The description of the flags of the utility is the same as the description of the Ant task attributes, described in the preceding section.

The `weblogic.webservice.clientgen` utility supports the following flags (see the equivalent attribute for a description of the flag):

- `-wsdl` *uri*

- `-ear` *pathname*

- `-clientJar` *pathname*

- `-packageName` *name*

- `-warName` *name*

- `-serviceName` *name*

- `-typeMappings` *pathname*

- `-useServerTypes`

servicegen

The `servicegen` Ant task takes as input an EJB JAR file or list of Java classes, and creates all the needed Web service components and packages them into a deployable EAR file.

In particular, the `servicegen` Ant task:

- Introspects the EJBs and Java classes, looking for public methods to convert into Web service operations.

- Creates a `web-services.xml` deployment descriptor file, based on the attributes of the `servicegen` Ant task and introspected information.

- Optionally creates the serialization class that converts the non-built-in data between its XML and Java representations. It also creates XML Schema representations of the Java objects and updates the `web-services.xml` file accordingly. This feature is referred to as *autotyping*.

- Optionally creates a client JAR file that includes the Web service-specific classes, stubs, and interfaces needed by client applications that invoke the Web service. The classes, stubs, and interfaces are based on the JAX-RPC API.

- Packages all the Web service components into a Web application WAR file, then packages the WAR and EJB JAR files into a deployable EAR file.

Warning: The serializer class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-17.

Note: The fully qualified name of the `servicegen` Ant task is `weblogic.ant.taskdefs.webservices.servicegen.ServiceGenTask`.

Example

```
<servicegen
    destEar="c:\myWebService.ear"
    warName="myWAR.war"
```

```
            contextURI="web_services" >
            <service
                ejbJar="c:\myEJB.jar"
                targetNamespace="http://www.bea.com/examples/Trader"
                serviceName="TraderService"
                serviceURI="/TraderService"
                generateTypes="True"
                expandMethods="True" >
            </service>
        </servicegen>
```

Attributes and Child Elements

The servicegen Ant task has four attributes and one child element (<service>) for each Web service you want to define in a single EAR file. You must specify at least one <service> element.

The <service> element has one optional <client> element.

The following graphic describes the hierarchy of the servicegen Ant task.
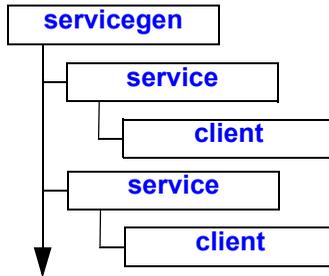
servicegen

The servicegen Ant task is the main task for automatically generating and assembling all the parts of a Web service and packaging it into a deployable EAR file.

The following table describes the attributes of the `servicegen` Ant task.

Attribute	Description	Required?
destEar	Pathname of the EAR file or exploded directory which will contain the Web service and all its components.	Yes
	To create or update an EAR file, use a `.ear` suffix when specifying the EAR file, such as `c:\mywebservice.ear`. If the attribute value does not have a `.ear` suffix, then the `servicegen` task creates an exploded directory.	
	If you specify an EAR file or directory that does not exist, the `servicegen` task creates a new one.	
overwrite	Specifies whether you want the components of an existing EAR file or directory to be overwritten. The components include the `web-services.xml` file, serialization class, client JAR files, and so on.	No
	Valid values for this attribute are `True` and `False`. The default value is `True`.	
	If you specify `False`, the `servicegen` Ant task attempts to merge the contents of the EAR file/directory and information in the `web-services.xml` file.	
warName	Name of the WAR file or exploded directory into which the Web service Web application is written. The WAR file or directory is created at the top level of the EAR file.	No
	The default value is a WAR file called `web-services.war`.	
	To specify a WAR file, use a `.war` suffix, such as `mywebserviceWAR.war`. If the attribute value does not have a `.war` suffix, then the `servicegen` task creates an exploded directory.	
contextURI	Context root of the Web service. You use this value in the URL that invokes the Web service.	No.
	The default value of the `contextURI` attribute is the value of the `warName` attribute.	

service

The `<service>` element describes a single Web service implemented with either a stateless session EJB or a Java class.

The following table describes the attributes of the `<service>` element of the `servicegen` Ant task. Include one `<service>` element for every Web service you want to package in a single EAR file.

