BEA WebLogic Server®

WebLogic Web Services: Getting Started

Version 10.0
Revised: April 28, 2008

Contents

5. Programming the JWS File

6. Implementing a JAX-WS 2.0 Web Service

7. Data Types and Data Binding

8. Invoking Web Services

9. Administering Web Services

10. Upgrading WebLogic Web Services From Previous Releases to 10.0

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*WebLogic Web Services: Getting Started*.

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 Web Services. 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 Java Platform, Enterprise Edition (Java EE) Version 5 and Web Services concepts, the Java programming language, 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.

WebLogic Web Services Documentation Set

This document is part of a larger WebLogic Web Services documentation set that covers a comprehensive list of Web Services topics. The full documentation set includes the following documents:

- WebLogic Web Services: Getting Started—Describes the basic knowledge and tasks required to program a simple WebLogic Web Service. This is the first document you should read if you are new to WebLogic Web Services. The guide includes Web Service overview information, use cases and examples, iterative development procedures, typical JWS programming steps, data type information, and how to invoke a Web Service.
- WebLogic Web Services: Security—Describes how to program and configure message-level (digital signatures and encryption), transport-level, and access control security for a Web Service.
- WebLogic Web Services: Advanced Programming—Describes how to program more advanced features, such as Web Service reliable messaging, callbacks, conversational Web Services, use of JMS transport to invoke a Web Service, and SOAP message handlers.
- WebLogic Web Services: Reference—Contains all WebLogic Web Service reference documenation about JWS annotations, Ant tasks, reliable messaging WS-Policy assertions, security WS-Policy assertions, and deployment descriptors.

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, “Iterative Development of WebLogic Web Services,” provides procedures for setting up your development environment and iterative programming of 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 7, “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 8, “Invoking Web Services,”describes how to write a client application (stand-alone or inside a WebLogic Web Service) that invokes a Web Service using the JAX-RPC stubs generated by the WebLogic Web Service Ant task `clientgen`.
- Chapter 9, “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 10, “Upgrading WebLogic Web Services From Previous Releases to 10.0,” describes how to upgrade an 8.1, 9.0, and 9.1Web Service to run on the new 10.0 Web Services runtime environment.

Related Documentation

This document contains information specific to basic WebLogic Web Services topics. See “WebLogic Web Services Documentation Set” on page 1-2 for a description of the related Web Services documentation.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applicationssee 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, Servlets, and JSPs 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 Applications to WebLogic Server is the primary source of information about deploying WebLogic Server applications. Use this guide for both development and production deployment of your applications.
- Configuring Applications for Production Deployment describes how to configure your applications for deployment to a production WebLogic Server environment.
- WebLogic Server Performance and Tuning contains information on monitoring and improving the performance of WebLogic Server applications.
- Overview of WebLogic Server System Administration is an overview of administering WebLogic Server and its deployed 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 and tutorials 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.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you

can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Server.

As companion documentation to the MedRec application, BEA provides development tutorials that provide step-by-step procedures for key development tasks, including Web Service-specific tasks. See Application Examples and Tutorials for the latest information.

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 can be found at http://dev2dev.bea.com. These examples include BEA-certified ones, as well as examples submitted by fellow developers.

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 and Resolved Issues lists known problems by general release, as well as service pack, for all WebLogic Server APIs, including Web Services.

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 metadata annotations. The Web Services programming model uses JWS annotations, defined by the Web Services Metadata for the Java Platform specification.

  See Chapter 5, "Programming the JWS File."

- Implementation of the *Web Services for J2EE*, Version 1.2 specification, which defines the standard Java EE runtime architecture for implementing Web Services in Java.

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

- Implementation of the Java API for XML Web Services (JAX-WS), the centerpiece of a newly rearchitected API stack for Web services, the so-called "integrated stack" that includes JAX-WS 2.0, JAXB 2.0, and SAAJ 1.3. JAX-WS is designed to take the place of JAX-RPC in Web services and Web applications.
- Asynchronous, loosely-coupled Web Services that take advantage of the following features, either separately or all together: Web Service reliable messaging, conversations, buffering, asynchronous request-response, and JMS transport.

