BEA WebLogic Server®

Programming Web Services for WebLogic Server

Version 9.0 BETA
Revised: December 15, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Introduction and Roadmap

2. Understanding WebLogic Web Services

3. Common Web Services Use Cases and Examples

4. Iteratively Developing WebLogic Web Services

5. Programming the JWS File

6. Advanced JWS Programming

7. Invoking Web Services

8. Data Types and Data Binding

9. Configuring Security

10. Administering Web Services

11. Publishing and Finding Web Services Using UDDI

12. Upgrading an 8.1 Web Service to 9.0

A. Ant Task Reference

B. JWS Annotation Reference

C. Reliable SOAP Messaging Policy Assertion Reference

D. WebLogic Web Service Deployment Descriptor Element Reference

E. Creating a J2EE Web Service Manually

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming Web Services for WebLogic Server®*.

Document Scope and Audience

This document is a resource for software developers who develop WebLogic Web Services. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server® or considering the use of WebLogic Web Services for a particular application.

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

This document does not address production phase administration, monitoring, or performance tuning Web Service topics. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-3.

It is assumed that the reader is familiar with J2EE and Web Services concepts, the Java programming language, Enterprise Java Beans (EJBs), and Web technologies. This document emphasizes the value-added features provided by WebLogic Web Services and key information about how to use WebLogic Server features and facilities to get a WebLogic Web Service application up and running.

Guide to This Document

This document is organized as follows:

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide and the features of WebLogic Web Services.
- Chapter 2, "Understanding WebLogic Web Services," provides an overview of how WebLogic Web Services are implemented, why they are useful, and the standard specifications that they implement or to which they conform.
- Chapter 3, "Common Web Services Use Cases and Examples," provides a set of common use case and examples of programming WebLogic Web Services, along with step by step instructions on reproducing the example in your own environment.
- Chapter 4, "Iteratively Developing WebLogic Web Services," provides procedures for setting up your development environment and iteratively programming a WebLogic Web Service.
- Chapter 5, "Programming the JWS File," provides details about using JWS annotations in a Java file to implement a basic Web Service. The section discusses both standard (JSR-181) JWS annotations as well as WebLogic-specific ones.
- Chapter 6, "Advanced JWS Programming," discusses how to configure reliable SOAP messaging for your Web Service and how to create and use SOAP message handlers to be able to intercept and process the SOAP messages.

- Chapter 7, “Invoking Web Services,”describes how to write a client application that invokes a Web Service using the JAX-RPC stubs generated by the WebLogic Web Service Ant task `clientgen`.
- Chapter 8, “Data Types and Data Binding,” discusses the built-in and user-defined XML Schema and Java data types that are supported by WebLogic Web Services.
- Chapter 9, “Configuring Security,” provides information about configuring different types of security for a WebLogic Web Service: message-level (digital signatures and encryption), transport-level (SSL), and access control.
- Chapter 10, “Administering Web Services,” provides information about the types of administrative tasks you typically perform with WebLogic Web Services and the different ways you can go about administering them: Administration Console, WebLogic Scripting Tool, and so on.
- Chapter 11, “Publishing and Finding Web Services Using UDDI,” describes how to use UDDI to publish and find Web Services.
- Chapter 12, “Upgrading an 8.1 Web Service to 9.0,” describes how to upgrade a Web Service built on WebLogic Server 8.1 to run on the new 9.0 Web Services runtime environment.
- Appendix A, “Ant Task Reference,” provides reference documentation about the WebLogic Web Services Ant tasks.
- Appendix B, “JWS Annotation Reference,”provides reference information about the JWS annotations (both standard JSR-181 and WebLogic-specific) that you can use in the JWS file that implements your Web Service.
- Appendix C, “Reliable SOAP Messaging Policy Assertion Reference,” provides reference information about the policy assertions you can add to a WS-Policy file to configure the reliable SOAP messaging feature of WebLogic Web Services.
- Appendix D, “WebLogic Web Service Deployment Descriptor Element Reference,” provides reference information about the elements in the WebLogic-specific Web Services deployment descriptor `weblogic-webservices.xml`.
- Appendix E, “Creating a J2EE Web Service Manually,” describes how to create a J2EE Web Service manually, or without using the WebLogic Web Service Ant tasks.

Related Documentation

This document contains Web Service-specific design and development information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- Developing WebLogic Server Applications is a guide to developing WebLogic Server components (such as Web applications and EJBs) and applications.
- Developing Web Applications for WebLogic Server is a guide to developing Web applications, including servlets and JSPs, that are deployed and run on WebLogic Server.
- Programming WebLogic Enterprise Java Beans is a guide to developing EJBs that are deployed and run on WebLogic Server.
- Programming WebLogic XML is a guide to designing and developing applications that include XML processing.
- Deploying WebLogic Server Applications is the primary source of information about deploying WebLogic Server applications.
- WebLogic Server Performance and Tuning contains information on monitoring and improving the performance of WebLogic Server applications.

Samples for the Web Services Developer

In addition to this document, BEA Systems provides a variety of code samples for Web Services developers. The examples illustrate WebLogic Web Services in action, and provide practical instructions on how to perform key Web Service development tasks.

BEA recommends that you run some or all of the Web Service examples before programming your own application that use Web Services.

Web Services Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in *WL_HOME*\samples\server\examples\src\examples\webservices, where *WL_HOME* is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server Start menu.

Additional Web Services Examples Available for Download

Additional API examples for download at http://dev2dev.bea.com. These examples are distributed as .zip files that you can unzip into an existing WebLogic Server samples directory structure.

You build and run the downloadable examples in the same manner as you would an installed WebLogic Server example. See the download pages of individual examples for more information.

Release-Specific WebLogic Web Services Information

For release-specific information, see these sections in *WebLogic Server Release Notes*:

- WebLogic Server Features and Changes lists new, changed, and deprecated features.
- WebLogic Server Known Issues lists known problems by general release, as well as service pack, for all WebLogic Server APIs, including Web Services.

Differences Between 8.1 and 9.0 WebLogic Web Services

Web Services is one of the most important themes of J2EE 1.4, and thus of WebLogic Server 9.0. J2EE 1.4 introduces a standard Java component model for authoring Web Services with the inclusion of new specifications such as *Implementing Enterprise Web Services* (JSR-921, the 1.1 maintenance version of JSR-109) and *Java API for XML Registries* (JAX-R), as well as the updated JAX-RPC and SAAJ specifications. Because the implementation of Web Services is now a J2EE standard, there have been many changes between 8.1 and 9.0 WebLogic Web Services.

In particular, the programming model used to create WebLogic Web Services has changed to take advantage of the powerful new metadata annotations feature introduced in Version 5.0 of the JDK (specified by JSR-175). In 9.0 you use JWS metadata annotations to annotate a Java file with information that specifies the shape and behavior of the Web Service. These JWS annotations include both the standard ones defined by the *Web Services Metadata for the Java Platform* specification (JSR-181), as well as additional WebLogic-specific ones. This JWS-based programming model is the same as that of WebLogic Workshop 9.0, and similar to that of WebLogic Workshop 8.1, although in 8.1 the metadata was specified via Javadoc tags. The programming model in 8.1, by contrast, used the many attributes of the Web Service Ant tasks, such as `servicegen`, to specify the shape and behavior of the Web Service. Occasionally programmers had to manually update the deployment descriptor file (`webservices.xml`) to specify characteristics of the Web Service. The new programming model makes implementing Web Services much easier and quicker.

See Chapter 5, “Programming the JWS File,” for more information.

Additionally, the runtime environment upon which WebLogic Web Services 9.0 run has been completely rewritten to support the *Implementing Enterprise Web Services*, Version 1.1, (JSR-921) specification. This means that Web Services created in 9.0 are internally implemented

differently from those created in 8.1 and both run on completely different runtime environments. The 8.1 runtime environment has been deprecated, although it will continue to be supported for a limited number of future WebLogic Server releases. This means that even though 8.1 WebLogic Web Services run correctly on WebLogic Server 9.0, this may not always be true and BEA recommends that you upgrade the 8.1 Web Services to run on the 9.0 runtime environment.

See "Anatomy of a WebLogic Web Service" on page 2-3 for more information.

Summary of WebLogic Web Services Features

The following list summarizes the main features of WebLogic Web Services and provides links for additional detailed information:

- Programming model based on the new JDK 5.0 metadata annotations feature (specified by JSR-175). The Web Services programming model uses JWS annotations, defined by the Web Services Metadata for the Java Platform specification (JSR-181).

  See Chapter 5, "Programming the JWS File."

- Implementation of the *Web Services for J2EE*, Version 1.1 specification, which defines the standard J2EE runtime architecture for implementing Web Services in Java.

  See "Anatomy of a WebLogic Web Service" on page 2-3.

- Asynchronous, loosely-coupled Web Services that take advantage of the following features: reliable SOAP messaging, conversations, callbacks, and addressing.

  See "Using Reliable SOAP Messaging" on page 6-1.

  Note: Documentation for conversations, callbacks, and addressing not available for Beta.

- Digital signatures and encryption of request and response SOAP messages, as specified by the WS-Security specification.

  See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2.

- Data binding between built-in and user-defined XML and Java data types.

  See Chapter 8, "Data Types and Data Binding."

- Use of WS-Policy files for the reliable messaging and digital signatures/encryption features.

  See "Use of WS-Policy Files for Message-Level Security Configuration" on page 9-4 for information on how policy files are used to configure message level security.

- Ant tasks that handle JWS files, generate a Web Service from a WSDL file, and create the JAX-RPC client classes needed to invoke a Web Service.

  See Appendix A, "Ant Task Reference."

- Implementation of and conformance with standard Web Services specifications.

  See "Standards Supported By WebLogic Web Services" on page 2-6.

Unsupported WebLogic Web Services Features

The following list describes the features that are not supported in this Beta release of WebLogic Web Services:

- WebLogic Web Services that use input parameters or return values of the following data types:
  - `javax.xml.soap.SOAPElement`
  - `javax.activation.DataHandler`
  - `XMLBeans`
- Document-literal-wrapped Web Services. (Document-literal-bare Web Services *are* supported.)
- Specifying your own WS-Policy file that contains security policy assertions to configure message-level security for a WebLogic Web Service. Instead, WebLogic Web Services provide three pre-packaged policy files (`Auth.xml`, `Sign.xml`, and `Encrypt.xml`) that you can use to specify simple message-level security.

  See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2 for more information.

- Use of the `javax.jws.security.SecurityIdentity` JWS annotation.

CHAPTER 2

Understanding WebLogic Web Services

The following sections provide an overview of WebLogic Web Services as implemented by WebLogic Server:

What Are Web Services?

A Web Service is a set of functions packaged into a single entity that is available to other systems on a network and can be shared by and used as a component of distributed Web-based applications. The network can be a corporate intranet or the Internet. Other systems, such as customer relationship management systems, order-processing systems, and other existing back-end applications, can call these functions to request data or perform an operation. Because Web Services rely on basic, standard technologies which most systems provide, they are an excellent means for connecting distributed systems together.

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 has evolved toward loosely coupled service-oriented applications that interact dynamically 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.

This 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.

Anatomy of a WebLogic Web Service

WebLogic Web Services are implemented according to the Enterprise Web Services 1.1 specification (JSR-921), which defines the standard J2EE runtime architecture for implementing Web Services in Java. The specification also describes a standard J2EE Web Service packaging format, deployment model, and runtime services, all of which are implemented by WebLogic Web Services.

Note: JSR-921 is the 1.1 maintenance release of JSR-109, which was the J2EE 1.3 specification for Web Services. JSR-921 is currently in final release of the JCP (Java Comunnity Process).

The Enterprise Web Services 1.1 specification describes that a J2EE Web Service is implemented by one of the following components:

- A Java class running in the Web container.
- A stateless session EJB running in the EJB container.

The code in the Java class or EJB is what implements the business logic of your Web Service. BEA recommends that, instead of coding the raw Java class or EJB directly, you use the JWS annotations programming model instead, which makes programming a WebLogic Web Service much easier.

This programing model takes advantage of the new JDK 5.0 metadata annotations feature (specified by JSR-175) in which you create an annotated Java file and then use Ant tasks to compile the file into the Java source code and generate all the associated artifacts. The Java Web Service (JWS) annotated file is the core of your Web Service. It contains the Java code that determines how your Web Service behaves. A JWS file is an ordinary Java class file that uses annotations to specify the shape and characteristics of the Web Service. The JWS annotations you can use in a JWS file include the standard ones defined by the *Web Services Metadata for the Java Platform* specification (JSR-181) as well as a set of WebLogic-specific ones.

For more information on the JWS programming model, see Chapter 5, “Programming the JWS File,”

If your Web Service is implemented with a Java class, then it is packaged in a standard Web application WAR file with all the standard WAR artifacts, such as the `web.xml` and `weblogic.xml` deployment descriptor files. The WAR file, however, contains additional artifacts to indicate that it is also a Web Service; these additional artifacts include the `webservices.xml` and `weblogic-webservices.xml` deployment descriptor files, the JAX-RPC data type mapping file, the WSDL file that describes the public contract of the Web Service, and so on.

Similarly, if your Web Service is implemented with a stateless session EJB, it is packaged in a standard EJB JAR file with all the usual artifacts, such as the `ejb-jar.xml` and `weblogic-ejb.jar.xml` deployment descriptor files. The EJB JAR file also contains additional Web Service artifacts, as described in the preceding paragraph, to indicate that it is a Web Service.

In addition to programming the implementation Java or EJB class of your Web Service, you can also configure one or more SOAP message handlers if you need to do additional processing of the request and response SOAP messages used in the invoke of a Web Service operation.

Once you have coded the basic WebLogic Web Service, you can configure it so that it can be invoked reliably (as specified by the WS-ReliableMessaging specification, dated February 4, 2004) and also specify that the SOAP messages be digitally signed and encrypted (as specified by the WS-Security specification). You configure these more advanced features of WebLogic Web Services using WS-Policy files, which is an XML file that adheres to the WS-Policy specification and contains security- or reliable messaging-specific XML elements that describe the security and reliable-messaging configuration, respectively.

Roadmap of Common Web Service Development Tasks

The following table provides a roadmap of common tasks for creating, deploying, and invoking WebLogic Web Services.

Table 2-1 Web Services Tasks

Major Task	Subtasks and Additional Information
Iteratively Develop Basic WebLogic Web Services	"Iteratively Developing WebLogic Web Services: Main Steps" on page 4-2
	"Integrating Web Services Into the WebLogic Split Development Directory Environment" on page 4-13
	"Programming the JWS File" on page 5-1
	"Supported Built-In Data Types" on page 8-2
	"Supported User-Defined Data Types" on page 8-6
	"Programming the User-Defined Java Data Type" on page 5-13
	"Throwing Exceptions" on page 5-16
	"Creating the Basic Ant build.xml File" on page 4-4
	"Running the jwsc WebLogic Web Services Ant Task" on page 4-5
	"Compiling User and Generated Java Code" on page 4-7
	"Packaging the Web Service into a WAR or JAR File" on page 4-8
Deploy the Web Service for testing purposes.	"Deploying and Undeploying WebLogic Web Services" on page 4-9
	"Invoking the WSDL and Home Page of the Web Service" on page 4-11
Invoke the Web Service.	"Invoking a Web Service from a Stand-Alone Client: Main Steps" on page 7-3
Add advanced features to the Web Service.	"Using Reliable SOAP Messaging" on page 6-1
	"Creating and Using SOAP Message Handlers" on page 6-13
Secure the Web Service.	"Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2
	"Configuring Transport-Level Security" on page 9-13
	"Configuring Access Control Security: Main Steps" on page 9-13

Table 2-1 Web Services Tasks

Major Task	Subtasks and Additional Information
Upgrade an 8.1 WebLogic Web Service to run on the 9.0 runtime.	"Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 9.0: Main Steps" on page 12-3
	"Upgrading an 8.1 EJB-Implemented WebLogic Web Service to 9.0: Main Steps" on page 12-8

Standards Supported By WebLogic Web Services

A Web Service requires the following standard specification implementations or conformance:

- A standard programming model used to develop the Web Service.

  The WebLogic Web Services programming model uses standard JDK 1.5 metadata annotations, as specified by the *Web Services Metadata for the Java Platform* specification (JSR-181) See "Web Services Metadata for the Java Platform (JSR-181)" on page 2-8.

- A standard implementation hosted by a server on the Web.

  WebLogic Web Services are hosted by WebLogic Server and are implemented using standard J2EE components, as defined by the *Implementing Enterprise Web Services* 1.1 specification (JSR-921, which is the 1.1 maintenance release of JSR-109). See "Enterprise Web Services 1.1" on page 2-8.

- A standard for transmitting data and Web Service invocation calls between the Web Service and the user of the Web Service.

  WebLogic Web Services use Simple Object Access Protocol (SOAP) 1.1 as the message format and HTTP as the connection protocol. See "SOAP 1.1" on page 2-9.

  WebLogic Web Services implement the SOAP with Attachments API for Java 1.2 specification to access any attachments to the SOAP message. See "SAAJ 1.2" on page 2-9.

- A standard for describing 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 2-10.

- A standard for client applications to invoke a Web Service.

WebLogic Web Services implement the Java API for XML-based RPC (JAX-RPC) 1.1 as part of a client JAR that client applications can use to invoke WebLogic and non-WebLogic Web Services. See "JAX-RPC 1.1" on page 2-11.

- A standard for digitally signing and encrypting the SOAP request and response messages between a client application and the Web Service it is invoking.

  WebLogic Web Services implement the following OASIS Standard 1.0 Web Services Security specifications, dated April 6 2004:

  – Web Services Security: SOAP Message Security

  – Web Services Security: Username Token Profile

  – Web Services Security: X.509 Token Profile

  For more information, see "Web Services Security (WS-Security)" on page 2-12.

- A standard for a Web Service to describe and communicate its policies.

  WebLogic Web Services conform to the WS-Policy specification when using policies to describe their reliable messaging and security (digital signatures and encryption) functionality.

- A standard way for two Web Services to communicate asynchronously.

  WebLogic Web Services conform to the WS-Addressing and WS-ReliableMessaging specifications when asynchronous features such as callbacks, addressing, conversations, and reliable messaging.

- A standard for client applications to find a registered Web Service and to register a Web Service.

  WebLogic Web Services implement two different registration specifications: UDDI 2.0 and JAX-R 1.0.

BEA Implementation of Web Service Specifications

Many of the specifications that define Web Service standards have been written in an intentionally vague way to allow for broad use of the specification throughout the industry. Because of this vagueness, BEA's implementation of a particular specification might not cover all possible usage scenarios covered by the specification.

BEA considers interoperability of Web Services platforms to be more important than providing support for all possible edge cases of the Web Services specifications. For this reason, BEA fully supports the Basic Profile 1.0 specification from the Web Services Interoperability Organization

and considers it to be the baseline for Web Services interoperability. BEA implements all requirements of the Basic Profile 1.0, although this guide does not necessarily document all of these requirements. This guide does, however, document features that are beyond the requirements of the Basic Profile 1.0."

Web Services Metadata for the Java Platform (JSR-181)

Although it is possible to program a WebLogic Web Service manually by coding the standard JSR-921 EJB or Java class from scratch and generating its associated artifacts by hand (deployment descriptor files, WSDL, data binding artifacts for user-defined data types, and so on), the entire process can be difficult and tedious. For this reason, BEA recommends that you take advantage of the new JDK 1.5 metadata annotations feature (specified by JSR-175) and use a programming model in which you create an annotated Java file and then use Ant tasks to convert the file into the Java source code of a standard JSR-921 Java class or EJB and automatically generate all the associated artifacts.

The Java Web Service (JWS) annotated file (called a *JWS file* for simplicity) is the core of your Web Service. It contains the Java code that determines how your Web Service behaves. A JWS file is an ordinary Java class file that uses JDK 1.5 metadata annotations to specify the shape and characteristics of the Web Service. The JWS annotations you can use in a JWS file include the standard ones defined by the *Web Services Metadata for the Java Platform* specification (JSR-181) as well as a set of WebLogic-specific ones.

Enterprise Web Services 1.1

The *Implementing Enterprise Web Services* 1.1 specification (JSR-921) defines the programming model and runtime architecture for implementing Web Services in Java that run on a J2EE application server, such as WebLogic Server. In particular, it specifies that programmers implement J2EE Web Services using one of the following two components:

- a Java class running in the Web container.
- a stateless session EJB running in the EJB container.

The specification also describes a standard J2EE Web Service packaging format, deployment model, and runtime services, all of which are implemented by WebLogic Web Services.

Note: JSR-921 is the 1.1 maintenance release of JSR-109, which was the J2EE 1.3 specification for Web Services.

SOAP 1.1

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 the SOAP 1.1 specification. 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.

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>
```

For more information, see SOAP 1.1 at http://www.w3.org/TR/SOAP.

SAAJ 1.2

The SOAP with Attachments API for Java (SAAJ) specification describes how developers can produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments notes.

The single package in the API, javax.xml.soap, provides the primary abstraction for SOAP messages with MIME attachments. Attachments may be entire XML documents, XML

fragments, images, text documents, or any other content with a valid MIME type. In addition, the package provides a simple client-side view of a request-response style of interaction with a Web Service.

For more information, see and SOAP With Attachments API for Java (SAAJ) 1.1 at http://java.sun.com/xml/saaj/index.html.

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 a client application connects 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>
```

For more information, see Web Services Description Language (WSDL) 1.1 at http://www.w3.org/TR/wsdl.

JAX-RPC 1.1

The Java API for XML-based RPC (JAX-RPC) 1.1 is a Sun Microsystems specification that defines the Java APIs for making XML-based remote procedure calls (RPC). In particular, these APIs are used to invoke and get a response from a Web Service using SOAP 1.1, and XML-based protocol for exchange of information in a decentralized and distributed environment.

WebLogic Server implements all required features of the JAX-RPC Version 1.1 specification. Additionally, WebLogic Server implements optional data type support, as specified in:

- "Supported Built-In Data Types" on page 8-2
- "Supported User-Defined Data Types" on page 8-6

WebLogic Server does not implement optional features of the JAX-RPC specification, other than what is described in these sections.

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

Table 2-2 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 invoke a Web Service dynamically.
JAXRPCException	Exception thrown if an error occurs while invoking a Web Service.

For detailed information on JAX-RPC, see 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.

Web Services Security (WS-Security)

The following description of Web Services Security is taken directly from the OASIS standard 1.0 specification, titled *Web Services Security: SOAP Message Security*, dated April 6, 2004:

> This specification proposes a standard set of SOAP extensions that can be used when building secure Web services to implement integrity and confidentiality. We refer to this set of extensions as the *Web Services Security Language* or *WS-Security.*
>
> WS-Security is flexible and is designed to be used as the basis for the construction of a wide variety of security models including PKI, Kerberos, and SSL. Specifically WS-Security provides support for multiple security tokens, multiple trust domains, multiple signature formats, and multiple encryption technologies.
>
> This specification provides three main mechanisms: security token propagation, message integrity, and message confidentiality. These mechanisms by themselves do not provide a complete security solution. Instead, WS-Security is a building block that can be used in conjunction with other Web service extensions and higher-level application-specific protocols to accommodate a wide variety of security models and encryption technologies.

These mechanisms can be used independently (for example, to pass a security token) or in a tightly integrated manner (for example, signing and encrypting a message and providing a security token hierarchy associated with the keys used for signing and encryption).

For more information, see the OASIS Web Service Security Web page at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

UDDI 2.0

The Universal Description, Discovery and Integration (UDDI) specification defines a standard for describing a Web Service; registering a Web Service in a well-known registry; and discovering other registered Web Services.

For more information, see http://www.uddi.org.

JAX-R 1.0

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

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

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

For more information, see Java API for XML Registries at http://java.sun.com/xml/jaxr/index.jsp.

WS-Addressing

The WS-Addressing specification provides transport-neutral mechanisms to address Web services and messages. In particular, the specification defines a number of XML elements used to identify Web service endpoints and to secure end-to-end endpoint identification in messages.

All the asynchronous features of WebLogic Web Services (callbacks, conversations, and reliable messaging) use addressing in their implementation, but Web Service programmers can also use the APIs that conform to this specification standalone if additional addressing functionality is needed.

See Web Services Addressing (WS-Addressing) at http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/.

WS-Policy

The Web Services Policy Framework (WS-Policy) specification provides a general purpose model and corresponding syntax to describe and communicate the policies of a Web Service. WS-Policy defines a base set of constructs that can be used and extended by other Web Services specifications to describe a broad range of service requirements, preferences, and capabilities.

See Web Services Policy Framework (WS-Policy) at http://www-106.ibm.com/developerworks/library/ws-polfram/.

WS-ReliableMessaging

The WS-ReliableMessaging specification (February 4, 2004) describes how two Web Services running on different WebLogic Server instances can communicate reliably in the presence of failures in software components, systems, or networks. In particular, the specification provides for an interoperable protocol in which a message sent from a source endpoint to a destination endpoint is guaranteed either to be delivered or to raise an error.

See Web Services Reliable Messaging Protocol (WS-ReliableMessaging) at http://www-106.ibm.com/developerworks/library/ws-rm/.

Additional Specifications Supported by WebLogic Web Services

- 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/

CHAPTER 3

Common Web Services Use Cases and Examples

The following sections provide information about the most common Web Service use cases:

These use cases provide step-by-step procedures for creating simple WebLogic Web Services and invoking an operation from a deployed Web Service. Each use case includes basic Java code and Ant `build.xml` files that you can use in your own development environment to recreate the example. The use cases do not go into detail about the tools and technologies used in the examples. For detailed information about specific features, see the relevant topics in this guide, in particular:

- Chapter 4, "Iteratively Developing WebLogic Web Services"
- Chapter 5, "Programming the JWS File"
- Chapter 6, "Advanced JWS Programming"
- Chapter 7, "Invoking Web Services"

- Appendix A, "Ant Task Reference"

Creating an EJB-Implemented Web Service Starting From Java

Another use case is to start from Java, design the Web Service operations from the point of view of Java methods, and then generate the Web Service from the Java implemenation. When starting from Java, you must decide on the backend implementation of the service: a simple Java class or a stateless session EJB. This example shows how to create an EJB-implemented Web Service.

To create an EJB-implemented Web Service, simply create a *JWS file* that includes a method for each operation you want in your Web Service. A JWS file is a standard Java file that uses JWS metadata annotations to specify the shape of the Web Service. Metadata annotations are a new JDK 5.0 feature, and the set of annotations used to annotate Web Service files are called JWS annotations. WebLogic Web Services use standard JWS annotations, as defined by JSR-181, as well as WebLogic-specific ones for added value. The example also shows how to use EJBGen annotations to specify the shape of the EJB that implements the Web Service.

After adding the business logic to the methods, you compile the JWS file, using the `jwsc` WebLogic Web Service Ant task, to generate the supporting artifacts (such as the deployment descriptors, serialization classes for any user-defineduser-defined data types, the WSDL file, and so on), compile the Java source files into class files, and finally archive all the artifacts into an EJB JAR file that you deploy to WebLogic Server.

The following example shows how to create an EJB-implemented Web Service using a JWS file. The Web Service has one method, `echoString()`, that simply returns the inputted String.

1. Open a command window and set your WebLogic Server environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the domain directory of WebLogic Server.
2. Create a working directory:

   ```
   prompt> mkdir /examples/ejbJWS
   ```

3. Create the JWS file that implements the Web Service by opening your favorite Java IDE or text editor and creating a Java file called `SimpleBean.java` using the Java code specified in "Sample SimpleBean JWS File" on page 3-4.

   The sample JWS file shows a class called `SimpleBean` that implements `javax.ejb.SessionBean`. The JWS and EJBGen annotations (specified using the `@` metadata identifier) determine what the generated Web Service and EJB look like. The `SimpleBean.java` file has one public method, `echoString()`, that will be listed as the single Web Service operation in the WSDL file. Finally, the standard EJB methods

(`ejbCreate()`, `ejbActivate()`, and so on) must be included in the JWS file, although typically you do not have to override them.

For more in-depth information about creating a JWS file, see Chapter 5, "Programming the JWS File."

4. Create a standard Ant `build.xml` file in the working directory and add the following calls to the `jwsc`, `javac`, and `jar` Ant tasks, wrapped inside of the `build-service` target:

```
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

<target name="build-service">

  <mkdir dir="output/temp/META-INF"/>

  <jwsc
    source="SimpleBean.java"
    destdir="output/temp/META-INF/" />

  <javac
    srcdir="output/temp/META-INF/"
    source="1.5"
    destdir="output/temp/"
    includes="*.java"/>

  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp"
    includes="*.java"/>

  <jar destfile="output/simpleBean.jar"
    basedir="output/temp"/>

</target>
```

Before you are able to execute the `jwsc` WebLogic Web Service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task. The `build-service` target includes other supporting standard Ant tasks in addition to `jwsc`: `mkdir` sets up the directory hierarchy for later packaging up an EJB JAR file under the temporary `output` directory; `javac` compiles the generated Java code into class files, and `jar` creates the EJB JAR archive file that is later deployed to WebLogic Server.

See "Sample Ant Build File" on page 3-6 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean` and `undeploy`.

5. Execute the `jwsc` Ant task, along with the other supporting Ant tasks, by specifying the `build-service` target at the command line:

```
prompt> ant build-service
```

See the `output/temp` and `output/temp/META-INF` directories to view the files and artifacts generated by the `jwsc` Ant task.

6. Start a WebLogic Server instance.
7. Deploy the Web Service to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In both cases, you deploy the EJB JAR file `SimpleBean.jar`, located in the `output` directory.

   To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

   ```
   <target name="deploy">

     <wldeploy action="deploy"
       name="SimpleBeanService"
       source="output/simpleBean.jar" user="system"
       password="gumby1234" verbose="true"
       adminurl="t3://localhost:7001"
       targets="myserver"/>

   </target>
   ```

   Substitute the values for `user`, `password`, `adminurl`, and `targets` that correspond to your WebLogic Server instance.

   Deploy the WAR file by executing the `deploy` target:

   ```
   prompt> ant deploy
   ```

8. Test that the Web Service is deployed correctly by invoking its Home Page in your browser:

   ```
   http://localhost:7001/ejbJWS/SimpleBean
   ```

   This URL is constructed using the values of the `contextPath` and `serviceUri` attributes of the `WLHttpTransport` JWS annotation in the `SimpleBean.java` JWS file. Use the hostname and port relevant to your WebLogic Server instance.

See "Invoking a Web Service from a Standalone JAX-RPC Java Client" on page 3-26 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

You can use the `build-service`, `undeploy`, and `deploy` targets in the `build.xml` file to iteratively update, rebuild, undeploy, and redeploy the EJB-implemented Web Service as part of your development process.

Sample SimpleBean JWS File

```
package examples.ejbJWS;

import java.rmi.RemoteException;
```

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

// Import the EJBGen annotation interfaces

import com.bea.wls.ejbgen.annotations.*;

// Import the WebLogic-specific JWS annotation interface called WLHttpTransport

import weblogic.jws.WLHttpTransport;

// EJBGen annotation that specifies that EJBGen should generate a stateless
// session EJB called Simple and its generated JAX-RPC Web service endpoint
// interface should be called "examples.ejbJWS.SimpleBeanPortType".

@Session(ejbName="Simple",
         serviceEndpoint="examples.ejbJWS.SimpleBeanPortType")

// Standard JWS annotation that specifies that the name of the Web Service is
// "myPortType", its public service name is "SimpleEjbService", and the
// targetNamespace used in the generated WSDL is "http://example.org"

@WebService(name="myPortType",
            serviceName="SimpleEjbService",
            targetNamespace="http://example.org")

// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are rpc-encoded.

@SOAPBinding(style=SOAPBinding.Style.RPC,
             use=SOAPBinding.Use.ENCODED)

// WebLogic-specific JWS annotation that specifies the port name is helloPort,
// and the context path and service URI used to build the URI of the Web
// Service is "ejbJWS/SimpleBean"

@WLHttpTransport(portName="helloPort",
                 contextPath="ejbJWS",
                 serviceUri="SimpleBean")

/**
 * This JWS file forms the basis of a stateless-session EJB implemented
 * WebLogic Web Service.
 *
 *
```

```
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class SimpleBean implements SessionBean {

  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: echoString.

  @WebMethod()
  public String echoString(String input) throws RemoteException {
    System.out.println("echoString got " + input);
    return input;
  }

  // Standard EJB methods.  Typically there's no need to override the methods.

  public void ejbCreate() {}
  public void ejbActivate() {}
  public void ejbRemove() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {}
}
```

Sample Ant Build File

```
<project default="all">

  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

  <target name="all" depends="clean,build-service,deploy" />

  <target name="clean">
    <delete dir="output" />
  </target>

  <target name="build-service">

    <mkdir dir="output/temp/META-INF"/>

    <jwsc
      source="SimpleBean.java"
      destdir="output/temp/META-INF/" />

    <javac
      srcdir="output/temp/META-INF/"
      source="1.5"
```

```
      destdir="output/temp/"
      includes="*.java"/>

    <javac
      srcdir="."
      source="1.5"
      destdir="output/temp"
      includes="*.java"/>

    <jar destfile="output/simpleBean.jar"
      basedir="output/temp"/>

  </target>

  <target name="deploy">

    <wldeploy action="deploy"
      name="SimpleBeanService"
      source="output/simpleBean.jar" user="system"
      password="gumby1234" verbose="true"
      adminurl="t3://localhost:7001"
      targets="myserver"/>

  </target>

  <target name="undeploy">

    <wldeploy action="undeploy"
      name="SimpleBeanService"
      user="system"
      password="gumby1234" verbose="true"
      adminurl="t3://localhost:7001"
      targets="myserver"/>

  </target>

</project>
```


Creating a Java Class-Implemented Web Service Starting from Java

Another use case when starting from Java is to use a simple Java class to implement the Web Service. This type of Web Service runs in the Web container of WebLogic Server.

To create a Java class-implemented Web Service, simply create a *JWS file* that includes a method for each operation you want in your Web Service. A JWS file is a standard Java file that uses JWS metadata annotations to specify the shape of the Web Service. Metadata annotations are a new JDK 5.0 feature, and the set of annotations used to annotate Web Service files are called JWS annotations. WebLogic Web Services use standard JWS annotations, as defined by JSR-181, as well as WebLogic-specific ones for added value.

After adding the business logic to methods, you compile the JWS file, using the `jwsc` WebLogic Web Service Ant task to generate the supporting artifacts (such as the deployment descriptors, serialization classes for any user-defined data types, the WSDL file, and so on), compile the Java source files into class files, and finally archive all the artifacts into Web application WAR file that you deploy to WebLogic Server.

The following example shows how to create a Java-class implemented Web Service using a JWS file. The Web Service has one method, `sayHello()`, that returns the inputted String with additional text.

1. Open a command window and set your WebLogic Server environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the domain directory of WebLogic Server.

2. Create a working directory:

   ```
   prompt> mkdir /examples/webJWS
   ```

3. Create the JWS file that implements the Web Service by opening your favorite Java IDE or text editor and creating a Java file called `SimpleImpl.java` using the Java code specified in "Sample SimpleImpl JWS File" on page 3-10.

   The sample JWS file shows a class called `SimpleImpl`. The JWS annotations (specified using the `@` metadata identifier) determine what the generated Web Service look like. The `SimpleImpl.java` file has one public method, `sayHello()`, that will be listed as the single Web Service operation in the WSDL file.

   For more in-depth information about creating a JWS file, see Chapter 5, "Programming the JWS File."

4. Create a standard Ant `build.xml` file in the working directory and add the following calls to the `jwsc`, `javac`, and `jar` Ant tasks, wrapped inside of the `build-service` target:

   ```
   <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

   <target name="build-service">

     <mkdir dir="output/temp/war/WEB-INF/classes"/>
   ```

```
    <jwsc
      source="SimpleImpl.java"
      destdir="output/temp/war/WEB-INF/" />

    <javac
      srcdir="output/temp/war/WEB-INF/"
      source="1.5"
      destdir="output/temp/war/WEB-INF/classes"
      includes="*.java"/>

    <javac
      srcdir="."
      source="1.5"
      destdir="output/temp/war/WEB-INF/classes"
      includes="*.java"/>

    <jar destfile="output/SimpleJavaClass.war"
      basedir="output/temp/war"/>

  </target>
```

Before you are able to execute the `jwsc` WebLogic Web Service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task. The `build-service` target includes other supporting standard Ant tasks in addition to `jwsc`: `mkdir` sets up the directory hierarchy for later packaging up Web Application WAR file under the temporary `output` directory; `javac` compiles the generated Java code into class files, and `jar` creates the WAR archive file that is later deployed to WebLogic Server.

See "Sample Ant Build File" on page 3-11 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean` and `undeploy`.

5. Execute the `jwsc` Ant task, along with the other supporting Ant tasks, by specifying the `build-service` target at the command line:

   ```
   prompt> ant build-service
   ```

   See the `output/temp/war/WEB-INF` directory to view the files and artifacts generated by the `jwsc` Ant task.

6. Start a WebLogic Server instance.

7. Deploy the Web Service to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In both cases, you deploy the Web Application file `SimpleJavaClass.war`, located in the `output` directory.

   To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

   ```
   <target name="deploy">
   ```

```
    <wldeploy action="deploy"
      name="SimpleJavaClassService"
      source="output/SimpleJavaClass.war" user="system"
      password="gumby1234" verbose="true"
      adminurl="t3://localhost:7001"
      targets="myserver"/>

  </target>
```

Substitute the values for `user`, `password`, `adminurl`, and `targets` that correspond to your WebLogic Server instance.

Deploy the WAR file by executing the `deploy` target:

```
prompt> ant deploy
```

8. Test that the Web Service is deployed correctly by invoking its Home Page in your browser:

   ```
   http://localhost:7001/webJWS/SimpleService
   ```

   This URL is constructed using the values of the `contextPath` and `serviceUri` attributes of the `WLHttpTransport` JWS annotation in the `SimpleImpl.java` JWS file. Use the hostname and port relevant to your WebLogic Server instance.

See "Invoking a Web Service from a Standalone JAX-RPC Java Client" on page 3-26 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

You can use the `build-service`, `undeploy`, and `deploy` targets in the `build.xml` file to iteratively update, rebuild, undeploy, and redeploy the Java class-implemented Web Service as part of your development process.

Sample SimpleImpl JWS File

```
package examples.webJWS;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

// Import the WebLogic-specific JWS annotation interfaces

import weblogic.jws.WLHttpTransport;

// Standard JWS annotation that specifies that the name of the Web Service is
// "Simple" and the targetNamespace used in the generated WSDL is
// "http://example.org"

@WebService(name="Simple", targetNamespace="http://example.org")
```

```
// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are rpc-encoded.

@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)

// WebLogic-specific JWS annotation that specifies the context path and
// service URI used to build the URI of the Web Service is
// "webJWS/SimpleService"

@WLHttpTransport(contextPath="webJWS", serviceUri="SimpleService")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 * @author Copyright (c) 2004 by BEA Systems.  All rights reserved.
 */

public class SimpleImpl {

  // Required constructor

  public SimpleImpl() {}

  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: sayHello.

  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```


Sample Ant Build File

```
<project default="all">

  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

  <target name="all" depends="clean,build-service,deploy" />

  <target name="clean">
    <delete dir="output" />
  </target>
```

```
<target name="build-service">
  <mkdir dir="output/temp/war/WEB-INF/classes"/>
  <jwsc
    source="SimpleImpl.java"
    destdir="output/temp/war/WEB-INF/" />
  <javac
    srcdir="output/temp/war/WEB-INF/"
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="*.java"/>
  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="*.java"/>
  <jar destfile="output/SimpleJavaClass.war"
    basedir="output/temp/war"/>
</target>
<target name="deploy">
  <wldeploy action="deploy"
    name="SimpleJavaClassService"
    source="output/SimpleJavaClass.war" user="system"
    password="gumby1234" verbose="true"
    adminurl="t3://localhost:7001"
    targets="myserver"/>
</target>
<target name="undeploy">
  <wldeploy action="undeploy"
    name="SimpleJavaClassService"
    user="system"
    password="gumby1234" verbose="true"
    adminurl="t3://localhost:7001"
    targets="myserver"/>
```

```
  </target>
</project>
```

Creating a Web Service that Uses User-Defined Data Types

The preceding use cases use only simple data types, such as integers or Strings, as the parameters and return values of the Web Service operations. This example shows how to create an EJB-implemented Web Service that uses a user-defined data type, in particular a JavaBean called `BasicStruct`, as both a parameter and a return value of its operation.

There is actually very little a programmer has to do to use a user-defined data type in a Web Service, other than create the Java source of the data type and use it correctly in the JWS file. The `jwsc` Ant task, when it encounters a user-defined data type in the JWS file, automatically generates all the data binding artifacts needed to convert data between its XML representation (used in the SOAP messages) and its Java representation (used in WebLogic Server.) The data binding artifacts include the XML Schema equivalent of the Java user-defined type, the JAX-RPC type mapping file, and so on.