Attribute	Description	Required?
ejbJar	JAR file or exploded directory that contains the EJBs that implement the backend component of a Web service operation. The `servicegen` Ant task introspects the EJBs to automatically generate all the components.	You must specify either the `ejbJar`, `javaClassComponents`, or `JMS*` attribute.
javaClassComponents	Comma-separated list of Java class names that implement the Web service operation. The Java classes must be compiled and in your CLASSPATH. For example: `javaClassComponents="my.FirstClass,my.SecondClass"` **Note:** Do not include the `.class` extension when specifying the class names. The `servicegen` Ant task introspects the Java classes to automatically generate all the needed components.	You must specify either the `ejbJar`, `javaClassComponents`, or `JMS*` attribute.
ignoreAuthHeader	Specifies that the Web Service ignore the `Authorization` HTTP header in the SOAP request. **Note:** Be careful using this attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it. Valid values are `True` and `False`. Default value is `False`.	No.

Attribute	Description	Required?
includeEJBs	Comma-separated list of EJB names for which non-built-in data type components should be generated. If you specify this attribute, the `servicegen` task processes only those EJBs on the list. The EJB names correspond to the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor in the EJB JAR file (specified with the `ejbJar` attribute).	No. Used only in combination with the `ejbJar` attribute.
excludeEJBs	Comma-separated list of EJB names for which non-built-in data type components should *not* be generated. If you specify this attribute, the `servicegen` task processes all EJBs *except* those on the list. The EJB names correspond to the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor in the EJB JAR file (specified with the `ejbJar` attribute).	No. Used only in combination with the `ejbJar` attribute.
serviceName	Name of the Web service which will be published in the WSDL. **Note:** If you specify more than one `<service>` element in your `build.xml` file that calls `servicegen`, and set the `serviceName` attribute for each element to the same value, `servicegen` attempts to merge the multiple `<service>` elements into a single Web service.	Yes.

Attribute	Description	Required?
serviceURI	Web service URI portion of the URL used by client applications to invoke the Web service. **Note:** Be sure to specify the leading "/", such as /TraderService. The full URL to invoke the Web service will be: *protocol://host:port/contextURI/serviceURI* where ■ *protocol* refers to the protocol attribute of the <service> element ■ *host* refers to the computer on which WebLogic Server is running ■ *port* refers to the port on which WebLogic Server is listening ■ *contextURI* refers to the contextURI attribute of the main servicegen Ant task ■ *serviceURI* refers to this attribute	Yes.
targetNamespace	The namespace URI of the Web service.	Yes.
protocol	Protocol over which this Web service is deployed. Valid values are http and https. The default value is http.	No.
expandMethods	Specifies whether the servicegen task, when generating the web-services.xml file, should create a separate <operation> element for each method of the EJB or Java class, or whether the task should implicitly refer to all methods by specifying only one <operation> element that contains a method="*" attribute. Valid values are True and False. Default value is False.	No.
generateTypes	Specifies whether the servicegen task should generate the serialization class and Java representations for non-built-in data types used as parameters or return values. Valid values are True and False. Default value is True. For the list of non-built-in data types for which servicegen can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.	No.

Attribute	Description	Required?
typeMappingFile	File that contains additional XML data type mapping information. The format of the information is the same as the data type mapping information in a `web-services.xml`. Use this attribute if you want to include extra XML data type information in the `<type-mapping>` element of the `web-services.xml` file, in addition to the required XML descriptions of data types used by the EJB or Java class that implements an operation. The `servicegen` task adds the extra information in the specified file to a generated `web-services.xml` file.	No.
style	Specifies whether the `servicegen` Ant task should generate RPC-oriented or document-oriented Web service operations. RPC-oriented WebLogic Web service operations use SOAP encoding. Document-oriented WebLogic Web service operations use literal encoding. If you specify `document` for this attribute, the methods that implement the operations of the generated Web service must have only *one* parameter. If `servicegen` encounters methods that have more than one parameter, `servicegen` ignores the method and does not generate a corresponding Web service operation for it. Valid values for this attribute are `rpc` and `document`. Default value is `rpc`. **Note:** Because the `style` attribute applies to an entire Web service, all operations in a single WebLogic Web service must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web service.	No.
JMSDestination	JNDI name of a JMS topic or queue.	Yes, if creating a JMS-implemented Web service.
JMSDestinationType	Type of JMS destination, either a Queue or a Topic. Valid values are `topic` or `queue`.	Yes, if creating a JMS-implemented Web service.