  See:

  – Using Web Service Reliable Messaging
  – Invoking a Web Service Using Asynchronous Request-Response
  – Using Callbacks to Notify Clients of Events
  – Creating Conversational Web Services
  – Creating Buffered Web Services
  – Using JMS Transport as the Connection Protocol
- Digital signatures and encryption of request and response SOAP messages, as specified by the WS-Security, as well as shared security contexts as described by the WS-SecureConversation specification.

  See Configuring Message-Level Security (Digital Signatures and Encryption).
- Use of WS-Policy files for the Web Service reliable messaging and digital signatures/encryption features.

  See Use of WS-Policy Files for Web Service Reliable Messaging Configuration and Using WS-Policy Files for Message-Level Security Configuration.
- Data binding between built-in and user-defined XML and Java data types.

  See Chapter 7, “Data Types and Data Binding.”
- SOAP message handlers that intercept the request and response SOAP message from an invoke of a Web Service.

  See Creating and Using SOAP Message Handlers.
- 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 Ant Task Reference.

- Implementation of and conformance with standard Web Services specifications.

  See "Standards Supported by WebLogic Web Services" on page 2-7.

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: monolithic systems such as those that ran on mainframes or client-server applications running on desktops. Although these architectures worked well for the purpose the applications were built to address, they were closed and their functionality could not be easily incorporated into new applications.

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 accessible 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 using that particular technology, often from a particular vendor. This requirement typically hinders widespread integration of the application's functionality into other services on the network. 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 using widely supported Web protocols such as HTTP.
- Web Services describe themselves using an XML-based description language.
- Web Services communicate with clients (both end-user applications or other Web Services) through simple XML messages that can be produced or parsed by virtually any programming environment or even by a person, if necessary.

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, Java Platform, Enterprise Edition (Java EE) Version 5 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 1.2 specification (JSR-109), which defines the standard Java EE runtime architecture for implementing Web Services in Java. The specification also describes a standard Java EE Web Service packaging format, deployment model, and runtime services, all of which are implemented by WebLogic Web Services.

The Enterprise Web Services 1.1 specification describes that a Java EE 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 in which you create an annotated Java file and then use Ant tasks to compile the file into a Java class 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 as well as a set of WebLogic-specific ones.

This release of WebLogic Server supports both Java API for XML-Based RPC 1.1 (JAX-RPC) and Java API for XML-Based Web Services 2.0 (JAX-JWS) Web Services. JAX-RPC, and older specification, defined APIs and conventions for supporting XML Web Services in the Java Platform as well support for the WS-I Basic Profile 1.0 to improve interoperability between JAX-RPC implementations. JAX-WS is a follow up to JAX-RPC 1.1.

WARNING: Although both JAX-RPC 1.1 and JAX-WS 2.0 are supported in this release of WebLogic Server, this document concentrates almost exclusively on describing how to create JAX-RPC style Web Services. This is because, in this release, all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and WebLogic value-added features (such as asynchronous request-response and callbacks) work *only* with JAX-RPC style Web Services. Therefore, unless

otherwise stated, you should assume that all descriptions and examples are for JAX-RPC Web Services.

For specific information about creating JAX-WS Web Services, see Chapter 6, "Implementing a JAX-WS 2.0 Web Service."

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

After you create the JWS file, you use the `jwsc` WebLogic Web Service Ant task to compile the JWS file, as described by the Enterprise Web Services 1.1 specification. The `jwsc` Ant task always compiles the JWS file into a plain Java class; the only time it implements a stateless session EJB is if you explicitly implemented `javax.ejb.SessionBean` in your JWS file. The `jwsc` Ant task also generates all the supporting artifacts for the Web Service, packages everything into an archive file, and creates an Enterprise Application that you can then deploy to WebLogic Server.

By default, the `jwsc` Ant task packages the Web service 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. If you execute `jwsc` against more than one JWS file, you can chose whether `jwsc` packages the Web Services in a single WAR file, or whether jwsc packages each Web Service in a separate WAR file. In either case, `jwsc` generates a single Enterprise Application.