The following procedure is very similar to the procedure in "Creating an EJB-Implemented Web Service Starting From Java" on page 3-2. For this reason, although the procedure does show all the needed steps, it provides details only for those steps that differ from the simple EJB example.

1. Open a command window and set your WebLogic Server environment.
2. Create a working directory:

   ```
   prompt> mkdir /examples/complex
   ```

3. Create the source for the `BasicStruct` JavaBean by opening your favorite Java IDE or text editor and creating a Java file called `BasicStruct.java`, in the working directory, using the Java code specified in "Sample BasicStruct JavaBean" on page 3-15.
4. Create the JWS file that implements the Web Service using the Java code specified in "Sample ComplexBean JWS File" on page 3-16.

   The sample JWS file differs from the simpler version only in that it imports the `examples.complex.BasicStruct` class and then uses the `BasicStruct` user-defined data type as both a parameter and return value of the `echoStruct()` method.
5. Create a standard Ant `build.xml` file in the working directory and add the following targets:

   ```
   <taskdef name="jwsc"
     classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />
   ```

```
<target name="build-service">
  <mkdir dir="output/temp/META-INF"/>
<target name="build-service">
  <mkdir dir="output/temp/META-INF"/>
  <jwsc
    source="ComplexBean.java"
    srcpath="../.."
    destdir="output/temp/META-INF/" />
  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp"
    includes="*.java"/>
  <javac
    srcdir="output/temp/META-INF/"
    source="1.5"
    destdir="output/temp/"
    includes="*.java"/>
  <jar destfile="output/complexBean.jar"
    basedir="output/temp"/>
</target>
```

This `build-service` target differs from the simpler example only in the extra attribute of the `jwsc` Ant task: `srcpath`. This attribute specifies the root directory of all the Java source files; because each Java file's package is `examples.complex`, the root is two directories above the working directory. This attribute is needed so the `jwsc` Ant task can find the source file for the `BasicStruct` user-defined data type and automatically generate the needed data binding artifacts, such as the corresponding XML Schema, JAX-RPC type mapping file, and so on.

See "Sample Ant Build File" on page 3-17 for a full sample `build.xml` file.

6. Execute the `jwsc` Ant task:

   ```
   prompt> ant build-service
   ```

   See the `output/temp/META-INF/schema` directory for the generated XML Schema files.

7. Start a WebLogic Server instance.

8. Deploy the Web Service, packaged in the `output/complexBean.jar` archive, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task.

9. Test that the Web Service is deployed correctly by invoking its Home Page in your browser:

```
http://localhost:7001/example/Complex
```

See "Invoking a Web Service from a Standalone JAX-RPC Java Client" on page 3-26 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

Sample BasicStruct JavaBean

```
package examples.complex;

/**
 * Defines a simple JavaBean called BasicStruct that has integer, String,
 * and String[] properties
 */

public class BasicStruct {

  // Properties

  private int intValue;
  private String stringValue;
  private String[] stringArray;

  // Constructor

  public BasicStruct() {}

  // Getter and setter methods

  public int getIntValue() {
    return intValue;
  }

  public void setIntValue(int intValue) {
    this.intValue = intValue;
  }

  public String getStringValue() {
    return stringValue;
  }

  public void setStringValue(String stringValue) {
    this.stringValue = stringValue;
  }

  public String[] getStringArray() {
    return stringArray;
  }

  public void setStringArray(String[] stringArray) {
    this.stringArray = stringArray;
  }
```

```
  public String toString() {
    return "IntValue="+intValue+", StringValue="+stringValue;
  }
}
```

Sample ComplexBean JWS File

```
package examples.complex;

import java.rmi.RemoteException;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

import javax.xml.soap.SOAPElement;

// Import the EJBGen annotation interfaces

import com.bea.wls.ejbgen.annotations.*;

// Import the WebLogic-specific JWS annotation interface

import weblogic.jws.WLHttpTransport;

// Import the BasicStruct JavaBean

import examples.complex.BasicStruct;

// EJBGen annotation that specifies that EJBGen should generate a stateless
// session EJB called Complex and its generated JAX-RPC Web service endpoint
// interface should be called "examples.complex.ComplexBeanPortType"

@Session(ejbName="Complex",
    serviceEndpoint="examples.complex.ComplexBeanPortType")

// Standard JWS annotation that specifies that the name of the Web Service is
// "ComplexPortType", its public service name is "ComplexService", and the
// targetNamespace used in the generated WSDL is "http://example.org"

@WebService(serviceName="ComplexService", name="ComplexPortType",
    targetNamespace="http://example.org")
```

```
// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are rpc-encoded.

@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)

// WebLogic-specific JWS annotation that specifies the context path and service
// URI used to build the URI of the Web Service is "example/Complex"

@WLHttpTransport(contextPath="example", serviceUri="Complex")

/**
 * This JWS file forms the basis of a stateless-session EJB implemented
 * WebLogic Web Service.  The Web Service has a single operation:
 * echoStruct(BasicStruct).
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class ComplexBean implements SessionBean {

  // Standard JWS annotation to expose method as an operation named
  // "echoStruct".

  @WebMethod()
  public BasicStruct echoStruct(BasicStruct struct)
      throws RemoteException
  {
    System.out.println("echoStruct called");
    return struct;
  }

  // Standard EJB methods.

  public void ejbCreate() {}
  public void ejbActivate() {}
  public void ejbRemove() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {}
}
```

Sample Ant Build File

```
<project default="all">

  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

  <target name="all" depends="clean,build-service,deploy" />
```

```
<target name="clean">
  <delete dir="output" />
</target>

<target name="build-service">

  <mkdir dir="output/temp/META-INF"/>

  <jwsc
    source="ComplexBean.java"
    srcpath="../.."
    destdir="output/temp/META-INF/" />

  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp"
    includes="*.java"/>

  <javac
    srcdir="output/temp/META-INF/"
    source="1.5"
    destdir="output/temp/"
    includes="*.java"/>

  <jar destfile="output/complexBean.jar"
    basedir="output/temp"/>

</target>

<target name="deploy">

  <wldeploy action="deploy"
    name="ComplexBeanService"
    source="output/complexBean.jar" user="system"
    password="gumby1234" verbose="true"
    adminurl="t3://localhost:7001"
    targets="myserver"/>

</target>

<target name="undeploy">

  <wldeploy action="undeploy"
    name="ComplexBeanService"
```

```
      user="system"
      password="gumby1234" verbose="true"
      adminurl="t3://localhost:7001"
      targets="myserver"/>
  </target>
</project>
```

Creating a Web Service From a WSDL File

A typical use case when creating a Web Service is to start from an existing WSDL file. The WSDL file can be thought of as a public contract that specifies what the Web Service looks like, such as the list of supported operations, the signature and shape of each operation, the protocols and transports that can be used when invoking the operations, and the XML Schema data types that are used when transporting the data over the wire. Based on this WSDL file, you generate the J2EE artifacts that implement the Web Service so that it can be deployed to the Web container of WebLogic Server. These artifacts include:

- The Java interface file that represents the implementation of your Web Service.
- The Java exception class for user-defined exceptions specified in the WSDL file.
- The deployment descriptor files required to deploy the Web Service: `webservices.xml`, `web.xml`, JAX-RPC mapping file, and all WebLogic-specific deployment descriptor files (`weblogic.xml`, `weblogic-webservices.xml`), if needed.
- The Java representation of any user-defined data types used in the operations of the Web Service, based on the XML Schema representation in the WSDL file.

You use the `wsdl2service` Ant task to generate these artifacts. After you run the Ant task for the first time, you code the Java class that contains the business logic of your Web Service, based on the generated interface file. In other words, you add code to the methods that implement the Web Service operations so that the operations behave as needed. Then you iteratively rerun the `wsdl2service` Ant task, regenerating code until the Web Service deploys and works as you want.

The following simple example shows how to create a Web Service from the WSDL file shown in "Sample WSDL File" on page 3-22. The Web Service has one operation, `getTemp`, that returns a temperature when passed a zip code.

1. Open a command window and set your WebLogic Server environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the domain directory of WebLogic Server.
2. Create a working directory:

```
prompt> mkdir /examples/wsdl2service
```

3. Put your WSDL file into an accessible directory on your computer. For the purposes of this example, it is assumed that your WSDL file is called `TemperatureService.wsdl` and is located in the `/examples/wsdls` directory. See "Sample WSDL File" on page 3-22 for a full listing of the file.
4. Create a standard Ant `build.xml` file in the working directory and add the following calls to the `wsdl2service`, `javac`, and `jar` Ant tasks, wrapped inside of the `build-service` target:

```
<taskdef name="wsdl2service"
  classname="weblogic.wsee.tools.anttasks.Wsdl2ServiceAntTask" />

<target name="build-service">

  <mkdir dir="output/temp/war/WEB-INF/classes"/>

  <wsdl2service destdir="output/temp/war/WEB-INF/"
                verbose="true"
                contextpath="/wsdl2service">

    <wsdl location="/examples/wsdls/TemperatureService.wsdl"
      packageName="examples.wsdl2service"
      impl="examples.wsdl2service.TemperatureImpl"
      uri="/temperature" />

  </wsdl2service>

  <javac
    srcdir="output/temp/war/WEB-INF/"
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="**/*.java"/>

  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="**/*.java"/>

  <jar destfile="output/temperature.war"
    basedir="output/temp/war"/>

</target>
```

Before you are able to execute the `wsdl2service` WebLogic Web Service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task. The `build-service` target includes other supporting standard Ant tasks in addition to `wsdl2service`: `mkdir` sets up an directory hierarchy for later packaging up a WAR file under the temporary `output` directory; `javac` compiles the generated Java code into class files, and `jar` creates the WAR archive file that is later deployed to WebLogic Server.

See "Sample Ant Build File" on page 3-24 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean` and `undeploy`.

5. Execute the `wsdl2service` Ant task, along with the other supporting Ant tasks, by specifying the `build-service` target at the command line:

   ```
   prompt> ant build-service
   ```

   The `wsdl2service` Ant task generates a Java interface file called `TemperaturePortType.java` in the `temp/war/WEB-INF/examples/wsdl2service` directory. This interface file implements the Web Service specified in the WSDL file.

   See the `output/temp/war/WEB-INF` directory for the other files and artifacts generated by the `wsdl2service` Ant task.

6. Create an implementation of the `TemperaturePortType.java` interface by opening your favorite Java IDE or text editor and creating a Java file in the working directory called `TemperatureImpl.java` using the Java code specified in "Sample Java Implementation File" on page 3-23.

   This implemention is where you add the actual business logic that specifies what the operations of the Web Service actually do. For simplicity, the sample `getTemp()` method in `TemperatureImpl.java` returns a hard-coded number; in real-life, the implementation of this method would actually look up the current temperature at the given zip code.

7. Rerun the `build-service` target to compile the `TemperatureImpl.java` class and add it to the `temperature.war` file:

   ```
   prompt> ant build-service
   ```

   You can iteratively keep rerunning this target until it builds successfully.

8. Start a WebLogic Server instance.

9. Deploy the Web Service to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In both cases, you deploy the Web Application file `temperature.war`, located in the `output` directory.

   To use the `wldeploy` Ant task, add the following target to the `build.xml` file:

```
<target name="deploy">

  <wldeploy action="deploy"
    name="TemperatureWebService"
    source="output/temperature.war" user="system"
    password="gumby1234" verbose="true"
    adminurl="t3://localhost:7001"
    targets="myserver"/>

</target>
```

Substitute the values for `user`, `password`, `adminurl`, and `targets` that correspond to your WebLogic Server instance.

Deploy the WAR file by executing the `deploy` target:

```
prompt> ant deploy
```

10. Test that the Web Service is deployed correctly by invoking its Home Page in your browser:

    ```
    http://localhost:7001/wsdl2service/temperature
    ```

    This URL is constructed using the values specified for the `contextpath` and `uri` attributes of the `wsdl2service` Ant task. Use the hostname and port relevant to your WebLogic Server instance.

See "Invoking a Web Service from a Standalone JAX-RPC Java Client" on page 3-26 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

You can use the `build-service`, `undeploy`, and `deploy` targets in the `build.xml` file to iteratively update, rebuild, undeploy, and redeploy the `Temperature` Web Service as part of your development process.

Sample WSDL File

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

<definitions name="TemperatureService"
   targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
   xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">

   <message name="getTempRequest">
         <part name="zipcode" type="xsd:string"/>
   </message>
```

```
    <message name="getTempResponse">
           <part name="return" type="xsd:float"/>
    </message>

    <portType name="TemperaturePortType">
         <operation name="getTemp">
              <input message="tns:getTempRequest"/>
              <output message="tns:getTempResponse"/>
          </operation>
    </portType>

    <binding name="TemperatureBinding" type="tns:TemperaturePortType">
        <soap:binding style="rpc"
                    transport="http://schemas.xmlsoap.org/Soap/http"/>
         <operation name="getTemp">
            <soap:operation soapAction=""/>
              <input>
                 <soap:body use="encoded"
                    namespace="urn:xmethods-Temperature"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
              </input>
              <output>
                 <soap:body use="encoded"
                    namespace="urn:xmethods-Temperature"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
              </output>
         </operation>
    </binding>

    <service name="TemperatureService">
       <documentation>Returns current temperature in a given U.S. zipcode
       </documentation>
       <port name="TemperaturePort" binding="tns:TemperatureBinding">
          <soap:address
             location="http://services.xmethods.net:80/soap/servlet/rpcrouter"/>
       </port>
    </service>

</definitions>
```

Sample Java Implementation File

```
package examples.wsdl2service;

/**
 * This class implements the TemperaturePortType interface that
 * was generated by the "wsdl2service" Ant task based on an existing WSDL
 * file that describes the TemperatureService Web Service.
```

```
 * <br>
 * The WSDL file specifies that the TemperatureService Web Service has
 * one operation, getTemp, that returns a float that represents the current
 * temperature at the inputted String zipcode location, the single parameter
 * to the operation.
 * <br>
 * The wsdl2service automatically generates the interface file that
 * represents the operation (or portType, using WSDL terminology).  It is
 * up to the user to then create the class that implements the interface.
 * <br>
 * @author Copyright (c) 2003 by BEA Systems. All Rights Reserved.
 */
public class TemperatureImpl implements TemperaturePortType {
  /**
   * This method implements the getTemp operation (or portType) of the
   * TemperatureService, as specified by the wsdl2service-generated
   * TemperaturePortType interface based on the existing WSDL of the
   * Web Service.
   * <br>
   * For the sake of simplicity, this implementation does nothing other than
   * return a hard-coded value: 100.1.  However, a real-world implementation
   * of the operation would return the actual current temperature of the
   * location specified by the inputted zipcode.
   */
  public float getTemp(String zipcode) {
    return 100.1f;
  }
}
```


Sample Ant Build File

```
<project default="all">
  <taskdef name="wsdl2service"
    classname="weblogic.wsee.tools.anttasks.Wsdl2ServiceAntTask" />
  <target name="all" depends="clean,build-service,deploy" />
```

```
<target name="clean">
  <delete dir="output" />
</target>

<target name="build-service">

  <mkdir dir="output/temp/war/WEB-INF/classes"/>

  <wsdl2service destdir="output/temp/war/WEB-INF/"
                verbose="true"
                contextpath="/wsdl2service">

    <wsdl location="/examples/wsdls/TemperatureService.wsdl"
      packageName="examples.wsdl2service"
      impl="examples.wsdl2service.TemperatureImpl"
      uri="/temperature" />

  </wsdl2service>

  <javac
    srcdir="output/temp/war/WEB-INF/"
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="**/*.java"/>

  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="**/*.java"/>

  <jar destfile="output/temperature.war"
    basedir="output/temp/war"/>

</target>

<target name="deploy">

  <wldeploy action="deploy"
    name="TemperatureWebService"
    source="output/temperature.war" user="system"
    password="gumby1234" verbose="true"
    adminurl="t3://localhost:7001"
    targets="myserver"/>
```

```
  </target>
  <target name="undeploy">
    <wldeploy action="undeploy"
      name="TemperatureWebService"
      user="system"
      password="gumby1234" verbose="true"
      adminurl="t3://localhost:7001"
      targets="myserver"/>
  </target>
</project>
```

Invoking a Web Service from a Standalone JAX-RPC Java Client

A different type of use case is to invoke an operation of a deployed Web Service from a client application. The Web Service could be deployed to WebLogic Server or to any other application server, such as .NET; all you need to know is the URL to its public contract file, or WSDL. This example shows how to create a standalone Java client application that invokes the EJB-implemented WebLogic Web Service that uses a user-defined data type described in "Creating a Web Service that Uses User-Defined Data Types" on page 3-13; it is assumed that you have successfully deployed the Web Service.

In addition to writing the Java client application, you must also run the `clientgen` WebLogic Web Service Ant task to generate the artifacts that your client application needs to invoke the Web Service operation. These artifacts include:

- The Java source code for the JAX-RPC `Stub` and `Service` interface implementations for the particular Web Service you want to invoke.
- The Java classes for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

The following example shows how to create a Java client application that invokes the `echoStruct` operation of the deployed WebLogic Web Service described in "Creating a Web Service that Uses User-Defined Data Types" on page 3-13. The `echoStruct` operation takes as a parameter the `BasicStruct` user-defined data type.

1. Open a command window and set your WebLogic Server environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the domain directory of WebLogic Server.
2. Create a working directory:

```
prompt> mkdir /examples/simpleClient
```

3. Create a standard Ant `build.xml` file in the working directory and add the following calls to the `clientgen` and `javac` Ant tasks, wrapped inside of the `build-client` target:

```
<taskdef name="clientgen"
  classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

<target name="build-client">
  <mkdir dir="clientclasses"/>

  <clientgen
    wsdl="http://localhost:7001/example/Complex?WSDL"
    destDir="clientclasses"
    packageName="examples.simpleClient"/>

  <javac
    srcdir="." destdir="clientclasses"
    includes="**/*.java"/>

  <javac
    srcdir="clientclasses" destdir="clientclasses"
    includes="examples/simpleClient/**/*.java" />

</target>
```

   Before you are able to execute the `clientgen` WebLogic Web Service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task. The `clientgen` Ant task uses the WSDL of the deployed WebLogic Web Service to generate the needed artifacts and puts them into the `clientclasses` directory, using the specified package name. The clientgen Ant task also generates the `examples.complex.BasicStruct` JavaBean class, which is the Java representation of the user-defined data type specified in the WSDL. The `build-client` target includes other supporting standard Ant tasks in addition to `clientgen`: `mkdir` creates a directory for the generated client source files and compiled classes and `javac` compiles all the Java code into class files.

   See "Sample Ant Build File" on page 3-29 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`.

4. Execute the `clientgen` Ant task, along with the other supporting Ant tasks, by specifying the `build-client` target at the command line:

```
prompt> ant build-client
```

See the `clientclasses` directory to view the files and artifacts generated by the `clientgen` Ant task.

5. Create the Java client application file that invokes the `echoStruct` operation by opening your favorite Java IDE or text editor, creating a Java file called `Main.java` using the code specified in "Sample Java Client Application" on page 3-29.

   The `Main` client application takes a single argument: the WSDL of the Web Service. The application then uses standard JAX-RPC procedures for invoking an operation of the Web Service using the Web Service-specific implemenation of the `Service` interface generated by `clientgen`. The application also imports and uses the `BasicStruct` user-defined type, generated by the `clientgen` Ant task, that is used as a parameter and return value for the `echoStruct` operation. For details, see Chapter 7, "Invoking Web Services."

6. Add the following targets to the `build.xml` file, used to execute the `Main` application:

```
<path id="client.class.path">
  <pathelement path="clientclasses"/>
  <pathelement path="${java.class.path}"/>
</path>

<target name="run" >
  <java fork="true" classname="examples.simpleClient.Main"
    failonerror="true" >
    <classpath refid="client.class.path"/>
    <arg line="http://localhost:7001/example/Complex?WSDL" />
  </java>

</target>
```

   The `path` task adds the `clientclasses` directory to the CLASSPATH. The `run` target invokes the `Main` application, passing it the WSDL of the deployed Web Service as its single argument.

7. Rerun the `build-client` target to regenerate the artifacts and recompile into classes, then execute the `run` target to invoke the `echoStruct` operation:

```
prompt> ant build-client run
```

   If the invoke was successful, you should see the following final output:

```
run:
     [java] echoStruct called. result: 0, null
```

You can use the `build-client` and `run` targets in the `build.xml` file to iteratively update, rebuild, and run the Java client application as part of your development process.

Sample Java Client Application

```
package examples.simpleClient;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;

import examples.complex.BasicStruct;

/**
 * This is a simple standalone client application that invokes the
 * the <code>echoStruct</code>  operation of the ComplexService Web service.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class Main {

  public static void main(String[] args)
      throws ServiceException, RemoteException{

    ComplexService service = new ComplexService_Impl (args[0]);
    ComplexPortType port = service.getComplexPortTypeSoapPort();

    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
    in.setStringValue("Hello Struct");

    BasicStruct result = port.echoStruct(in);
    System.out.println("echoStruct called. result: " + result.getIntValue()+ ",

" + result.getStringValue());

  }

}
```


Sample Ant Build File

```
<project default="all">

  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

  <path id="client.class.path">
    <pathelement path="clientclasses"/>
    <pathelement path="${java.class.path}"/>
  </path>
```

```
  <target name="all" depends="clean,build-client,run" />

  <target name="clean">
    <delete dir="clientclasses" />
  </target>

  <target name="build-client">
    <mkdir dir="clientclasses"/>

    <clientgen
      wsdl="http://localhost:7001/example/Complex?WSDL"
      destDir="clientclasses"
      packageName="examples.simpleClient"/>

    <javac
      srcdir="." destdir="clientclasses"
      includes="**/*.java"/>

    <javac
      srcdir="clientclasses" destdir="clientclasses"
      includes="examples/simpleClient/**/*.java" />

  </target>

  <target name="run" >
    <java fork="true" classname="examples.simpleClient.Main"
      failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg line="http://localhost:7001/example/Complex?WSDL" />
    </java>

  </target>

</project>
```


Other Web Service Use Cases

Other Web Service use cases include:

- Create a Web Service starting from an XML Schema file.

  [[More information to come.]]

- Invoke a Web Service from a component running on WebLogic Server, such as an EJB, servlet, or another Web Service.

[[More information to come.]]

- Create a conversational Web Service using JWS

  [[More information to come.]]

- Create a Web Service that uses callbacks

  [[More information to come.]]

- Create a reliable Web Service

  [[More information to come.]]

- Create a data-secured Web Service that uses encryption and digital signatures.

  [[More information to come.]]

CHAPTER 4

Iteratively Developing WebLogic Web Services

The following sections provide information about the steps needed to iteratively develop WebLogic Web Services:

Overview of the WebLogic Web Service Programming Model

The WebLogic Web Services programming model centers around *JWS files* (Java files that use *JWS annotations* to specify the shape and behavior of the Web Service) and Ant tasks that execute on the JWS file. JWS annotations are based on the new metadata feature of Version 5.0 of the JDK (specified by JSR-175), and include both the standard annotations defined by the

Web Services Metadata for the Java Platform specification (JSR-181), as well as additional WebLogic-specific ones.

In this programming model, you create a JWS file that contains a public method for each Web Service operation, and then code the method so that the operation does what you want it to do. You then use JWS annotations to specify other Web Service characteristics, such as whether the service is implemented with a Java class or a stateless session EJB, the type of SOAP bindings of the Web Service (rpc/document or literal/encoded), the context path and service URI, and so on. If your Web Service operations use user-defined data types, you create the Java class that describes this data type, and then import this class into your JWS file and specify it as a return value or parameter to the method.

Once you've programmed the JWS file and created all user-defined Java data types, you then use the `jwsc` WebLogic Web Service Ant task to generate all the supporting artifacts: deployment descriptors, WSDL files, JAX-RPC type mapping files, and so on. Finally, you compile the Java files into class files, package all artifacts into a deployable WAR or JAR file, then deploy the archive as usual.

Note: The use case of starting from a WSDL file is slightly different from the starting-from-Java use case in that it does not use JWS files. Instead, you use the `wsdl2service` Ant task to generate the Java interface file that describes the J2EE implementation of the Web Service, and then you create a standard Java class (that does *not* use JWS annotations) that implements the generated interface file. The rest of the programming model is the same. For details of this specific use case, see "Creating a Web Service From a WSDL File" on page 3-19.

Iteratively Developing WebLogic Web Services: Main Steps

Iterative development refers to setting up your development environment in such a way so that you can repeatedly code, compile, package, deploy, and test a Web Service until it works as you want.

The WebLogic Web Service programming model uses Ant tasks to perform most of the steps of the iterative development processs. Typically, you create a single `build.xml` file that contains targets for all the steps, then repeatedly run the targets, after you have updated your JWS file with new Java code, to test that the updates work as you expect.

This section describes the general procedure for iteratively developing WebLogic Web Services using JWS files and Ant tasks. See Chapter 3, "Common Web Services Use Cases and Examples," for specific examples of this process based on common use cases. The procedure is

just a recommendation; if you have already set up your own development environment, you can use this procedure as a guide to updating your environment to develop WebLogic Web Services.

This procedure also does not use the WebLogic Web Services split development directory environment. If you are using this development environment, and would like to integrate Web Services development into it, see "Integrating Web Services Into the WebLogic Split Development Directory Environment" on page 4-13 for details.

To iteratively develop a WebLogic Web Service, follow these steps.

1. Open a command window and set your WebLogic Server environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.
2. Create a source directory that will contain the JWS file, Java source for any user-defined data types, and the Ant `build.xml` file.
3. In the source directory, create the JWS file that implements your Web Service.

   See Chapter 5, "Programming the JWS File."
4. If your Web Service uses user-defined data types, create the JavaBean that describes it.

   See "Programming the User-Defined Java Data Type" on page 5-13.
5. In the source directory, create a basic Ant build file called `build.xml`.

   See "Creating the Basic Ant build.xml File" on page 4-4.
6. Run the `jwsc` Ant task against the JWS file to generate source code, data binding artifacts, deployment descriptors, and so on, into the destination directory.

   See "Running the jwsc WebLogic Web Services Ant Task" on page 4-5
7. Compile all code (user and generated) into class files.

   "Compiling User and Generated Java Code" on page 4-7.
8. Package all files into a WAR or JAR file.

   See "Packaging the Web Service into a WAR or JAR File" on page 4-8.
9. Deploy the Web Service to WebLogic Server.

   See "Deploying and Undeploying WebLogic Web Services" on page 4-9.

10. Invoke the Web Service Home Page to test that the Web Service was deployed correctly.

    See "Invoking the WSDL and Home Page of the Web Service" on page 4-11.

To make changes to the Web Service, update the JWS file, undeploy the Web Service as described in "Deploying and Undeploying WebLogic Web Services" on page 4-9, then repeat the steps starting from running the `jwsc` Ant task.

See Chapter 7, "Invoking Web Services," for information on writing client applications that invoke a Web Service.

Creating the Basic Ant build.xml File

Ant uses build files written in XML (default name `build.xml`) that contain one project and one or more targets that specify different stages in the Web Services development process. Each target contains one or more tasks, or pieces of code that can be executed. This section describes how to create a basic Ant build file; later sections describe how to add targets to the build file that specify how to exeucte various stages of the Web Services development process, such as running the `jwsc` Ant task to process a JWS file, compiling Java source code, and deploying the Web Service to WebLogic Server.

The following skeleton `build.xml` file specifies a default `all` target that calls all the other targets that will be added in later sections:

```
<project default="all">

  <target name="all"
          depends="clean,build-service,compileSource,package,deploy" />

  <target name="clean">
    <delete dir="output" />
  </target>

  <target name="build-service">
     <!--add jwsc and related tasks here -->
  </target>

  <target name="compileSource">
     <!--add tasks to compile the source code here -->
  </target>

  <target name="package">
     <!--add tasks to package the web service here -->
  </target>
```

```
  <target name="deploy">
      <!--add wldeploy task here -->
 </target>

</project>.
```

Running the jwsc WebLogic Web Services Ant Task

The `jwsc` Ant task takes as input a JWS file that contains both standard (JSR-181) and WebLogic-specific JWS annotations and generates all the artifacts you need to create a WebLogic Web Service. These generated artifacts include:

- Java source files that implement a standard JSR-921 Web Service. The generated source files include the various EJB-related Java files, such as the `Home` and `Remote` interfaces, in the case of an EJB-implemented Web Service.
- All required deployment descriptors. In addition to the standard `webservices.xml` and JAX-RPC mapping files, the `jwsc` Ant task also generates the WebLogic-specific Web Services deployment descriptor (`weblogic-wesbservices.xml`), the `web.xml` and `weblogic.xml` files for Java class-implemented Web Services and the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files for EJB-implemented Web Services.
- The XML Schema representation of any Java user-defined types used as parameters or return values to the Web Service operations.
- The WSDL file that publicly describes the Web Service.

To run the `jwsc` Ant task, add the following `taskdef` and `build-service` target to the `build.xml` file:

```
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

<target name="build-service">

  <mkdir dir="temp_output_directory">

  <jwsc
    source="JWS_file"
    destdir="destination_directory" />

</target>
```

where

- *`temp_output_directory`* refers to a temporary directory that will contain all the generated artifacts. The contents of this directory will later be packaged into a deployable

Web Service. Because EJBs and Web applications are packaged slightly differently, you should structure the output directory according to the implemenation of your Web Service (EJB or Java class). In particular:

  – If your Web Service is implemented with a stateless session EJB, create an output directory that contains a `META-INF` subdirectory.

  – If your Web Service is implemented with a Java class, create an output directory that contains a `WEB-INF/classes` subdirectory.

- *`JWS_file`* refers to the name of your JWS file.
- *`destination_directory`* refers to the directory into which the `jwsc` Ant task should put the generated artifacts. Depending on the type of Web Service implementation, this might also be `temp_output_directory`.

The required `taskdef` element specifies the full class name of the `jwsc` Ant task.

The following `build.xml` excerpt shows an example of running the `jwsc` Ant task on a JWS file for an EJB-implemented Web Service:

```
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

<target name="build-service">

  <mkdir dir="output/temp/META-INF"/>

  <jwsc
    source="SimpleBean.java"
    destdir="output/temp/META-INF/" />

</target>
```

The following `build.xml` excerpt shows the equivalent `build-service` target for a Java class implemented Web Service:

```
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

<target name="build-service">

  <mkdir dir="output/temp/war/WEB-INF/classes"/>

  <jwsc
    source="SimpleImpl.java"
    destdir="output/temp/war/WEB-INF/" />

</target>
```

Compiling User and Generated Java Code

The Web Services Ant tasks, such as `jwsc` and `wsdl2service`, generate Java source code based on the JWS file or the WSDL file, respectively. This source code, along with the JWS file, needs to be compiled into Java classes before the service can be packaged and deployed to WebLogic Server.

Use the standard `javac` Ant task to compile the source code by adding the following `compileSource` target to your `build.xml` file:

```
<target name="compileSource">

  <javac
    srcdir="JWSCDestinationDir"
    source="1.5"
    destdir="RootDirforCompiledClasses"
    includes="*.java"/>

  <javac
    srcdir="JWSLocationDir"
    source="1.5"
    destdir="RootDirforCompiledClasses"
    includes="*.java"/>

</target>
```

where

- *`JWSCOutputDir`* refers to the destination directory of either the `jwsc` or `wsdl2service` Ant task, or in other words, the directory into which the Ant tasks generate Java source code. This is the value of the `destdir` attribute of either Ant task.
- *`RootDirforCompiledClasses`* refers to the top-level directory into which the Java classes should be compiled. This top-level directory depends on whether your Web Service is implemented with a Java class or a stateless session EJB, because of the structure of the correspoding WAR or EJB package.

  If your Web Service is implemented with a Java class, the *`RootDirforCompiledClasses`* directory should be something like *`output`*`/WEB-INF/classes`, where *`output`* refers to an output directory; this is because WAR files include compiled Java classes in the `WEB-INF/classes` directory. If your Web Service is implemented with an EJB, the *`RootDirforCompiledClasses`* directory should be the top-level output directory; this is because EJB JAR files include compiled Java classes at the root level of the archive.
- *`JWSLocationDir`* refers to the directory that contains the source Java files, such as the JWS file in the case of using the `jwsc` Ant task.

For example, the following `javac` tasks are used to compile Java code in the case of a Java class-implemented Web Service:

```
<target name="compileSource">

  <javac
    srcdir="output/temp/war/WEB-INF/"
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="*.java"/>

  <javac
    srcdir="."
    source="1.5"
    destdir="output/temp/war/WEB-INF/classes"
    includes="*.java"/>

</target>
```

Packaging the Web Service into a WAR or JAR File

Web Services are packaged as WAR or JAR files, depending on whether you have chosen a Java class or a stateless session EJB to implement the service.

Note: It is assumed in this section that you have generated the Web Service artifacts and compiled the source code into a directory hierarchy that is ready to be packaged into a standard J2EE WAR or JAR file. See preceding sections for more information.

Use the standard `jar` Ant task to package the Web Service into a deployable WAR or JAR file. To use the `jar` task, add the following target to your `build.xml` file:

```
<target name="package">

  <jar
    destfile="ArchiveFile"
    basedir="RootExplodedDirectory" />

</target>
```

where

- *`ArchiveFile`* refers to the name of the generated archive file. If your Web Service is implemented with a Java class, specify a `.war` extension; if your Web Service is implemented with a stateless session EJB, specify a `.jar` extension.
- *`RootExplodedDirectory`* refers to the top-level directory that contains the compiled source code and artifacts generated by the `jwsc` or `wsdl2service` Ant tasks. The root directory contains either a `WEB-INF` directory (for creating WAR files) or a `META-INF` directory (for creating EJB files.)

For example, the following `jar` task creates a WAR file called `TemperatureService.war` in the output directory, under the current directory, by packaging up the files under the `output/war` directory:

```
<target name="package">

  <jar
    destfile="output/TemperatureService.war"
    basedir="output/war" />

</target>
```

For the `jar` task to complete successfully, there must exist an `output/war/WEB-INF` directory that contains all the generated deployment descriptors, as well as all the other artifacts needed to build a Web application.

To actually package the Web Service, execute the `package` target at the command-line:

```
prompt> ant package
```

There are many more attributes available for the `jar` Ant task; for details, see the Apache Ant documentation at http://ant.apache.org/manual/index.html.

Deploying and Undeploying WebLogic Web Services

Because Web Services are packaged as either standard Web applications or EJBs, depending on the type of implementation you have chosen, deploying a Web Service simply means deploying the corresponding WAR or JAR file.

There are a variety of ways to deploy WebLogic applications, from using the Administration Console to using the `weblogic.Deployer` Java utility. There are also various issues you must consider when deploying an application to a production environment as opposed to a development environment. For a complete discussion about deployment, see *Deploying WebLogic Server Applications*.

This guide, because of its development nature, discusses just two ways of deploying Web Services:

- Using the wldeploy Ant Task to Deploy Web Services
- Using the Administration Console to Deploy Web Services

Using the wldeploy Ant Task to Deploy Web Services

The easiest way to quickly deploy a Web Service as part of the iterative development process is to add a target that executes the `wldeploy` WebLogic Ant task to your `build.xml` file that

contains the `jwsc`, `javac`, and other Ant tasks. You can add tasks to both deploy and undeploy the Web Service so that as you add more Java code and regenerate the service, you can redeploy and test it iteratively.

To use the `wldeploy` Ant task, add the following target to your `build.xml` file:

```
<target name="deploy">

    <wldeploy action="deploy"
      name="DeploymentName"
      source="SourcFile" user="AdminUser"
      password="AdminPassword"
      adminurl="AdminServerURL"
      targets="ServerName"/>

</target>
```

where

- *DeploymentName* refers to the deployment name of the application, or the name that appears in the Administration Console under the list of deployments.
- *SourceFile* refers to the name of the actual WAR or JAR file that is being deployed.
- *AdminUser* refers to administrative username.
- *AdminPassword* refers to the administrative password.
- *AdminServerURL* refers to the URL of the Administration Server, typically `t3://localhost:7001`.
- *ServerName* refers to the name of the WebLogic Server instance to which you are deploying the Web Service.

For example, the following `wldeploy` task specifies that the Web applciation packaged in the `output/TemperatureService.war` file, relative to the current directory, be deployed to the `myServer` WebLogic Server instance. Its deployed name is `TemperatureService`.

```
<target name="deploy">

    <wldeploy action="deploy"
      name="TemperatureService"
      source="output/TemperatureService.war" user="weblogic"
      password="weblogic"
      adminurl="t3://localhost:7001"
      targets="myServer"/>

</target>
```

To actually deploy the Web Service, execute the `deploy` target at the command-line:

```
prompt> ant deploy
```

You can also add a target to easily undeploy the Web Service so that you can make changes to its source code, then redeploy it:

```
<target name="undeploy">

    <wldeploy action="undeploy"
      name="TemperatureService"
      user="weblogic"
      password="weblogic"
      adminurl="t3://localhost:7001"
      targets="myServer"/>

</target>
```

When undeploying a Web Service, you do not specify the `source` attribute, but rather undeploy it by its name.

Using the Administration Console to Deploy Web Services

To use the Administration Console to deploy the Web Service, first invoke it in your browser using the following URL:

```
http://[host]:[port]/console
```

where:

- *`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`).

Then use the deployment assistants to help you deploy the WAR or JAR file. For more information on the Administration Console, see the Online Help.

Invoking the WSDL and Home Page of the Web Service

Every Web Service deployed on WebLogic Server has a Home Page that you can invoke in your browser. From the Home page you can:

- View the WSDL that describes the service.
- Test each operation to ensure that it is working correctly.
- View the SOAP request and response messages from a successful test of an operation.

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

```
http://[host]:[port]/[contextPath]/[serviceUri]
http://[host]:[port]/[contextPath]/[serviceUri]?WSDL
```

where:

- *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`).
- *contextPath* refers to the value of the `contextPath` attribute of the `@WLHttpTransport` JWS annotation of the JWS file that implements your Web Service.
- *serviceUri* refers to the value of the `serviceUri` attribute of the `@WLHttpTransport` JWS annotation of the JWS file that implements your Web Service.

For example, assume you used the following `@WLHttpTransport` annotation in the JWS file that implements your Web Service

```
...
@WLHttpTransport(portName="helloPort",
                 contextPath="ejbJWS",
                 serviceUri="SimpleBean")

/**
 * This JWS file forms the basis of a stateless-session EJB implemented
 * WebLogic Web Service.
 *
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public class SimpleBean implements SessionBean {

...
```

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/ejbJWS/SimpleBean
```

The URL to get view the WSDL of the Web Service is:

```
http://ariel:7001/ejbJWS/SimpleBean?WSDL
```

Integrating Web Services Into the WebLogic Split Development Directory Environment

This section describes how to integrate Web Services development into the WebLogic split development directory environment. It is assumed that you understand this WebLogic feature and have already set up this type of environment for developing standard J2EE applications and modules, such as EJBs and Web applications, and you want to update the single `build.xml` file to include Web Services development.

For detailed information about the WebLogic split development directory environment, see Creating a Split Development Directory for an Application at http://e-docs.bea.com/wls/docs90/programming/splitcreate.html and the `splitdir` example installed with WebLogic Server, located in the *BEA_HOME*`/weblogic90/samples/server/examples/src/examples/splitdir` directory, where *BEA_HOME* refers to the main installation directory for BEA products, such as `c:/bea`.

1. In the main project directory, create a directory that will contain the JWS file that implements your Web Service.

   For example, if your main project directory is called `/src/helloEar`, then create a directory called `/src/helloEar/helloWebService`:

   ```
   prompt> mkdir /src/helloEar/helloWebService
   ```

2. Put your JWS file in the just-created Web Service subdirectory of your main project directory.

3. In the `build.xml` file that builds the Enterprise application, create a new target to build the Web Service, adding a call to the `jwsc` WebLogic Web Service Ant task, as described in "Running the jwsc WebLogic Web Services Ant Task" on page 4-5. Also include a `mkdir` task to create an equivalent Web Service subdirectory (typically with the same name as the source directory) in the destination directory.

   Make sure that the `jwsc source` attribute points to the JWS file in its Web Service source subdirectory and the `jwsc destdir` attribute points to the directory created with the `mkdir` task, as shown in the following example:

   ```
   <target name="build-webservice">
     <mkdir dir="${dest.dir}/helloWebService/META-INF" />
     <jwsc
         source="helloWebService/SimpleBean.java"
         destdir="${dest.dir}/helloWebService/META-INF" />
   </target>
   ```

In the example, `${dest.dir}` refers to the destination directory of the other split development directory environment Ant tasks, such as `wlappc` and `wlcompile`. In the example, the Web Service is implemented with an EJB; for a Java class implemented Web Service, you would create a `${dest.dir}/helloWebService/WEB-INF/classes` output directory into which the `jwsc` Ant task would generate artifacts.