Attribute	Description	Required?
JMSAction	Specifies whether the client application that invokes this JMS-implemented Web service sends or receives messages to or from the JMS destination. Valid values are `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, if creating a JMS-implemented Web service.
JMSConnectionFactory	JNDI name of the `ConnectionFactory` used to create a connection to the JMS destination.	Yes, if creating a JMS-implemented Web service.
JMSOperationName	Name of the operation in the generated WSDL file. Default value is either `send` or `receive`, depending on the value of the JMSAction attribute.	No.
JMSMessageType	Data type of the single parameter to the send or receive operation. Default value is `java.lang.String`. If you use this attribute to specify a non-built-in data type, and set the `generateTypes` attribute to `True`, be sure the Java represenation of this non-built-in data type is in your CLASSPATH.	No.

client

The optional `<client>` element describes how to create the client JAR file that client applications use to invoke the Web service. Specify this element only if you want the `servicegen` Ant task to create a client JAR file.

Note: You do not have to create the client JAR file when you assemble your Web service. You can later use the `clientgen` Ant task to generate the JAR file.

The following table describes the attributes of the `<client>` element.

Attribute	Description	Required?
clientJarName	Name of the generated client JAR file. When the `servicegen` task packages the Web service, it puts the client JAR file in the top-level directory of the Web service WAR file of the EAR file. Default name is *serviceName*_client.jar, where *serviceName* refers to the name of the Web service (the serviceName attribute) **Note:** If you want a link to the client JAR file to automatically appear in the Web service Home Page, you should not change its default name.	No.
packageName	Package name into which the generated client interfaces and stub files are packaged.	Yes.
useServerTypes	Specifies where the `servicegen` task gets the implementation of any non-built-in Java data types used in a Web service: either the task generates the Java code or the task gets it from the EAR file that contains the full implementation of the Web service. Valid values are True (use the Java code in the EAR file) and False. Default value is False. For the list of non-built-in data types for which servicegen can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-13.	No.
saveWSDL	When set to True, saves the WSDL file of the Web service in the generated client JAR file. This means that client applications do not need to download the WSDL file every time they create a stub to the Web service, possibly improving performance of the client because of reduced network usage. Valid values are True and False. Default value is True.	No.

Equivalent Command-Line Utility

The equivalent command-line utility of the `servicegen` Ant task is called `weblogic.webservice.servicegen`. The description of the flags of the utility is the same as the description of the Ant task attributes, described in the preceding sections.

Warning: If you use the `weblogic.webservice.servicegen` command-line utility to automatically assemble a Web service, you can create only *one* Web service in the `web-services.xml` file.

The `weblogic.webservice.servicegen` utility supports the following flags (see the equivalent attribute for a description of the flag):

- `-destEar` *pathname*
- `-warName` *name*
- `-ejbJar` *pathname*
- `-javaClassComponents` *list_of_classnames*
- `-serviceName` *name*
- `-serviceURI` *uri*
- `-targetNamespace` *uri*
- `-protocol` *protocol*
- `-expandMethods`
- `-clientPackageName` *name*
- `-clientJarName` *name*

source2wsdd

The `source2wsdd` Ant task generates a `web-services.xml` deployment descriptor file from the Java source file for a Java class-implemented WebLogic Web service.

The source2wsdd Ant task does *not* generate data type mapping information for any non-built-in data types used as parameters or return values of the methods of your Java class If your Java class uses non-built-in data types, you must first run the autotype Ant task to generate the needed components, then point the typesInfo attribute of the source2wsdd Ant task to the types.xml file generated by the autotype Ant task.

If your Java class refers to other Java class files, be sure to set the sourcePath attribute to the directory that contains them.

Note: The fully qualified name of the source2wsdd Ant task is
weblogic.ant.taskdefs.webservices.autotype.JavaSource2DD.

Example

```
<source2wsdd
      javaSource="c:\source\MyService.java"
      typesInfo="c:\autotype\types.xml"
      ddFile="c:\ddfiles\web-services.xml"
      serviceURI="/MyService"
/>
```

Attributes

The following table describes the attributes of the source2wsdd Ant task.

Attribute	Description	Required?
javaSource	Name of the Java source file that implements your Web service component.	Yes.
ddFile	Full pathname of the Web services deployment descriptor file (web-services.xml) which will contain the generated deployment descriptor information.	Yes.