4. In the `build.xml` file, create a new target to compile the code generated by the `jwsc` Ant task, as described in "Compiling User and Generated Java Code" on page 4-7.

   Be sure you set the `srcdir` and `destdir` of the `javac` task to point to the Web Service source and destination directories of the split development directory environment, as shown in the following example:

```
<target name="compile-webservice">

   <javac
     srcdir="${dest.dir}/helloWebService/META-INF/"
     source="1.5"
     destdir="${dest.dir}/helloWebService/"
     includes="*.java"/>

   <javac
     srcdir="helloWebService/"
     source="1.5"
     destdir="${dest.dir}/helloWebService/"
     includes="*.java"/>
 </target>
```

5. Update the main build target of the `build.xml` file to call the Web Service-related targets:

```
<!-- Builds the entire helloWorldEar application -->

<target name="build"
  description="Compiles helloWorldEar application and runs appc"
  depends="build-webservice,compile-webservice,compile,appc" />
```

   Warning: When you actually build your Enterprise Application, be sure you run the `jwsc` Ant task and compile the generated code *before* you run the `wlappc` Ant task. This is because `wlappc` requires some of the artifacts generated by `jwsc` for it to execute successfully. In the example, this means that you should specify the `build-webservice` and `compile-webservice` targets *before* the `appc` target.

6. Update the `application.xml` file in the `META-INF` project source directory, adding either an `<ejb>` or `<web>` module, depending on the type of Web Service implementation. Specify the name of the subdirectory that contains the JWS file.

   For example, if your Web Service is implemented with an EJB, add the following to the application.xml file:

```
<module>
  <ejb>helloWebService</ejb>
</module>
```

The following example shows how to specify a Java class implemented Web Service:

```
<module>
  <web>
    <web-uri>helloWebService</web-uri>
  </web>
</module>
```

Your split development directory environment has now been updated to include Web Service development. After you have rebuilt and deployed the entire Enterprise Application, the Web Service will also be deployed as part of the EAR. You invoke the Web Service in the standard way described in "Invoking the WSDL and Home Page of the Web Service" on page 4-11, even though the Web Service is deployed as part of an Enterprise Application.

CHAPTER 5

Programming the JWS File

The following sections provide information about programming the JWS file that implements your Web Service:

Overview of JWS Files and JWS Annotations

One way to program a WebLogic Web Service is to code the standard JSR-921 EJB or Java class from scratch and generate its associated artifacts manually (deployment descriptor files, WSDL file, data binding artifacts for user-defined data types, and so on), but it can be difficult and tedious. BEA recommends that you take advantage of the new JDK 1.5 metadata annotations feature (specified by JSR-175) and use a programming model in which you create an annotated Java file and then use Ant tasks to compile the file into the Java source code and generate all the associated artifacts.

The Java Web Service (JWS) annotated file is the core of your Web Service. It contains the Java code that determines how your Web Service behaves. A JWS file is an ordinary Java class file that uses JDK 1.5 metadata annotations to specify the shape and characteristics of the Web Service. The JWS annotations you can use in a JWS file include the standard ones defined by the

Web Services Metadata for the Java Platform specification (JSR-181) as well as a set of WebLogic-specific ones.

A Web Service can be implemented with either a stateless session EJB that runs in the EJB container, or a Java class that runs in the Web container. If your Web Service is implemented with a stateless session EJB, the JWS Java class implements `SessionBean` and includes the standard EJB methods, such as `ejbCreate()`, `ejbActivate()`, and so on. The JWS file also uses EJBGen annotations to specify the shape and behavior of the EJB. If your Web Service is implemented with a Java class, then the JWS need not implement any particular interface.

See EJBGen Reference for more information on EJBGen annotations.

The full set of JDK 1.5 annotations you can use in a JWS file include the standard ones specified by the Java Community Process JSR-181 specification (*Web Services Metadata for the Java Platform*) and WebLogic-specific ones documented in this guide.

Programimng Basic Web Service Features Using Common JWS Annotations

This section describes how to use standard (JSR-181) and WebLogic-specific annotations in your JWS file to program basic Web Service features. In particular, it discusses the following JWS and EJBGen annotations:

- `@WebService` (standard)
- `@SOAPBinding` (standard)
- `@WLHttpTransport` (WebLogic-specific)
- `@Session` (EJBGen)
- `@WebMethod` (standard)
- `@OneWay` (standard)
- `@WebParam` (standard)
- `@WebResult` (standard)

See the *Web Services Metadata for the Java Platform* specification for detailed reference documentation about the standard JWS annotations. For reference documentation about the WebLogic-specific JWS annotations, see Appendix B, "JWS Annotation Reference."

Programming the JWS File: Java Requirements

When you program your JWS file, you must follow a set of requirements, as specified by JSR-181 specification (*Web Services Metadata for the Java Platform*). In particular, the stateless session EJB or Java class that implements the Web Service:

- Must be an outer public class, must not be final, and must not be abstract.
- Must have a default public constructor.
- Must not define a `finalize()` method.
- Must include, at a minimum, a `@WebService` JWS annotation at the class level to indicate that the JWS file implements a Web Service.
- May reference the service endpoint interface by using the `@WebService.endpointInterface` annotation. In this case, it is assumed that the service endpoint interface exists (rather than `jwsc` generating one) and you cannot specify any other JWS annotations in the JWS file other than `@WebService.endpointInterface` and `@WebService.serviceName`.
- Must specify the `@WebMethod` annotation for each method that is to be exposed as a public Web Service operation.

Programming the JWS File: Typical Steps

The following procedure describes the typical steps when programming the JWS file that implements a Web Service. See "Example of an EJB-Implemented JWS File" on page 5-5 and "Example of a Java Class-Implemented JWS File" on page 5-7 for code examples.

1. Import the standard JWS annotations used in your JWS file. The standard JWS annotations are in either the `javax.jws` or `javax.jws.soap` package. For example:

   ```
   import javax.jws.WebMethod;
   import javax.jws.WebService;
   import javax.jws.soap.SOAPBinding;
   ```

2. Import the WebLogic-specific annotations used in your JWS file. The WebLogic-specific annotations are in the `weblogic.jws` package. For example:

   ```
   import weblogic.jws.WLHttpTransport;
   ```

3. Add the standard `@WebService` JWS annotation at the class level to specify that the EJB or Java class implements a Web Service.

   For details, see "Specifying That the Bean Implements a Web Service" on page 5-8.

4. Add the standard `@SOAPBinding` JWS annotation at the class level to specify the mapping between the Web Service and the SOAP message protocol. In particular, use this annotation to specify whether the Web Service is document-literal, RPC-encoded, and so on.

   For details, see "Specifying the Mapping of the Web Service to the SOAP Message Protocol" on page 5-8.

5. Add the WebLogic-specific `@WLHttpTransport` JWS annotation at the class level to specify the context path and service URI used in the URL that invokes the Web Service.

   For details, see "Specifying the Context Path and Service URI of the Web Service" on page 5-9.

6. If your Web Service is implemented with an EJB, follow these steps:

   a. Import the required stateless session EJB classes:

   ```
   import javax.ejb.SessionBean;
   import javax.ejb.SessionContext;
   ```

   b. If you are going to use EJBGen annotations to simplify programming the EJB (recommended), then import the annotations. EJBGen annotations are in the `com.bea.wls.ejbgen.annotations` package. For example:

   ```
   import com.bea.wls.ejbgen.annotations.*;
   ```

   c. Ensure that the JWS file implements `javax.ejb.SessionBean`:

   ```
   public class SimpleBean implements SessionBean {...
   ```

   d. Add the standard EJB methods; typically there is no need to override these methods:

   ```
   public void ejbCreate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
   ```

   e. Add EJBGen annotations to your JWS file to simplify the programming of the standard EJB. At a minimum, add the `@Session` annotation.

   For details, see "Using EJBGen Annotations to Simplify Programming the Stateless Session EJB" on page 5-10.

7. For each method in the JWS file that you want to expose as a public operation, add a standard `@WebMethod` annotation. You can optionally specify that the operation takes only input parameters but does not return any value by using the standard `@OneWay` annotation.

For details, see "Specifying That a Bean Method Be Exposed as a Public Operation" on page 5-10.

8. Optionally customize the name and behavior of the input parameters of the exposed operations by adding standard `@WebParam` annotations.

   For details, see "Customizing the Mapping Between Operation Parameters and WSDL Parts" on page 5-11.

9. Optionally customize the name and behavior of the return value of the exposed operations by adding standard `@WebResult` annotations.

   For details, see "Customizing the Mapping Between the Operation Return Value and a WSDL Part" on page 5-12.

Example of an EJB-Implemented JWS File

The following sample JWS file shows how to implement a simple Web Service with a stateless session EJB.

```
package examples.ejbJWS;

import java.rmi.RemoteException;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

// Import the EJBGen annotation interfaces

import com.bea.wls.ejbgen.annotations.*;

// Import the WebLogic-specific JWS annotation interface called WLHttpTransport

import weblogic.jws.WLHttpTransport;

// EJBGen annotation that specifies that EJBGen should generate a stateless
// session EJB called Simple and its generated JAX-RPC Web service endpoint
// interface should be called "examples.ejbJWS.SimpleBeanPortType".

@Session(ejbName="Simple",
         serviceEndpoint="examples.ejbJWS.SimpleBeanPortType")
```

```
// Standard JWS annotation that specifies that the name of the Web Service is
// "myPortType", its public service name is "SimpleEjbService", and the
// targetNamespace used in the generated WSDL is "http://example.org"

@WebService(name="myPortType",
            serviceName="SimpleEjbService",
            targetNamespace="http://example.org")

// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are rpc-encoded.

@SOAPBinding(style=SOAPBinding.Style.RPC,
             use=SOAPBinding.Use.ENCODED)

// WebLogic-specific JWS annotation that specifies the port name is helloPort,
// and the context path and service URI used to build the URI of the Web
// Service is "ejbJWS/SimpleBean"

@WLHttpTransport(portName="helloPort",
                 contextPath="ejbJWS",
                 serviceUri="SimpleBean")

/**
 * This JWS file forms the basis of a stateless-session EJB implemented
 * WebLogic Web Service.
 *
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class SimpleBean implements SessionBean {

  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: echoString.

  @WebMethod()
  public String echoString(String input) throws RemoteException {
    System.out.println("echoString got " + input);
    return input;
  }

  // Standard EJB methods.  Typically there's no need to override the methods.

  public void ejbCreate() {}
  public void ejbActivate() {}
  public void ejbRemove() {}
  public void ejbPassivate() {}
```

```
  public void setSessionContext(SessionContext sc) {}
}
```

Example of a Java Class-Implemented JWS File

The following sample JWS file shows how to implement a simple Web Service using a Java class.

```
package examples.webJWS;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

// Import the WebLogic-specific JWS annotation interfaces

import weblogic.jws.WLHttpTransport;

// Standard JWS annotation that specifies that the name of the Web Service is
// "Simple" and the targetNamespace used in the generated WSDL is
// "http://example.org"

@WebService(name="Simple", targetNamespace="http://example.org")

// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are rpc-encoded.

@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)

// WebLogic-specific JWS annotation that specifies the context path and
// service URI used to build the URI of the Web Service is
// "webJWS/SimpleService"

@WLHttpTransport(contextPath="webJWS", serviceUri="SimpleService")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 * @author Copyright (c) 2004 by BEA Systems.  All rights reserved.
 */

public class SimpleImpl {

  // Required constructor

  public SimpleImpl() {}
```

```
  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: sayHello.

  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

Specifying That the Bean Implements a Web Service

Use the standard `@WebService` annotation to specify, at the class level, that the EJB or Java class of the JWS file implements a Web Service, as shown in the following code excerpt:

```
@WebService(name="funWebService",
            serviceName="SimpleEjbService",
            targetNamespace="http://example.org")

public class SimpleBean implements SessionBean {...
```

In the example, the name of the Web Service is `funWebService`, which will later map to the `wsdl:portType` element in the WSDL file generated by the `jwsc` Ant task. The service name is `SimpleEjbService`, which will map to the `wsdl:service` element in the generated WSDL file. The target namespace used in the generated WSDL is `http://example.org`.

You can also specify the following additional attributes of the `@WebService` annotation:

- `wsdlLocation`—Relative or absolute URL of a pre-defined WSDL file. If you specify this attribute, the jwsc Ant task does not generate a WSDL file, and returns an error if the JWS file is inconsistent with the port types and bindings in the WSDL file.
- `endpointInterface`—Fully qualified name of an existing service endpoint interface file. If you specify this attribute, the jwsc Ant task does not generate the interface for you, but assumes you have already created it and it is in your CLASSPATH.

None of the attributes of the `@WebService` annotation is required. See the *Web Services Metadata for the Java Platform* for the default values of each attribute.

Specifying the Mapping of the Web Service to the SOAP Message Protocol

It is assumed that you want your Web Service to be available over the SOAP 1.1 message protocol; for this reason, your JWS file should always include the standard `@SOAPBinding`

annotation, at the class level, to specify the SOAP bindings of the Web Service (such as RPC-encoded or document-literal-wrapped), as shown in the following code exceprt:

```
@SOAPBinding(style=SOAPBinding.Style.Document,
             use=SOAPBinding.Use.Literal)

public class SimpleBean implements SessionBean {...
```

In the example, the Web Service uses document-style encodings and literal message formats.

You can also use the `parameterStyle` attribute (in conjunction with the `style=SOAPBinding.Style.DOCUMENT` attribute) to specify whether the Web Service operation parameters represent the entire SOAP message body, or whether the parameters are elements wrapped inside a top-level element with the same name as the operation.

The following table lists the possible and default values for the three attributes of the `@SOAPBinding` annotation.

Table 5-1 Attributes of the @SOAPBinding Annotation

Attribute	Possible Values	Default Value
style	`SOAPBinding.Style.RPC` `SOAPBinding.Style.Document`	`SOAPBinding.Style.Document`
use	`SOAPBinding.Use.Literal` `SOAPBinding.Use.Encoded`	`SOAPBinding.Use.Literal`
parameterStyle	`SOAPBinding.ParameterStyle.Bare` `SOAPBinding.ParameterStyle.Wrapped`	`SOAPBinding.ParameterStyle.Wrapped`

Specifying the Context Path and Service URI of the Web Service

Use the Weblogic-specific `@WLHttpTransport` annotation to specify the context path and service URI sections of the URL used to invoke the Web Service over the HTTP transport, as well as the name of the port in the generated WSDL, as shown in the following code exceprt:

```
@WLHttpTransport(portName="helloPort",
                 contextPath="ejbJWS",
                 serviceUri="SimpleBean")

public class SimpleBean implements SessionBean {...
```

In the example, the name of the port in the WSDL (in particular, the `name` attribute of the `<port>` element) file generated by the `jwsc` Ant task is `helloPort`. The URL used to invoke the Web Service over HTTP includes a context path of `ejbJWS` and a service URI of `SimpleBean`, as shown in the following example:

```
http://host:port/ejbJWS/SimpleBean
```

For reference documentation on this and other WebLogic-specific annotations, see Appendix B, "JWS Annotation Reference."

Using EJBGen Annotations to Simplify Programming the Stateless Session EJB

If you decide to implement your Web Service with a stateless session EJB, you must program the EJB as usual. BEA recommends that you use EJBGen annotations to simplify this task. Using EJBGen annotations, you can automatically generate, rather than create manually, the remote and home interface classes and the deployment descriptor files for an EJB, reducing to one the number of EJB files you need to edit and maintain.

At a minimum, use the `@Session` annotation at the class level to specify the name and service endpoint of the EJB, as shown in the following code excerpt:

```
@Session(ejbName="Simple",
         serviceEndpoint="examples.ejbJWS.SimpleBeanPortType")

public class SimpleBean implements SessionBean {...
```

For more information on using EJBGen annotations, see the EJBGen Reference in the *Programming WebLogic Enterprise JavaBeans* guide.

Specifying That a Bean Method Be Exposed as a Public Operation

Use the standard `@WebMethod` annotation to specify that a method of the JWS file should be exposed as a public operation of the Web Service, as shown in the following code excerpt:

```
public class SimpleBean implements SessionBean {

  @WebMethod(operationName="funOperation")

  public String echoString(String input) throws RemoteException {
    System.out.println("echoString got " + input);
    return input;
  }
```

```
...
```

In the example, the `echoString()` method of the `SimpleBean` EJB is exposed as a public operation of the Web Service. The `operationName` attribute specifies, however, the public name of the operation in the WSDL file is `funOperation`. If you do not specify the `operationName` attribute, the public name of the operation is the name of the method itself.

You can also use the `action` attribute to specify the action of the operation. When using SOAP as a binding, the value of the `action` attribute determines the value of the `SOAPAction` header in the SOAP messages.

You can specify that an operation not return a value to the calling application by using the standard `@OneWay` annotation, as shown in the following example:

```
public class SimpleBean implements SessionBean {

  @WebMethod()
  @OneWay()

  public void ping() {
    System.out.println("ping operation");
  }

...
```

If you specify that an operation is one-way, the implementing method is required to return `void`, cannot use a Holder class as a parameter, and cannot throw any exceptions.

None of the attributes of the `@Webmethod` annotation is required. See the *Web Services Metadata for the Java Platform* for the default values of each attribute, as well as additional information about the `@WebMethod` and `@OneWay` annotations.

Customizing the Mapping Between Operation Parameters and WSDL Parts

Use the standard `@WebParam` annotation to customize the mapping between operation input parameters of the Web Service and elements of the generated WSDL file, as well as specify the behavior of the parameter, as shown in the following code excerpt:

```
public class SimpleBean implements SessionBean {

  @WebMethod()
  @WebResult(name="IntegerOutput",
             targetNamespace="http://example.org/docLiteralBare")
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/docLiteralBare")
```

```
        int input)
        throws RemoteException
    {
        System.out.println("echoInt '" + input + "' to you too!");
        return input;
    }
...
```

In the example, the name of the parameter of the `echoInt` operation in the generated WSDL is `IntegerInput`; if the `@WebParam` annotation were not present in the JWS file, the name of the parameter in the generated WSDL file would be the same as the name of the method's parameter: `input`. The `targetNamespace` attribute specifies that the XML namespace for the parameter is `http://example.org/docLiteralBare`; this attribute is relevant only when using document-style SOAP bindings where the parameter maps to an XML element.

You can also specify the following additional attributes of the `@WebParam` annotation:

- `mode`—The direction in which the parameter is flowing (IN, OUT, or INOUT). The OUT and INOUT modes may be specified only for parameter types that conform to the JAX-RPC definition of `Holder` types. OUT and INOUT modes are only supported for RPC-style encodings or for parameters that map to headers.
- `header`—Boolean attribute that, when set to `true`, specifies that the value of the parameter should be retrieved from the SOAP header, rather than the default body.

None of the attributes of the `@WebParam` annotation is required. See the *Web Services Metadata for the Java Platform* for the default value of each attribute.

Customizing the Mapping Between the Operation Return Value and a WSDL Part

Use the standard `@WebResult` annotation to customize the mapping between the Web Service operation return value and the corresponding element of the generated WSDL file, as well as specify the behavior of the return value, as shown in the following code excerpt:

```
public class SimpleBean implements SessionBean {

  @WebMethod()
  @WebResult(name="IntegerOutput",
             targetNamespace="http://example.org/docLiteralBare")
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/docLiteralBare")
      int input)
      throws RemoteException
```

```
    {
        System.out.println("echoInt '" + input + "' to you too!");
        return input;
    }
...
```

In the example, the name of the return value of the `echoInt` operation in the generated WSDL is `IntegerOutput`; if the `@WebResult` annotation were not present in the JWS file, the name of the return value in the generated WSDL file would be the hard-coded name `result`. The `targetNamespace` attribute specifies that the XML namespace for the return value is `http://example.org/docLiteralBare`; this attribute is relevant only when using document-style SOAP bindings where the return value maps to an XML element.

None of the attributes of the `@WebResult` annotation is required. See the *Web Services Metadata for the Java Platform* for the default value of each attribute.

Programming the User-Defined Java Data Type

The methods of the JWS file that are exposed as Web Service operations do not necessarily take built-in data types (such as Strings and integers) as parameters and return values, but rather, might use a Java data type that you create yourself. An example of a user-defined data type is `TradeResult`, which has two fields: a `String` stock symbol and an integer number of shares traded.

If your JWS file uses user-defined data types as parameters or return values of one or more of its methods, you must create the Java code of the data type yourself, and then import the class into your JWS file and use it appropriately. The `jwsc` Ant task will later take care of creating all the necessary data binding artifacts, such as the corresponding XML Schema represenation of the Java user-defined data type, the JAX-RPC type mapping file, and so on.

Follow these basic requirements when writing the Java class for your user-defined data type:

- Define a default constructor, which is a constructor that takes no parameters.
- Define both `getXXX()` and `setXXX()` methods for each member variable that you want to publicly expose.
- Make the data type of each exposed member variable one of the built-in data types, or another user-defined data type that consists of built-in data types.

These requirements are specified by JAX-RPC 1.1; for more detailed information and the complete list of requirements, see the JAX-RPC specification at http://java.sun.com/xml/jaxrpc/index.jsp.

The `jwsc` Ant task can generate data binding artifacts for most common XML and Java data types. For the list of supported user-defined data types, see "Supported User-Defined Data Types" on page 8-6. See "Supported Built-In Data Types" on page 8-2 for the full list of supported built-in data types.

The following example shows a simple Java user-defined data type called `BasicStruct`:

```
package examples.complex;

/**
 * Defines a simple JavaBean called BasicStruct that has integer, String,
 * and String[] properties
 */

public class BasicStruct {

  // Properties

  private int intValue;
  private String stringValue;
  private String[] stringArray;

  // Constructor

  public BasicStruct() {}

  // Getter and setter methods

  public int getIntValue() {
    return intValue;
  }

  public void setIntValue(int intValue) {
    this.intValue = intValue;
  }

  public String getStringValue() {
    return stringValue;
  }

  public void setStringValue(String stringValue) {
    this.stringValue = stringValue;
  }
```

```
    public String[] getStringArray() {
      return stringArray;
    }

    public void setStringArray(String[] stringArray) {
      this.stringArray = stringArray;
    }

    public String toString() {
      return "IntValue="+intValue+", StringValue="+stringValue;
    }

  }
```

The following snippets from a JWS file show how to import the `BasicStruct` class and use it as both a parameter and return value for one of its methods; for the full JWS file, see "Sample ComplexBean JWS File" on page 3-16:

```
package examples.complex;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

...

// Import the BasicStruct JavaBean

import examples.complex.BasicStruct;

...

@WebService(serviceName="ComplexService", name="ComplexPortType",
    targetNamespace="http://example.org")

public class ComplexBean implements SessionBean {

  @WebMethod()
  public BasicStruct echoStruct(BasicStruct struct)
      throws RemoteException
  {
    System.out.println("echoStruct called");
    return struct;
  }

...
```

```
}
```

Throwing Exceptions

When you write the error-handling Java code in methods of the JWS file, you can either throw your own user-defined exceptions or throw a `javax.xml.rpc.soap.SOAPFaultException` exception. If you throw a `SOAPFaultException`, WebLogic Server maps it to a SOAP fault and sends it to the client application that invokes the operation.

If your JWS file throws any type of Java exception other than `SOAPFaultException`, WebLogic Server tries to map it to a SOAP fault as best it can. However, if you want to control what the client application receives and send it the best possible exception information, you should explicitly throw a `SOAPFaultException` exception or one that extends the exception. See the JAX-RPC 1.1 specification at http://java.sun.com/xml/jaxrpc/index.jsp for detailed information about creating and throwing your own user-defined exceptions.

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() {...}
}
```

Use the SOAP with Attachments API for Java 1.1 (SAAJ) `javax.xml.soap.SOAPFactory.createDetail()` method to create the `Detail` object, which is a container for `DetailEntry` objects that provide detailed application-specific information about the error. You can use your own implementation of the SOAPFactory, or use BEA's, which is `com.bea.saaj.SOAPFactoryImpl`.

The following JWS file shows an example of creating and throwing a `SOAPFaultException` from within a method that implements an operation of your Web Service; the sections in bold highlight the exception code:

```
package examples.soapfaultexception;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
```

```
import javax.xml.namespace.QName;
import javax.xml.soap.Detail;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.rpc.soap.SOAPFaultException;

import weblogic.jws.WLHttpTransport;

/**
 * This JWS file implements a Java class Web Service that shows how to
 * throw SOAPFaultExceptions.  It uses the SAAJ javax.xml.soap.SOAPFactory
 * class via BEA's implementation com.bea.saaj.SOAPFactoryImpl.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

@WebService(name="Simple",
            serviceName="SimpleImpl", targetNamespace="http://example.org")
@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)
@WLHttpTransport(contextPath="runtime_faults", serviceUri="ExceptionService")

public class SimpleImpl {

  public SimpleImpl() {

    // Set the SOAPFactory implementation class
    System.setProperty("javax.xml.soap.SOAPFactory",
      "com.bea.saaj.SOAPFactoryImpl");
  }

  @WebMethod()
  public void soapFaultException() {

    Detail detail = null;

    try {
      detail = SOAPFactory.newInstance().createDetail();
    } catch (SOAPException e) {
      e.printStackTrace();
    }

    QName faultCode = null;
    String faultString = null;
    String faultActor = null;
    throw new SOAPFaultException(faultCode, faultString, faultActor, detail);
  }
}
```

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.

JWS Programming Best Practices

The following list provides some best practices when programming the JWS file:

- When you create a document-style Web Service, use the `@WebParam` JWS annotation to ensure that all input parameters for all operations of a given Web Service have a unique name.

  You create a document-style Web Service by setting the `style` attribute of the `soap.SOAPBinding` JWS annotation to `SOAPBinding.Style.Document`. Because of the nature of document-style Web Services, if you do not explicitly use the `@WebParam` annotation to specify the name of the input parameters, WebLogic Server creates one for you and run the risk of duplicating the names of the parameters across a Web Service.

- In general, create document-literal-wrapped Web Services for maximum interoperability of your Web Service with other Web Services, rather than the less-interoperable RPC-encoded.

- Use the `@WebResult` JWS annotation to explicitly set the name of the returned value of an operation, rather than always relying on the hard-coded name `result`, which is the default name of the returned value if you do not use the `@WebResult` annotation in your JWS file.

- Use EJBGen annotations to specify the characteristics and shape of the stateless session EJB that implements a Web Service. See EJBGen Reference for more information on EJBGen annotations.

- Use `SOAPFaultExceptions` in your JWS file if you want to control the exception information that is passed back to a client application when an error is encountered while invoking a the Web Service.

- Even though it is not required, BEA recommends you always specify the `portName` attribute of the WebLogic-specific `@WLHttpTransport` annotation in your JWS file. If you do not specify this attribute, the `jwsc` Ant task will generate a port name for you when generating the WSDL file, but this name might not be very user-friendly. A consequence of this is that the `getXXX()` method you use in your client applications to invoke the Web Service will not be very well-named. To ensure that your client applications use the most

user-friendly methods possible when invoking the Web Service, specify a relevant name of the Web Service port by using the `portName` attribute.

CHAPTER 6

Advanced JWS Programming

The following sections provide information about advanced JWS programming topics:

Using Reliable SOAP Messaging

The following section describes how to use reliable SOAP messaging:

Conceptual Overview of Reliable SOAP Messaging

Reliable SOAP messaging is a framework whereby an application running in one WebLogic Server instance can reliably invoke a Web Service running on another WebLogic Server instance. *Reliable* is defined as the ability to guarantee message delivery between the two Web Services.

WebLogic Web Services 9.0 conform to the WS-ReliableMessaging specification (February 4, 2004), which describes how two Web Services running on different WebLogic Server instances

can communicate reliably in the presence of failures in software components, systems, or networks. In particular, the specification describes an interoperable protocol in which a message sent from a *source endpoint* to a *destination endpoint* is guaranteed either to be delivered, according to one or more delivery assurances, or to raise an error.

A reliable WebLogic Web Service provides the following two delivery assurances:

- **ExactlyOnce**—Every message is delivered exactly once, without duplication.
- **InOrder**—Messages are delivered in the order that they were sent.

Use of WS-Policy Files for Reliable SOAP Messaging Configuration

WebLogic Web Services use WS-Policy files to enable a destination endpoint to describe and advertise its reliable messaging capabilities and requirements. The WS-Policy specification provides a general purpose model and syntax to describe and communicate the policies of a Web service.

These WS-Policy files are XML files that describe features such as the version of the WS-ReliableMessaging specification that is supported, the source endpoint's retransmission interval, the destination endpoint's acknowledgment interval, and so on.

You specify the names of the WS-Policy files that are attached to your Web Service using the `@Policy` JWS annotation in your JWS file.

See "Creating the WS-Policy File for Reliable Messaging" on page 6-9 for See Appendix C, "Reliable SOAP Messaging Policy Assertion Reference," for reference information about writing a reliable messaging policy file.

Reliable SOAP Messaging Architecture

[[More information to come.]]

Terminology

[[More information to come.]]

Using Reliable SOAP Messaging: Main Steps

Configuring reliable messaging for a WebLogic Web Service involves some standard JMS tasks, such as creating JMS servers and Store and Forward (SAF) agents, as well as Web

Service-specific tasks, such as adding additional JWS annotations to your JWS file and creating WS-Policy files that describe the reliable messaging capabilities of the reliable Web Service.

Configuration tasks must be performed on both the WebLogic Server instance in which the source endpoint (Web Service or other J2EE component that includes client code to invoke the Web Service reliably) is deployed, as well as the WebLogic Server instance in which the reliable Web Service itself is deployed.

To configure reliable messaging for a WebLogic Web Service and a client that invokes the service, follow these high-level steps. Later sections describe the steps in more detail.

Note: It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web Service and you want to update it so that its operations can be invoked reliably. It is also assumed that you use Ant build scripts to iteratively develop your Web Service and that you have a working `build.xml` file that you can update with new information. Finally, it is assumed that you have an existing Web Service or EJB on the source WebLogic Server that invokes the Web Service running on the destination server. If these assumptions are not true, see:

- Chapter 5, "Programming the JWS File"
- Chapter 4, "Iteratively Developing WebLogic Web Services"
- Chapter 7, "Invoking Web Services"

1. Configure the source WebLogic Server instance for reliable SOAP messaging.

   See "Configuring the Source WebLogic Server" on page 6-4.

2. Configure the destination WebLogic Server instance for reliable SOAP messaging.

   See "Configuring the Destination WebLogic Server" on page 6-5.

3. Create a WS-Policy file that describes the reliable messaging cabalities of the Web Service running on the destination WebLogic Server.

   See "Creating the WS-Policy File for Reliable Messaging" on page 6-9.

4. Update your JWS file, adding JWS annotations needed to specify that the operations of the Web Service can be invoked reliably.

   See "Updating the JWS File" on page 6-10.

5. Update your the client running in the source WebLogic Server instance to invoke the destination Web Service reliably.

   See "Updating the Client To Invoke a Web Service Reliably" on page 6-13.

Configuring the Source WebLogic Server

Configuring the WebLogic Server instance on which the source endpoint is deployed involves configuring JMS and store and forward (SAF) resources. The following procedure describes how to use the Administration Console to perform these tasks.

Note: For clarity, the procedure provides suggested names for the newly created JMS and SAF resources. This is so later sections of the procedure can refer to these resources by name. These are just suggestions, however; you are not required to name the resources this way.

1. Invoke the Administration Console for the domain that contains the source WebLogic Server in your browser. It is assumed in this procedure that the name of the source WebLogic Server that hostst the source endpoint is `sourceServer`.

   See "Invoking the Administration Console" on page 10-3.

2. Click the Lock & Edit button in the top left corner of the console to enable changes to the WebLogic Server configuration.

3. Create a JMS persistent store that will be used by the source WebLogic Server to store internal reliable SOAP messaging information.:

   a. In the left pane, click the Services->Persistent Stores node.

   b. In the right pane, click the New button, and click Create File Store.

      Note: If you want to use a JDBC store, you must create all the JDBC resources first.

   c. Enter `sourceFileStore` in the Name field.

   d. Choose `sourceServer` for the Target.

   e. Enter the full directory pathname where the file store is kept in the Directory field.

   f. Click Finish.

4. Create a JMS Server:

   a. In the left pane, click the Services->JMS->Servers node.

   b. In the right pane, click the New button.

   c. Enter `sourceJMSServer` in the Name field.

   d. Choose `sourceFileStore` for the Persistent Store.

   e. Click Next.

f. Choose `sourceServer` for the Target.

g. Click Finish.

5. Create a store and forward (SAF) agent:

   a. In the left pane, click Services->Store-and-Forward Agents.

   b. In the right pane, click New.

   c. Enter `sourceSAFAgent` in the Name field.

   d. Choose `sourceFileStore` for the Persistent Store.

   e. Chose Sending and Receiving for the Agent Type.

   f. Click Next.

   g. Select the `sourceServer` server.

   h. Click Finish.

6. Click the Activate Changes button in the top left corner of the console to activate your changes.

Configuring the Destination WebLogic Server

As with the source WebLogic Server, configuring the WebLogic Server instance on which the destination endpoint is deployed involves configuring JMS and store and forward (SAF) resources. In addition, you must deploy two J2EE components that are packaged in the `WL_HOME/server/lib` directory of the WebLogic Server installation and are needed by the reliable SOAP messaging feature:

- A message-driven bean (MDB) called `WsrmDestMDB.jar` that listens to a particular JMS queue.
- A Web Service used internally by the reliable messaging feature.

You only need to deploy these components once for each destination WebLogic Server.

The following procedure describes how to use the Administration Console to perform these tasks.

Note: For clarity, the procedure provides suggested names for the newly created JMS and SAF resources. This is so later sections of the procedure can refer to these resources by name. These are just suggestions, however; you are not required to name the resources this way.

The only requirement in the following procedure is the JNDI name of the JMS queue; you must name it exactly as specified because the deployed reliable messaging MDB is configured to listen at that JNDI name.

1. Invoke the Administration Console for the domain that contains the destination WebLogic Server in your browser. It is assumed in this procedure that the name of the destination WebLogic Server that hostst the destination endpoint is `destinationServer`.

   See "Invoking the Administration Console" on page 10-3.

2. Click the Lock & Edit button in the top left corner of the console to enable changes to the WebLogic Server configuration.

3. Create a JMS persistent store that will be used by the destination WebLogic Server to store internal reliable SOAP messaging information.:

   a. In the left pane, click the Services->Persistent Stores node.

   b. In the right pane, click the New button, and click Create File Store.

      Note: If you want to use a JDBC store, you must create all the JDBC resources first.

   c. Enter `destinationFileStore` in the Name field.

   d. Choose `destinationServer` for the Target.

   e. Enter the full directory pathname where the file store is kept in the Directory field.

   f. Click Finish.

4. Create a JMS Server:

   a. In the left pane, click the Services->JMS->Servers node.

   b. In the right pane, click the New button.

   c. Enter `destinationJMSServer` in the Name field.

   d. Choose `destinationFileStore` for the Persistent Store.

   e. Click Next.

   f. Choose `destinationServer` for the Target.

   g. Click Finish.

5. Create a JMS module that contains a JMS queue:

a. In the left pane, click the Services->JMS->JMS Modules node.

b. In the right pane, click New.

c. Enter `destinationJMSModule` in the Name field.

d. Enter destinationJMSModule in the Descriptor File Name field.

e. Enter the location you want to store the module file name in the Location in Domain field.

f. Click Next.

g. Select the `destinationServer` target.

h. Click Finish.

i. In the table of JMS modules, click `destinationJMSModule`.

j. Click the Configuration->Entities tab.

k. Click New.

l. Choose Queue for the Type.

m. Click Next.

n. Enter `destinationQueue` in the Name field.

o. Enter `weblogic.wsee.reliability.DestinationQueue` in the JNDI Name field.

Warning: You *must* enter this exact value in the JNDI Name field, or the reliable SOAP messaging feature will not work.

p. Click Next.

q. Select the `destinationServer` server.

r. Click Finish.

6. Create a store and forward (SAF) agent:

a. In the left pane, click Services->Store-and-Forward Agents.

b. In the right pane, click New.

c. Enter `destinationSAFAgent` in the Name field.

d. Choose `destinationFileStore` for the Persistent Store.

e. Chose Sending and Receiving for the Agent Type.

f. Click Next.

g. Select the `destinationServer` server.

h. Click Finish.

7. Deploy the special message-driven bean that is used internally by the reliable messaging feature:

   a. In the left pane, click the Deployments node.

   b. In the right pane, click the Install button.

   c. Browse to the *`WL_HOME`*`/server/lib` directory, where *`WL_HOME`* refers to the main WebLogic Server installation directory, such as `c:\bea\weblogic90`.

   d. Select the `WsrmDestMDB.jar` archive.

   e. Click Next.

   f. Select `destinationServer`.

   g. Click Next.

   h. Click Finish.

8. Deploy the special Web Service that is used internally by the reliable messaging feature:

   a. In the left pane, click the Deployments node.

   b. In the right pane, click the Install button.

   c. Browse to the *`WL_HOME`*`/server/lib` directory, where *`WL_HOME`* refers to the main WebLogic Server installation directory, such as `c:\bea\weblogic90`.

   d. Select the `WsrmAsyncService.jar` archive.

   e. Click Next.

   f. Select `destinationServer`.

   g. Click Next.

   h. Click Finish.

9. Click the Activate Changes button in the top left corner of the console to activate your changes.

Creating the WS-Policy File for Reliable Messaging

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declarations:

```
<wsp:Policy wsp:Name="NameOfFile"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2003/11/policy"
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
```

The following reliable messaging policy assertions are required:

- `<SpecVersion>`—For this release, set the `URI` attribute to `http://schemas.xmlsoap.org/ws/2004/03/rm`.
- `<SequenceCreation>`—This element has no attributes and takes no value.

The following reliable messaging policy assertions are optional; set them only if the default values are not adequate:

- `<InactivityTimeout>`—The number of milliseconds, specificed with the `Milliseconds` attribute, which defines an inactivity interval. After this amount of time, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may consider the sequence to have terminated due to inactivity. By default, sequences never timeout.
- `<AcknowledgmentInterval>`—Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand alone acknowledgement. The default value is set by the SAF agent on the destination endpoint's WebLogic Server instance.
- `<BaseRetransmissionInterval>`—Specifies the interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message. Default value is set by the SAF agent on the source endpoint's WebLogic Server instance.
- `<ExponentialBackoff>`—Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm. This element has no attributes.
- `<Expires>`—Specifies the expiration time of a sequence of messages. By default, sequences never expire.

The following example shows a simple policy file:

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

```
<wsp:Policy wsp:Name="DocLitHelloWorldPolicy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2003/11/policy"
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <wsrm:SpecVersion
      URI="http://schemas.xmlsoap.org/ws/2004/03/rm" />
  <wsrm:InactivityTimeout
      Milliseconds="150000" />
  <wsrm:AcknowledgementInterval
      Milliseconds="2000" />
  <wsrm:BaseRetransmissionInterval
      Milliseconds="500" />
  <wsrm:ExponentialBackoff />
  <wsrm:SequenceCreation />
</wsp:Policy>
```

Updating the JWS File

There are two JWS annotations you must add to your JWS file to specify that the operations can be invoked reliably:

- `@Policy`
- `@OneWay`

The following example shows how to use the `@Policy` and `@OneWay` JWS annotations, with the relevant sections shown in bold; see "@Policy" on page 6-11 and "@OneWay" on page 6-13 for additional information about using these annotations:

```
package rmexample;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.Stub;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.Oneway;
import javax.jws.soap.SOAPBinding;

import weblogic.jws.WLHttpTransport;

import weblogic.jws.Policy;
```

```
/**
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

  @WebService(name="DocLitHelloWorldPortType", serviceName="DocLitHelloWorld")
  @SOAPBinding(style=SOAPBinding.Style.DOCUMENT, use=SOAPBinding.Use.LITERAL,
               parameterStyle=SOAPBinding.ParameterStyle.BARE)
  @Policy(uri="policy:DocLitHelloWorldPolicy.xml",
          direction=Policy.Direction.inbound,
          attachToWsdl=true)
  @WLHttpTransport(portName="DocLitHelloWorld",
      contextPath="DocLitHelloWorld",
      serviceUri="DocLitHelloWorld")

  public class DocLitHelloWorld {

  public DocLitHelloWorld() {}

  @WebMethod()
  @Oneway()
  public void helloWorld(String input) {
    System.out.println(" Hello World " + input);

  }

}
```

In the example, the `DocLitHelloWorldPolicy.xml` file is attached to the Web Service at the class level, which means that the policy file is applied to all public operations of the Web Service. The policy file is applied only to the request SOAP message (as required by the reliable messaging feature) and it is attached to the WSDL file.

The `helloWorld()` method has been marked with both the `@WebMethod` and `@OneWay` JWS annotations, which means that the corresponding `helloWorld` operation can be invoked reliably.

@Policy

Use the `@Policy` annotation in your JWS file to specify that the Web Service has a WS-Policy file attached to it that contains reliable SOAP messaging assertions. You must follow these requiements when using the `@Policy` annotation for reliable messaging:

- You can specify only *one* `@Policy` JWS annotation in a JWS file. In other words, you cannot use the `@Policies` JWS annotation to specify a set of `@Policy` files that contain reliable messaging assertions.
- In addition, you can specify the `@Policy` annotation at *only* the class-level.

- Finally, because reliable SOAP messaging is applied only to the request SOAP message, you must set the `direction` attribute of the `@Policy` annotation to `Policy.Direction.inbound`.

Use the `uri` attribute to specify the location of the policy file, as described below:

- To specify that the policy file is located in the JAR or WAR archive in which the Web Service is packaged, specify the `policy:` prefix along with the name of the policy file. You must include the policy file in a specific location of the J2EE archive in which the Web Service is packaged:
  - For EJB-implemented Web Services, the policy file must be located in the `META-INF/policies` directory of the EJB JAR file.
  - For Java class-implemented Web Services, the policy file must be located in the `WEB-INF/policies` directory of the Web application WAR file.

  In the following example, the `MyPolicy.xml` file must be located in the `META-INF/policies/reliable` (or `WEB-INF/policies/reliable`, depending on the implementation of the Web Service):

  ```
  @Policy(uri="policy:reliable/MyPolicy.xml")
  ```

  You can also put your policy file in the `META-INF/policies` or `WEB-INF/policies` directory of a shared J2EE library if you want to share the policy file with multiple Web Services packaged in different J2EE archives. You specify the policy file in the same way as if it were packaged in the same archive at the Web Service. See Creating Shared J2EE Libraries and Optional Packages at http://e-docs.bea.com/wls/docs90/programming/libraries.html for information on creating shared libraries and setting up your environment so the Web Service can find the policy files.

- To specify that the policy file is published somewhere on the Web, use the `http:` prefix along with the URL, as shown in the following example:

  ```
  @Policy(uri="http://someSite.com/policies/mypolicy.xml")
  ```

You can also set the `attachToWsd` attribute of the `@Policy` annotation to specify whether the policy file should be attached to the WSDL file that describes the public contract of the Web Service. Typically you want to publicly publish the policy so that client applications know the reliable SOAP messaging capabilities of the Web Service. For this reason, the default value of this attribute is `true`.

@OneWay

You *must* specify the `@OneWay` JWS annotation for any operation in a JWS file that you want a client to invoke reliably. The annotation has no attributes.

Updating the Client To Invoke a Web Service Reliably

[[More information to come.]]

Creating and Using SOAP Message Handlers

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 access 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.

The following table lists the standard JWS annotations that you can use in your JWS file to specify that a Web Service has a handler chain configured; later sections discuss how to use the annotations in more detail. For additional information, see the *Web Services MetaData for the Java Platform* (JSR-181) specification at http://www.jcp.org/en/jsr/detail?id=181.

Table 6-1 JWS Annotations Used To Configure SOAP Message Handler Chains

JWS Annotation	Description
`javax.jws.HandlerChain`	Associates the Web Service with an externally defined handler chain. Use this annotation (rather than `@SOAPMessageHandlers`) when multiple Web Services need to share the same handler configuration, or if the handler chain consists of handlers for multiple transports.
`javax.jws.soap.SOAPMessageHandlers`	Specifies a list of SOAP handlers that run before and after the invocation of each Web Service operation. Use this annotation (rather than `@HanderChain`) if embedding handler configuration information in the JWS file itself is preferred, rather than having an external configuration file. The `@SOAPMessageHandler` annotation is an array of `@SOAPMessageHandlers`. The handlers are executed in the order they are listed in this array.
`javax.jws.soap.SOAPMessageHandler`	Specifies a single SOAP message handler in the `@SOAPMessageHandlers` array. T

The following table describes the main classes and interfaces of the `javax.xml.rpc.handler` API, some of which you use when creating the handler itself. These APIs are discussed in detail

in a later section. For additional information about these APIs, see the JAX-RPC 1.1 specification at http://java.sun.com/xml/jaxrpc/index.jsp.

Table 6-2 JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
`Handler`	Main interface that is implemented when creating a handler. Contains methods to handle the SOAP request, response, and faults.
`GenericHandler`	Abstract class that implements the `Handler` interface. User should extend this class when creating a handler, rather than implement `Handler` directory. The `GenericHandler` class is a convenience abstract class that makes writing Handlers easy. This class provides default implementations of the lifecycle methods init and destroy and also different handle methods. A Handler developer should only override methods that it needs to specialize as part of the derived Handler implementation class.
`HandlerChain`	Interface that represents a list of `Handlers`. An implementation class for the `HandlerChain` interface abstracts the policy and mechanism for the invocation of the registered handlers.
`HandlerRegistry`	Interface that provides support for the programmatic configuration of handlers in a HandlerRegistry.
`HandlerInfo`	Class that contains information about the handler in a handler chain. A `HandlerInfo` instance is passed in the `Handler.init` method to initialize a `Handler` instance.
`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.

Main Steps

The following procedure describes the high-level steps to add SOAP message handlers to your Web Service.

It is assumed that you have already created a basic JWS file that implements a Web Service and that you want to update the Web Service by adding SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes targets for running the `jwsc` Ant task and compiling the Java code. For more information, see Chapter 4, "Iteratively Developing WebLogic Web Services," and Chapter 5, "Programming the JWS File."

1. Design the handlers and handler chains.

   See "Designing the SOAP Message Handlers and Handler Chains" on page 6-16.

2. For each handler in the handler chain, create a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class.

   See "Creating the GenericHandler Class" on page 6-18..

3. Update your JWS file, adding annotations to configure the SOAP message handlers.

   See "Configuring Handlers in the JWS File" on page 6-27.

4. If you are using the `@HandlerChain` standard annotation in your JWS file, create the handler chain configuration file.

   See "Creating the Handler Chain Configuration File" on page 6-30.

5. Compile all handler classes in the handler chain and rebuild your Web Service.

   See "Compiling And Rebuilding the Web Service" on page 6-32.

Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers and handler chains, you must decide:

- The number of handlers needed to perform all the work
- The sequence of execution

Each handler in a handler chain has one method for handling the request SOAP message and another method for handling the response SOAP message. An ordered group of handlers is

referred to as a *handler chain*. You specify that a Web Service has a handler chain attached to it with one of two JWS annotations: `@HandlerChain` or `@SOAPMessageHandler`. When to use which is discussed in a later section.

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 by the JWS annotation. 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 back-end component that implements the Web Service (EJB or Java class), passing it the final SOAP message request.
3. When the back-end component has finished executing, the `handleResponse()` methods of the handlers in the handler chain are executed in the *reverse* order specified in by the JWS annotation. 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.

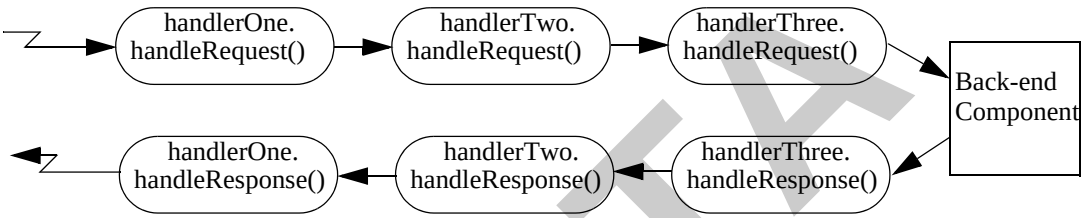
For example, assume that you are going to use the `@HandlerChain` JWS annotation in your JWS file to specify an external configuration file, and the configration file defines a handler chain called `SimpleChain` that contains three handlers, as shown in the following sample:

```
<handler-config
      xmlns="http://www.bea.com/2003/03/wls/handler/config/"
      xmlns:soap1="http://HandlerInfo.org/Server1"
      xmlns:soap2="http://HandlerInfo.org/Server2">

   <handler-chain name="SimpleChain">
     <handler>
       <handler-name>handlerOne</handler-name>
       <handler-class>examples.handlerOne</handler-class>
     </handler>

     <handler>
       <handler-name>handlerTwo</handler-name>
       <handler-class>examples.handlerTwo</handler-class>
     </handler>
```

```
      <handler>
        <handler-name>handlerThree</handler-name>
        <handler-class>examples.handlerThree</handler-class>
      </handler>
    </handler-chain>
</handler-config>
```

The following graphic shows the order in which WebLogic Server executes the `handleRequest()` and `handleResponse()` methods of each handler:

Figure 6-1 Order of Execution of Handler Methods



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.

Creating the GenericHandler Class

Your SOAP message handler class should extend the `javax.rpc.xml.handler.GenericHandler` abstract class, which itself implements the `javax.rpc.xml.handler.Handler` interface.

The `GenericHandler` class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the lifecycle methods `init()` and `destroy()` and the various `handleXXX()` methods of the `Handler` interface. When you write your handler class, only override those methods that you need to customize as part of your `Handler` implementation class.

In particualr, the `Handler` interface contains the following methods that you can implement in your handler class that extends `GenericHandler`:

- `init()`

  See "Implementing the Handler.init() Method" on page 6-21.

- `destroy()`

  See "Implementing the Handler.destroy() Method" on page 6-21.

- `getHeaders()`

  See "Implementing the Handler.getHeaders() Method" on page 6-22.

- `handleRequest()`

  See "Implementing the Handler.handleRequest() Method" on page 6-22.

- `handleResponse()`

  See "Implementing the Handler.handleResponse() Method" on page 6-23.

- `handleFault()`

  See "Implementing the Handler.handleFault() Method" on page 6-24.

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the `javax.xml.soap.SOAPMessage` abstract class, which is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification For details, see "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 6-25.

The following example demonstrates a simple SOAP message handler that prints out the SOAP request and response messages to the WebLogic Server log file:

```
package examples.webservices.SOAPHandlers.globalHandler;

import javax.xml.namespace.QName;

import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;

import javax.xml.rpc.JAXRPCException;

import weblogic.logging.NonCatalogLogger;

/**
 * This class implements a handler in the handler chain, used to access the SOAP
 * request and response message.
```

```
 * <p>
 * This class extends the <code>javax.xml.rpc.handler.GenericHandler</code>
 * abstract classs and simply prints the SOAP request and response messages to
 * the server log file before the messages are processed by the backend
 * Java class that implements the Web Service itself.
 */

public class ServerHandler1 extends GenericHandler {

  private NonCatalogLogger log;

  private HandlerInfo handlerInfo;

  /**
   *  Initializes the instance of the handler.  Creates a nonCatalogLogger to
   *  log messages to.
   */
  public void init(HandlerInfo hi) {

    log = new NonCatalogLogger("WebService-LogHandler");
    handlerInfo = hi;

  }

  /**
   * Specifies that the SOAP request message be logged to a log file before the
   * message is sent to the Java class that implements the Web Service.
   */
  public boolean handleRequest(MessageContext context) {

   SOAPMessageContext messageContext = (SOAPMessageContext) context;
   System.out.println("** Request: "+messageContext.getMessage().toString());
   log.info(messageContext.getMessage().toString());
   return true;

  }

  /**
   * Specifies that the SOAP response message be logged to a log file before the
   * message is sent back to the client application that invoked the Web
   * service.
   */
  public boolean handleResponse(MessageContext context) {

    SOAPMessageContext messageContext = (SOAPMessageContext) context

    System.out.println("** Response: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;

  }
```

```
  /**
   * Specifies that a message be logged to the log file if a SOAP fault is
   * thrown by the Handler instance.
   */
  public boolean handleFault(MessageContext context) {

   SOAPMessageContext messageContext = (SOAPMessageContext) context;

   System.out.println("** Fault: "+messageContext.getMessage().toString());
   log.info(messageContext.getMessage().toString());
       return true;

  }

  public QName[] getHeaders() {

    return handlerInfo.getHeaders();
  }

}
```

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,. Use the `HandlerInfo.getHandlerConfig()` method to get the parameters; the method returns a `java.util.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 that can be 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,SOAPFaultException  {}
```

Implement this method to perform such tasks as decrypting 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 "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 6-25.

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 either the next `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the next `@SOAPMessageHandler` in the array specified by the `@SOAPMessageHandlers` annotation. If there are no more handlers in the chain, the method either invokes the back-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 back-end 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 back-end 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 perform such tasks as encrypting 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 "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 6-25.

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 either the preceding `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the preceding `@SOAPMessageHandler` in the array specified by the `@SOAPMessageHandlers` annotation. (Remember that responses on the handler chain execute in the *reverse* order that they are specified in the JWS file. See "Designing the SOAP Message Handlers and Handler Chains" on page 6-16 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 back-end 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 "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 6-25.

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`.

Directly Manipulating the SOAP Request and Response Message Using SAAJ

The `javax.xml.soap.SOAPMessage` abstract class is part of the SOAP With Attachments API for Java 1.1 (SAAJ) 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 SAAJ Javadocs for the full description of the `SOAPMessage` class. For more information on SAAJ, go to http://java.sun.com/xml/saaj/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 `javax.xml.soap.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.

Warning: If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see "Manipulating Image Attachments in a SOAP Message Handler" on page 6-26 for important information.

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`.

Manipulating Image Attachments in a SOAP Message Handler

It is assumed in this section that you are creating a SOAP message handler that accesses a `java.awt.Image` attachment and that the `Image` has been sent from a client application that uses the client JAX-RPC stubs generated by the `clientgen` Ant task.

In the client code generated by the `clientgen` Ant task, a `java.awt.Image` attachment is sent to the invoked WebLogic Web Service with a MIME type of `text/xml` rather than `image/gif`,

and the Image is serialized into a stream of integers that represents the image. In particular, the client code serializes the `Image` using the following format:

- `int width`
- `int height`
- `int[] pixels`

This means that, in your SOAP message handler that manipulates the received Image attachment, you must deserialize this stream of data to then re-create the original Image.

Configuring Handlers in the JWS File

There are two standard annotations you can use in your JWS file to configure a handler chain for a Web Service: `@HandlerChain` and `@SOAPMessageHandlers`.

@HandlerChain

When you use the `@HandlerChain` annotation, you use the `file` attribute to specify an external file that contains the configuration of the handler chain you want to associate with the Web Service. The configuration includes the list of handlers in the chain, the order in which they execute, the initialization parameters, and so on.

Use the `@HandlerChain` annotation, rather than the `@SOAPMessageHandlers` annotation, in your JWS file if:

- You want multiple Web Services to share the same configuration.
- Your handler chain includes handlers for multiple transports.
- You want to be able to change the handler chain configuration for a Web Service without recompiling the JWS file that implements it.

The following JWS file shows an example of using the `@HandlerChain` annotation; the relevant Java code is shown in bold:

```
package examples.webservices.SOAPHandlers.globahHandler;

import java.io.Serializable;
import java.rmi.RemoteException;

// Import the standard JWS annotation interfaces, including HandlerChain
import javax.jws.HandlerChain;
import javax.jws.WebService;
import javax.jws.WebMethod;

import weblogic.jws.WLHttpTransport;
```

```
@WebService(name="HandlerChainWS", serviceName="HandlerChain")

// Standard JWS annotation that specifies that the handler chain called
// "SimpleChain", configured in the HandlerConfig.xml file, should fire
// each time an operation of the Web Service is invoked.

@HandlerChain(file="HandlerConfig.xml", name="SimpleChain")

@WLHttpTransport(contextPath="HandlerChain", serviceUri="HandlerChain")

public class HandlerChainBean  {

   public HandlerChainBean() {}

   @WebMethod()
   public String sayHello(String input) throws RemoteException {
     weblogic.utils.Debug.say( "in backend component. input:" +input );
     return "'" + input + "' to you too!";
   }
}
```

Before you use the `@HandlerChain` annotation, you must import it into your JWS file, as shown in the preceding example.

Use the `file` attribute of the `@HandlerChain` annotation to specify the name of the external file that contains configuration information for the handler chain. The value of this attribute is a URL, which may be relative or absolute. Relative URLs are relative to the location of the JWS file at the time you run the `jwsc` Ant task to compile the file.

Use the `name` attribute to specify the name of the handler chain in the configuration file that you want to associate with the Web Service. The value of this attribute corresponds to the `name` attribute of the `<handler-chain>` element in the configuration file.

Warning: It is an error to specify more than one `@HandlerChain` annotation in a single JWS file. It is also an error to combine the `@HandlerChain` annotation with the `@SOAPMessageHandlers` annotation.

For details about creating the external configuration file, see "Creating the Handler Chain Configuration File" on page 6-30.

For additional detailed information about the standard JWS annotations discussed in this section, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

@SOAPMessageHandlers

When you use the `@SOAPMessageHandlers` annotation, you specify, within the JWS file itself, an array of SOAP message handlers (specified with the `@SOAPMessageHandler` annotation) that

execute before and after the operations of a Web Service. The `@SOAPMessageHandler` annotation includes attributes to specify the class name of the handler, the initialization parameters, list of SOAP headers processed by the handler, and so on. Because you specify the list of handlers within the JWS file itself, the configuration of the handler chain is embedded within the Web Service.

Use the `@SOAPMessageHandlers` annotation if:

- You prefer to embed the configuration of the handler chain inside the Web Service itself, rather than specify the configuration in an external file.
- Your handler chain includes only SOAP handlers and none for any other transport.
- You prefer to recompile the JWS file each time you change the handler chain configuration.

The following JWS file shows a simple example of using the `@SOAPMessageHandlers` annotation; the relevant Java code is shown in bold:

```
package examples.webservices.SOAPHandlers.simple;

import java.io.Serializable;
import java.rmi.RemoteException;

// Import the standard JWS annotation interfaces, including
// SOAPMessageHandler/s

import javax.jws.soap.SOAPMessageHandlers;
import javax.jws.soap.SOAPMessageHandler;
import javax.jws.WebService;
import javax.jws.WebMethod;

import weblogic.jws.WLHttpTransport;

@WebService(name="SimpleChainWS", serviceName="SimpleChain")

// Standard JWS annotation that specifies a list of SOAP message handlers
// that exeucte before and after an invocation of all operations in the
// Web Serice.

@SOAPMessageHandlers ( {
 @SOAPMessageHandler (
  className="examples.webservices.SOAPHandlers.simple.ServerHandler1"),
 @SOAPMessageHandler (
  className="examples.webservices.SOAPHandlers.simple.ServerHandler2")
       } )

@WLHttpTransport(contextPath="SimpleChain", serviceUri="SimpleChain")

public class SimpleSOAPMessage  {
   @WebMethod()
```

```
    public String sayHello(String input) throws RemoteException {
      weblogic.utils.Debug.say( "in backend component. input:" +input );
      return "'" + input + "' to you too!";
    }
}
```

Before you use the `@SOAPMessageHandlers` and `@SOAPMessageHandler` annotations, you must import them into your JWS file, as shown in the preceding example. Note that these annotations are in the `javax.jws.soap` package.

The order in which you list the handlers (using the `@SOAPMessageHandler` annotation) in the `@SOAPMessageHandlers` array specifies the order in which the handlers execute: in forward order before the operation, and in reverse order after the operation. The preceding example configures two handlers in the handler chain, whose class names are `examples.webservices.SOAPHandlers.simple.ServerHandler1` and `examples.webservices.SOAPHandlers.simple.ServerHandler2`.

Use the `initParams` attribute of `@SOAPMessageHandler` to specify an array of initialization parameters expected by a particular handler. Use the `@InitParam` standard JWS annotation to specify the name/value pairs, as shown in the following example:

```
@SOAPMessageHandler(
  className = "examples.webservices.SOAPHandlers.simple.ServerHandler1",
   initParams = { @InitParam(name="logCategory", value="MyService")}
)
```

The `@SOAPMessageHandler` annotation also includes the `roles` attribute for listing the SOAP roles implemented by the handler, and the `headers` attribute for listing the SOAP headers processed by the handler.

Warning: It is an error to combine the `@SOAPMessageHandlers` annotation with the `@HandlerChain` annotation.

For additional detailed information about the standard JWS annotations discussed in this section, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

Creating the Handler Chain Configuration File

If you decide to use the `@HandlerChain` annotation in your JWS file to associate a handler chain with a Web Service, you must create an external configuration file that specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on.

Because this file is external to the JWS file, you can configure multiple Web Services to use this single configuration file to standardize the handler configuration file for all Web Services in your enterprise. Additionally, you can change the configuration of the handler chains without needing to recompile all your Web Services. Finally, if you include handlers in your handler chain that use a non-SOAP transport, then you are required to use the `@HandlerChain` annotation rather than the `@SOAPMessageHandler` annotation.

The configuration file uses XML to list one or more hander chains, as shown in the following simple example:

```
<handler-config xmlns="http://www.bea.com/2003/03/wls/handler/config/"
   xmlns:soap1="http://HandlerInfo.org/Server1"
   xmlns:soap2="http://HandlerInfo.org/Server2">

   <handler-chain name="SimpleChain">
     <handler>
       <handler-name>handler1</handler-name>
       <handler-class>
           examples.webservices.SOAPHandlers.globalHandler.ServerHandler1
       </handler-class>
     </handler>

     <handler>
       <handler-name>handler2</handler-name>
       <handler-class>
           examples.webservices.SOAPHandlers.globalHandler.ServerHandler2
       </handler-class>
     </handler>

   </handler-chain>
</handler-config>
```

In the example, the handler chain called `SimpleChain` contains two handlers: `handler1` and `handler2`, implemented with the class names specified with the `<handler-class>` element. The two handlers execute in forward order before the relevant Web Service operation executes, and in reverse order after the operation executes.

Use the `<init-param>`, `<soap-role>`, and `<soap-header>` child elements of the `<handler>` element to specify the handler initialization parameters, SOAP roles implemented by the handler, and SOAP headers processed by the handler, respectively.

For the XML Schema that defines the external configuration file, additional information about creating it, and additional examples, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

Compiling And Rebuilding the Web Service

It is assumed in this section that you have a working `build.xml` Ant file that compiles and builds your Web Service, and you want to update the build file to include handler chain. See Chapter 4, "Iteratively Developing WebLogic Web Services," for information on creating this `build.xml` file.

Follow these guidelines to update your development environment to include message handler compilation and building:

- Store the Java source for the class that extends `GenericHandler` in a location where the current `javac` task can find it, or update the `javac` task to find the location.

  For example, if you configure the handler chain for a Web Service using the `@SOAPMessageHandlers` annotation, consider storing the handler source code in the same location as the JWS file that implements the Web Service. If you use the `@HandlerChain` annotation, however, consider storing the source code in a common directory so that the same class file can be used by multiple Web Services, then update the `javac` command (or add a new one) to find the source code.

  The destination directory of the compiled handler classes is the standard location of classes in archive files: `WEB-INF/classes` of a WAR file (for Java class implemented Web Services) or the root directory of an EJB JAR file (for EJB implemented Web Services)

- After you have updated the JWS file with either the `@HandlerChain` or `@SOAPMessageHandlers` annotation, you must rerun the jwsc Ant task to recompile the JWS file and generate a new Web Service. This is true anytime you make a change to an annotation in the JWS file.

  If you used the `@HandlerChain` annotation in your JWS file, reran the `jwsc` Ant task to regenerate the Web Service, and subsequently changed just the external configuration file, you do not need to rerun `jwsc` for the change to take affect.

- You deploy and invoke a Web Service that has a handler chain associated with it in the same way you deploy and invoke one that has no handler chain. The only difference is that when you invoke any operation of the Web Service, the handlers in the handler chain are executed by the Web Services runtime both before and after the operation invoke.

CHAPTER 7

Invoking Web Services

The following sections provide information about invoking Web Services:

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 .NET, and so on.

Note: This chapter uses the term *client application* to refer to both a stand-alone client that uses the WebLogic client classes to invoke a Web Service hosted on both WebLogic and non-WebLogic Servers, and a client that runs inside of a J2EE component deployed to WebLogic Server, such as an EJB.

The sections that follow describe how to use BEA's implementation of the JAX-RPC specification (Version 1.1) to invoke a Web Service from a Java client application. You can use this implementation to invoke Web Services running on any application server, both WebLogic and non-WebLogic. In addition, you can create a standalone client application or one that runs as part of a WebLogic Server.

JAX-RPC

The Java API for XML based RPC (JAX-RPC) is a Sun Microsystems specification that defines the APIs used to invoke a Web Service.

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

Table 7-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Service	Main client interface.
ServiceFactory	Factory class for creating `Service` instances.
Stub	Base class of the client proxy used to invoke the operations 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 implements the JAX-RPC 1.1 specification.

For detailed information on JAX-RPC, see 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.

The clientgen Ant Task

The `clientgen` WebLogic Web Services Ant task generates, from an existing WSDL file, the client artifaces that client applications use to invoke both WebLogic and non-WebLogic Web Services. These artifacts include:

- The Java source code for the JAX-RPC `Stub` and `Service` interface implementations for the particular Web Service you want to invoke.
- The Java source code for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

For additional information about the clientgen Ant task, such as all the available attributes, see Appendix A, "Ant Task Reference."

Examples of Clients That Invoke Web Services

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

For detailed instructions on how to build and run the examples, open the *WL_HOME*/samples/server/docs/index.html Web page in your browser and click the WebLogic Server Examples->Examples->API->Web Services node.

Invoking a Web Service from a Stand-Alone Client: Main Steps

It is assumed in the following procedure that you use Ant in your development environment to build your client application, compile Java files, and so on, and that you have an existing build.xml file that you want to update with Web Services client tasks.

For general information about using Ant in your development environment, see "Creating the Basic Ant build.xml File" on page 4-4. For a full example of a build.xml file used in this section, see "Sample Ant Build File" on page 7-8.

To create a Java stand-alone client application that invokes a Web Service, follow these steps:

1. Open a command shell and set your environment.

   On Windows NT, execute the setDomainEnv.cmd command, located in your domain directory. The default location of WebLogic Server domains is *BEA_HOME*\user_projects\domains*domainName*, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

   On UNIX, execute the setDomainEnv.sh command, located in your domain directory. The default location of WebLogic Server domains is *BEA_HOME*/user_projects/domains/*domainName*, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

2. Update your build.xml file to execute the clientgen Ant task to generate the needed client-side artifacts to invoke a Web Service.

   See "Using the clientgen Ant Task To Generate Client Artifacts" on page 7-4.

3. Get information about the Web Service, such as the signature of its operations and the name of the ports.

See "Getting Information About a Web Service" on page 7-5.

4. Write the client application Java code that includes an invoke of the Web Service operation.

   See "Writing the Java Client Application Code" on page 7-6.

5. Compile and run your Java client application.

   See "Compiling and Running the Client Application" on page 7-7.

Using the clientgen Ant Task To Generate Client Artifacts

Update your `build.xml` file, adding a call to the `clientgen` Ant task, as shown in the following example:

```
<taskdef name="clientgen"
  classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

<target name="build-client">
  <mkdir dir="clientclasses"/>

  <clientgen
    wsdl="http://localhost:7001/example/Complex?WSDL"
    destDir="clientclasses"
    packageName="examples.simpleClient"/>

</target>
```

Before you are able to execute the `clientgen` WebLogic Web Service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task.

You must include the `wsdl`, `destDir`, and `packageName` attributes of the `clientgen` Ant task to specify the WSDL file from which you want to create client-side artifacts, the directory into which these artifacts should be generated, and the package name that all generated Java files should be in. The preceding example first creates a `clientclasses` directory, under the current directory, into which `clientgen` generates all artifacts.

If the WSDL file specifies that user-defined data types are used as input parameters or return values of Web Service operations, `clientgen` automatically generates a JavaBean class which is the Java representation of the XML Schema data type defined in the WSDL. The JavaBean classes are generated into the `destDir` directory.

See "Sample Ant Build File" on page 7-8 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`.

To execute the `clientgen` Ant task, along with the other supporting Ant tasks, specify the `build-client` target at the command line:

```
prompt> ant build-client
```

See the `clientclasses` directory to view the files and artifacts generated by the `clientgen` Ant task.

Getting Information About a Web Service

You need to know the name of the Web Service and the signature of its operations before you write your Java client application code to invoke an operation. There are a variety of ways to find this information.

If you are invoking a WebLogic Web Service, you can use its Home Page to get the full signature of each operation. For details, see "Invoking the WSDL and Home Page of the Web Service" on page 4-11.

Another way to get the signature of a Web Service operation is to use the `clientgen` Ant task to generate the Web Service-specific JAX-RPC stubs and look at the generated `*.java` files:

- The *`ServiceName`*`.java` source file contains the methods for getting the Web Service port, where *`ServiceName`* refers to the name of the Web Service. If you are invoking a WebLogic Web Service you implemented with a JWS file, the name is specified using the `serviceName` attribute of the `@WebService` JWS annotation.
- The *`PortType`*`.java` file contains the method signatures that correspond to the public operations of the Web Service, where *`PortType`* refers to the port type of the Web Service. If you are invoking a WebLogic Web Service you implemented with a JWS file, the port type is specified using the `name` attribute of the `@WebService` JWS annotation.

Finally, you can examine the actual 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>
```

Writing the Java Client Application Code

The following code shows an example of invoking a Web Service operation.

The client application takes a single argument: the WSDL of the Web Service. The application then uses standard JAX-RPC API code for invoking an operation of the Web Service using the Web Service-specific implemenation of the `Service` interface, generated by `clientgen`.

The example also shows how to invoke an operation that has a user-defined data type (examples.complex.BasicStruct) as an input parameter and return value. The the `clientgen` Ant task automatically generates the Java code for this user-defined data type for you.


```
package examples.simpleClient;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;

import examples.complex.BasicStruct;

/**
 * This is a simple standalone client application that invokes the
 * the <code>echoStruct</code>  operation of the ComplexService Web service.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class Main {

  public static void main(String[] args)
      throws ServiceException, RemoteException{

    ComplexService service = new ComplexService_Impl (args[0]);
    ComplexPortType port = service.getComplexPortTypeSoapPort();

    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
    in.setStringValue("Hello Struct");
```

```
    BasicStruct result = port.echoStruct(in);
    System.out.println("echoStruct called. result: " + result.getIntValue()+ ",
" + result.getStringValue());
  }
}
```

In the preceding example:

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

  ```
  ComplexService service = new ComplexService_Impl (args[0]);
  ComplexPortType port = service.getComplexPortTypeSoapPort();
  ```

  The `ComplexService_Impl` stub factory implements the JAX-RPC `Service` interface. The constructor of `ComplexService_Impl` creates a stub based on the provided WSDL URI (`args[0]`). The `getComplexPortTypeSoapPort()` method is used to return an instance of the `ComplexPortType` stub implementation.

- The following code shows how to invoke the `echoStruct()` operation of the `ComplexService` Web Service:

  ```
  BasicStruct result = port.echoStruct(in);
  ```

  The `echoStruct()` operation returns the user-defined data type called `BasicStruct`..

Compiling and Running the Client Application

Add `javac` tasks to the `build-client` target in the `build.xml` file to compile all the Java files (both of your client application and those generated by clientgen) into class files, as shown in the following example

```
<target name="build-client">
    <mkdir dir="clientclasses"/>

    <clientgen
      wsdl="http://localhost:7001/example/Complex?WSDL"
      destDir="clientclasses"
      packageName="examples.simpleClient"/>

    <javac
      srcdir="."
      destdir="clientclasses"
      includes="**/*.java"/>

    <javac
      srcdir="clientclasses"
```

```
        destdir="clientclasses"
        includes="examples/simpleClient/**/*.java" />
    </target>
```

In the example, the first `javac` task compiles the Java files in the current directory (with the assumption that the client application file is in the same directory as the `build.xml` file) and the second `javac` task compiles the Java files in the `clientclasses` directory that were generated by `clientgen`.

To run the client application, add a `run` target to the `build.xml` that includes a call to to the `java` task, as shown below:

```
<path id="client.class.path">
    <pathelement path="clientclasses"/>
    <pathelement path="${java.class.path}"/>
</path>

<target name="run" >
    <java fork="true" classname="examples.simpleClient.Main"
      failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg line="http://localhost:7001/example/Complex?WSDL" />
    </java>

</target>
```

The `path` task adds the `clientclasses` directory to the CLASSPATH. The `run` target invokes the `Main` application, passing it the WSDL of the deployed Web Service as its single argument.

Rerun the `build-client` target to regenerate the artifacts and recompile into classes, then execute the `run` target to invoke the `echoStruct` operation:

```
prompt> ant build-client run
```

You can use the `build-client` and `run` targets in the `build.xml` file to iteratively update, rebuild, and run the Java client application as part of your development process.

Sample Ant Build File

```
<project default="all">

  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

  <path id="client.class.path">
    <pathelement path="clientclasses"/>
```

```
    <pathelement path="${java.class.path}"/>
  </path>

  <target name="all" depends="clean,build-client,run" />

  <target name="clean">
    <delete dir="clientclasses" />
  </target>

  <target name="build-client">
    <mkdir dir="clientclasses"/>

    <clientgen
      wsdl="http://localhost:7001/example/Complex?WSDL"
      destDir="clientclasses"
      packageName="examples.simpleClient"/>

    <javac
      srcdir="." destdir="clientclasses"
      includes="**/*.java"/>

    <javac
      srcdir="clientclasses" destdir="clientclasses"
      includes="examples/simpleClient/**/*.java" />

  </target>

  <target name="run" >
    <java fork="true" classname="examples.simpleClient.Main"
      failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg line="http://localhost:7001/example/Complex?WSDL" />
    </java>

  </target>

</project>
```

CHAPTER 8

Data Types and Data Binding

The following sections provide information about supported data types (both built-in and user-defined) and data binding:

Overview of Data Types and Data Binding

As in previous releases, WebLogic Web Services support a full set of built-in XML Schema, Java, and SOAP types, as specified by the JAX-RPC 1.1 specification, that you can use in your Web Service operations without performing any additional programming steps. Built-in data types are those such as `integer`, `string`, and `time`.

Additionally, you can use a variety of user-defined XML and Java data types, including XMLBeans, as input parameters and return values of your Web Service. User-defined data types are those that you create from XML Schema or Java building blocks, such as `<xsd:complexType>` or JavaBeans. The WebLogic Web Services Ant tasks, such as `jwsc` and `clientgen`, automatically generate the data binding artifacts needed to convert the user-defined data types between their XML and Java representations. The XML representation is used in the SOAP request and response messages, and the Java represenation is used in the Java class or stateless session EJB that implements the Web Services. The conversion of data between its XML and Java represenations is called *data binding*.

Supported 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 back-end 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 user-defined data types, then you must create the data binding artifacts that convert the data between XML and Java.WebLogic Server includes the `jwsc` and `wsdl2service` Ant tasks that can generate the data binding artifacts for most user-defined data types. See "Supported User-Defined Data Types" on page 8-6 for a list of supported XML and Java data types.

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

The following table lists the supported XML Schema data types (target namespace `http://www.w3.org/2001/XMLSchema`) and their corresponding Java data types.

For a list of the supported user-defined XML data types, see "Java-to-XML Mapping for Built-In Data Types" on page 8-4.

Table 8-1 Mapping XML Schema Data Types to Java 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

Table 8-1 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
decimal	java.math.BigDecimal
string	java.lang.String
dateTime	java.util.Calendar
base64Binary	byte[]
hexBinary	byte[]
duration	java.lang.String
time	java.util.Calendar
date	java.util.Calendar
gYearMonth	java.lang.String
gYear	java.lang.String
gMonthDay	java.lang.String
gDay	java.lang.String
gMonth	java.lang.String
anyURI	java.lang.String
NOTATION	java.lang.String
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[]

Table 8-1 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
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 user-defined Java data types, see "Supported Java User-Defined Data Types" on page 8-8.

Table 8-2 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Schema 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
java.util.Calendar	dateTime
java.util.Date	dateTime

Table 8-2 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Schema Data Type
byte[]	base64Binary
javax.xml.namespace.QName	Qname
java.net.URI	anyURI

Supported User-Defined Data Types

The tables in the following sections list the user-defined XML and Java data types for which the `jwsc` and `wsdl2service` Ant tasks can generate data binding artifacts, such as the corresponding Java or XML representation, the JAX-RPC type mapping file, 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 "Supported Built-In Data Types" on page 8-2, then you must create the user-defined data type artifacts manually.

Supported XML User-Defined Data Types

The following table lists the XML Schema data types supported by the `jwsc` and `wsdl2service` Ant tasks and their equivalent Java data type or mapping mechanism.

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

Table 8-3 Supported User-Defined 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

Table 8-3 Supported User-Defined XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
Derivation of new simple types by restriction of an existing simple type.	Equivalent Java data type of simple type.
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 optionally 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 `_value` whose type is mapped from the simple type according to the rules in this section.
`<xsd:anyType>`	java.lang.Object.
`<xsd:union>`	Common parent type of union members.
`<xsi: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	Mapped using Java inheritance.
Abstract types	Abstract Java data type.

Supported Java User-Defined Data Types

The following table lists the Java user-defined data types supported by the the `jwsc` and `wsdl2service` Ant tasks and their equivalent XML Schema data type.

Table 8-4 Supported User-Defined Java Data Types

Java Data Type	Equivalent XML Schema Data Type
JavaBean whose properties are any supported data type.	`<xsd:complexType>` whose content model is a `<xsd:sequence>` of elements corresponding to JavaBean properties.
Array of any supported data type (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.List (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.ArrayList (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.LinkedList (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.Vector (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.Stack (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.Collection (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.Set (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.

Table 8-4 Supported User-Defined Java Data Types

Java Data Type	Equivalent XML Schema Data Type
java.util.HashSet (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.SortedSet (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.util.TreeSet (JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
java.lang.Object **Note:** The data type of the runtime object must be a known type.	<xsd:anyType>
JAX-RPC-style enumeration class	<xsd:simpleType> with enumeration facets

CHAPTER 9

Configuring Security

The following sections provide information on configuring differents kinds of security for you Web Service:

Overview of Web Services Security

To secure your WebLogic Web Service, you configure one or more of three conceptually different types of security:

- Message-level security, in which data in a SOAP message is digitally signed or encrypted.

  See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2.

- Transport-level security, in which SSL is used to secure the connection between a client application and the Web Service.

  See "Configuring Transport-Level Security" on page 9-13.

- Access control security, which specifies which roles are allowed to access Web Services.

See "Configuring Access Control Security: Main Steps" on page 9-13.

What Type of Security Should You Configure?

Access control security answers the question "who can do what?" First you specify the list of roles that are allowed to access a Web Service. Then, when a client application attempts to invoke a Web Service operation, the client authenticates itself to WebLogic Server, using HTTP, and if the client has the authorization, it is allowed to continue with the invocation. Access control security secures only WebLogic Server resources. This means that if you configure *only* access control security, the connection between the client application and WebLogic Server is not secure and the SOAP message is in plain text.

With transport-level security, you secure the connection between the client application and WebLogic Server with Secure Sockets Layer (SSL). SSL provides secure connections by allowing two applications connecting over a network to authenticate the other's identity and by encrypting the data exchanged between the applications. Authentication allows a server, and optionally a client, to verify the identity of the application on the other end of a network connection. Encryption makes data transmitted over the network intelligible only to the intended recipient.

Transport-level security, however, secures only the connection itself. This means that if there is an intermediary between the client and WebLogic Server, such as a router or message queue, the intermediary gets the SOAP message in plain text. When the intermediary sends the message to a second receiver, the second receiver does not know who the original sender was. Additionally, the encryption used by SSL is "all or nothing": either the entire SOAP message is encrypted or it is not encrypted at all. There is no way to specify that only selected parts of the SOAP message be encrypted.

Message-level security includes all the security benefits of SSL, but with additional flexibility and features. Message-level security is end-to-end, which means that a SOAP message is secure even when the transmission involves one or more intermediaries. The SOAP message itself is digitally signed and encrypted, rather than just the connection. And finally, you can specify that only parts of the message be signed or encrypted.

Configuring Message-Level Security (Digital Signatures and Encryption)

Message-level security specifies whether the SOAP messages between a client application and the Web Service it is invoking should be digitally signed or encrypted or both.

WebLogic Web Services implement the following OASIS Standard 1.0 Web Services Security specifications, dated April 6 2004:

- Web Services Security: SOAP Message Security
- Web Services Security: Username Token Profile
- Web Services Security: X.509 Token Profile

These specifications provide three main mechanisms: security token propagation, message integrity, and message confidentiality. These mechanisms can be used independently (such as passing a username security token for user authentication) or together (such as digitally signing and encrypting a SOAP message.)