Attribute	Description	Required?
typesInfo	Name of the file that contains the XML Schema representation and data type mapping information for any non-built-in data types used as parameters or return value of the Web service.	Yes.
	The format of the data type mapping information is the same as that in the `<type-mapping>` element of the `web-services.xml` file.	
	Typically you have already run the `autotype` Ant task to generate this information into a file called `types.xml`.	
serviceURI	Web service URI portion of the URL used by client applications to invoke the Web service.	Yes.
	Note: Be sure to specify the leading "/", such as `/TraderService`.	
	The value of this attribute becomes the uri attribute of the `<web-service>` element in the generated web-services.xml deployment descriptor.	
sourcePath	Full pathname of the directory that contains any additional classes referred to by the Java source file specified with the javaSource attribute.	No.
ignoreAuthHeader	Specifies that the Web Service ignore the `Authorization` HTTP header in the SOAP request.	No.
	Note: Be careful using this attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it.	
	Valid values are `True` and `False`. Default value is `False`.	

wsdl2Service

The `wsdl2Service` Ant task takes as input an existing WSDL file, and generates the following WebLogic Web service components:

- the `web-services.xml` deployment descriptor file

- a Java source file that partially implements the Web service.

The Java source file contains a template for the full Java class-implemented WebLogic Web service. The template includes full method signatures that correspond to the operations in the WSDL file. You write the actual code for these methods so that they function as you want.

The `wsdl2Service` Ant task generates a partial implementation for only one service in a WSDL file (specified by the `<service>` element.) Use the `serviceName` attribute to specify a particular service; if you do not specify this attribute, the `wsdl2Service` Ant task generates a partial implementation for the first `<service>` element in the WSDL.

The `wsdl2Service` Ant task does *not* generate data type mapping information for any non-built-in data types used as parameters or return values of the operations in the WSDL file. If the WSDL uses non-built-in data types, you must first run the `autotype` Ant task to generate the data type mapping information, then point the `typeMappingFile` attribute of the `wsdl2Service` Ant task to the `types.xml` file generated by the `autotype` Ant task.

Note: The fully qualified name of the `wsdl2Service` Ant task is
`weblogic.ant.taskdefs.webservices.wsdl2service.WSDL2Service`.

Example

```
<wsdl2service
    wsdl="c:\wsdls\myService.wsdl"
    destDir="c:\myService\implementation"
    typeMappingFile="c:\autotype\types.xml"
    packageName="example.ws2j.service"
/>
```

Attributes

The following table describes the attributes of the wsdl2Service Ant task.

Attribute	Description	Required?
wsdl	The full path name or URL of the WSDL that describes a Web service for which a partial WebLogic Web service implementation will be generated.	Yes.
destDir	The full pathname of the directory that will contain the generated components (web-services.xml file and Java source file that partially implements the Web service.)	Yes.
packageName	The package name for the generated Java source file that partially implements the Web service.	Yes.
serviceName	The name of the Web service in the WSDL file for which a partial WebLogic implementation will be generated. The name of a Web service in a WSDL file is the value of the name attribute of the <service> element. If you do not specify this attribute, the wsdl2Service Ant task generates a partial implementation for the first <service> element it finds in the WSDL file. **Note:** The wsdl2Service Ant task generates a partial WebLogic Web service implementation for only *one* service in a WSDL file. If your WSDL file contains more than one Web service, then you must run wsdl2Service multiple times, changing the value of this attribute each time.	No.

Attribute	Description	Required?
typeMappingFile	File that contains data type mapping information for all non-built-in data types referred to by the operations of the Web service in the WSDL file. The format of the information is the same as the data type mapping information in the `<type-mapping>` element of the `web-services.xml` file. Typically, you first run the `autotype` Ant task (specifying the `wsdl` attribute) against the same WSDL file and generate all the non-built-in data type components. One of the components is a file called `types.xml` that contains the non-built-in data type mapping information. Set the `typeMappingFile` attribute equal to this file.	Required only if the operations of the Web service in the WSDL file refer to any non-built-in data types.

wspackage

The `wspackage` Ant task packages the various components of a WebLogic Web service into a deployable EAR file. It is assumed that you have already generated these components, which can include:

- The `web-services.xml` deployment descriptor file

- The EJB JAR file that contains the EJBs the implement a Web service

- The Java class file that implements a Web service

- A client JAR file that users can download and use to invoke the Web service

- Implementations of SOAP handlers

- Components for any non-built-in data types used as parameters and return values for the Web service. These components include the XML and Java representations of the data type and the serialization class that converts the data between its two representations.