Message-level security is configured using security policy statements, as specified by the WS-Policy (dated September 2004) specification.

The following sections provide information about message-level security:

- "Main Use Cases" on page 9-3
- "Use of WS-Policy Files for Message-Level Security Configuration" on page 9-4
- "Configuring Message-Level Security: Main Steps" on page 9-4

Main Use Cases

BEA's implemenation of the *Web Services Security: SOAP Message Security* specification is designed to fully support the following use cases:

- Use an X.509 certificate to encrypt and sign a SOAP message, starting from the client application that invokes the message-secured Web Service, to the WebLogic Server instance that is hosting the Web Service and back to the client application. The SOAP message itself contains all the security information, so intermediaries between the client application and Web Service can also play a part without compromising any security of the message.
- Provide flexibility over what parts of the SOAP message are signed and encrypted. By default, when you enable WebLogic Web Service message-level security, the entire SOAP message body is encrypted and signed. You can, however, specify that all occurrences of a specific element in the SOAP message be signed, encrypted, or both.
- Include an encrypted and signed username and password in the SOAP message (rather than in the HTTP header, as is true for SSL and access control security) for further downstream processing.

Use of WS-Policy Files for Message-Level Security Configuration

You specify the details of message-level security for a WebLogic Web Service with WS-Policy files. The WS-Policy specification provides a general purpose model and syntax to describe and communicate the policies of a Web service.

The WS-Policy files used for message-level security are XML files that describe how a SOAP message should be digitally signed or encrypted. They also can specify that a client application authenticate itself with username and password.

You specify the names of the WS-Policy files that are attached to your Web Service using the `@Policy` JWS annotation in your JWS file.

WebLogic Server includes three simple WS-Policy files that you can specify in your JWS file if you do not want to create your own WS-Policy files:

- `Auth.xml`: specifies that the client application invoking the Web Service must authenticate itself with a username and password.
- `Encrypt.xml`: specifies that the entire body of the SOAP message be encrypted.
- `Sign.xml`: specifies that the body and WebLogic-specific system headers of the SOAP message be digitally signed.

Using the `@Policy` and `@Policies` JWS annotations, you can specify one or more of these pre-packaged WS-Policy files in your JWS file, at either the class or method level.

Configuring Message-Level Security: Main Steps

Configuring message-level security for a WebLogic Web Service involves some standard security tasks, such as obtaining digital certificates, creating keystores, and users, as well as Web Service-specific tasks, such as adding security-related JWS annotations to your JWS file and optionally creating WS-Policy files if you do not want to use the pre-packaged ones included in WebLogic Server.

To configure message-level security for a WebLogic Web Service and a client that invokes the service, follow these steps. Later sections describe some steps in more detail.

Note: It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web Service and you want to update it so that the SOAP messages are digitally signed and encrypted. It is also assumed that you use Ant build scripts to iteratively develop your Web Service and that you have a working `build.xml`

file that you can update with new information. Finally, it is assumed that you have a client application that invokes the non-secured Web Service. If these assumptions are not true, see:

- Chapter 5, "Programming the JWS File"
- Chapter 4, "Iteratively Developing WebLogic Web Services"
- Chapter 7, "Invoking Web Services"

1. Obtain two sets of key pair and digital certificates to be used by WebLogic Web Services. Although not required, BEA recommends that you obtain key pairs and certificates that will be used *only* by WebLogic Web Services.

   Warning: BEA requires that the key length be 1024 bits or larger.

   You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   For details, see *Obtaining Private Keys and Digital Signatures* at http://e-docs.bea.com/wls/docs90/secmanage/identity_trust.html#get_keys_certs_trustedcas.

2. Create, if one does not currently exist, a custom identity keystore for WebLogic Server and load the key pairs and digital certificates you obtained in the preceding step into the identity keystore.

   If you have already configured WebLogic Server for SSL, then you have already created a identity keystore which you can also use for WebLogic Web Services data security purposes.

   You can use WebLogic's `ImportPrivateKey` utility and Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   For details, see *Creating a Keystore and Loading Private Keys and Trusted Certificate Authorities Into the Keystore* at http://e-docs.bea.com/wls/docs90/secmanage/identity_trust.html#keystore_creating.

3. Using the Administration Console, configure WebLogic Server to locate the keystore you created in the preceding step. If you are using a keystore that has already been configured for WebLogic Server, you do not need to perform this step.

   For details, see *Configuring Keystores for Production* at http://e-docs.bea.com/wls/docs90/secmanage/identity_trust.html#ConfiguringKeystores.

4. Create a keystore used by the client application. BEA recommends that you create one client keystore per application user.

You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

For details, see *Obtaining Private Keys and Digital Signatures* at http://e-docs.bea.com/wls/docs90/secmanage/identity_trust.html#get_keys_certs_trustedcas.

5. Create a key pair and a digital certificate, and load them into the client keystore. The same key pair will be used to digitally sign the SOAP request and encrypt the SOAP responses. The digital certificate will be mapped to a user of WebLogic Server, created in a later step.

   Warning: BEA requires that the key length be 1024 bits or larger.

   You can use Sun Microsystem's `keytool` utility to perform this step.

6. Using the Administration Console, configure an Identity Asserter provider for your WebLogic Server security realm.

   WebLogic Server provides a default security realm, called `myrealm`, which is configured with a default Identity Asserter provider. Use this default security realm if you do not want to configure your own Identity Asserter provider. You must, however, perform additional configuration tasks to ensure that the default Identity Asserter Provider works correctly with message-secured WebLogic Web Services.

   For details, see "Configuring The Identity Asserter Provider for the myrealm Security Realm" on page 9-7.

7. Using the Administration Console, create users for authentication in your security realm.

   For details, see *Creating Users* at http://e-docs.bea.com/wls/docs90/secwlres/usrs_grps.html.

   Later sections of this guide assume you created a user `auth_user` with password `auth_user_password`.

8. Update your JWS file, adding WebLogic-specific `@Policy` and `@Policies` JWS annotations to specify the WS-Policy files that are attached to either the entire Web Service or to particular operations.

   See "Updating the JWS File With @Policy and @Policies Annotations" on page 9-8.

9. If you do not want to use the WS-Policy files that are packaged with WebLogic Server, create your own WS-Policy file.

   See "Use of WS-Policy Files for Message-Level Security Configuration" on page 9-4 for descriptions of the pre-packaged WS-Policy files. See "Creating a WS-Policy File" on page 9-10 for details on creating your own WS-Policy file.

10. Update your client application to invoke the message-secured Web Service.

    For details, see "Updating a Client Application to Invoke a Message-Secured Web Service" on page 9-11.

Configuring The Identity Asserter Provider for the myrealm Security Realm

You can use the default Identity Asserter provider, configured for the default myrealm security realm, with message-secured WebLogic Web Services. You must, however, perform some additional configuration tasks:

1. Invoke the Administration Console in your browser.

   See "Invoking the Administration Console" on page 10-3.

2. Click the Lock & Edit button in the top left hand corner to be able to modify WebLogic Server.
3. In the left pane, click Security Realms. A table of all the security realms configured for WebLogic Server appears in the right pane.
4. In the right pane, click the `myrealm` security realm listed in the table.
5. Click the Providers->Authentication tab.
6. Click DefaultIdentityAsserter in the displayed table.
7. Click the Configuration->Provider Specific tab.
8. Scroll down to the Active Types box.
9. Move `X.509` from the Available box to the Chosen box.
10. Ensure that Use Default User Name Mapper is checked.
11. Click Save.
12. Click the Activate Changes button in the top left corner of the console to activate your changes.

For additional information about configuring the Identity Asserter, see:

- WebLogic Identity Asserter Provider -> Details at http://e-docs.bea.com/wls/docs90/ConsoleHelp/security_defaultidentityasserter_details.html

- WebLogic Identity Asserter Provider -> General at http://e-docs.bea.com/wls/docs90/ConsoleHelp/security_defaultidentityasserter_general.html

Updating the JWS File With @Policy and @Policies Annotations

Use the `@Policy` and `@Policies` annotations in your JWS file to specify that the Web Service has one or more WS-Policy files attached to it. You can use these annotations at either the class or method level.

The `@Policies` annotation simply groups two or more `@Policy` annotations together. Use the `@Policies` annotation if you want to attach two or more policy files to the class or method. If you want to attach just one policy file, you can use `@Policy` on its own.

The `@Policy` annotation describes a single policy file and whether the policy applies to the request or response SOAP message, or both. Use the `uri` attribute to specify the location of the policy file, as described below:

- To specify one of the three pre-packaged policy files that are installed with WebLogic Server, use the `policy:` prefix and the name of one of the policy files (either `Auth.xml`, `Encrypt.xml`, or `Sign.xml`), as shown in the following example:

  ```
  @Policy(uri="policy:Encrypt.xml")
  ```

  If you use the pre-packaged policy files, you do not have to create one yourself or package it in an accessible location. For this reason, BEA recommends that you use the pre-packaged policy files whenever you can.

  See "Use of WS-Policy Files for Message-Level Security Configuration" on page 9-4 for information on the various types of message-level security provided by the pre-packaged policy files.

- To specify that a user-created policy file is located in the JAR or WAR archive in which the Web Service is packaged, also specify the `policy:` prefix along with the name of the policy file. However, in this case you must include the policy file in a specific location of the J2EE archive in which the Web Service is packaged:
  - For EJB-implemented Web Services, the policy file must be located in the `META-INF/policies` directory of the EJB JAR file.
  - For Java class-implemented Web Services, the policy file must be located in the `WEB-INF/policies` directory of the Web application WAR file.

In the following example, the `MyPolicy.xml` file must be located in the `META-INF/policies/encryption` (or `WEB-INF/policies/encryption`, depending on the implementation of the Web Service):

```
@Policy(uri="policy:encryption/MyPolicy.xml")
```

You can also put your policy file in the `META-INF/policies` or `WEB-INF/policies` directory of a shared J2EE library if you want to share the policy file with multiple Web Services packaged in different J2EE archives. You specify the policy file in the same way as if it were packaged in the same archive at the Web Service. See Creating Shared J2EE Libraries and Optional Packages at http://e-docs.bea.com/wls/docs90/programming/libraries.html for information on creating shared libraries and setting up your environment so the Web Service can find the policy files.

- To specify that the policy file is published somewhere on the Web, use the `http:` prefix along with the URL, as shown in the following example:

```
@Policy(uri="http://someSite.com/policies/mypolicy.xml")
```

You can also set the following attributes of the `@Policy` annotation:

- `direction`—Specifies whether the policy file should be applied to the request (inbound) SOAP message, the response (outbound) SOAP message, or both. The default value if you do not specify this attribute is both. The `direction` attribute accepts the following values:
  - `Policy.Direction.both`
  - `Policy.Direction.inbound`
  - `Policy.Direction.outbound`
- `attachToWsdl`—Specifies whether the policy file should be attached to the WSDL file, published by WebLogic Server when the Web Service is deployed, that describes the public contract of the Web Service. Typically you want to publicly publish the policy so that client applications know what sort of message-level security they have to provide when invoking the Web Service. For this reason, the default value of this attribute is `true`.

The following example shows how to use the `@Policy` and `@Policies` JWS annotations, with the relevant sections shown in bold:

```
package examples.policy.jws.service;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Policies;
import weblogic.jws.Policy;
```

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

/**
 *
 *  Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
@WebService(name="HelloWorld",
            serviceName="HelloWorldService",
            targetNamesp"http://www.bea.com")
@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)
@WLHttpTransport(contextPath="SecureHelloWorldService",
                 serviceUri="SecureHeorldService")
@Policies({
    @Policy(uri="policy:Auth.xml", direction=Policy.Direction.inbound),
    @Policy(uri="policy:Sign.xml"),
    @Policy(uri="policy:Encrypt.xml")})

public class SecureHelloWorldService {
  @WebMethod()
  public String sayHello(String s) {
    return "Hello " + s;
  }
}
```

In the example, three policy files are attached to the Web Service at the class level, which means that all three policy files are applied to all public operations of the Web Service. The specified policy files are those pre-packaged with WebLogic Server; this means that the developer does not need to create their own files or package them in the corresponding archive.

The `Auth.xml` file is applied to *only* the request (inbound) SOAP message, as specified by the `direction` attribute. This means that only the client application needs to provide a UsernameToken; when WebLogic Server responds to the invoke, it does not provide a UsernameToken. The `Sign.xml` policy file specifies that the body and WebLogic system headers of both the request and response SOAP message be digitally signed. The `Encrypt.xml` policy file specifies that the body of both the request and response SOAP messages be encrypted.

By default, all three policy files are attached to the WSDL of the deployed Web Service.

Creating a WS-Policy File

You can create your own WS-Policy file if you do not want to use the pre-packaged ones that are installed with WebLogic Server. For example, you might want to specify that particular parts of the body of a SOAP message be encrypted, rather than the entire body, as specified by the `Encrypt.xml` pre-packaged policy file.

[[More information to come.]]

Updating a Client Application to Invoke a Message-Secured Web Service

When you update your Java code to invoke a message-secured Web Service, you must load a key pair and digital certificate from the client's keystore and pass this information, along with a username and password for user authentication if so required by the policy, to the secure WebLogic Web Service being invoked.

The following example shows a Java client application that invokes the message-secured WebLogic Web Service described by the JWS file in "Updating the JWS File With the @SecurityRoles Annotation" on page 9-14. The client application takes five arguments:

- the client's username, used for client authentication
- the client's password for client authentication
- the client's private key file
- the client's digital certificate
- the WSDL of the deployed Web Service

The the security-specific code in the sample client appliation is shown in bold (and described after the example):

```
package examples.policy.jws.client;

import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;

import javax.xml.rpc.Stub;
import java.util.List;
import java.util.ArrayList;

/**
 *  Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public class SecureHelloWorldClient {

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

    //username or password for the UsernameToken

    String username = args[0];
    String password = args[1];
```

```
    //client private key file

    String keyFile = args[2];

    //client certificate

    String clientCertFile = args[3];

    String wsdl = args[4];

    HelloWorldService service = new HelloWorldService_Impl(wsdl);
    HelloWorld port = service.getHelloWorldSoapPort();

    //create credential provider and set it to the Stub

    List credProviders = new ArrayList();

    //client side BinarySecurityToken credential provider -- x509

   CredentialProvider cp = new ClientBSTCredentialProvider(clientCertFile, keyF
ile);
    credProviders.add(cp);

    //client side UsernameToken credential provider

    cp = new ClientUNTCredentialProvider(username, password);
    credProviders.add(cp);



    Stub stub = (Stub)port;

    stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
credProviders)

    String response = port.sayHello("World");
    System.out.println("response = " + response);
  }
}
```

The main points to note about the preceding code are:

- Use the `ClientBSTCredentialProvider` WebLogic API to create a BinarySecurityToken from the client's certificate and private key:

  ```
  CredentialProvider cp =
     new ClientBSTCredentialProvider(clientCertFile, keyFile);
  ```

- Use the ClientUNTCredentialProvider WebLogic API to create a UsernameToken from the client's username and password:

  ```
  cp = new ClientUNTCredentialProvider(username, password);
  ```

- Use the `WSSecurityContext.CREDENTIAL_PROVIDER_LIST` property to pass a `List` object that contains the BinarySecurityToken and UsernameToken information to the JAX-RPC Stub, as required by the security policy of the Web Service:

  ```
  stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
  credProviders)
  ```

Configuring Transport-Level Security

Transport-level security refers to securing the connection between a client application and a Web Service with Secure Sockets Layer (SSL).

[[More information to come.]]

Configuring Access Control Security: Main Steps

Access control security refers to configuring the Web Service to control the users who are allowed to access it, and then coding your client application to authenticate itself, using HTTP, to the Web Service when the client invokes one of its operations.

Because WebLogic Web Services are implemented with either a stateless session EJB or a Java class, you can secure the Web Service by securing these components in the standard J2EE way. See *Securing WebLogic Resources* at http://e-docs.bea.com/wls/docs90/secwlres/intro.html.

If your Web Service is implemented with an EJB, you can also use a Web Service-specific JWS annotation (`@SecurityRoles`) in your JWS file to specify the roles that are allowed to access the entire Web Service or selected operations. This section describes how to do this.

The following procedure describes the high-level steps; later sections in the chapter describe the steps in more detail.

Note: It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web Service and you want to update it with access control security. It is also assumed that you use Ant build scripts to iteratively develop your Web Service and that you have a working `build.xml` file that you can update with new information. Finally, it is assumed that you have a client application that invokes the non-secured Web Service. If these assumptions are not true, see:

- Chapter 5, “Programming the JWS File”
- Chapter 4, “Iteratively Developing WebLogic Web Services”
- Chapter 7, “Invoking Web Services”

1. Update your JWS file, adding the standard `@SecurityRoles` annotation at the class or method level to specify the roles that are allowed to access either the entire Web Service or selected operations.

   See "Updating the JWS File With the @SecurityRoles Annotation" on page 9-14.

2. Using the Administration Console, create the role and users that map to the role.

   See Security Roles at http://e-docs.bea.com/wls/docs90/secwlres/secroles.html.

3. Update your client application to authenticate itself when invoking an access-secured WebLogic Web Service.

   See "Updating a Client Application to Authenticate Itself to a Web Service" on page 9-16.

Updating the JWS File With the @SecurityRoles Annotation

Use the JSR-181 standard `@SecurityRoles` JWS annotation in your JWS file to specify the roles that are allowed to invoke the operations of a Web Service. You can set this annotation at both the class level or at the method level. At the class level, the roles apply to all public operations. You can add additional roles to a particular operation by specifying the annotation at the method level.

Note: You can use the `@SecurityRoles` annotation only for EJB-implemented Web Services.

The `@SecurityRoles` annotation has two attributes:

- `rolesAllowed`—Specifies the roles that are allowed to access the operations.

  This annotation is the equivalent of the `<method-permission>` element in the `ejb-jar.xml` deployment descriptor of the stateless session EJB that implements the Web Service.

- `rolesReference`—Specifies a list of roles referenced by the Web Service.

  This annotation is the equivalent of the `<security-role-ref>` element in the `ejb-jar.xml` deployment descriptor of the stateless session EJB that implements the Web Service.

The following example shows how to use the `@SecurityRoles` annotation in a JWS file, with the relevant sections shown in bold:

```
package examples.webservices.jws_basic.ejb;

import java.rmi.RemoteException;
```

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.security.SecurityRoles;

import com.bea.wls.ejbgen.annotations.*;

import weblogic.jws.WLHttpTransport;

@Session(ejbName="Simple",
        serviceEndpoint="examples.webservices.jws_basic.ejb.SimpleBeanPortType"
)
@WebService(name="myPortType",
            serviceName="SimpleEjbService",
            targetNamespace="http://example.org")
@SOAPBinding(style=SOAPBinding.Style.RPC,
             use=SOAPBinding.Use.ENCODED)
@WLHttpTransport(portName="helloPort",
                 contextPath="jws_basic_ejb",
                 serviceUri="SimpleBean")

// Specifies the roles that are allowed to invoke the operations of the Web
// Service.

@SecurityRoles (rolesAllowed="admin", rolesReferenced={"charlie","lucy"} )

/**
 * This JWS file forms the basis of a stateless-session EJB implemented
 * WebLogic Web Service.
 *
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class SimpleBean implements SessionBean {

  @WebMethod(operationName="myoperation")

  // Specifies the roles that are allowed to invoke this operation, in addition
  // to those specified at the class level.
  @SecurityRoles (rolesAllowed="guest")

  public int echoInt(int input) throws RemoteException {
    System.out.println("got input '" + input + "'");
    return input;
  }
```

```
  @WebMethod()
  public String echoString(String input) throws RemoteException {
    System.out.println("echoString got " + input);
    return input;
  }

  public void ejbCreate() {}
  public void ejbActivate() {}
  public void ejbRemove() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {}
}
```

Updating a Client Application to Authenticate Itself to a 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");
```

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.

CHAPTER 10

Administering Web Services

The following sections provide information on administering WebLogic Web Services:

Overview of Administering WebLogic Web Services

WebLogic Web Services are packaged and deployed either as EJB JAR file or Web application WAR files, depending on the type of component that implements the Web Service (stateless session EJB or Java class, respectively). Therefore, basic administration of Web Services is very similar to basic administration of EJBs or Web applications.

Standard tasks when administering Web Services include:

- Installing the Web Service.
- Starting and stoping the deployed Web Service

- Configuring the basic Web Service. You can configure general characteristics, such as the deployment order, or module-specific characteristics, such as session timeout for Web applications or transaction type for EJBs.
- Monitoring the Web Service.

There are a variety of ways to administer J2EE modules and applications that run on WebLogic Server, including Web Services; use the tool that best fits your needs:

- Using the Administration Console
- Using the WebLogic Scripting Tool
- Using WebLogic Ant Tasks
- Using the Java Management Extensions (JMX)
- Using the J2EE Deployment API

Using the Administration Console

The BEA WebLogic Server Administration Console is a Web browser-based, graphical user interface you use to manage a WebLogic Server domain, one or more WebLogic Server instances, clusters, and applications, including Web Services, that are deployed to the server or cluster.

One instance of WebLogic Server in each domain is configured as an Administration Server. The Administration Server provides a central point for managing a WebLogic Server domain. All other WebLogic Server instances in a domain are called Managed Servers. In a domain with only a single WebLogic Server instance, that server functions both as Administration Server and Managed Server. The Administration Server hosts the Administration Console, which is a Web Application accessible from any supported Web browser with network access to the Administration Server.

You can use the System Administration Console to:

- Configure, start, and stop WebLogic Server Instances
- Configure WebLogic Server Clusters
- Configure WebLogic Server Services, such as database connectivity (JDBC), and messaging (JMS).
- Configure security parameters, including managing users, groups, and roles.

- Configure and Deploy your applications.
- Monitor server and application performance.
- View server and domain log files.
- View application deployment descriptors.
- Edit selected runtime application deployment descriptor elements.

Invoking the Administration Console

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

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

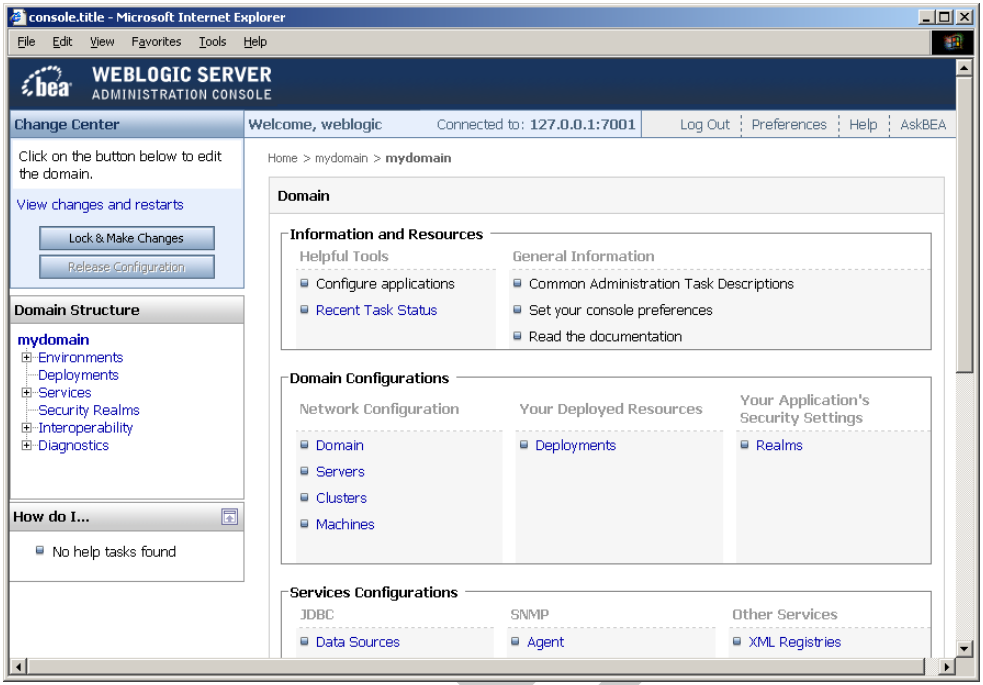
where

- *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.

Click the **Help** button, located at the top right corner of the Administration Console, to invoke the Online Help for detailed instructions on using the Administration Console.

The following figure shows the main Administration Console window.

Figure 10-1 WebLogic Server Administration Console Main Window



How Web Services Are Displayed In the Administration Console

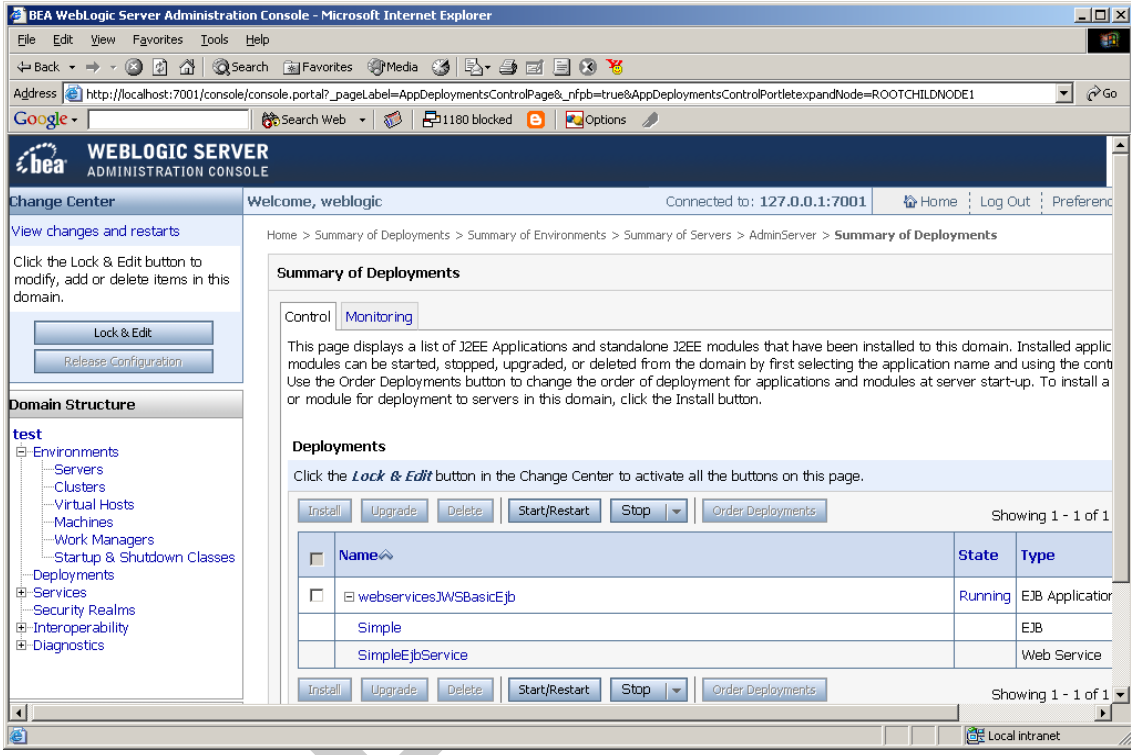
Web Services are essentially Web applications or EJBs, with additional artifacts so that the Web Service can be invoked using SOAP. These additional artifacts include Web Service deployment descriptors, the WSDL file, and so on. Web Services are packaged as WAR or EJB JAR files, which is how they are displayed in the Administration Console.

To view and update the Web Service-specific configuration information about a Web Service using the Administration Console, click on the Deployments node and find the Web application or EJB archive in which the Web Service is packaged. When you expand the archive, you will see the Web Service listed below, with a Deployment Type of `Web Service`. Click on the name of the Web Service to view or update its configuration.

The following figure shows how a Web Service is displayed in the main Deployments table of the Administration Console. The top-level listing of the Web Service in the Administration Console Deployments table is as the EJB Application `webservicesJWSBasicEjb`. Once you expand this EJB application, however, you see that it is made up of an EJB called `Simple` and a

Web Service called `SimpleEjbService`. Both are essentially the same component, but the component can be invoked as either an EJB or a Web Service, they are listed separately.

Figure 10-2 Web Service Displayed in Deployments Table of Administration Console



Using the WebLogic Scripting Tool

The WebLogic Scripting Tool (WLST) is a command-line scripting interface that you can use to interact with and configure WebLogic Server domains and instances, as well as deploy J2EE modules and applications (including Web Services) to a particular WebLogic Server instance. Using WLST, system administrators and operators can initiate, manage, and persist WebLogic Server configuration changes.

Typically, the types of WLST commands you use to administer Web Services fall under the Deployment category.

For more information on using WLST, see WebLogic Scripting Tool at http://e-docs.bea.com/wls/docs90/config_scripting/index.html.

Using WebLogic Ant Tasks

WebLogic Server includes a variety of Ant tasks that you can use to centralize many of the configuration and administrative tasks into a single Ant build script. These Ant tasks can:

- Build your application from your source directory, using `wlcompile`, `appc`, and `jwsc`.
- Create, start, and configure a new WebLogic Server domain, using the `wlserver` and `wlconfig` Ant tasks.
- Deploy a compiled application to the newly-created domain, using the `wldeploy` Ant task.

See Using Ant Tasks to Configure a WebLogic Server Domain and wldeploy Ant Task Reference for specific information about the non-Web Services related WebLogic Ant tasks.

Using the Java Management Extensions (JMX)

As an application developer, you can greatly reduce the cost of operating and maintaining your applications by building in management facilities directly into your application. The simplest facility is message logging, which reports events within your applications as they occur and writes messages to a file or other repository. Depending on the criticality of your application, the complexity of the production environment, and the types of monitoring systems your organization uses in its operations center, your needs might be better served by building richer management facilities based on Java Management Extensions (JMX). JMX enables a generic management system to monitor your application, raise notifications when the application needs attention, and change the configuration or runtime state of your application to remedy problems.

JMX uses MBeans, or managed beans, which are Java objects that represent resources to be managed. An MBean has a management interface consisting of:

- Named and typed attributes that can be read and written.
- Named and typed operations that can be invoked.
- Typed notifications that can be emitted by the MBean.

WebLogic Web Services have a set of MBeans associated with them that you can manipulate, using JMX, to programmatically add configuration and monitoring capabilities to your Web Service.

There are two types of MBeans: runtime (for read-only monitoring information) and configuration (for configuring the Web Service after it has been deployed).

The configuration Web Services MBeans are;

- [More information to come.]

The runtime Web Services Mbeans are:

- `weblogic.management.runtime.WseeRuntimeMbean` (Parent MBean of all other runtime Web Services MBeans)
- `weblogic.management.runtime.WseeHandlerRuntimeMBean`
- `weblogic.management.runtime.WseePortRuntimeMbean`
- `weblogic.management.runtime.WseeOperationRuntimeMbean`
- `weblogic.management.runtime.WseePolicyRuntimeMBean`

To access runtime information about the underlying EJB or Web Application that implements the Web Service, use the `weblogic.management.runtime.J2EEApplicationRuntimeMbean` and `weblogic.management.runtime.WebAppRuntimeMBean` objects, respectively.

The descriptor Web Services MBeans are:

- [More information to come.]

For information on using JMX to programmatically add configuration and monitoring capabilities to your Web Service, see Developing Manageable Applications with JMX. For reference information about the WebLogic Web Services MBeans, see WebLogic Server MBean Reference.

Using the J2EE Deployment API

In J2EE 1.4, the J2EE Application Deployment specification (JSR-88) defines a standard API that you can use to configure an application for deployment to a target application server environment.

The specification describes the J2EE 1.4 Deployment architecture, which in turn defines the contracts that enable tools or application programmers to configure and deploy applications on any J2EE platform product. The contracts define a uniform model between tools and J2EE platform products for application deployment configuration and deployment. The Deployment architecture makes it easier to deploy applications: Deployers do not have to learn all the features

of many different J2EE deployment tools in order to deploy an application on many different J2EE platform products.

See Deploying Applications to WebLogic Server for more information.

CHAPTER 11

Publishing and Finding Web Services Using UDDI

The following sections provide information about publishing and finding Web Services using UDDI:

Overview of 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 11-3.

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 11-1 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:

- UDDI 2.0 Server

- UDDI Directory Explorer
- UDDI Client API.
- Pluggable tModel

UDDI 2.0 Server

The UDDI 2.0 Server is part of WebLogic Server and is started automatically when WebLogic Server is started. The UDDI Server implements the UDDI 2.0 server specification at http://www.uddi.org/specification.html.

Configuring the UDDI 2.0 Server

To configure the UDDI 2.0 Server:

1. Stop WebLogic Server.
2. Update the `uddi.properties` file, located in the *`WL_HOME`*`/server/lib` directory, where *`WL_HOME`* refers to the main WebLogic Platform installation directory.
3. Restart WebLogic Server.

Never edit the `uddi.properties` file while WebLogic Server is running. Should you modify this file in a way that prevents the successful startup of UDDI Server, refer to the *`WL_HOME`*`/server/lib/uddi.properties.booted` file for the last known good configuration.

To restore your configuration to its default, remove the `uddi.properties` file from the *`WL_HOME`*`/server/lib` directory. BEA strongly recommends that you move this file to a backup location, because a new `uddi.properties` file will be created and with its successful startup the `uddi.properties.booted` file will also be overwritten. After removing the properties file, start the server. Minimal default properties will be loaded and written to a newly created `uddi.properties` file.

The following section describes the UDDI Server properties that you can include in the `uddi.properites` file. The list of properties has been divided according to component, usage and functionality. At any given time, you do not need all these properties to be present.

Description of Properties in the uddi.properties File

The following tables describe all the properties of the `uddi.properties` file, categorized by the type of UDDI feature they configure:

- Basic UDDI Configuration
- UDDI User Defaults
- General Server Configuration
- Logger Configuration
- Connection Pools
- LDAP Datastore Configuration
- Replicated LDAP Datastore Configuration
- File Datastore Configuration
- General Security Configuration
- LDAP Security Configuration
- File Security Configuration

Table 11-2 Basic UDDI Configuration

UDDI Property Key	Description
auddi.discoveryurl	Specifies the DiscoveryURL prefix that is set for each saved business entity. This will typically be the full URL to the uddilistener servlet, so that the full DiscoveryURL results in the display of the stored BusinessEntity data.
auddi.inquiry.secure	Permissible values are `true` and `false`. When set to `true`, inquiry calls to UDDI Server will be limited to secure https connections only. Any UDDI inquiry calls through a regular http URL will be rejected.
auddi.publish.secure	Permissible values are `true` and `false`. When set to `true`, publish calls to UDDI Server will be limited to secure https connections only. Any UDDI publish calls through a regular http URL will be rejected.
auddi.search.maxrows	The value of this property specifies the maximum number of returned rows for search operations. When the search results in a higher number of rows then the limit set by this property, the result will be truncated.

Table 11-2 Basic UDDI Configuration

UDDI Property Key	Description
auddi.search.timeout	The value of this property specifies a timeout value for search operations. The value is indicated in milliseconds.
auddi.siteoperator	This property determines the name of the UDDI registry site operator. The specified value will be used as the operator attribute, saved in all future BusinessEntity registrations. This attribute will later be returned in responses, and indicates which UDDI registry has generated the response.
security.cred.life	The value of this property, in seconds, specifies the credential life for authentication. Upon authentication of a user, an AuthToken is assigned which will be valid for the duration specified by this property.
pluggableTModel.file.list	UDDI Server is pre-populated with a set of Standard TModels. You can further customize the UDDI server by providing your own taxonomies, in the form of TModels. Taxonomies must be defined in XML files, following the provided XML schema. The value of this property a comma-separated list of URIs to such XML files. Values that refer to these TModels will be checked and validated against the specified taxonomy.

Table 11-3 UDDI User Defaults

UDDI Property Key	Description
auddi.default.lang	The value of this property determines a user's initial language, assigned to his user profile by default at the time of creation. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.assertion	The value of this property determines a user's initial assertion quota, assigned to his user profile by default at the time of creation. The assertion quota is the maximum number of publisher assertions that the user is allowed to publish. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.

Table 11-3 UDDI User Defaults

UDDI Property Key	Description
auddi.default.quota.binding	The value of this property determines a user's initial binding quota, assigned to his user profile by default at the time of creation. The binding quota is the maximum number of binding templates that the user is allowed to publish, per each business service. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.entity	The value of this property determines a user's initial business entity quota, assigned to his user profile by default at the time of creation. The entity quota is the maximum number of business entities that the user is allowed to publish. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.messageSize	The value of this property determines a user's initial message size limit, assigned to his user profile by default at the time of creation. The message size limit is the maximum size of a SOAP call that the user may send to UDDI Server. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.service	The value of this property determines a user's initial service quota, assigned to his user profile by default at the time of creation. The service quota is the maximum number of business services that the user is allowed to publish, per each business entity. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.tmodel	The value of this property determines a user's initial TModel quota, assigned to his user profile by default at the time of creation. The TModel quota is the maximum number of TModels that the user is allowed to publish. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.

Table 11-4 General Server Configuration

UDDI Property Keys	Description
auddi.datasource.type	This property allows you to configure the physical storage of UDDI data. This value defaults to `WLS`. The value of `WLS` for this property indicates that the internal LDAP directory of WebLogic Server is to be used for data storage. Other permissible values include `LDAP`, `ReplicaLDAP`, and `File`.
auddi.security.type	This property allows you to configure UDDI Server's security module (authentication). This value defaults to `WLS`. The value of `WLS` for this property indicates that the default security realm of WebLogic Server is to be used for UDDI authentication. As such, a WebLogic Server user would be an UDDI Server user and any WebLogic Server administrator would also be an UDDI Server administrator, in addition to members of the UDDI Server administrator group, as defined in UDDI Server settings. Other permissible values include `LDAP` and `File`.
auddi.license.dir	The value of this property specifies the location of the UDDI Server license file. In the absence of this property, the *`WL_HOME`*`/server/lib` directory is assumed to be the default license directory, where *`WL_HOME`* is the main WebLogic Platform installation directory. Some WebLogic users are exempt from requiring an UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (e.g., UDDI Server Browser).
auddi.license.file	The value of this property specifies the name of the license file. In the absence of this property, `uddilicense.xml` is presumed to be the default license filename. Some WebLogic users are exempt from requiring an UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (e.g., UDDI Server Browser).

Table 11-5 Logger Configuration

UDDI Property Key	Description
logger.file.maxsize	The value of this property specifies the maximum size of logger output files (if output is sent to file), in Kilobytes. Once an output file reaches maximum size, it is closed and a new log file is created.
logger.indent.enabled	Permissible values are `true` and `false`. When set to `true`, log messages beginning with "+" and "-", typically TRACE level logs, cause an increase or decrease of indentation in the output.
logger.indent.size	The value of this property, an integer, specifies the size of each indentation (how many spaces for each indent).
logger.log.dir	The value of this property specifies an absolute or relative path to a directory where log files are stored.
logger.log.file.stem	The value of this property specifies a string that is prefixed to all log file names.
logger.log.type	The value of this property determines whether log messages are sent to the screen, to a file or to both destinations. Permissible values for this property, respectively are: `LOG_TYPE_SCREEN`, `LOG_TYPE_FILE`, and `LOG_TYPE_SCREEN_FILE`.
logger.output.style	The value of this property determines whether logged output will simply contain the message, or if thread and timestamp information will be included. The two permissible values are `OUTPUT_LONG` and `OUTPUT_SHORT`.
logger.quiet	The value of this property determines whether the logger itself displays information messages or not. Permissible values are `true` and `false`.
logger.verbosity	The value of this property determines the logger's verbosity level. Permissible values (case sensitive) are `TRACE`, `DEBUG`, `INFO`, `WARNING` and `ERROR`, where each severity level includes the following ones accumulatively.

Table 11-6 Connection Pools

UDDI Property Key	Description
datasource.ldap.pool.increment	When all connections in the pool are busy, the value of this property specifies the number of new connections to create and add to the pool.
datasource.ldap.pool.initialsize	The value of this property specifies the number of connections to be stored, at the time of creation and initialization of the pool.
datasource.ldap.pool.maxsize	The value of this property specifies the maximum number of connections that the pool may hold.
datasource.ldap.pool.systemmaxsize	The value of this property specifies the maximum number of connections created, even after the pool has reached its capacity. Once the pool reaches its maximum size, and all connections are busy, connections are temporarily created and returned to the client, but not stored in the pool. However once the system max size is reached, all requests for new connections are blocked until a previously busy connection becomes available.

Table 11-7 LDAP Datastore Configuration

UDDI Property Key	Description
datasource.ldap.manager.uid	The value of this property specifies back-end LDAP server administrator or privileged user ID, (e.g. cn=Directory Manager) who can save data in LDAP.
datasource.ldap.manager.password	The value of this property is the password for the above user ID, and is used to establish connections with the LDAP directory used for data storage.
datasource.ldap.server.url	The value of this property is an "ldap://" URL to the LDAP directory used for data storage.
datasource.ldap.server.root	The value of this property is the root entry of the LDAP directory used for data storage (e.g., dc=acumenat, dc=com).

Note: In a replicated LDAP environment, there are "m" LDAP masters and "n" LDAP replicas, respectively numbered from 0 to (m-1) and from 0 to (n-1). The fifth part of the property keys below, quoted as "i", refers to this number and differs for each LDAP server instance defined.

Table 11-8 Replicated LDAP Datastore Configuration

UDDI Property Key	Description
datasource.ldap.server.master.i.manager.uid	The value of this property specifies the administrator or privileged user ID for this "master" LDAP server node, (e.g. cn=Directory Manager) who can save data in LDAP.
datasource.ldap.server.master.i.manager.password	The value of this property is the password for the matching above user ID, and is used to establish connections with the relevant "master" LDAP directory to write data.
datasource.ldap.server.master.i.url	The value of this property is an "ldap://" URL to the corresponding LDAP directory node.
datasource.ldap.server.master.i.root	The value of this property is the root entry of the corresponding LDAP directory node (e.g., dc=acumenat, dc=com).
datasource.ldap.server.replica.i.manager.uid	The value of this property specifies the user ID for this "replica" LDAP server node, (e.g. cn=Directory Manager) who can read the UDDI data from LDAP.
datasource.ldap.server.replica.i.manager.password	The value of this property is the password for the matching above user ID, and is used to establish connections with the relevant "replica" LDAP directory to read data.
datasource.ldap.server.replica.i.url	The value of this property is an "ldap://" URL to the corresponding LDAP directory node.
datasource.ldap.server.replica.i.root	The value of this property is the root entry of the corresponding LDAP directory node (e.g., dc=acumenat, dc=com).

Table 11-9 File Datastore Configuration

UDDI Property Key	Description
datasource.file.directory	The value of this property specifies the directory where UDDI data is stored in the file system.

Table 11-10 General Security Configuration

UDDI Property Key	Description
security.custom.group.operators	The value of this property specifies a security group name, where the members of this group will be treated as UDDI administrators.

Table 11-11 LDAP Security Configuration

UDDI Property Key	Description
security.custom.ldap.manager.uid	The value of this property specifies security LDAP server administrator or privileged user ID, i.e. cn=Directory Manager who can save data in LDAP.
security.custom.ldap.manager.password	The value of this property is the password for the above user ID, and is used to establish connections with the LDAP directory used for security.
security.custom.ldap.url	The value of this property is an "ldap://" URL to the LDAP directory used for security.
security.custom.ldap.root	The value of this property is the root entry of the LDAP directory used for security (e.g., dc=acumenat, dc=com).

Table 11-11 LDAP Security Configuration

UDDI Property Key	Description
security.custom.ldap.userroot	The value of this property specifies the users root entry on the security LDAP server. For example, ou=People.
security.custom.ldap.group.root	The value of this property specifies the operator entry on the security LDAP server. For example, "cn=UDDI Administrators, ou=Groups". This entry contains IDs of all UDDI administrators.

Table 11-12 File Security Configuration

UDDI Property Key	Description
security.custom.file.userdir	The value of this property specifies the directory where UDDI security information (users and groups) is stored in the file system.

UDDI Directory Explorer

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.)

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 **Explorer Help** link on the main page.

UDDI Client API

WebLogic Server includes an implementation of the client-side UDDI API that you can use in your Java client applications to programmatically 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/docs90/javadocs/index.html.

Pluggable tModel

A taxonomy is basically a tModel used as reference by a categoryBag or identifierBag. A major distinction is that in contrast to a simple tModel, references to a taxonomy are typically checked and validated. WebLogic Server's UDDI Server takes advantage of this concept and extends this capability by introducing custom taxonomies, called "pluggable tModels". Pluggable tModels allow users (UDDI administrators) to add their own checked taxonomies to the UDDI registry, or overwrite standard taxonomies.

To add a pluggable tModel:

1. Create an XML file conforming to the specified format described in "XML Schema for Pluggable tModels" on page 11-17, for each tModelKey/categorization.
2. Add the comma-delimited, fully qualified file names to the `pluggableTModel.file.list` property in the `uddi.properties` file used to configure UDDI Server. For example:

   ```
   pluggableTModel.file.list=c:/temp/cat1.xml,c:/temp/cat2.xml
   ```

   See "Configuring the UDDI 2.0 Server" on page 11-5 for details about the uddi.properties file.
3. Restart WebLogic Server.

The following sections include a table detailing the XML elements and their permissible values, the XML schema against which pluggable tModels are validated, and a sample XML.

XML Elements and Permissible Values

The following table describes the elements of the XML file that describes your pluggable tModels.

Table 11-13 Description of the XML Elements to Configure Pluggable tModels

Element/Attribute	Required	Role	Values	Comments
Taxonomy	Required	Root Element		
checked	Required	Whether this categorization is checked or not.	true / false	If false, keyValue will not be validated.
type	Required	The type of the tModel.	categorization / identifier / valid values as defined in uddi-org-types	See uddi-org-types tModel for valid values.

Table 11-13 Description of the XML Elements to Configure Pluggable tModels

Element/Attribute	Required	Role	Values	Comments
applicability	Optional	Constraints on where the tModel may be used.		No constraint is assumed if this element is not provided
scope	Required if the applicability element is included.		businessEntity / businessService / bindingTemplate / tModel	tModel may be used in tModelInstanceInfo if scope "bindingTemplate" is specified.
tModel	Required	The actual tModel, according to the UDDI data structure.	Valid tModelKey must be provided.	
categories	Required if checked is set to true.			
category	Required if element categories is included	Holds actual keyName and keyValue pairs.	keyName / keyValue pairs	category may be nested for grouping or tree structure.
keyName	Required			
keyValue	Required			

XML Schema for Pluggable tModels

The XML Schema against which pluggable tModels are validated is as follows:

```
<simpleType name="type">
  <restriction base="string"/>
</simpleType>
```

```
<simpleType name="checked">
  <restriction base="NMTOKEN">
    <enumeration value="true"/>
    <enumeration value="false"/>
  </restriction>
</simpleType>

<element name="scope" type="string"/>

<element name = "applicability" type = "uddi:applicability"/>

<complexType name = "applicability">
  <sequence>
    <element ref = "uddi:scope" minOccurs = "1" maxOccurs = "4"/>
  </sequence>
</complexType>

<element name="category" type="uddi:category"/>

<complexType name = "category">
  <sequence>
   <element ref = "uddi:category" minOccurs = "0" maxOccurs = "unbounded"/>
  </sequence>
  <attribute name = "keyName" use = "required" type="string"/>
  <attribute name = "keyValue" use = "required" type="string"/>
</complexType>

<element name="categories" type="uddi:categories"/>

<complexType name = "categories">
  <sequence>
   <element ref = "uddi:category" minOccurs = "1" maxOccurs = "unbounded"/>
  </sequence>
</complexType>

<element name="Taxonomy" type="uddi:Taxonomy"/>

<complexType name="Taxonomy">
  <sequence>
    <element ref = "uddi:applicability" minOccurs = "0" maxOccurs = "1"/>
    <element ref = "uddi:tModel" minOccurs = "1" maxOccurs = "1"/>
    <element ref = "uddi:categories" minOccurs = "0" maxOccurs = "1"/>
  </sequence>
```

```
      <attribute name = "type" use = "required" type="uddi:type"/>
      <attribute name = "checked" use = "required" type="uddi:checked"/>
    </complexType>
```

Sample XML for a Pluggable tModel

The following shows a sample XML for a pluggable tModel:

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

  <SOAP-ENV:Envelope
      xmlns:SOAP_ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <SOAP-ENV:Body>

   <Taxonomy checked="true" type="categorization" xmlns="urn:uddi-org:api_v2" >
    <applicability>
        <scope>businessEntity</scope>
        <scope>businessService</scope>
        <scope>bindingTemplate</scope>
    </applicability>
    <tModel  tModelKey="uuid:C0B9FE13-179F-41DF-8A5B-5004DB444tt2" >
        <name> sample pluggable tModel </name>
        <description>used for test purpose only </description>
            <overviewDoc>
                <overviewURL>http://www.abc.com </overviewURL>
            </overviewDoc>
    </tModel>
    <categories>
        <category keyName="name1 " keyValue="1">
            <category keyName="name11" keyValue="12">
                <category keyName="name111" keyValue="111">
                    <category keyName="name1111" keyValue="1111"/>
                    <category keyName="name1112" keyValue="1112"/>
                </category>
                <category keyName="name112" keyValue="112">
                    <category keyName="name1121" keyValue="1121"/>
                    <category keyName="name1122" keyValue="1122"/>
                </category>
            </category>
        </category>
        <category keyName="name2 " keyValue="2">
            <category keyName="name21" keyValue="22">
                <category keyName="name211" keyValue="211">
                    <category keyName="name2111" keyValue="2111"/>
                    <category keyName="name2112" keyValue="2112"/>
```

```
                </category>
                <category keyName="name212" keyValue="212">
                    <category keyName="name2121" keyValue="2121"/>
                    <category keyName="name2122" keyValue="2122"/>
                </category>
            </category>
        </category>
    </categories>
   </Taxonomy>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

CHAPTER 12

Upgrading an 8.1 Web Service to 9.0

The following sections provide information about upgrading an 8.1 Web Service to run on the 9.0 Web Service runtime:

Overview of Upgrading

This section describes how to upgrade an 8.1 WebLogic Web Service to use the new Version 9.0 Web Services runtime. This runtime is based on the *Implementing Enterprise Web Services* 1.1 specification (JSR-921, which is the 1.1 maintenance release of JSR-109). The 9.0 programming model uses standard JDK 1.5 metadata annotations, as specified by the *Web Services Metadata for the Java Platform* specification (JSR-181).

Note: 8.1 WebLogic Web Services will continue to run, without any changes, on Version 9.0 of WebLogic Server because the 8.1 Web Services runtime is still supported in 9.0, although it is deprecated and will be removed from the product in future releases. For this reason, BEA highly recommends that you follow the instructions in this chapter to upgrade your 8.1 Web Service to 9.0.

Upgrading your 8.1 Web Service includes the following high-level tasks (the procedures in later sections go into more detail):

- Update the Java source code of the Java class or stateless session EJB that implements the Web Service to now use JWS annotations.
- Update the Ant build script that builds the Web Service to call the 9.0 WebLogic Web Service Ant task `jwsc` instead of the 8.1 `servicegen` task.
- Add additional tasks to your Ant build script to compile generated Java source code and package the 9.0 Web Service into a deployable archive.

  Previously, in 8.1, the `servicegen` Ant task would automatically execute many of these tasks for you. However, in 9.0, the equivalent Ant task (`jwsc`) only generates Web Service artifacts (WSDL file, deployment descriptors, XML Schemas, Web Service endpoint interface and so on) and places them in an output directory. It is then up to the developer to compile the Java source into class files and package everything into an EJB JAR file or WAR file, as appropriate. The main reason for this change in functionality is to give the developer more control over the smaller tasks, and to be consistent with other WebLogic Ant tasks, such as `wlcompile` and `wlappc`.

It is assumed in this section that:

- You previously used `servicegen` to generate your 8.1 Web Service and that, more generally, you use Ant scripts in your development environment to iteratively develop Web Service and other J2EE artifacts that run on WebLogic Server. The procedures in this section direct you to update existing Ant `build.xml` files.
- You have access to the Java class or EJB source code for your 8.1 Web Service.

This section does *not* discuss the following topics:

- Upgrading a JMS-implemented 8.1 Web Service.
- Upgrading Web Services from versions previous to 8.1.
- Upgrading a client application that invokes an 8.1 Web Service to one that invokes a 9.0 Web Service. For details on how to write a client application that invokes a 9.0 Web Service, see Chapter 7, “Invoking Web Services.”

Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 9.0: Main Steps

The following procedure describes the basic guidelines to upgrade an 8.1 Java class-implemented Web Service to use the 9.0 WebLogic Web Services runtime.

1. Create a working directory to contain the upgraded Web Service.
2. Copy the old Java class that implements the 8.1 Web Service to the working directory. Rename the file, if desired.
3. Edit the Java file, as described in the following steps. See the old and new sample Java files in "Example of an 8.1 and Updated 9.0 Java Class File" on page 12-5 for specific examples.

   a. If needed, change the package name and class name of the Java file.

   b. Add import statements to import the JWS annotations.

   c. Add, at a minimum, the following JWS anntotations:

   – The standard `@WebService` annotation at the Java class level to specify that the JWS file implements a Web Service.

   – The standard `@SOAPBinding` annotation to specify the type of Web Service, such as document-literal or RPC-encoded.

   – The WebLogic-specific `@WLHttpTransport` annotation to specify the the context and service URIs that are used in the URL that invokes the deployed Web Service.

   – The standard `@WebMethod` annotation for each method that is exposed as a Web Service operation.

   See Chapter 5, "Programming the JWS File," for general information about using JWS annotations in a Java file.

   d. You might need to add additional annotations to your JWS file, depending on the 8.1 Web Service features you want to carry forward to 9.0. In 8.1, many of these features were configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations" on page 12-19 for a table that lists equivalent JWS annotation, if available, for features you enabled in 8.1 using `servicegen` attributes.

4. Update your Ant `build.xml` file to execute the `jwsc` Ant task, along with other supporting tasks, instead of `servicegen`.

BEA recommends that you create a new target, such as `build-service`, in your Ant build file and add all the tasks needed to build a 9.0 Web Service based on the JWS file you created in the preceding steps. Once this target is working correctly, you can remove the old `servicegen` Ant task.

The following procedure lists the main steps to update your `build.xml` file; for details on the steps, see the standard iterative development process outlined in Chapter 4, "Iteratively Developing WebLogic Web Services."

See "Example of an 8.1 and Updated 9.0 Ant Build File For Java Class-Implemented Web Services" on page 12-6 for specific examples of the steps in the following procedure.

a. Add the `jwsc` taskdef to the `build.xml` file.

b. Create a `build-service` target and add all the tasks needed to build the 9.0 Web Service, as described in the following steps.

c. Add a `mkdir` task to create a *`temp_dir`*`/war/WEB-INF/classes` directory, where *`temp_dir`* refers to a temporary output directory to hold the artifacts generated by `jwsc` until they are ready to be packaged into a WAR file.

d. Add the `jwsc` task to the build file. Set the `source` attribute to the name of the JWS file you created in the preceding steps and the `destdir` attribute to the *`temp_dir`*`/war/WEB-INF` directory.

   You may need to specify additional attributes to the `jwsc` task, depending on the 8.1 Web Service features you want to carry forward to 9.0. In 8.1, many of these features were configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations" on page 12-19 for a table that describes if there is an equivalent `jwsc` attribute for features you enabled using `servicegen` attributes.

e. Add `javac` tasks to compile the JWS file and the Java files generated by the `jwsc` Ant task. Set the `destdir` attribute to the *`temp_dir`*`/war/WEB-INF/classes` directory.

f. Add a `jar` task to package the files under the *`temp_dir`*`/war` directory into a WAR file.

5. Execute the `build-service` Ant target. Assuming all the tasks complete successfully, the resulting WAR file is your upgraded 9.0 Web Service.

See "Deploying and Undeploying WebLogic Web Services" on page 4-9 and "Invoking the WSDL and Home Page of the Web Service" on page 4-11 for additional information about deploying and testing your Web Service.

Example of an 8.1 and Updated 9.0 Java Class File

Assume that the following sample Java class implemented a 8.1 Web Service:

```
package examples.javaclass;

/**
 * Simple Java class that implements the HelloWorld Web service.   It takes
 * as input an integer and a String, and returns a message that includes these
 * two parameters.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public final class HelloWorld81 {

  /**
   * Returns a text message that includes the integer and String input
   * parameters.
   *
   */
  public String sayHello(int num, String s) {

    System.out.println("sayHello operation has been invoked with arguments " +
s + " and " + num);

    String returnValue = "This message brought to you by the letter "+s+" and
the number "+num;

    return returnValue;
  }

}
```


An equivalent JWS file for a 9.0 Java class-implemented Web Service is shown below, with the differences shown in bold. Note that some of the JWS annotation values are taken from attributes of the 8.1 servicegen Ant task shown in "Example of an 8.1 and Updated 9.0 Ant Build File For Java Class-Implemented Web Services" on page 12-6:

```
package examples.upgradeJavaWS;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

import weblogic.jws.WLHttpTransport;
```

```
/**
 * Simple Java class that implements the HelloWorld90 Web service.   It takes
 * as input an integer and a String, and returns a message that includes these
 * two parameters.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

@WebService (name= "upgradedWebService",
             targetNamespace="http://example.org",
             serviceName="HelloWorld")

@SOAPBinding (style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)

@WLHttpTransport(contextPath="jwsWeb", serviceUri="HelloWorld")

public final class HelloWorld90 {

  /**
   * Returns a text message that includes the integer and String input
   * parameters.
   *
   */

  @WebMethod()
  public String sayHello(int num, String s) {

    System.out.println("sayHello operation has been invoked with arguments " +
s + " and " + num);

    String returnValue = "This message brought to you by the letter "+s+" and
the number "+num;

    return returnValue;
  }

}
```


Example of an 8.1 and Updated 9.0 Ant Build File For Java Class-Implemented Web Services

The following simple `build.xml` file shows the 8.1 way to build a WebLogic Web Service using the `servicegen` Ant task:

```
<project name="javaclass-webservice" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property name="source" value="."/>
  <property name="build" value="${source}/build"/>
```

```
  <property name="war_file" value="HelloWorldWS.war" />
  <property name="ear_file" value="HelloWorldApp.ear" />
  <property name="namespace" value="http://examples.org" />

  <target name="all" depends="clean, ear"/>

  <target name="clean">
    <delete dir="${build}"/>
  </target>

  <!-- example of old 8.1 servicegen call to build Web Service -->

  <target name="ear">
    <servicegen
       destEar="${build}/${ear_file}"
       warName="${war_file}">
       <service
         javaClassComponents="examples.javaclass.HelloWorld81"
         targetNamespace="${namespace}"
         serviceName="HelloWorld"
         serviceURI="/HelloWorld"
         generateTypes="True"
         expandMethods="True">
       </service>
     </servicegen>
  </target>

</project>
```


An equivalent `build.xml` file that calls the `jwsc` Ant task to build a 9.0 Web Service is shown below, with the relevant tasks in bold:

```
<project name="javaclass-webservice" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property name="source" value="."/>
  <property name="build" value="${source}/build"/>
  <property name="war_file" value="HelloWorldWS.war" />
  <property name="ear_file" value="HelloWorldApp.ear" />
  <property name="namespace" value="http://examples.org" />
```

```
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

  <target name="all" depends="clean, build-service"/>

  <target name="clean">
    <delete dir="${build}"/>
  </target>

  <target name="build-service">

    <mkdir dir="${build}/war/WEB-INF/classes" />

    <jwsc source="HelloWorld90.java"
          destdir="${build}/war/WEB-INF/" />

    <javac
      srcdir="${build}/war/WEB-INF/"
      source="1.5"
      destdir="${build}/war/WEB-INF/classes"
      includes="*.java"/>

    <javac
      srcdir="."
      source="1.5"
      destdir="${build}/war/WEB-INF/classes"
      includes="*.java"/>

    <jar destfile="${build}/${war_file}"
      basedir="${build}/war"/>

  </target>
</project>
```


Upgrading an 8.1 EJB-Implemented WebLogic Web Service to 9.0: Main Steps

The following procedure describes the basic guidelines to upgrade an 8.1 EJB-implemented Web Service to use the 9.0 WebLogic Web Services runtime.

It is assumed that in 8.1, you programmed the EJB by creating all the needed artifacts: the Bean class that implements `javax.ejb.SessionBean`; the Home and Remote interfaces; and the `ejb-jar.xml` deployment descriptor file. The 9.0 EJB, however, makes use of EJBGen

annotations to simplify programming an EJB: you need only create the Bean implementation class and all the other EJB artifacts are automatically generated. The Bean implementation class also has the JWS annotations, so it also acts as the JWS file. The `jwsc` Ant task automatically calls the EJBGen task when it encounters an EJBGen annotation.

1. Create a working directory to contain the upgraded Web Service.
2. Copy the 8.1 Bean class that implemented `javax.ejb.SessionBean` to the working directory. Rename the file, if desired.
3. Edit the new Bean file, as described in the following steps. See the old and new sample Bean classes and interfaces in "Example of an 8.1 and Updated 9.0 EJB Classes" on page 12-11 for specific examples.

   a. If needed, change the package name and class name of the Bean file.

   b. Add import statements to import the JWS annotations.

   c. Add, at a minimum, the following JWS anntotations:

   – The standard `@WebService` annotation at the class level to specify that the JWS file implements a Web Service.

   – The standard `@SOAPBinding` annotation to specify the type of Web Service, such as document-literal or RPC-encoded.

   – The WebLogic-specific `@WLHttpTransport` annotation to specify the the context and service URIs that are used in the URL that invokes the deployed Web Service.

   – The standard `@WebMethod` annotation for each method that is exposed as a Web Service operation.

   See Chapter 5, "Programming the JWS File," for general information about using JWS annotations in a Java file.

   d. Add the EJBGen annotations needed to generate all the EJB artifacts.

   At a minimum, add the `@Session` annotation to specify that the Bean class implements a stateless session EJB. Set the `serviceEndpoint` attribute equal to the full classname of the Bean you are editing, with a `PortType` suffix. For example, if the full classname of the Bean is `examples.service.HelloBean`, set `serviceEndpoint` to `examples.service.HelloBeanPortType`. This class will later be automatically generated by the `jwsc` Ant task.

   e. You might need to add additional annotations to your JWS file, depending on the 8.1 Web Service features you want to carry forward to 9.0. In 8.1, many of these features were

configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations" on page 12-19 for a table that lists equivalent JWS annotation, if available, for features you enabled in 8.1 using `servicegen` attributes.

f. Make sure that the methods you are exposing as Web Service operations throw `java.rmi.RemoteException`.

4. Update your Ant `build.xml` file to execute the `jwsc` Ant task, along with other supporting tasks, instead of `servicegen`.

   BEA recommends that you create a new target, such as `build-service`, in your Ant build file and add all the tasks needed to build a 9.0 Web Service based on the JWS file you created in the preceding steps. Once this target is working correctly, you can remove the old `servicegen` Ant task.

   The following procedure lists the main steps to update your `build.xml` file; for details on the steps, see the standard iterative development process outlined in Chapter 4, "Iteratively Developing WebLogic Web Services."

   See "Example of an 8.1 and Updated 9.0 Ant Build File For EJB-Implemented Web Services" on page 12-16 for specific examples of the steps in the following procedure.

   a. Add the `jwsc` taskdef to the `build.xml` file.

   b. Create a `build-service` target and add all the tasks needed to build the 9.0 Web Service, as described in the following steps.

   c. Add a `mkdir` task to create a *`temp_dir`*`/jar/META-INF` directory, where *`temp_dir`* refers to a temporary output directory to hold the artifacts generated by `jwsc` until they are ready to be packaged into an EJB JAR file.

   d. Add the `jwsc` task to the build file. Set the `source` attribute to the name of the JWS file you created in the preceding steps and the `destdir` attribute to the *`temp_dir`*`/jar/META-INF` directory.

      You may need to specify additional attributes to the `jwsc` task, depending on the 8.1 Web Service features you want to carry forward to 9.0. In 8.1, many of these features were configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations" on page 12-19 for a table that describes if there is an equivalent `jwsc` attribute for features you enabled using `servicegen` attributes.

   e. Add `javac` tasks to compile the JWS file and the Java files generated by the `jwsc` Ant task. Set the `destdir` attribute to the *`temp_dir`*`/jar` directory.

f. Add a `jar` task to package the files under the *`temp_dir`*`/jar` directory into an EJB JAR file.

5. Execute the `build-service` Ant target. Assuming all the tasks complete successfully, the resulting JAR file is your upgraded 9.0 Web Service.

See "Deploying and Undeploying WebLogic Web Services" on page 4-9 and "Invoking the WSDL and Home Page of the Web Service" on page 4-11 for additional information about deploying and testing your Web Service.

Example of an 8.1 and Updated 9.0 EJB Classes

Assume that the Bean, Home, and Remote classes and interfaces, shown in the next three sections, implemented the 8.1 stateless session EJB which in turn implemented an 8.1 Web Service. The equivalent 9.0 Bean class is shown in "Equivalent 9.0 SessionBean Class File That Uses EJBGen and JWS Annotations" on page 12-14. The differences between the 8.1 and 9.0 classes are shown in bold. Note that some of the JWS annotation values are taken from attributes of the 8.1 `servicegen` Ant task shown in "Example of an 8.1 and Updated 9.0 Ant Build File For EJB-Implemented Web Services" on page 12-16.

8.1 SessionBean Class

```
package examples.statelessSession;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

/**
 * HelloWorldBean is a stateless session EJB.  It has a single method,
 * sayHello(), that takes an integer and a String and returns a String.
 * <p>
 * The sayHello() method is the public operation of the Web service based on
 * this EJB.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class HelloWorldBean81 implements SessionBean {

  private static final boolean VERBOSE = true;
  private SessionContext ctx;

  // You might also consider using WebLogic's log service
  private void log(String s) {
```

```
    if (VERBOSE) System.out.println(s);
  }

  /**
   *   Single EJB business method.
   */
  public String sayHello(int num, String s) {

    System.out.println("sayHello in the HelloWorld EJB has "+
      "been invoked with arguments " + s + " and " + num);

    String returnValue = "This message brought to you by the "+
      "letter "+s+" and the number "+num;

    return returnValue;
  }

  /**
   * 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 is required by the EJB Specification,
   * but is not used by this example.
   *
   */
  public void ejbCreate () throws CreateException {
    log("ejbCreate called");
  }

}
```

8.1 Remote Interface

```
package examples.statelessSession;

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

/**
 * The methods in this interface are the public face of HelloWorld.
 * The signatures of the methods are identical to those of the EJBean, except
 * that these methods throw a java.rmi.RemoteException.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public interface HelloWorld81 extends EJBObject {

  /**
   * Simply says hello from the EJB
   *
   * @param num              int number to return
   * @param s                String string to return
   * @return                 String returnValue
   * @exception              RemoteException if there is
   *                         a communications or systems failure
   */
  String sayHello(int num, String s)
    throws RemoteException;
}
```

8.1 EJB Home Interface

```
package examples.statelessSession;
```

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

/**
 * This interface is the Home interface of the HelloWorld stateless session EJB.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public interface HelloWorldHome81 extends EJBHome {

  /**
   * This method corresponds to the ejbCreate method in the
   * HelloWorldBean81.java file.
   */
  HelloWorld81 create()
    throws CreateException, RemoteException;

}
```

Equivalent 9.0 SessionBean Class File That Uses EJBGen and JWS Annotations

The differences between the 8.1 and 9.0 classes are shown in bold; these differences include use of EJBGen and JWS annotations. Note that the value of some of the JWS annotations are taken from attributes of the 8.1 `servicegen` Ant task shown in "Example of an 8.1 and Updated 9.0 Ant Build File For EJB-Implemented Web Services" on page 12-16

```
package examples.upgradeEJBWS;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

import com.bea.wls.ejbgen.annotations.*;

import weblogic.jws.WLHttpTransport;

@Session(ejbName="HelloWorld90",
         serviceEndpoint="examples.upgradeEJBWS.HelloWorldBean90PortType")

@WebService(serviceName="HelloWorldEJB",
            targetNamespace="http://example.org")

@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.ENCODED)
```

```
@WLHttpTransport(contextPath="jwsEJB", serviceUri="HelloWorldEJB")

/**
 * HelloWorldBean90 is a stateless session EJB.  It has a single method,
 * sayHello(), that takes an integer and a String and returns a String.
 * <p>
 * The sayHello() method is the public operation of the Web service based on
 * this EJB.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class HelloWorldBean90 implements SessionBean {

  private static final boolean VERBOSE = true;
  private SessionContext ctx;

  // You might also consider using WebLogic's log service
  private void log(String s) {
    if (VERBOSE) System.out.println(s);
  }

  /** the sayHello method will become the public operation of the Web
   *  Service.
   */

  @WebMethod()
  public String sayHello(int num, String s) {

    System.out.println("sayHello in the HelloWorld EJB has "+
      "been invoked with arguments " + s + " and " + num);

    String returnValue = "This message brought to you by the "+
      "letter "+s+" and the number "+num;

    return returnValue;
  }

  /**
   * 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 is required by the EJB Specification,
   * but is not used by this example.
   *
   */
  public void ejbCreate () throws CreateException {
    log("ejbCreate called");
  }

}
```


Example of an 8.1 and Updated 9.0 Ant Build File For EJB-Implemented Web Services

The following simple `build.xml` file shows the 8.1 way to build an EJB-implemented WebLogic Web Service using the `servicegen` Ant task:

```
<project name="ejb-webservice" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property name="source" value="."/>
  <property name="build" value="${source}/build"/>
  <property name="ejb_file" value="HelloWorldWS.jar" />
```

```
<property name="war_file" value="HelloWorldWS.war" />
<property name="ear_file" value="HelloWorldApp.ear" />
<property name="namespace" value="http://examples.org" />

<target name="all" depends="clean,ear"/>

<target name="clean">
  <delete dir="${build}"/>
</target>

<!-- example of old 8.1 servicegen call to build Web Service -->

<target name="ejb">
  <delete dir="${build}" />
  <mkdir dir="${build}"/>
  <mkdir dir="${build}/META-INF"/>
  <copy todir="${build}/META-INF">
    <fileset dir="${source}">
      <include name="ejb-jar.xml"/>
    </fileset>
  </copy>
  <javac srcdir="${source}" includes="HelloWorld*.java"
         destdir="${build}" />
  <jar jarfile="${ejb_file}" basedir="${build}" />
  <wlappc source="${ejb_file}" />
</target>

<target name="ear" depends="ejb">
  <servicegen
    destEar="${build}/${ear_file}"
    warName="${war_file}">
    <service
      ejbJar="${ejb_file}"
      targetNamespace="${namespace}"
      serviceName="HelloWorldEJB"
      serviceURI="/HelloWorldEJB"
      generateTypes="True"
      expandMethods="True">
    </service>
  </servicegen>
</target>
```

```
</project>
```

An equivalent `build.xml` file that calls the `jwsc` Ant task to build a 9.0 EJB-implemented Web Service is shown below, with the relevant tasks in bold:

```
<project name="ejb-webservice" default="all" basedir=".">

  <!-- set global properties for this build -->
  <property name="source" value="."/>
  <property name="build" value="${source}/build"/>
  <property name="ejb_file" value="HelloWorldWS.jar" />
  <property name="war_file" value="HelloWorldWS.war" />
  <property name="ear_file" value="HelloWorldApp.ear" />
  <property name="namespace" value="http://examples.org" />

  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

  <target name="all" depends="clean, build-service"/>

  <target name="clean">
    <delete dir="${build}"/>
  </target>

  <target name="build-service">

    <mkdir dir="${build}/jar/META-INF" />

    <jwsc source="HelloWorldBean90.java"
          destdir="${build}/jar/META-INF" />

    <javac
      srcdir="${build}/jar/META-INF/"
      source="1.5"
      destdir="${build}/jar/"
      includes="*.java"/>

    <javac
      srcdir="."
      source="1.5"
      destdir="${build}/jar/"
      includes="*.java"/>

    <jar destfile="${build}/${ejb_file}"
      basedir="${build}/jar"/>
```

```
  </target>
</project>
```

Mapping of servicegen Attributes to JWS Annotations

The following table maps the attributes of the `servicegen` Ant task to their equivalent JWS annotation or `jwsc` attribute.

The attributes listed in the first column are a mixture of attributes of the main `servicegen` Ant task and attributes of the four child elements of `servicegen` (`<service>`, `<client>`, `<handlerChain>`, and `<security>`).

Table 12-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`contextURI`	`contextPath` attribute of the WebLogic-specific `@WLHttpTransport` annotation.
`destEAR`	No direct equivalent because the `jwsc` Ant task does not package Web Services into archive files. Indirect equivalent is the `destDir` attribute of the `jwsc` Ant task which specifies the destination directory of generated Java code and other artifacts.
`keepGenerated`	No equivalent.
`mergeWithExistingWS`	No equivalent.
`overwrite`	`overwrite` attribute of the `jwsc` Ant task.
`warName`	No equivalent because the `jwsc` Ant task does not package Web Services into archives. That task is left up to the developer.
`ejbJAR` (attribute of the `service` child element)	No direct equivalent, because the `jwsc` Ant task generates Web Service artifacts from a JWS file, rather than a compiled EJB or Java class. Indirect equivalent is the `source` attribute of the `jwsc` Ant task that specifies the name of the JWS file.
`excludeEJBs` (attribute of the `service` child element)	No equivalent.

Table 12-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`expandMethods` (attribute of the `service` child element)	No equivalent.
`generateTypes` (attribute of the `service` child element)	No equivalent.
`ignoreAuthHeader` (attribute of the `service` child element)	No equivalent.
`includeEJBs` (attribute of the `service` child element)	No equivalent.
`javaClassComponents` (attribute of the `service` child element)	No direct equivalent, because the `jwsc` Ant task generates Web Service artifacts from a JWS file, rather than a compiled EJB or Java class. Indirect equivalent is the `source` attribute of the `jwsc` Ant task that specifies the name of the JWS file.
`JMSAction` (attribute of the `service` child element)	No equivalent.
`JMSConnectionFactory` (attribute of the `service` child element)	No equivalent.
`JMSDestination` (attribute of the `service` child element)	No equivalent.
`JMSDestinationType` (attribute of the `service` child element)	No equivalent.

Table 12-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`JMSMessageType` (attribute of the `service` child element)	No equivalent.
`JMSOperationName` (attribute of the `service` child element)	No equivalent.
`protocol` (attribute of the `service` child element)	`https` attribute of the WebLogic-specific `@WLHttpTransport` annotation VERIFY THAT THIS INDEED GETS IMPLEMENTED AS I INDICATE (SEE CR201513)
`serviceName` (attribute of the `service` child element)	`serviceName` attribute of the standard `@WebService` annotation.
`serviceURI` (attribute of the `service` child element)	`serviceUri` attribute of the WebLogic-specific `@WLHttpTransport` annotation.
`style` (attribute of `service` child element)	`style` attribute of the standard `@SOAPBinding` annoation.
`typeMappingFile` (attribute of the `service` child element)	No equivalent.
`targetNamespace` (attribute of the `service` child element)	`targetNamespace` attribute of the standard `@WebService` annotation.
`userSOAP12` (attribute of the `service` child element)	No equivalent.
`clientJarName` (attribute of `client` child element)	No equivalent.

Table 12-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`packageName` (attribute of the `client` child element)	No direct equivalent. Use the `packageName` attribute of the `clientgen` Ant task to generate client-side Java code and artifacts.
`saveWSDL` (attribute of the `client` child element)	No equivalent.
`userServerTypes` (attribute of the `client` child element)	No equivalent.
`handlers` (attribute of the `handlerChain` child element)	Standard `@HandlerChain` or `@SOAPMessageHandlers` annotation.
`name` (attribute of the `handlerChain` child element)	Standard `@HandlerChain` or `@SOAPMessageHandlers` annotation.
`duplicateElimination` (attribute of the `reliability` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains reliable SOAP messaging policy assertions. See "Using Reliable SOAP Messaging" on page 6-1.
`persistDuration` (attribute of the `reliability` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains reliable SOAP messaging policy assertions. See "Using Reliable SOAP Messaging" on page 6-1.

Table 12-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`enablePasswordAuth` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2.
`encryptKeyName` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2
`encryptKeyPass` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2
`password` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2

Table 12-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`signKeyName` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2
`signKeyPass` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2
`username` (attribute of the `security` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains message-level security policy assertions. See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 9-2

APPENDIX A

Ant Task Reference

The following sections provide information about the WebLogic Web Services Ant tasks:

Overview of WebLogic Web Services Ant Tasks

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 Web Service-related artifacts.

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/.

List of Web Services Ant Tasks

The following table provides an overview of the Web Service Ant tasks provided by BEA.

Table A-1 WebLogic Web Services Ant Tasks

Ant Task	Description
clientgen	Generates the JAX-RPC Service stubs and other client-side files used to invoke a Web Service.
jwsc	Compiles a JWS-annotated file into a standard non-annotated Java source file.
wsdl2service	Generates a partial Web Service implementation based on a WSDL file.

Using the Web Services Ant Tasks

To use the Ant tasks, follow these steps:

1. Set your environment.

   On Windows NT, execute the `setDomainEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setDomainEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a file called `build.xml` that will contain a call to the Web Services Ant tasks.

   The following example shows a simple `build.xml` file with a single target called `clean`:

```
<project name="buildWebservice">

<target name="clean">
    <delete>
      <fileset dir="." />
    </delete>
  </target>

</project>
```

   Later sections provide examples of specifying the Ant task in the `build.xml` file.

3. For each WebLogic Web Service Ant task you want to execute, add an appropriate task definition and target to the `build.xml` file using the `<taskdef>` and `<target>` elements. The following example shows how to add the `jwsc` Ant task to the build file; the attributes of the task have been removed for clarity:

```
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

<target name="compile-JWS">
    <jwsc attributes go here...>
    ...
    </jwsc>
</target>
```

   You can, of course, name the WebLogic Web Services Ant tasks anything you want by changing the value of the `name` attribute of the relevant `<taskdef>` element. For consistency, however, this document uses the names `jwsc`, `clientgen`, `wsdl2service`, `java2schema`, and `schema2java` throughout.

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

```
prompt> ant compile-JWS
```

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 `jwsc` Ant task to add a new directory to the CLASSPATH variable:

```
<jwsc source="MyJWSFile.java"
            classpath="${java.class.path};my_fab_directory"
   ...
</jwsc>
```

The following example shows how to add to the CLASSPATH by using the `<classpath>` element:

```
<jwsc ...>
   <classpath>
       <pathelement path="${java.class.path}" />
       <pathelement path="my_fab_directory" />
   </classpath>
```

```
...
</jwsc>
```

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="myClassID">
   <pathelement path="${java.class.path}"/>
   <pathelement path="${additional.path1}"/>
   <pathelement path="${additional.path2}"/>
</path>

<jwsc ....>
   <classpath refid="myClassID" />
...
</jwsc>
```

Warning: The WebLogic Web Services Ant tasks support the standard Ant property `build.sysclasspath`. The default value for this property is `ignore`. This means that if you specifically set the CLASSPATH in the `build.xml` file as described in this section, the Ant task you want to run ignores the system CLASSPATH (or the CLASSPATH in effect when Ant is run) and uses only the one that you specifically set. It is up to you to include in your CLASSPATH setting all the classes that the Ant task needs to successfully run. To change this default behavior, set the `build.sysclasspath` property to `last` to concatenate the system CLASSPATH to the end of the one you specified, or `first` to concatenate your specified CLASSPATH to the end of the system one.

For more information on the `build.sysclasspath` property, see the Ant documentation.

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 to set various Ant-specific variables, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation If you need to update these Ant variables, 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.

The Ant tasks process these files in a case-sensitive way. This means that if, for example, the XML Schema file specifies two user-defined 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 user-defined 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.

clientgen

The `clientgen` Ant task generates, from an existing WSDL file, the client component files that client applications use to invoke both WebLogic and non-WebLogic Web Services. These files include:

- The Java source code for the JAX-RPC `Stub` and `Service` interface implementations for the particular Web Service you want to invoke.
- The Java source code for any user-defined XML Schema data types included in the WSDL file.

- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

Two types of client applications use the generated artifacts of `clientgen` to invoke Web Services:

- Standalone Java clients that do not use the J2EE client container.
- J2EE clients, such as EJBs, JSPs, and Web Services, that use the J2EE client container.

Taskdef Classname

```
<taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
```

Example

```
<taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

...