Typically you use other Ant tasks, such as `clientgen`, `autotype`, `source2wsdd`, and `wsdl2Service`, to generate the preceding components.

Note: The fully qualified name of the `wspackage` Ant task is `weblogic.ant.taskdefs.webservices.wspackage.WSPackage`.

Example

```
<wspackage
     output="c:\myWebService.ear"
     contextURI="web_services"
     codecDir="c:\autotype"
     webAppClasses="example.ws2j.service.SimpleTest"
     ddFile="c:\ddfiles\web-services.xml"
/>
```

Attributes

The following table describes the attributes of the `wspackage` Ant task.

Attribute	Description	Required?
output	Pathname of the EAR file or exploded directory which will contain the Web service and all its components.	Yes
	To create or update an EAR file, use a `.ear` suffix when specifying the EAR file, such as `c:\mywebservice.ear`. If the attribute value does not have a `.ear` suffix, then the `wspackage` task creates an exploded directory.	
	If you specify an EAR file or directory that does not exist, the `wspackage` task creates a new one.	
warName	Name of the WAR file into which the Web service is written. The WAR file is created at the top level of the EAR file.	No
	The default value is `web-services.war`.	
	Note: If you specified an exploded directory with the `output` attribute, the `wspackage` task creates an exploded Web application directory, even if you specify a `.war` suffix for the `warName` attribute.	

Attribute	Description	Required?
contextURI	Context root of the Web service. You use this value in the URL that invokes the Web service. The default value of the `contextURI` attribute is the value of the `warName` attribute.	No.
ddFile	Full pathname of an existing Web services deployment descriptor file (`web-services.xml`).	Yes.
filesToEar	Comma-separated list of files to be packaged in the root directory of the EAR. Use this attribute to specify the EJB JAR files that implement a Web service, as well as any other supporting EJB JAR files.	No.
filesToWar	Comma-separated list of additional files, such as the client JAR file, to be packaged in the root directory of the Web service's Web application.	No.
webAppClasses	Comma-separated list of class files that should be packaged in the `WEB-INF/classes` directory of the Web service's Web application. Use this attribute to specify the Java class that implements a Web service, SOAP handler classes, and so on.	No.
codecDir	Name of the directory that contains the serialization classes for any non-built-in data types used as parameters or return values in your Web service.	No.
overwrite	Specifies whether you want the components of an existing EAR file or directory to be overwritten. The components include the `web-services.xml` file, serialization class, client JAR files, and so on. Valid values for this attribute are `True` and `False`. The default value is `True`. If you specify `False`, the `wspackage` Ant task attempts to merge the contents of the EAR file/directory and information in the `web-services.xml` file.	No

wsgen

The `wsgen` Ant task upgrades a Version 6.1 WebLogic Web service to Version 7.0. In particular, it:

- Creates a new `web-services.xml` file that describes the upgraded Web service. The task uses values found in the 6.1 `build.xml` file and default values to generate the deployment descriptor.

- Optionally creates a client JAR file that includes all the classes, stubs, and interfaces needed by client applications that invoke the Web service. The classes, stubs, and interfaces are based on the JAX-RPC API.

- Packages all the Web service components into a WAR file called `web-services.war`, then packages the WAR and EJB JAR files into a deployable EAR file.

The task takes as input the same `build.xml` file you used in Version 6.1 to create a Web service.   You can optionally add two new attributes, as described in "Additional Attributes" on page B-37. The task also takes as input the same archive file or exploded directory that you used to create the 6.1 Web service.

Warning: Do not use this Ant task for assembling a *new* 7.0 Web service. Instead, use the `servicegen` Ant task. Because the `wsgen` Ant task is used only for upgrading Web services from Version 6.1 to 7.0, the task is deprecated in this version of WebLogic Server.

Note: The fully qualified name of the `wsgen` Ant task is `weblogic.ant.taskdefs.webservices.wsgen.WSGenTask`.

Example

The following `build.xml` excerpt is taken from the 6.1 RPC-style Web services example. It includes a new attribute `targetNameSpace`.

```
<wsgen
   destpath="weather.ear"
   context="/weather">
   <rpcservices path="weather.jar">
```

```
        <rpcservice bean="statelessSession"
                    uri="/weatheruri"
                    targetNameSpace="http://www.bea.com/examples"/>
    </rpcservices>
</wsgen>
```

Additional Attributes

Because you use the 6.1 `build.xml` file when using `wsgen` to upgrade your 6.1 Web service to Version 7.0, the attributes and elements of the `build.xml` file are those documented in the 6.1 version of the *Programming WebLogic Web Services* guide. For details, see build.xml Attributes and Elements at http://e-docs.bea.com/wls/docs61/webServices/build_xml.html.

You can, however, add the two attributes listed in the following table to your 6.1 `build.xml` file and `wsgen` will add the appropriate information to the new `web-services.xml` deployment descriptor file.

Attribute	Element	Description
targetNameSpace	rpcservice, messageservice	Namespace URI of the Web service. The default value of this attribute, if you do not add it to your existing 6.1 `build.xml` file, is `http://example.org`.
packageName	clientjar	Package name into which the generated client interfaces and stub files should be packaged in the client JAR file. The default value of this attribute, if you do not add it to your existing 6.1 `build.xml` file, is `org.example`.

C Customizing WebLogic Web Services

The following sections describe how to customize your WebLogic Web service by updating the Web application deployment descriptor files of your Web service WAR file:

Publishing a Static WSDL File

By default, WebLogic Server dynamically generates the WSDL of a WebLogic Web service, based on the contents of its `web-services.xml` deployment descriptor file. See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22 for details on getting the URL of the dynamically generated WSDL.

You can, however, include a static version of the WSDL file in the Web services EAR file and publish its URL as the public description of your Web service. One reason for publishing a static WSDL is to be able to add more custom documentation than what the dynamically generated WSDL contains.

Warning: If you publish a static WSDL as the public description of your Web service, you must always ensure that it remains up to date with the actual Web service. In other words, if you change your Web service, you must also manually change the static WSDL to reflect the changes you made to

your Web service. One advantage of using the dynamic WebLogic-generated WSDL is that it is always up to date.

To include a static WSDL file in your Web services EAR file and publish it, rather than the dynamically generated WSDL, to the Web, follow these steps:

1. Un-JAR the WebLogic Web services EAR file and then the WAR file that contains the `web-services.xml` file.

2. Put the static WSDL file in a directory of the exploded Web application. This procedure assumes you put the file at the top-level directory.

3. Update the `web.xml` file of the Web application, adding a `<mime-mapping>` element to map the extension of your WSDL file to an XML mime type.

   For example, if the name of your static WSDL file is `myService.wsdl`, the corresponding entry in the `web.xml` file is as follows:

   ```
   <mime-mapping>
           <extension>wsdl</extension>
           <mime-type>text/xml</mime-type>
   </mime-mapping>
   ```

4. Re-JAR the Web services WAR and EAR files.

5. Invoke the static WSDL file using the standard URL to invoke a static file in a Web application.

   For example, use the following URL to invoke the `myService.wsdl` file in a Web application that has a context root of `web_services`:

   `http://host:port/web_services/myService.wsdl`

Creating a Custom WebLogic Web Service Home Page

Every WebLogic Web service has a default Home Page that contains links to view the WSDL of the Web service, test the service, download the client JAR file, and view the SOAP requests and responses of a client application invoking the Web service. See "The WebLogic Web Services Home Page and WSDL URLs" on page 8-22 for details.

WebLogic Server dynamically generates the Web services Home page and thus it cannot be customized. If you want to create your own custom Home Page, add an HTML or JSP file to the Web services WAR file. For more information on creating JSPs, see *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs70/jsp/index.html.

D Specifications Supported by WebLogic Web Services

WebLogic Web services support the following specifications:

- JAX-RPC 1.0 at http://java.sun.com/xml/jaxrpc/index.html

- SOAP 1.1 at http://www.w3.org/TR/SOAP

- SOAP Messages With Attachments at http://www.w3.org/TR/SOAP-attachments

- Web Services Description Language (WSDL) 1.1 at http://www.w3.org/TR/wsdl

- UDDI 2.0 at http://www.uddi.org

- XML Schema Part 1: Structures at http://www.w3.org/TR/xmlschema-1/

- XML Schema Part 2: Data Types at http://www.w3.org/TR/xmlschema-2/

- JSSE at http://java.sun.com/products/jsse