<target name="build_client">

<clientgen
    wsdl="http://example.com/myapp/myservice.wsdl"
    destDir="/output/clientclasses"
    packageName="myapp.myservice.client"
    serviceName="StockQuoteService" />

<javac ... />

</target>
```


Attributes

The following table describes the attributes of the `clientgen` Ant task.

Table 12-2 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
destDir	Directory into which the `clientgen` Ant task generates the client source code, WSDL, and client deployment descriptor files. You can set this attribute to any directory you want. However, if you are generating the client component files to invoke a Web Service from an EJB, JSP, or other Web Service, you typically set this attribute to the corresponding META-INF or WEB-INF directory. If you are invoking the Web Service from a standalone client, then you can generate the client component files into the same source code directory hierarchy as your client application code.	String	Yes.
generatePolicyMethods	Specifies whether the `clientgen` Ant task should include policy-loading methods in the generated JAX-RPC stubs. These methods can be used by client applications to load a local policy statement. If you specify `True`, four flavors of a method called `get`*`XXX`*`SoapPort()` are added as extensions to the JAX-RPC `Service` interface in the generated client stubs, where *`XXX`* refers to the name of the Web Service. Client applications can use these methods to load and apply local policy statements, rather than apply any policy statements deployed with the Web Service itself. Client applications can specify whether the local policy statement applies to inbound, outbound, or both SOAP messages and whether to load the local policy from an InputStream or a URI. Valid values for this attribute are `True` or `False`. The default value is `False`, which means the additional methods are *not* generated.	Boolean	No.

Table 12-2 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
overwrite	Specifies whether the client component files (source code, WSDL, and deployment descriptor files) generated by this Ant task should be overwritten if they already exist. If you specify `True`, new artifacts are always generated and any existing artifacts are overwritten. If you specify `False`, the Ant task overwrites only those artifacts that have changed, based on the timestamp of any existing artifacts. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.
packageName	Package name into which the generated JAX-RPC client interfaces and stub files should be packaged.	String	Yes.

Table 12-2 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
serviceName	Name of the Web Service in the WSDL file for which the corresponding client component files should be generated. The Web Service name corresponds to the `<service>` element in the WSDL file. The generated JAX-RPC mapping file and client-side copy of the WSDL file will use this name. For example, if you set `serviceName` to `cuteService`, the JAX-RPC mapping file will be called `cuteService.xml` and the client-side copy of the WSDL will be called `cuteService.wsdl`.	String	This attribute is required *only* if the WSDL file contains more than one `<service>` element. The Ant task returns an error if you do not specify this attribute and the WSDL file contains more than one `<service>` element.
wsdl	Full path name or URL of the WSDL that describes a Web Service (either WebLogic or non-WebLogic) for which the client component files should be generated. The generated stub factory classes in the client JAR file use the value of this attribute in the default constructor.	String	Yes.

jwsc

The `jwsc` Ant task takes as input a JWS file that contains both standard (JSR-181) and WebLogic-specific JWS annotations and generates all the artifacts you need to create a WebLogic Web Service. These generated artifacts include:

- Java source files that implement a standard JSR-921 Web Service. The generated source files include the service endpoint interface (called *`JWS_ClassName`*`PortType.java`, where *`JWS_ClassName`* refers to the JWS class) and the various EJB-related Java files, such as the `Home` and `Remote` interfaces, in the case of an EJB-implemented Web Service.
- All required deployment descriptors. In addition to the standard `webservices.xml` and JAX-RPC mapping files, the `jwsc` Ant task also generates the WebLogic-specific Web

Services deployment descriptor (`weblogic-wesbservices.xml`), the `web.xml` and `weblogic.xml` files for Java class-implemented Web Services and the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files for EJB-implemented Web Services.

- The XML Schema representation of any Java user-defined types used as parameters or return values to the methods of the JWS files that are specified to be exposed as public operations.
- The WSDL file that publicly describes the Web Service.

After generating all the artifacts, you compile the Java source files into classes, package all the artifacts into their correct archive (a JAR file for EJB-implemented Web Services and a WAR file for Java class-implemented Web Services), and deploy the archive as usual.

When deciding where to specify the `jwsc` Ant task to generate artifacts, consider specifying a directory that can be easily packaged into an EJB or WAR file, depending on the type of implementation you have chosen for your Web Service. For example, set the `destDir` attribute to `/output/META-INF` when creating an EJB-implemented Web Service so as to generate the deployment descriptors into the corerct directory, compile the Java source code into a directory hierarchy parallel with META-INF, and then you can easily use the `jar` Ant task on the root directory to create an EJB JAR file.

Taskdef Classname

```
<taskdef name="jwsc"
      classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />
```

Example

```
<taskdef name="jwsc"
      classname="weblogic.wsee.tools.anttasks.JWSWSEEGenTask" />

<target name="build_service">

    <jwsc
      source="SimpleBean.java"
      destdir="/output/META-INF/"
      srcPath="/src/myWebService"
    />

    <javac ... />

</target>
```

Attributes

The following table describes the attributes of the `jwsc` Ant task.

Table 12-3 Attributes of the jwsc Ant Task

Attribute	Description	Data Type	Required?
destDir	The full pathname of the directory that will contain the compiled JWS files, XML Schemas, WSDL, and generated deployment descriptor files. Consider generating compiled source code into a directory hierarchy that can be easily packaged into an EJB or WAR file, depending on the type of implementation you have chosen for your Web Service. For example, set the `destDir` attribute to `/output/META-INF` when creating an EJB-implemented Web Service to as to generate the deployment descriptors into the corerct directory. Compile the Java source code into a directory hierarchy parallel with META-INF.	String	Yes.
overwrite	Specifies whether the Java source files and artifacts generated by this Ant task should be overwritten if they already exist. If you specify `True`, new Java source files and artifacts are always generated and any existing artifacts are overwritten. If you specify `False`, the Ant task overwrites only those artifacts that have changed, based on the timestamp of any existing artifacts. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.
source	Name of the JWS file that forms the basis of the Web Service and from which the Java source files and artifacts are generated.	String	Yes.

Table 12-3 Attributes of the jwsc Ant Task

Attribute	Description	Data Type	Required?
srcPath	The full pathname of the root directory, or the top-level directory within a package hierarchy, that contains the Java source for any user-defined data types used by the JWS file pointed to by the `source` attribute. For example, if your JWS file uses the `types.complex.myDatatype` user-defined data type, and the Java source for this data type is stored in the `/source/types/complex` directory, then set the `srcPath` attribute to `/source`.	String	You must specify this attribute if you use user-defined data types in your JWS file.
verbose	Enables verbose output for debugging purposes. Valid values for this attribute are `True` or `False`. The default value is `False`, which means verbose output is not enabled.	Boolean	No.
wsdlOnly	Specifies that *only* a WSDL file should be generated. **Note:** Although the other artifacts, such as the deployment descriptors and service endpoint interface, are not generated, data binding artifacts *are* generated because the WSDL must include the XML Schema that describes the data types of the parameters and return values of the Web Service operations. The WSDL is generated into the `destDir` directory. The name of the file is *`JWS_ClassName`*`Service.wsdl`, where *`JWS_ClassName`* refers to the name of the JWS class. *`JWS_ClassNameService`* is also the name of Web Service in the generated WSDL file. Valid values for this attribute are `True` or `False`. The default value is `False`, which means that all artifacts are generated by default, not just the WSDL file.	Boolean.	No.

wsdl2service

The `wsdl2service` Ant task generates, from an existing WSDL file, almost all the artifacts you need to create and deploy a Java-class implemented WebLogic Web Service that implements the specifications of the WSDL file.

In particular, the Ant task takes as input a WSDL file and generates:

- the interface file that represents the Java-class implementation of your Web Service that runs in the Web container of WebLogic Server.

  The name of the file is *ServiceName*`PortType.java`, where *ServiceName* is the name of the Web Service (in particular, the `name` attribute of the `<service>` element in the WSDL file.)

- the Java exception class for user-defined exceptions specified in the WSDL file.
- the `webservices.xml`, `web.xml`, and `weblogic.xml` deployment descriptor files needed to deploy the Java-class implemented Web Service.
- The JAX-RPC mapping files that describe the mapping between Java and XML of the data types used in the Web Service operations.
- The Java representation of any user-defined data types used in the operations of the Web Service, based on the XML Schema representation in the WSDL file.

The generated Java interface file describes the 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 must then write your own Java class that implements this interface so that the methods function as you want.

The `wsdl2Service` Ant task generates a Java interface for only one Web Service in a particular WSDL file. Use the `serviceName` attribute of the Ant task to specify the exact service, which corresponds to the `name` attribute of the `<service>` element in the WSDL file.

You can, however, generate Java implementations for more than one WSDL for a particular `wsdl2service` execution by specifying multiple `<wsdl>` child elements to the `<wsdl2service>` task.

Once you have written the Java implementation class and coded all the business logic needed to make the Web Service work as you want, use the `jar` Ant task to package all the artifacts into a single WAR file that can be deployed on WebLogic Server. If you specified more than one WSDL file, all the Web Services will be packaged into the same WAR file and deployed using the same context URI.

The `wsdl2service` Ant task generates the interface files into a directory hierarchy (corresponding to the specified package name) under the `destDir` directory. For example, if you specify a package name of `my.package` and the destination directory of `/output/war/WEB-INF`, the interface files are generated into the `/output/war/WEB-INF/my/package` directory

Taskdef Classname

```
<taskdef name="wsdl2service"
    classname="weblogic.wsee.tools.anttasks.Wsdl2ServiceAntTask" />
```

Example

```
<taskdef name="wsdl2service"
  classname="weblogic.wsee.tools.anttasks.Wsdl2ServiceAntTask" />

...

<target name="build_service">

<wsdl2service destdir="/output/war/WEB-INF/"
  verbose="true" contextpath="/wsdl2service">

    <wsdl location="/wsdls/TemperatureService.wsdl"
      packageName="examples.wsdl2service"
      impl="examples.wsdl2service.TemperatureImpl"
      uri="/temperature" />

    <wsdl location="/wsdls/CurrencyExchangeService.wsdl"
      packageName="examples.wsdl2service"
      impl="examples.wsdl2service.CurrencyExchangeImpl"
      uri="/currency" />

</wsdl2service>

<javac ... />

</target>
```


The example shows how to create the needed artifacts for two Web Services in two different WSDL files. The implementation Java source files will be generated into the directory `/output/war/WEB-INF/examples/wsdl2service`, and they will be called *`ServiceName`*`PortType.java`, such as `TemperatureServicePortType.java`. The implementation classes will be called `TemperatureImpl` and `CurrencyExchangeImpl` and they will both be in the package `examples.wsdl2service`.

Attributes

The `wsdl2service` Ant task has three attributes and one child element: `<wsdl>`.

The following table describes the attributes of the `wsdl2service` Ant task.

Table 12-4 Attributes of the wsdl2service Ant Task

Attribute	Description	Data Type	Required?
contextPath	Context root of the Web Service. You use this value in the URL that invokes the Web Service.	String	No.
destDir	The full pathname of the directory that will contain the generated artifacts. Consider generating the artifacts into a directory hierarchy that can be easily packaged into a WAR file. For example, set the `destDir` attribute to `/output/war/WEB-INF`.	String	Yes.
verbose	Enables verbose output for debugging purposes. Valid values for this attribute are `True` or `False`. The default value is `False`, which means verbose output is not enabled.	Boolean	No.

wsdl Child Element

Use the `<wsdl>` child element of the `wsdl2service` Ant task to specify the WSDL files from which the needed artifacts are generated. You must specify at least one `<wsdl>` element, and can specify as many as you want.

The following table describes the attributes of the `wsdl` child element of the `wsdl2service` Ant task.

Table 12-5 Attributes of the wsdl Child Element of the wsdl2service Ant Task

Attribute	Description	Data Type	Required?
location	The full path name or URL of the WSDL that describes a Web Service for which the artifacts needed to create a WebLogic Web Service implementation will be generated.	String	Yes.
packageName	The package name of the generated Java interface file that represents the implementation of your Web Service.	String	Yes.

Table 12-5 Attributes of the wsdl Child Element of the wsdl2service Ant Task

Attribute	Description	Data Type	Required?
uri	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: `http(s)://host:port/contextPath/uri` where • *`host`* refers to the computer on which WebLogic Server is running • *`port`* refers to the port on which WebLogic Server is listening • *`contextPath`* refers to the `contextPath` attribute of the main `wsdl3service` Ant task • *`uri`* refers to this attribute	String	Yes.
impl	The full name of the Java class that implements the generated interface.	String	Yes.
serviceName	The name of the Web Service in the WSDL file for which `wsdl2service` generates artifacts. 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 artifacts for the first `<service>` element it finds in the WSDL file. **Note:** The `wsdl2Service` Ant task generates artifacts for only *one* service in a WSDL file. If your WSDL file contains more than one Web Service, and you want to create a Java implementation for each service, then you must run `wsdl2Service` multiple times, changing the value of this attribute each time.	String	No.

JWS Annotation Reference

The following sections provide reference documentation about standard (JSR-181) and WebLogic-specific JWS annotations:

Overview of JWS Annotation Tags

The WebLogic Web Services programming model uses the new JDK 5.0 metadata annotations feature (specified by JSR-175). In this programming model, you create an annotated Java file and then use Ant tasks to compile the file into the Java source code and generate all the associated artifacts.

The Java Web Service (JWS) annotated file is the core of your Web Service. It contains the Java code that determines how your Web Service behaves. A JWS file is an ordinary Java class file that uses annotations to specify the shape and characteristics of the Web Service. The JWS annotations you can use in a JWS file include the standard ones defined by the *Web Services Metadata for the Java Platform* specification (JSR-181) as well as a set of WebLogic-specific ones. This chapter provides reference information about both of these set of annotations.

You can target a JWS annotation at either the class-, method- or parameter-level in a JWS file. Some annotations can be targetted at more than one level, such as `@SecurityRoles` that can be targetted at both the class- and method-level. The documentation in this section lists the level to which you can target each annotation.

The following example shows a simple JWS file that uses both standard JSR-181 and WebLogic-specific JWS annotations, shown in bold:

```
package examples.ejbJWS;

import java.rmi.RemoteException;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

// Import the standard JWS annotation interfaces

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

// Import the EJBGen annotation interfaces

import com.bea.wls.ejbgen.annotations.*;

// Import the WebLogic-specific JWS annotation interface called WLHttpTransport

import weblogic.jws.WLHttpTransport;

// EJBGen annotation that specifies that EJBGen should generate a stateless
// session EJB called Simple and its generated JAX-RPC Web service endpoint
// interface should be called "examples.ejbJWS.SimpleBeanPortType".

@Session(ejbName="Simple",
         serviceEndpoint="examples.ejbJWS.SimpleBeanPortType")

// Standard JWS annotation that specifies that the name of the Web Service is
// "myPortType", its public service name is "SimpleEjbService", and the
// targetNamespace used in the generated WSDL is "http://example.org"

@WebService(name="myPortType",
            serviceName="SimpleEjbService",
            targetNamespace="http://example.org")

// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are rpc-encoded.

@SOAPBinding(style=SOAPBinding.Style.RPC,
             use=SOAPBinding.Use.ENCODED)

// WebLogic-specific JWS annotation that specifies the port name is helloPort,
// and the context path and service URI used to build the URI of the Web
// Service is "ejbJWS/SimpleBean"
```

```
@WLHttpTransport(portName="helloPort",
                 contextPath="ejbJWS",
                 serviceUri="SimpleBean")

/**
 * This JWS file forms the basis of a stateless-session EJB implemented
 * WebLogic Web Service.
 *
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class SimpleBean implements SessionBean {

  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: echoString.

  @WebMethod()
  public String echoString(String input) throws RemoteException {
    System.out.println("echoString got " + input);
    return input;
  }

  // Standard EJB methods.  Typically there's no need to override the methods.

  public void ejbCreate() {}
  public void ejbActivate() {}
  public void ejbRemove() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {}
}
```


Standard JSR-181 JWS Annotations Reference

The Web Services Metadata for the Java Platform (JSR-181) specification defines the standard annotations you can use in your JWS file to specify the shape and behavior of your Web Service. This section briefly describes each annotation, along with its attributes. See Chapter 5, "Programming the JWS File," for examples. For more detailed information about the annotations, such as the Java annotation type definition and additional examples, see the specification.

This section documents the following standard JWS anntoations:

- javax.jws.WebService
- javax.jws.WebMethod

- javax.jws.Oneway
- javax.jws.WebParam
- javax.jws.WebResult
- javax.jws.HandlerChain
- javax.jws.soap.SOAPBinding
- javax.jws.soap.SOAPMessageHandler
- javax.jws.soap.InitParam
- javax.jws.soap.SOAPMessageHandlers
- javax.jws.security.SecurityRoles
- javax.jws.security.SecurityIdentity

javax.jws.WebService

Description

Target: Class

Specifies that a Java class or stateless session EJB implements a Web Service.

Attributes

Table 12-6 Attributes of the javax.jws.WebService JWS Annotation

Name	Description	Data Type	Required?
name	Name of the Web Service. Maps to the `<wsdl:portType>` element in the WSDL file. Default value is the unqualified name of the Java class or stateless session EJB.	String	No.
targetNamespace	The XML namespace used for the WSDL and XML elements generated from this Web Service. The default value is specified by the JAX-RPC specification.	String.	No.

Table 12-6 Attributes of the javax.jws.WebService JWS Annotation

Name	Description	Data Type	Required?
serviceName	Service name of the Web Service. Maps to the `<wsdl:service>` element in the WSDL file. Default value is the unqualified name of the Java class or stateless session EJB appended with the string `Service`.	String	No.
wsdlLocation	Relative or absolute URL of a pre-defined WSDL file. If you specify this attribute, the jwsc Ant task does not generate a WSDL file, and returns an error if the JWS file is inconsistent with the port types and bindings in the WSDL file.	String.	No.
endpointInterface	Fully qualified name of an existing service endpoint interface file. If you specify this attribute, the jwsc Ant task does not generate the interface for you, but assumes you have already created it and it is in your CLASSPATH.	String.	No.

javax.jws.WebMethod

Description

Target: Method

Specifies that the method is exposed as a public operation of the Web Service. You must explicitly use this annotation to expose a method; if you do not specify this annotation, the method by default is not exposed.

Attributes

Table 12-7 Attributes of the javax.jws.WebMethod JWS Annotation

Name	Description	Data Type	Required?
operationName	Name of the operation. Maps to the `<wsdl:operation>` element in the WSDL file. Default value is the name of the method.	String	No.
action	The action for this operation. For SOAP bindings, the value of this attribute determines the value of the `SOAPAction` header in the SOAP messages.	String	No.

javax.jws.Oneway

Description

Target: Method

Specifies that the method has only input parameters, but does not return a value. This annotation must be used only in conjunction with the `@WebMethod` annotation.

It is an error to use this annotation on a method that returns anything other than `void` or takes a `Holder` class as an input parameter.

This annotation does not have any attributes.

javax.jws.WebParam

Description

Target: Parameter

Customizes the mapping between operation input parameters of the Web Service and elements of the generated WSDL file. Also used to specify the behavior of the parameter.

Attributes

Table 12-8 Attributes of the javax.jws.WebParam JWS Annotation

Name	Description	Data Type	Required?
name	Name of the parameter in the WSDL file. For RPC-style Web Services, the name maps to the `<wsdl:part>` element that represents the parameter. For document-style Web Services, the name is the local name of the XML element that respresents the parameter. The default value is the name of the method's parameter.	String	No.
targetNamespace	The XML namespace of the parameter. This value is used only used for document-style Web Services, in which the parameter maps to an XML element. The default value is the targetNamespace of the Web Service.	String	No.
mode	The direction in which the parameter is flowing. Valid values are IN, OUT, and INOUT. Default value is IN. If you specify OUT or INOUT, then the data type of the parameter must be `Holder`, or extend `Holder`. For details, see the JAX-RPC specification. OUT and INOUT modes are only supported for RPC-style Web Services or for parameters that map to headers.	String	No.
header	Specifies whether the value of the parameter is found in the SOAP header. By default parameters are in the SOAP body. Valid values are `true` and `false`. Default value is `false`.	Boolean	No.

javax.jws.WebResult

Description

Target: Parameter

Customizes the mapping between the Web Service operation return value and the corresponding element of the generated WSDL file. Also used to specify the behavior of the return value.

Attributes

Table 12-9 Attributes of the javax.jws.WebResult JWS Annotation

Name	Description	Data Type	Required?
name	Name of the parameter in the WSDL file. For RPC-style Web Services, the name maps to the `<wsdl:part>` element that represents the return value. For document-style Web Services, the name is the local name of the XML element that respresents the return value. The default value is the hard-coded name `result`.	String	No.
targetNamespace	The XML namespace of the return value. This value is used only used for document-style Web Services, in which the return value maps to an XML element. The default value is the targetNamespace of the Web Service.	String	No.

javax.jws.HandlerChain

Description

Target: Class

Associates a Web Service with an external file that contains the configuration of a handler chain. The configuration includes the list of handlers in the chain, the order in which they execute, the initialization parameters, and so on.

Use the `@HandlerChain` annotation, rather than the `@SOAPMessageHandlers` annotation, in your JWS file if:

- You want multiple Web Services to share the same configuration.
- Your handler chain includes handlers for multiple transports.

- You want to be able to change the handler chain configuration for a Web Service without recompiling the JWS file that implements it.

It is an error to combine this annotation with the `@SOAPMessageHandlers` annotation.

For the XML Schema of the external configuration file, additional information about creating it, and additional examples, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

Attributes

Table 12-10 Attributes of the javax.jws.HandlerChain JWS Annotation

Name	Description	Data Type	Required?
file	URL, either relative or absolute, of the handler chain configuration file. Relative URLs are relative to the location of JWS file.	String	Yes
name	Name of the handler chain (in the configuration file pointed to by the `file` attribute) that you want to associate with the Web Service.	String	Yes.

javax.jws.soap.SOAPBinding

Description

Target: Class

Specifies the mapping of the Web Service onto the SOAP message protocol.

Attributes

Table 12-11 Attributes of the javax.jws.soap.SOAPBinding JWS Annotation

Name	Description	Data Type	Required?
style	Specifies the encoding style of the request and response SOAP messages. Valid values are `SOAPBinding.Style.RPC` and `SOAPBinding.Style.Document`. Default value is `SOAPBinding.Style.Document`.	String	No.
use	Specifies the formatting style of the request and response SOAP messages. Valid values are `SOAPBinding.Use.Literal` and `SOAPBinding.Use.Encoded`. Default value is `SOAPBinding.Use.Literal`.	String	No.
parameterStyle	Determines whether method parameters represent the entire message body, or whether the parameters are elements wrapped inside a top-level element named after the operation. Valid values are `SOAPBinding.ParameterStyle.Bare` and `SOAPBinding.ParameterStyle.Wrapped`. Default value is `SOAPBinding.ParameterStyle.Wrapped`.	String	No.

javax.jws.soap.SOAPMessageHandler

Description

Target: Class

Specifies a particular SOAP message handler in a `@SOAPMessageHandler` array. The annotation includes attributes to specify the class name of the handler, the initialization parameters, list of SOAP headers processed by the handler, and so on.

Attributes

Table 12-12 Attributes of the javax.jws.soap.SOAPMessageHandler JWS Annotation

Name	Description	Data Type	Required?
name	Name of the SOAP message handler. The default value is the name of the class that implements the `Handler` interface (or extends the `GenericHandler` abstract class.)	String	No.
className	Name of the handler class.	String	Yes.
initParams	Array of name/value pairs that is passed to the handler class during initialization.	Array of `@InitParam`	No.
roles	List of SOAP roles implemented by the handler.	Array of String	No.
headers	List of SOAP headers processed by the handler. Each element in this array contains a QName which defines the header element processed by the handler.	Array of String	No.

javax.jws.soap.InitParam

Description

Target: Class

Use this annotation in the `initParams` attribute of the `@SOAPMessageHandler` annotation to specify the array of parameters (name/value pairs) that are passed to a handler class during initiazliation.

Attributes

Table 12-13 Attributes of the javax.jws.soap.InitParam JWS Annotation

Name	Description	Data Type	Required?
name	Name of the initialization parameter.	String	Yes.
value	Value of teh initialization parameter.	String	Yes.

javax.jws.soap.SOAPMessageHandlers

Description

Target: Class

Specifies an array of SOAP message handlers that exeucte before and after the operations of a Web Service. Use the `@SOAPMessageHandler` annotation to specify a particular handler. Because you specify the list of handlers within the JWS file itself, the configuration of the handler chain is embedded within the Web Service.

Use the `@SOAPMessageHandlers` annotation, rather than `@HandlerChain`, if:

- You prefer to embed the configuration of the handler chain inside the Web Service itself, rather than specify the configuration in an external file.
- Your handler chain includes only SOAP handlers and none for any other transport.
- You prefer to recompile the JWS file each time you change the handler chain configuration.

The `@SOAPMessageHandlers` annotation is an array of `@SOAPMessageHandler` types. The handlers run in the order in which they appear in the annotation, starting with the first handler in the array.

This annotation does not have any attributes.

javax.jws.security.SecurityRoles

Description

Target: Class, Method

Specifies the roles that are allowed to access the operations of the Web Service.

If you specify this annotation at the class level, then the specified roles apply to all public operations of the Web Service. You can also specify a list of roles at the method level if you want to associate different roles to different operations of the same Web Service.

Warning: This annotation can only be used for Web Services that are implemented with a stateless session EJB.

Attributes

Table 12-14 Attributes of the javax.jws.security.SecurityRoles JWS Annotation

Name	Description	Data Type	Required?
rolesAllowed	Specifies the list of roles that are allowed to access the Web Service. This annotation is the equivalent of the `<method-permission>` element in the `ejb-jar.xml` deployment descriptor of the stateless session EJB that implements the Web Service.	Array of String	No.
rolesReferenced	Specifies a list of roles referenced by the Web Service. The Web Service may access other resources using the credentials of the listed roles. This annotation is the equivalent of the `<security-role-ref>` element in the `ejb-jar.xml` deployment descriptor of the stateless session EJB that implements the Web Service.	Array of String	No.

javax.jws.security.SecurityIdentity

Description

Target: Class

Specifies the identity assumed by the Web Service when it is invoked.

Unless otherwise specified, a Web Service assumes the identity of the authenticated invoker. This annotation allows the developer to override this behavior so that the Web Service instead

executes as a particular role. The role must map to a user or group in the WebLogic Server security realm.

Attributes

Table 12-15 Attributes of the javax.jws.security.SecurityIdentity JWS Annotation

Name	Description	Data Type	Required?
value	Specifies the role which the Web Service assumes when it is invoked. The role must map to a user or group in the WebLogic Server security realm.	String	Yes.

WebLogic-Specific JWS Annotations Reference

WebLogic Web Services define a set of JWS annotations that you can use to specify behavior and features in addition to the standard JSR-181 JWS annotations. In particular, the WebLogic-specific annotations are:

- "weblogic.jws.Policies" on page B-14
- "weblogic.jws.Policy" on page B-15
- "weblogic.jws.WLHttpTransport" on page B-17

weblogic.jws.Policies

Description

Target: Class, Method

Specifies an array of `@weblogic.jws.Policy` annotations.

Use this annotation if you want to attach more than one WS-Policy files to a class or method of a JWS file. If you want to attach just one policy file, you can use the `@weblogic.jws.Policy` on its own.

This JWS annotation does not have any attributes.

Example

```
@Policies({
    @Policy(uri="policy:firstPolicy.xml"),
    @Policy(uri="policy:secondPolicy.xml")
  })
```

weblogic.jws.Policy

Description

Target: Class, Method

Specifies that a WS-Policy file, which contains information about digital signatures, encryption, or reliable messaging, should be applied to the request or response SOAP messages.

This annotation can be used on its own to apply a single WS-Policy file to a class or method. If you want to apply more than one WS-Policy file to a class or method, use the `@weblogic.jws.Policies` annotation to group them together.

If this annotation is specified at the class level, the indicated policy file or files are applied to every public operation of the Web Service. If the annotation is specified at the method level, then only the corresponding operation will have the policy file applied.

By default, WS-Policy files are applied to both the request (inbound) and response (outbound) SOAP messages. You can change this default behavior with the `direction` attribute.

By default, the specified WS-Policy file is attached to the generated and published WSDL file of the Web Service so that consumers can view all the policy requirements of the Web Service. Use the `attachToWsdl` attribute to change this default behavior.

Attributes

Table 12-16 Attributes of the weblogic.jws.Policies JWS Annotation Tag

Name	Description	Data Type	Required?
uri	Specifies the location from which to retrieve the WS-Policy file. Use the `http:` prefix to specify the URL of a policy file on the Web. Use the `policy`: prefix to specify that the policy file is packaged in the Web Service archive file or in a shareable J2EE library of WebLogic Server, as shown in the following example: `@Policy(uri="policy:MyPolicyFile.xml")` If you are going to publish the policy file in the Web Service arrchive, the policy XML file must be located in either the `META-INF/policies` or `WEB-INF/policies` directory of the EJB JAR file (for EJB implemented Web Services) or WAR file (for Java class implemented Web Services), respectively. For information on publishing the policy file in a library, see Creating Shared J2EE Libraries and Optional Packages.	String	Yes.
direction	Specifies when to apply the policy: on the inbound request SOAP message, the outbound response SOAP message, or both (default). Valid values for this attribute are: • `Policy.Direction.both` • `Policy.Direction.inbound` • `Policy.Direction.outbound` The default value is `Policy.Direction.both`.	Sting.	No.
attachToWsdl	Specifies whether the WS-Policy file should be attached to the WSDL that describes the Web Service. Valid values are `true` and `false`. Default value is `true`.	Boolean	No.

Example

```
@Policy(uri="policy:myPolicy.xml",
        attachToWsdl=true,
        direction=Policy.Direction.outbound)
```

weblogic.jws.WLHttpTransport

Description

Target: Class

Specifies the context path and service URI sections of the URL used to invoke the Web Service over the HTTP transport, as well as the name of the port in the generated WSDL.

Attributes

Table 12-17 Attributes of the weblogic.jws.WLHttpTransport JWS Annotation Tag

Name	Description	Data Type	Required?
contextPath	Context root of the Web Service. You use this value in the URL that invokes the Web Service.	String	Yes.
serviceUri	Web Service URI portion of the URL used by client applications to invoke the Web Service.	String	Yes.
portName	The name of the port in the generated WSDL. This attribute maps to the `name` attribute of the `<port>` element in the WSDL. If you do not specify this attribute, the `jwsc` generates a default name based on the name of the class that implements the Web Service.	String	No.

Example

```
@WLHttpTransport(portName="helloPort",
                 contextPath="ejbJWS",
                 serviceUri="SimpleBean")
```

APPENDIX C

Reliable SOAP Messaging Policy Assertion Reference

The following sections provide reference information about reliable SOAP messaging policy assertions in a WS-Policy file:

Overview of a WS-Policy File That Contains Reliable SOAP Messaging Assertions

You use WS-Policy files to configure the reliable SOAP messaging capabilities of a WebLogic Web Service running on a destination endpoint. Use the `@Policy` JWS anntoation in the JWS file that implements the Web Service to specify the name of the WS-Policy file that is associated with a WebLogic Web Service.

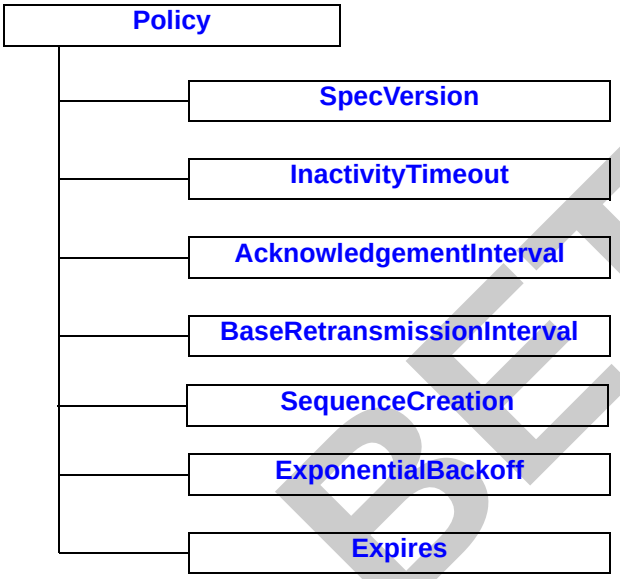
A WS-Policy file is an XML file that conforms to the WS-Policy specification. The root element of a WS-Policy file is always `<Policy>`. In the case of reliable SOAP messaging, the child elements of the `<Policy>` root element are a set of policy assertions that conform to the WS-PolicyAssertions specification. This section describes these reliable messaging policy assertions.

See "Using Reliable SOAP Messaging" on page 6-1 for task-oriented information about creating a reliable WebLogic Web Service.

Graphical Representation

The following graphic describes the element hierarchy of reliable SOAP messaging policy assertions in a WS-Policy file.

Figure 12-1 Element Hierarchy of Reliable SOAP Messaging Policy Assertions



Example of a WS-Policy File With Reliable SOAP Messaging Assertions

The following example shows a simple WS-Policy file used to configure reliable SOAP messaging for a WebLogic Web Service:

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

<wsp:Policy wsp:Name="DocLitHelloWorldPolicy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2003/11/policy"
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
```

```
  <wsrm:SpecVersion
      URI="http://schemas.xmlsoap.org/ws/2004/03/rm" />
  <wsrm:InactivityTimeout
      Milliseconds="150000" />
  <wsrm:AcknowledgementInterval
      Milliseconds="2000" />
  <wsrm:BaseRetransmissionInterval
      Milliseconds="500" />
  <wsrm:ExponentialBackoff />
  <wsrm:SequenceCreation />
</wsp:Policy>
```

Element Description

AcknowledgementInterval

Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand alone acknowledgement.

A destination endpoint can send an acknowledgement on the return message immediately after it has received a message from a source endpoint, or it can send one separately in a stand alone acknowledgement. In the case that a return message is not available to send an acknowledgement, a destination endpoint may wait for up to the acknowledgement interval before sending a stand alone acknowledgement. If there are no unacknowledged messages, the destination endpoint may choose not to send an acknowledgement.

This assertion does not alter the formulation of messages or acknowledgements as transmitted. Its purpose is to communicate the timing of acknowledgements so that the source endpoint may tune appropriately.

This element is optional. If you do not specify this element, the default value is set by the store and forward (SAF) agent configured for the destination endpoint. If using the Administration Console to configure the SAF agent, this value is labeled Retry Delay Base.

Table 12-18 Attributes of <AcknowledgementInterval>

Attribute	Description	Required?
Milliseconds	Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand alone acknowledgement.	Yes.

BaseRetransmissionInterval

Specifies the interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message.

If the source endpoint does not receive an acknowledgement for a given message within the interval specified by this element, the source endpoint retransmits the message. The source endpoint can modify this retransmission interval at any point during during the lifetime of the sequence of messages. This assertion does not alter the formulation of messages as transmitted, only the timing of their transmission.

This element can be used in conjunctions with the `<ExponentialBackoff>` element to specify that the retransmission interval will be adjusted using the algorithm specified by the `<ExponentialBackoff>` element.

This element is optional. If you do not specify this element, the default value is set by the store and forward (SAF) agent configured for the source endpoint. If using the Administration Console to configure the SAF agent, this value is labeled Retry Delay Base.

Table 12-19 Attributes of <BaseRetransmissionInterval>

Attribute	Description	Required?
Milliseconds	Number of milliseconds the source endpoint waits to retransmit message.	Yes.

Expires

Specifies the expiration time of a sequence of messages.

The data type of the value of the element is xsd:dateTime.

The following example shows how to specify an expiration date for a sequence of messages:

```
<wsu:Expires>2100-01-01T12:00:00.000-00:00</wsu:Expires>
```

This element is optional. If not specified, the sequence of messages never expires.

This element has no attributes.

The element is defined in the Web Services Security Addendum specification

Warning: While the expression of time for the expiration of a sequence of messages is absolute, the correct operation of the WS-ReliableMessaging protocol does not depend upon the synchronization of clocks between participating endpoints.

ExponentialBackoff

Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm.

This element is used in conjunction with the `<BaseRetransmissionInterval>` element. If a destination endopint does not acknowledge a sequence of messages for the amount of time specified by `<BaseRetransmissionInterval>`, the exponential backoff algorithm will be used for timing of successive retransmissions by the source endpoint, should the message continue to go unacknowledged.

The exponential backoff algorithm specifies that successive retransmission intervals should increase exponentially, based on the base retransmission interval. For example, if the base retransmission interval is 2 seconds, and the exponential backoff element is set in the WS-Policy file, successive retransmission intervals if messages continue to be unacknowledged are 2, 4, 8, 16, 32, and so on.

This element is optional. If not set, the same retransmission interval is used in successive retries, rather than the interval increasing exponentially.

This element has no attributes.

InactivityTimeout

Specifies (in milliseconds) a period of inactivity for a sequence of messages. A sequence of messages is defined as a set of messages, identified by a unique sequence number, for which a particular delivery assurance applies; typically a sequence originates from a single source endpoint. If, during the duration specified by this element, a destination endpoint has received no messages from the source endpoint, the destination endpoint may consider the sequence to have been terminated due to inactivity.

This element is optional. If it is not set in the WS-Policy file, then sequences never timeout due to inactivity.

Table 12-20 Attributes of <InactivityTimeout>

Attribute	Description	Required?
Milliseconds	The number of milliseconds that defines a period of inactivity.	Yes.

SequenceCreation

Specifies that the destination endpoint is responsible for creating enw sequences of messages and the source endpoint must use the Sequence protocol, defined in the WS-ReliableMessaging specification, to initiate a new sequence.

This element is required.

This element has no attributes.

SpecVersion

Specifies the version of the WS-ReliableMessaging specification that is supported by the WebLogic Web Services reliable SOAP messaging feature.

This element is required.

Table 12-21 Attributes of <SpecVersion>

Attribute	Description	Required?
URI	The URI of the supported WS-ReliableMessaging specification. Valid value is: `http://schemas.xmlsoap.org/ws/2004/03/rm`	Yes.
Usage	Indicates the usage of this assertion, such as `Required`, `Optional`, and so on. For details, see the WS-Policy specification.	No.
Preference	Specifies the preference, as an integer value, of this particular alternative. The higher the value of the preference, the greater the weighting of the expressed preference. If no preference is specified, a value of zero is assumed.	No.

APPENDIX D

WebLogic Web Service Deployment Descriptor Element Reference

The following sections provide information about the WebLogic-specific Web Services deployment descriptor file, `weblogic-webservices.xml`:

Overview of weblogic-webservices.xml

The standard J2EE deployment descriptor for Web Services is called `webservices.xml`. This file specifies the set of Web Services that are to be deployed to WebLogic Server and the dependencies they have on container resources and other services. See the Web Services XML Schema for a full description of this file.

The WebLogic equivalent to the standard J2EE `webservices.xml` deployment descriptor file is called `weblogic-webservices.xml`. This file contains WebLogic-specific information about a WebLogic Web Service, such as the URL used to invoke the deployed Web Service, and so on.

Both deployment descriptor files are located in the same location on the J2EE archive that contains the Web Service. In particular:

- For Java class-implemented Web Services, the Web Service is packaged as a Web application WAR file and the deployment descriptors are located in the WEB-INF directory.
- For stateless session EJB-implemented Web Services, the Web Service is packaged as an EJB JAR file and the deployment descriptors are located in the META-INF directory.

The structure of the `weblogic-webservices.xml` file is similar to the structure of the J2EE `webservices.xml` file in how it lists and identifies the Web Services that are contained within the archive. For example, for each Web Service in the archive, both files have a `<webservice-description>` child element of the appropriate root element (`<webservices>` for the J2EE `webservices.xml` file and `<weblogic-webservices>` for the `weblogic-webservices.xml` file)

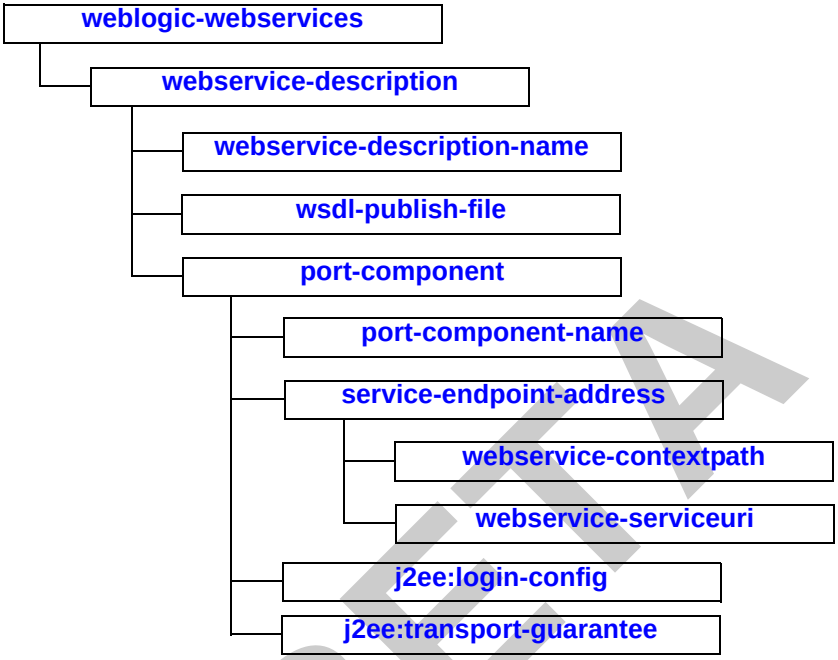
Typically users never need to update either deployment descriptor files, because the `jwsc` Ant task automatically generates the files for you based on the value of the JWS anntoations in the JWS file that implements the Web Service. For this reason, this section is published for informational purposes only.

The data type definitions of two elements in the `weblogic-webservices.xml` file (j2ee:login-config and j2ee:transport-guarantee) are imported from the J2EE Schema for the `web.xml` file. See the Servlet Deployment Descriptor Schema for details about these elements and data types.

Graphical Representation

The following graphic describes the element hierarchy of the `weblogic-webservices.xml` deployment descriptor file.

Figure 12-2 Element Hierarchy of weblogic-webservices.xml



XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://www.bea.com/ns/weblogic/90"
  xmlns:wls="http://www.bea.com/ns/weblogic/90"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  >
  <import namespace="http://java.sun.com/xml/ns/j2ee"
    schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"/>

  <element name="weblogic-webservices" type="wls:weblogic-webservicesType"/>
```

```
<complexType name="weblogic-webservicesType">
    <sequence>
      <xsd:element name="webservice-description"
                   type="wls:webservice-descriptionType"
                   minOccurs="1"
                   maxOccurs="unbounded" >
         <xsd:annotation>
          <xsd:documentation>

          Top level wsee internal DD element

          </xsd:documentation>
          </xsd:annotation>
      </xsd:element>
    </sequence>
</complexType>

<complexType name="webservice-descriptionType">
    <sequence>
      <xsd:element name="webservice-description-name"
                   type="j2ee:string" />

      <xsd:element name="wsdl-publish-file"
                   type="j2ee:string"
                   minOccurs="0"/>

      <xsd:element name="port-component"
                   type="wls:port-componentType"
                   minOccurs="0"
                   maxOccurs="unbounded" />
    </sequence>
</complexType>

<complexType name="port-componentType">
    <sequence>
      <xsd:element name="port-component-name"
                   type="j2ee:string" />
      <xsd:element name="service-endpoint-address"
                   type="wls:webservice-addressType"
                   minOccurs="0"/>
      <xsd:element name="login-config"
                   type="j2ee:login-configType"
                   minOccurs="0"/>
      <xsd:element name="transport-guarantee"
                   type="j2ee:transport-guaranteeType"
                   minOccurs="0"/>
    </sequence>
</complexType>
```

```
  <complexType name="webservice-addressType">
      <sequence>
        <xsd:element name="webservice-contextpath"
                     type="j2ee:string" />

        <xsd:element name="webservice-serviceuri"
                     type="j2ee:string" />
      </sequence>
  </complexType>

</schema>
```

Example of a weblogic-webservices.xml Deployment Descriptor File

The following example shows a simple `weblogic-webservices.xml` deployment descriptor:

```
<weblogic-webservices xmlns="http://www.bea.com/ns/weblogic/90">
  <webservice-description>
    <webservice-description-name>SimpleEjbService</webservice-description-name>
    <port-component>
      <port-component-name>helloPort</port-component-name>
      <service-endpoint-address>
        <webservice-contextpath>/jws_basic_ejb</webservice-contextpath>
        <webservice-serviceuri>/SimpleBean</webservice-serviceuri>
      </service-endpoint-address>
    </port-component>
  </webservice-description>
</weblogic-webservices>
```

Element Description

j2ee:login-config

The `j2ee:login-config` element specifies the authentication method that should be used, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism.

The XML Schema data type of the `j2ee:login-config` element is `j2ee:login-configType`, and is defined in the J2EE Schema that describes the standard `web.xml` deployment descriptor. For the full reference information, see http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd.

port-component

The <port-component> element associates a WSDL port with a WebLogic Web Service implementation. The value of the <port-component-name> child element specifies which particular port in the WSDL file is implemented by an EJB or Java class contained in the archive.

The child elements of the <port-component> element specify WebLogic-specific characteristics of the Web Service port, such as the context path and service URI used to invoke the Web Service after it has been deployed to WebLogic Server.

port-component-name

The <port-component-name> child element of the <port-component> element specifies a port component's name in the WSDL file. The value of the <port-component-name> element is associated with the name attribute of the corresponding <port> element of the WSDL file.

The value of this element must be unique for all <port-component-name> elements within a single weblogic-webservices.xml file.

service-endpoint-address

The <service-endpoint-address> element groups the WebLogic-specific context path and service URI values that together make up the Web Service endpoint address, or the URL that invokes the Web Service after it has been deployed to WebLogic Server.

These values are specified with the <webservice-contextpath> and <webserivce-serviceuri> child elements.

j2ee:transport-guarantee

The j2ee:transport-guarantee element specifies the type of communication between the client application invoking the Web Service and WebLogic server.

The value of this element is either NONE, INTEGRAL, or CONFIDENTIAL. NONE means that the application does not require any transport guarantees. A value of INTEGRAL means that the application requires that the data sent between the client and server be sent in such a way that it cannot be changed in transit. CONFIDENTIAL means that the application requires that the data be transmitted in a way that prevents other entities from observing the contents of the transmission. In most cases, the presence of the INTEGRAL or CONFIDENTIAL flag indicates that the use of SSL is required.

The XML Schema data type of the `j2ee:transport-guarantee` element is `j2ee:transport-guaranteeType`, and is defined in the J2EE Schema that describes the standard `web.xml` deployment descriptor. For the full reference information, see http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd.

weblogic-webservices

The `<weblogic-webservices>` element is the root element of the WebLogic-specific Web Services deployment descriptor (`weblogic-webservices.xml`).

The element specifies the set of Web Services contained in the J2EE component archive in which the deployment descriptor is also contained. The archive is either an EJB JAR file (for stateless session EJB-implemented Web Services) or a WAR file (for Java class-implemented Web Services)

webservice-contextpath

The `<webservice-contextpath>` element specifies the context path portion of the URL used to invoke the Web Service.

The URL to invoke a Web Service deployed to WebLogic Server is:

`http://`*`host`*`:`*`port`*`/`*`contextPath`*`/`*`serviceURI`*

where

- *host* is the host computer on which WebLogic Server is running.
- *port* is the port address to which WebLogic Server is listening.
- *contextPath* is the value of this element
- *serviceURI* is the value of the webservice-serviceuri element.

This value of the <*webservice-contextpath*> element is taken from the `contextPath` attribute of the WebLogic-specific `@WLHttpTransport` annotation in the JWS file that implements the Web Service contained in the archive.

webservice-description

The `<webservice-description>` element defines a single Web Service, corresponding to a `<service>` element in the WSDL file. The name of the Web Service is specified with the `<webservice-description-name>` child element.

The <webservice-description> element also defines a set of port components (specified using one or more <port-component> child elements) that are associated with the WSDL ports defined in the WSDL document.

There may be multiple <webservice-description> elements defined within a single weblogic-webservices.xml file, each corresponding to a particular stateless session EJB or Java class contained within the archive.

webservice-description-name

The <webservice-description-name> element specifies the name of the Web Service. Its value corresponds to the name attribute of the aoosicated <service> element in the WSDL file.

The value of this element must be unique for all <webservice-description-name> elements within a single weblogic-webservices.xml file.

webservice-serviceuri

The <webservice-serviceuri> element specifies the Web Service URI portion of the URL used to invoke the Web Service.

The URL to invoke a Web Service deployed to WebLogic Server is:

```
http://host:port/contextPath/serviceURI
```

where

- *host* is the host computer on which WebLogic Server is running.
- *port* is the port address to which WebLogic Server is listening.
- *contextPath* is the value of the webservice-contextpath element
- *serviceURI* is the value of this element.

This value of the <webservice-serviceuri> element is taken from the serviceURI attribute of the WebLogic-specific @WLHttpTransport annotation in the JWS file that implements the Web Service contained in the archive.

wsdl-publish-file

The <wsdl-publish-file> element specifies a directory to which WebLogic Server should write the WSDL file of a deployed Web Service.

Warning: Only specify this element if client applications that invoke the Web Service need to access the WSDL via non-HTTP ways; typically, client applications access the WSDL using HTTP, as described in "Invoking the WSDL and Home Page of the Web Service" on page 4-11.

The value of this element should be an absolute directory pathname. This directory must exist on *every* machine which hosts a WebLogic Server instance or cluster to which you deploy the Web Service.

APPENDIX E

Creating a J2EE Web Service Manually

The following sections describe how to create a J2EE Web Service (compliant with JSR-109) manually:

Overview of Creating a J2EE Web Service

This section describes how to create a standard J2EE Web Service manually, or in other words, without using the WebLogic Web Services tools, such as the `jwsc` Ant task. Standard J2EE Web Services follow the *Implementing Enterprise Web Services* 1.1 specification (JSR-921, which is the 1.1 maintenance release of JSR-109).

J2EE Web Services can be implemented with two different backend components: stateless session EJBs or Java classes. Depending on the type of backend component, the Web Service is packaged in either an EJB JAR file or Web application WAR file. The JAR or WAR archives contain their standard deployment descriptors (`ejb-jar.xml` for EJBs and `web.xml` for Web applications), updated to include additional elements that specify that the component is also a Web Service. The archive also contains additional Web Service-specific deployment descriptors, such as `webservices.xml` and the JAX-RPC data type mapping file.

Creating an EJB-Implemented J2EE Web Service: Main Steps

To create a J2EE Web Service that is implemented with a stateless session EJB, follow these steps:

1. Create a standard stateless session EJB whose business methods you want to expose as Web Service operations. You can also use an existing EJB.

   There are no requirements or limitations to the stateless session EJB that implements your Web Service, other than the standard guidelines outlined in the Enterprise JavaBeans specification. See *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs90/ejb/index.html for detailed information about creating an EJB.

   See "Simple Example of a Stateless Session EJB" on page E-3 for sample Java code for a Bean implementation class and its Home and Remote interfaces.

2. Create, if they do not already exist, the standard and WebLogic-specific EJB deployment descriptors:

   - `ejb-jar.xml`
   - `weblogic-ejb-jar.xml`

   For details about writing or generating EJB deployment descriptors, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs90/ejb/index.html.

   See "Sample EJB Deployment Descriptors" on page E-4 for sample deployment descriptor files that describe the EJB created in the preceding step.

3. Create the service endpoint interface (SEI) from the EJB.

   See "Creating the Service Endpoint Interface from the EJB" on page E-6.

4. Update the `ejb-jar.xml` deployment descriptor with Web Service information.

   Note: You do not have to update the `weblogic-ejb-jar.xml` deployment descriptor file.

   See "Updating the ejb-jar.xml Deployment Descriptor with Web Services Information" on page E-7.

5. Create the WSDL file which describes the public contract of the Web Service.

   See "Creating the WSDL File for an EJB-Implemented Web Service" on page E-8.

6. Create the standard Web Service deployment descriptors: `webservices.xml` and the JAX-RPC mapping file. Depending on what your Web Service looks like, you might also need to create the WebLogic-specific Web Services deployment descriptor, `weblogic-webservices.xml`.

   See "Writing the Deployment Descriptor Files for an EJB-Implemented Web Service" on page E-11.

7. Compile all Java and EJB source code into class files.

   For detailed information, see Compiling Java Code at http://e-docs.bea.com/wls/docs90/programming/topics.html.

8. Package all the various artifacts into a deployable EJB JAR file.

   See "Packaging All Artifacts Into a JAR File" on page E-15.

9. Deploy the EJB JAR file as usual.

   For details, see *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs90/deployment/index.html.

Simple Example of a Stateless Session EJB

The following sections provide sample Java code for a simple stateless session EJB that implements a J2EE Web Service. The EJB defines one business method, `sayHello()`, that takes as input a `String` and returns the same `String` slightly modified.

Bean Implementation Class

```
package examples.ejbDeploy;

import java.rmi.RemoteException;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

/**
 * EJB implementation of HelloWs
 */

public class HelloBean implements SessionBean {

   public void ejbCreate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
```

```
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

    public String sayHello(String input) throws RemoteException {
      System.out.println(":input" +input );
      return "'" + input + "' to you too!";
    }
}
```

Home Interface

```
package examples.ejbDeploy;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface HelloHome extends EJBHome {
  HelloRemote create() throws RemoteException, CreateException;
}
```

Remote Interface

```
package examples.ejbDeploy;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

public interface HelloRemote extends EJBObject {

  public String sayHello(String input) throws RemoteException;

}
```


Sample EJB Deployment Descriptors

The following sections list sample `ejb-jar.xml` and `weblogic-ejb-jar.xml` files for the `HelloBean` EJB.

ejb-jar.xml

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

```
<ejb-jar
      version="2.1"
      xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
           http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <display-name>EjbDeployBean</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>EjbDeployBean</ejb-name>
      <home>examples.ejbDeploy.HelloHome</home>
      <remote>examples.ejbDeploy.HelloRemote</remote>
      <ejb-class>examples.ejbDeploy.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>EjbDeployBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

weblogic-ejb-jar.xml

```
<!DOCTYPE weblogic-ejb-jar
      PUBLIC '-//BEA Systems, Inc.//DTD WebLogic 8.1.0 EJB//EN'
      'http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd'>
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
```

```
    <ejb-name>EjbDeployBean</ejb-name>
      <stateless-session-descriptor>
        <pool>
          <max-beans-in-free-pool>100</max-beans-in-free-pool>
          <initial-beans-in-free-pool> 10 </initial-beans-in-free-pool>
        </pool>
      </stateless-session-descriptor>
    <jndi-name>EjbDeployName</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Creating the Service Endpoint Interface from the EJB

The service endpoint interface (SEI) specifies which business methods of the stateless session EJB you want to expose as Web Service operations. These operations are published in the WSDL as a PortType that a client applications uses to invoke the Web Service. The SEI is similar to the standard Remote EJB interface that defines the business methods used by a client to the EJB.

When you create the SEI, follow these guidelines:

- The interface must extend `java.rmi.Remote`.
- Each method must throw `java.rmi.RemoteException`.

The Bean implementation class can also implement the SEI, although it is not required. This means that you can expose an existing stateless session EJB as a Web Service without having to update the Java implementation code.

The following sample SEI for the `HelloBean` EJB shows how to expose the `sayHello()` method as a Web Service operation:

```
package examples.ejbDeploy;

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

public interface HelloWs extends Remote {

   public String sayHello(String input) throws RemoteException;
}
```

Updating the ejb-jar.xml Deployment Descriptor with Web Services Information

Once you have created the service endpoint interface (SEI), you must update the `ejb-jar.xml` deployment descriptor with information about the interface. You do this by adding a `<service-endpoint>` child element to the `<session>` element that describes the stateless session EJB, similar to how you specify the standard Home and Remote interfaces, as shown in the following example:

```
<ejb-jar...>
     ...
  <enterprise-beans>
    <session>
      <ejb-name>EjbDeployBean</ejb-name>
      <ejb-class>examples.ejbDeploy.HelloBean</ejb-class>
      <home>examples.ejbDeploy.HelloHome</home>
      <remote>examples.ejbDeploy.HelloRemote</remote>
      <service-endpoint>examples.ejbDeploy.HelloWS</service-endpoint>
      ...
    </session>
  </enterprise-beans>
```

The following example shows the same `ejb-jar.xml` file from "Sample EJB Deployment Descriptors" on page E-4 after it has been udpated with an entry for the `HelloWS` SEI (see element in bold):

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

<ejb-jar
      version="2.1"
      xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
           http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <display-name>EjbDeployBean</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>EjbDeployBean</ejb-name>
      <home>examples.ejbDeploy.HelloHome</home>
```

```
      <remote>examples.ejbDeploy.HelloRemote</remote>
      <service-endpoint>examples.ejbDeploy.HelloWS</service-endpoint>
      <ejb-class>examples.ejbDeploy.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>EjbDeployBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```


Creating the WSDL File for an EJB-Implemented Web Service

The WSDL file defines the public contract of the Web Service. This includes:

- The name of the Web Service and a list of its ports. Each port has its own endpoint address, typically a URL that points to the implementation of the Web Service.
- The list of available operations for a given port. In the language of WSDL, this set of operations is called a *portType*.
- Binding information that specifies details about message formats and protocols when invoking an operation., or *how* to invoke an operation. Typical bindings include SOAP, HTTP, and JMS.
- Descriptions of the messages that are sent and received as a Web Service operation is invoked.
- Descriptions of the data types used in the messages.

Note: This section provides only high-level information about creating a WSDL file manually. For details about the structure and contents of a WSDL file, see the WSDL specification at http://www.w3.org/TR/wsdl.

Follow these high-level steps to create a WSDL file for your Web Service:

1. Use information from the service endpoint interface (SEI) to create the `<portType>` and `<message>` elements.

   The JAX-RPC specification defines how the SEI from the EJB that implements a Web Service maps to a WSDL file. In particular:

   – The name of the interface maps to a `<portType>` WSDL element.

   – Each method of the SEI maps to an `<operation>` child element of the `<portType>` element.

   – The input and return paramters of each method correspond to `<input>` and `<output>` child elements of the `<operation>` element, respectively. The `message` attribute of `<input>` and `<output>` maps to a `<message>` element that describes what the parameter looks like, such as its data type.

2. Create a `<binding>` element that references the `<portType>` element you created in the preceding step.

   You typically use the `<binding>` element to describe what the SOAP request and response messages look like when a client application invokes a Web Service operation. For example, use the `<soap:binding>` child element to specify whether the operation is RPC- or document-style. Then, for each operation defined in the portType, use the `<operation>` child element of `<binding>` to specify what the input and output parameters look like in the SOAP message.

3. Create a `<service>` element which defines the Web Service as a whole. You will later use this information when creating the Web Service deployment descriptors.

   Use the `name` attribute to specify the name of the Web Service. For each EJB that implements a Web Service, create a `<port>` child element of `<service>` that specifies its name, a reference to the binding you created in the preceding step, and the service endpoint address (or the URL of the Web Service once deployed on WebLogic Server.)

Note: You can call the WSDL file anything you want because the Web Services deployment descriptor files contain an explicit reference to this file.

The following sample WSDL file for the Web Service implemented with the `HelloBean` EJB shows how its service endpoint interface maps to elements of the WSDL file. The sections marked in bold highlight where the SEI and WSDL link up:

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

<definitions name="EjbDeploy"
             targetNamespace="http://EjbDeploy.org"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:tns="http://EjbDeploy.org"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
              xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
  </types>

  <message name="HelloWs_sayHelloResponse">
    <part name="result" type="xsd:string"/>
  </message>

  <message name="HelloWs_sayHello">
    <part name="String_1" type="xsd:string"/>
  </message>

  <portType name="HelloWs">
    <operation name="sayHello" parameterOrder="String_1">
      <input message="tns:HelloWs_sayHello"/>
      <output message="tns:HelloWs_sayHelloResponse"/>
    </operation>
  </portType>

  <binding name="HelloWsBinding" type="tns:HelloWs">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body
                  use="encoded"
                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  namespace="http://EjbDeploy.org"/>
      </input>
      <output>
        <soap:body
                  use="encoded"
```

```
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://EjbDeploy.org"/>
      </output>
    </operation>
  </binding>

  <service name="EjbDeploy">
    <port name="HelloWsPort" binding="tns:HelloWsBinding">
      <soap:address
             location="http://localhost:7001/ejbDeploy/EjbDeployBean"/>
    </port>
  </service>

</definitions>
```

Writing the Deployment Descriptor Files for an EJB-Implemented Web Service

This section describes how to create the following J2EE Web Service deployment descriptor files:

- webservices.xml
- JAX-RPC Mapping File

Note: This section provides only high-level information about manually creating the J2EE Web Services deployment descriptor files. For full details about the structure and contents of these files, see the *Web Services for J2EE* specification at http://jcp.org/aboutJava/communityprocess/final/jsr109/index.html.

webservices.xml

The `webservices.xml` file deployment descriptor file describes one or more Web Services that are contained in EJB JAR file in which the descriptor is contained.

The root element of the `webservices.xml` file is `<webservices>`. You can specify one or more Web Services using the `<webservice-description>` child element. Each Web Service can include the following information:

- The internal name of the Web Service using the `<webservice-description-name>` element.

- The location (relative to the root directory of the EJB JAR file) of the WSDL file using the `<wsdl-file>` element.
- The location (relative to the root directory of the EJB JAR file) of the JAX-RPC type mapping file, using the `<jaxrpc-mapping-file>` element.
- For each EJB Bean that implements a Web Service, a port component description using the `<port-component>` element. The `<port-component>` element can include the following information:
  - The internal name of the port using the `<port-component-name>` child element.
  - The name of the port in the WSDL file to which this port component definition refers using the `<wsdl-port>` element. The value of this element must be exactly the same as the value of the `name` attribute of the `<port>` child element of the `<service>` element of the WSDL file you created in "Creating the WSDL File for an EJB-Implemented Web Service" on page E-8, along with the `wsdl` namespace.
  - The full class name of the service endpoint interface you created from the EJB that implements the Web Service using the `<service-endpoint-interface>` element.
  - The name of the EJB that implements the Web Service, using the `<ejb-link>` child element of the `<service-impl-bean>` element. The value of this element must be exactly the same as the `<ejb-name>` child element of the `<session>` element in the `ejb-jar.xml` file that describes the EJB that you created in "Sample EJB Deployment Descriptors" on page E-4.

The following sample `webservices.xml` file describes the Web Service example being used in this section:

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

<webservices xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wsdl="http://EjbDeploy.org"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
  version="1.1">

    <webservice-description>
         <webservice-description-name>EjbDeploy
         </webservice-description-name>
         <wsdl-file>META-INF/wsdl/EjbDeploy.wsdl</wsdl-file>
         <jaxrpc-mapping-file>META-INF/EjbDeploy.xml</jaxrpc-mapping-file>
         <port-component>
```

```
            <port-component-name>EjbDeployPC</port-component-name>
            <wsdl-port>wsdl:HelloWsPort</wsdl-port>
            <service-endpoint-interface>examples.ejbDeploy.HelloWs
            </service-endpoint-interface>
            <service-impl-bean>
                <ejb-link>EjbDeployBean</ejb-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

JAX-RPC Mapping File

The JAX-RPC mapping file contains information that describes the relationship between the Java interfaces of the Web Service and the WSDL file that describes its public contract. The Java interfaces include both the service interface, used in a client that invokes the Web Service, and the service endpoint interface, used in the server-side implementation of the Web Service.

The JAX-RPC file also describes the mapping between the Java and XML representations (specified in the WSDL) of any user-defined data types used as input or output parameters by the Web Service.

The JAX-RPC mapping file does not have a standard name. Rather, the `webservices.xml` file makes an explicit reference to the file, including its name. BEA recommends, however, that you use the `.xml` extension in its name.

The following sample JAX-RPC mapping file describes the mapping between the Java interfaces and the WSDL file of the example Web Service discussed in this chapter:

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

<java-wsdl-mapping
     xmlns="http://java.sun.com/xml/ns/j2ee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:wsdl="http://EjbDeploy.org"
     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
     version="1.1">

  <package-mapping>
    <package-type>examples.ejbDeploy</package-type>
```

```
    <namespaceURI>http://EjbDeploy.org</namespaceURI>
  </package-mapping>

  <service-interface-mapping>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-service-name>wsdl:EjbDeploy</wsdl-service-name>
    <port-mapping>
      <port-name>HelloWsPort</port-name>
      <java-port-name>EjbDeploy</java-port-name>
    </port-mapping>
  </service-interface-mapping>

  <service-endpoint-interface-mapping>
    <service-endpoint-interface>examples.ejbDeploy.HelloWs
    </service-endpoint-interface>
    <wsdl-port-type>wsdl:HelloWs</wsdl-port-type>
    <wsdl-binding>wsdl:HelloWsBinding</wsdl-binding>

    <service-endpoint-method-mapping>
      <java-method-name>sayHello</java-method-name>
      <wsdl-operation>sayHello</wsdl-operation>

      <method-param-parts-mapping>
        <param-position>0</param-position>
        <param-type>java.lang.String</param-type>
        <wsdl-message-mapping>
          <wsdl-message>HelloWs_sayHello</wsdl-message>
          <wsdl-message-part-name>String_1</wsdl-message-part-name>
          <parameter-mode>IN</parameter-mode>
        </wsdl-message-mapping>
      </method-param-parts-mapping>

      <wsdl-return-value-mapping>
        <method-return-value>java.lang.String</method-return-value>
        <wsdl-message>HelloWs_sayHelloResponse</wsdl-message>
        <wsdl-message-part-name>result</wsdl-message-part-name>
      </wsdl-return-value-mapping>
    </service-endpoint-method-mapping>
  </service-endpoint-interface-mapping>
</java-wsdl-mapping>
```

Packaging All Artifacts Into a JAR File

To package all the artifacts into a JAR file, follow these steps:

1. In a staging directory, create the standard EJB JAR file directories and copy the compiled class files, deployment descriptors, and WSDL file into their appropriate directories. In particular:
   - The `webservices.xml` deployment descriptor lives in the META-INF directory, the same as the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files.
   - The WSDL and JAX-RPC mapping file live in the directories specified by the `webservices.xml` file with the `<wsdl-file>` and `<jaxrpc-mapping-file>` elements.

   The following listing shows, relative to the staging directory, the location of the files used by the Web Service example in this chapter:

   ```
   ./META-INF/MANIFEST.MF
   ./META-INF/ejb-jar.xml
   ./META-INF/weblogic-ejb-jar.xml
   ./META-INF/webservice.xml
   ./META-INF/EjbDeploy.xml
   ./META-INF/wsdl/EjbDeploy.wsdl
   ./examples/ejbDeploy/HelloBean.class
   ./examples/ejbDeploy/HelloHome.class
   ./examples/ejbDeploy/HelloRemote.class
   ./examples/ejbDeploy/HelloWs.class
   ```

2. Using the `jar` utility, create an EJB JAR file from the staging directory, as shown in the following example:

   ```
   prompt> jar cvf myWebService.jar .
   ```

Creating a Java Class-Implemented J2EE Web Service: Main Steps

To create a J2EE Web Service that is implemented with a Java class, follow these steps:

1. Program the Java class.

There are some requirements you must follow when programming the Java class that implements the J2EE Web Service. For details, see "Programming the Java Class" on page E-17.

2. Create, or update if it already exists, the standard J2EE Web Application deployment descriptor (`web.xml`) to describe the Java class you created in the preceding step. Use the `<servlet-class>` element of `web.xml` to specify the Java class.

   Note: Although you use the `<servlet-class>` element to specify the Java class, this does not mean that your Java class has to be a servlet or that it has to, for example, extend the `javax.servlet.http.HttpServlet` interface.

   See "Sample web.xml Deployment Descriptor" on page E-18 for a sample `web.xml` deployment descriptor file that describes a Web Application that contains the Java class you created in the preceding step.

   For details about writing or generating Web Application deployment descriptors, see *Developing Web Applications for WebLogic Server* at http://e-docs.bea.com/wls/docs90/webapp/index.html.

3. Create the service endpoint interface (SEI) from the Java class.

   See "Creating the Service Endpoint Interface from the Java Class" on page E-19.

4. Create the WSDL file which describes the public contract of the Web Service.

   See "Creating the WSDL File for a Java Class-Implemented Web Service" on page E-19.

5. Create the standard Web Service deployment descriptors: `webservices.xml` and the JAX-RPC mapping file. Depending on what your Web Service looks like, you might also need to create the WebLogic-specific Web Services deployment descriptor, `weblogic-webservices.xml`.

   See "Writing the Deployment Descriptor Files for a Java Class-Implemented Web Service" on page E-22.

6. Compile all the Java source code into class files.

   For detailed information, see Compiling Java Code at http://e-docs.bea.com/wls/docs90/programming/topics.html.

7. Package all the various artifacts into a deployable WAR file.

   See "Packaging All Artifacts Into a WAR File" on page E-26.

8. Deploy the WAR file as usual.

For details, see *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs90/deployment/index.html.

Programming the Java Class

When you program the Java class that implements your Web Service, follow these guidelines:

- The Java class must be public, not final, and not abstract.
- The Java class must not define the `finalize()` method.
- The Java class must have a default public constructor.
- The business methods of the Java class must be public and not static.
- The Java class must implement all the method signatures of the service endpoint interface (SEI). The Java class can implement other methods in addition to those defined in the SEI, but only those methods in the SEI will be exposed as Web Service operations.

  Note: Creating the SEI is discussed in "Creating the Service Endpoint Interface from the Java Class" on page E-19.

- Although not required, the Java class can implement the SEI.
- If the Java class does not implement the SEI, the business methods of the Java class must not be final.
- The Java class can implement the `javax.xml.rpc.server.ServiceLifeCycle` interface if an instance of the Java class, when deployed as a Web Service, wants to be notified by WebLogic Server of impending changes in its state. The Java class can then use the notification to prepare its internal state for the transition.

The following sample shows a simple Java class that implements a J2EE Web Service; in this case, the Java class implements the SEI called `HelloWs` (described in "Creating the Service Endpoint Interface from the Java Class" on page E-19) :

```
package examples.warDeploy;

import java.io.Serializable;
import java.rmi.RemoteException;

/*
 * JavaBean implementation of HelloWs
  */

public class HelloJavaBean implements HelloWs, Serializable {
```

```
    public HelloJavaBean() {}

    public String sayHello(String input) throws RemoteException {
      weblogic.utils.Debug.say( "(manoj):input" +input );
      return "'" + input + "' to you too!";
    }
}
```

Sample web.xml Deployment Descriptor

The following sample `web.xml` file shows how to describe the `example.warDeploy.HelloJavaBean` Java class so that it can be exposed as a J2EE Web Service:

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

<web-app version="2.4"
            xmlns="http://java.sun.com/xml/ns/j2ee"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
               http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>WarDeployApp_web</display-name>

    <servlet>
       <servlet-name>WarDeployBean</servlet-name>
       <servlet-class>examples.warDeploy.HelloJavaBean</servlet-class>
       <load-on-startup>0</load-on-startup>
    </servlet>

    <servlet-mapping>
       <servlet-name>WarDeployBean</servlet-name>
       <url-pattern>/ws4ee/warDeploy</url-pattern>
    </servlet-mapping>

    <session-config>
       <session-timeout>54</session-timeout>
    </session-config>

</web-app>
```

Note: Although you use the `<servletXXX>` elements to specify the Java class, this does not mean that your Java class has to be a servlet or that it has to, for example, extend the `javax.servlet.http.HttpServlet` interface.

Creating the Service Endpoint Interface from the Java Class

The service endpoint interface (SEI) specifies which business methods of the Java class you want to expose as Web Service operations. These operations are published in the WSDL as a PortType that a client applications uses to invoke the Web Service.

When you create the SEI, follow these guidelines:

- The interface must extend `java.rmi.Remote`.
- Each method must throw `java.rmi.RemoteException`.

The following sample SEI for the `HelloJavaBean` shows how to expose the `sayHello()` method as a Web Service operation:

```
package examples.warDeploy;

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

public interface HelloWs extends Remote {

   public String sayHello(String input) throws RemoteException;
}
```


Creating the WSDL File for a Java Class-Implemented Web Service

The WSDL file defines the public contract of the Web Service. This includes:

- The name of the Web Service and a list of its ports. Each port has its own endpoint address, typically a URL that points to the implementation of the Web Service.
- The list of available operations for a given port. In the language of WSDL, this set of operations is called a *portType*.
- Binding information that specifies details about message formats and protocols when invoking an operation., or *how* to invoke an operation. Typical bindings include SOAP, HTTP, and JMS.
- Descriptions of the messages that are sent and received as a Web Service operation is invoked.
- Descriptions of the data types used in the messages.

Note: This section provides only high-level information about creating a WSDL file manually. For details about the structure and contents of a WSDL file, see the WSDL specification at http://www.w3.org/TR/wsdl.

Follow these high-level steps to create a WSDL file for your Web Service:

1. Use information from the service endpoint interface (SEI) to create the `<portType>` and `<message>` elements.

   The JAX-RPC specification defines how the SEI from the Java class that implements a Web Service maps to a WSDL file. In particular:

   – The name of the interface maps to a `<portType>` WSDL element.

   – Each method of the SEI maps to an `<operation>` child element of the `<portType>` element.

   – The input and return paramters of each method correspond to `<input>` and `<output>` child elements of the `<operation>` element, respectively. The `message` attribute of `<input>` and `<output>` maps to a `<message>` element that describes what the parameter looks like, such as its data type.

2. Create a `<binding>` element that references the `<portType>` element you created in the preceding step.

   You typically use the `<binding>` element to describe what the SOAP request and response messages look like when a client application invokes a Web Service operation. For example, use the `<soap:binding>` child element to specify whether the operation is RPC- or document-style. Then, for each operation defined in the portType, use the `<operation>` child element of `<binding>` to specify what the input and output parameters look like in the SOAP message.

3. Create a `<service>` element which defines the Web Service as a whole. You will later use this information when creating the Web Service deployment descriptors.

   Use the `name` attribute to specify the name of the Web Service. For each Java class that implements a Web Service, create a `<port>` child element of `<service>` that specifies its name, a reference to the binding you created in the preceding step, and the service endpoint address (or the URL of the Web Service once deployed on WebLogic Server.)

Note: You can call the WSDL file anything you want because the Web Services deployment descriptor files contain an explicit reference to this file.

The following sample WSDL file for the Web Service implemented with the `HelloJavaBean` Java class shows how its service endpoint interface maps to elements of the WSDL file. The sections marked in bold highlight where the SEI and WSDL link up:

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

<definitions name="WarDeploy"
             targetNamespace="http://WarDeploy.org"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:tns="http://WarDeploy.org"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
  </types>

  <message name="HelloWs_sayHelloResponse">
    <part name="result" type="xsd:string"/>
  </message>

  <message name="HelloWs_sayHello">
    <part name="String_1" type="xsd:string"/>
  </message>

  <portType name="HelloWs">
    <operation name="sayHello" parameterOrder="String_1">
      <input message="tns:HelloWs_sayHello"/>
      <output message="tns:HelloWs_sayHelloResponse"/>
    </operation>
  </portType>

  <binding name="HelloWsBinding" type="tns:HelloWs">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body
              use="encoded"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://WarDeploy.org"/>
      </input>
      <output>
        <soap:body
               use="encoded"
```

```
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://WarDeploy.org"/>
      </output>
    </operation>
  </binding>

  <service name="WarDeploy">
    <port name="HelloWsPort" binding="tns:HelloWsBinding">
      <soap:address
              location="http://localhost:7001/warDeploy/ws4ee/warDeploy"/>
    </port>
  </service>

</definitions>
```

Writing the Deployment Descriptor Files for a Java Class-Implemented Web Service

This section describes how to create the following J2EE Web Service deployment descriptor files:

- webservices.xml
- JAX-RPC Mapping File

Note: This section provides only high-level information about manually creating the J2EE Web Services deployment descriptor files. For full details about the structure and contents of these files, see the *Web Services for J2EE* specification at http://jcp.org/aboutJava/communityprocess/final/jsr109/index.html.

webservices.xml

The `webservices.xml` file deployment descriptor file describes one or more Web Services that are contained in a Web Application WAR archive in which the descriptor is contained.

The root element of the `webservices.xml` file is `<webservices>`. You can specify one or more Web Services using the `<webservice-description>` child element. Each Web Service can include the following information:

- The internal name of the Web Service using the `<webservice-description-name>` element.

- The location (relative to the root directory of the Web Application WAR file) of the WSDL file using the `<wsdl-file>` element.
- The location (relative to the root directory of the Web Application WAR file) of the JAX-RPC type mapping file, using the `<jaxrpc-mapping-file>` element.
- For each Java class that implements a Web Service, a port component description using the `<port-component>` element. The `<port-component>` element can include the following information:
  - The internal name of the port using the `<port-component-name>` child element.
  - The name of the port in the WSDL file to which this port component definition refers using the `<wsdl-port>` element. The value of this element must be exactly the same as the value of the `name` attribute of the `<port>` child element of the `<service>` element of the WSDL file you created in "Creating the WSDL File for a Java Class-Implemented Web Service" on page E-19, along with the `wsdl` namespace.
  - The full class name of the service endpoint interface you created from the EJB that implements the Web Service using the `<service-endpoint-interface>` element.
  - The name of the Java class that implements the Web Service, using the `<servlet-link>` child element of the `<service-impl-bean>` element. The value of this element must be exactly the same as the `<servlet-name>` child element of the `<servlet>` element in the `web.xml` file that describes the EJB that you created in "Sample web.xml Deployment Descriptor" on page E-18.

The following sample `webservices.xml` file describes the Web Service example being used in this section:

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

<webservices
      xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:wsdl="http://WarDeploy.org"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
      version="1.1">

     <webservice-description>
       <webservice-description-name>WarDeploy
     </webservice-description-name>
     <wsdl-file>WEB-INF/wsdl/WarDeploy.wsdl</wsdl-file>
     <jaxrpc-mapping-file>WarDeploy.xml</jaxrpc-mapping-file>
```

```
      <port-component>
           <port-component-name>WarDeployPC</port-component-name>
           <wsdl-port>wsdl:HelloWsPort</wsdl-port>
           <service-endpoint-interface>examples.warDeploy.HelloWs
           </service-endpoint-interface>
           <service-impl-bean>
               <servlet-link>WarDeployBean</servlet-link>
           </service-impl-bean>
      </port-component>
      </webservice-description>
</webservices>
```

JAX-RPC Mapping File

The JAX-RPC mapping file contains information that describes the relationship between the Java interfaces of the Web Service and the WSDL file that describes its public contract. The Java interfaces include both the service interface, used in a client that invokes the Web Service, and the service endpoint interface, used in the server-side implementation of the Web Service.

The JAX-RPC file also describes the mapping between the Java and XML representations (specified in the WSDL) of any user-defined data types used as input or output parameters by the Web Service.

The JAX-RPC mapping file does not have a standard name. Rather, the `webservices.xml` file makes an explicit reference to the file, including its name. BEA recommends, however, that you use the `.xml` extension in its name.

The following sample JAX-RPC mapping file describes the mapping between the Java interfaces and the WSDL file of the example Web Service discussed in this chapter:

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

<java-wsdl-mapping
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wsdl="http://WarDeploy.org"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
  version="1.1">

  <package-mapping>
    <package-type>examples.warDeploy</package-type>
```

```
    <namespaceURI>http://WarDeploy.org</namespaceURI>
  </package-mapping>

  <service-interface-mapping>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-service-name>wsdl:WarDeploy</wsdl-service-name>
    <port-mapping>
      <port-name>HelloWsPort</port-name>
      <java-port-name>WarDeploy</java-port-name>
    </port-mapping>
  </service-interface-mapping>

  <service-endpoint-interface-mapping>
    <service-endpoint-interface>examples.warDeploy.HelloWs
    </service-endpoint-interface>
    <wsdl-port-type>wsdl:HelloWs</wsdl-port-type>
    <wsdl-binding>wsdl:HelloWsBinding</wsdl-binding>

    <service-endpoint-method-mapping>
      <java-method-name>sayHello</java-method-name>
      <wsdl-operation>sayHello</wsdl-operation>
      <method-param-parts-mapping>
        <param-position>0</param-position>
        <param-type>java.lang.String</param-type>
        <wsdl-message-mapping>
          <wsdl-message>HelloWs_sayHello</wsdl-message>
          <wsdl-message-part-name>String_1</wsdl-message-part-name>
          <parameter-mode>IN</parameter-mode>
        </wsdl-message-mapping>
      </method-param-parts-mapping>

      <wsdl-return-value-mapping>
        <method-return-value>java.lang.String</method-return-value>
        <wsdl-message>HelloWs_sayHelloResponse</wsdl-message>
        <wsdl-message-part-name>result</wsdl-message-part-name>
      </wsdl-return-value-mapping>
    </service-endpoint-method-mapping>

  </service-endpoint-interface-mapping>

</java-wsdl-mapping>
```

Packaging All Artifacts Into a WAR File

To package all the artifacts into a Web Application WAR file, follow these steps:

1. In a staging directory, create the standard Web Application WAR file directories and copy the compiled class files, deployment descriptors, and WSDL file into their appropriate directories. In particular:
   - The `webservices.xml` deployment descriptor lives in the WEB-INF directory, the same as the `web.xml` and optional `weblogic.xml` files.
   - The WSDL and JAX-RPC mapping file live in the directories specified by the `webservices.xml` file with the `<wsdl-file>` and `<jaxrpc-mapping-file>` elements.

   The following listing shows, relative to the staging directory, the location of the files used by the Web Service example in this chapter:

   ```
   ./META-INF/MANIFEST.MF
   ./WarDeploy.xml
   ./WEB-INF/web.xml
   ./WEB-INF/webservices.xml
   ./WEB-INF/wsdl/WarDeploy.wsdl
   ./WEB-INF/classes/examples/warDeploy/HelloJavaBean.class
   ./WEB-INF/classes/examples/warDeploy/HelloWs.class
   ```

2. Using the `jar` utility, create a Web Application WAR file from the staging directory, as shown in the following example:

   ```
   prompt> jar cvf myWebService.war .
   ```