If you explicitly implement `javax.ejb.SessionBean` in your JWS file, then the `jwsc` Ant task packages the Web Service 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-specific artifacts, as described in the preceding paragraph, to indicate that it is a Web Service. Similarly, you can choose whether multiple JWS files are packaged in a single or multiple EJB JAR files.

In addition to programming the JWS file, 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 program and configure additional advanced features. These include being able to invoke the Web Service reliably (as specified by the WS-ReliableMessaging specification, dated February 4, 2005) 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 Web Service 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
Get started.	"Understanding WebLogic Web Services" on page 2-1
	"Anatomy of a WebLogic Web Service" on page 2-3
	"Standards Supported by WebLogic Web Services" on page 2-7
	"Creating a Simple HelloWorld Web Service" on page 3-2
	"Common Web Services Use Cases and Examples" on page 3-1

Table 2-1 Web Services Tasks

Major Task	Subtasks and Additional Information
Iteratively develop a basic WebLogic Web Service.	"Iterative Development of WebLogic Web Services Starting From Java: Main Steps" on page 4-4
	"Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps" on page 4-5
	"Integrating Web Services Into the WebLogic Split Development Directory Environment" on page 4-22
	"Programming the JWS File" on page 5-1
	"Supported Built-In Data Types" on page 7-2
	"Supported User-Defined Data Types" on page 7-6
	"Programming the User-Defined Java Data Type" on page 5-20
	"Throwing Exceptions" on page 5-22
	"Accessing Runtime Information about a Web Service Using the JwsContext" on page 5-11
	"Should You Implement a Stateless Session EJB?" on page 5-17
	"Creating the Basic Ant build.xml File" on page 4-7
	"Running the jwsc WebLogic Web Services Ant Task" on page 4-8
Deploy the Web Service for testing purposes.	"Deploying and Undeploying WebLogic Web Services" on page 4-15
	"Browsing to the WSDL of the Web Service" on page 4-18
Invoke the Web Service.	"Invoking a Web Service from a Stand-alone Client: Main Steps" on page 8-4
	"Invoking a Web Service from Another Web Service" on page 8-12
	Invoking a Web Service Using Asynchronous Request-Response
	Creating and Using Client-Side SOAP Message Handlers
	Using a Client-Side Security WS-Policy File

Table 2-1 Web Services Tasks

Major Task	Subtasks and Additional Information
Add advanced features to the Web Service.	Using Web Service Reliable Messaging
	Using Callbacks to Notify Clients of Events
	Creating Conversational Web Services
	Creating Buffered Web Services
	Using JMS Transport as the Connection Protocol
	Creating and Using SOAP Message Handlers3
Secure the Web Service.	Configuring Message-Level Security (Digital Signatures and Encryption)
	Configuring Transport-Level Security
	Configuring Access Control Security: Main Steps
Upgrade an 8.1, 9.0, or 9.1WebLogic Web Service to run in the 10.0 runtime.	"Upgrading a 9.0 or 9.1 WebLogic Web Service to 10.0" on page 10-1
	"Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 10.0: Main Steps" on page 10-3
	"Upgrading an 8.1 EJB-Implemented WebLogic Web Service to 10.0: Main Steps" on page 10-10

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 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) 2.0" on page 2-9.

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

WebLogic Web Services are hosted by WebLogic Server and are implemented using standard Java EE components, as defined by the *Implementing Enterprise Web Services* 1.2 specification (JSR-109). See "Enterprise Web Services 1.2" on page 2-10.

- 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) as the message format and HTTP as the connection protocol; both versions 1.1 and 1.2 are supported. See "SOAP 1.1 and 1.2" on page 2-10.

  WebLogic Web Services implement the SOAP with Attachments API for Java (SAAJ) specification to access any attachments to the SOAP message. See "SAAJ" on page 2-11.

- 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 2-12.

  WebLogic Web Services uses WS-Policy to describe additional functionality and requirements not addressed in WSDL 1.1. WebLogic Web Services conform to the WS-Policy specification when using policies to describe their reliable messaging and security (digital signatures and encryption) functionality. See "WS-Policy 1.2 (April 2006 Member Submission)" on page 2-16.

- 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 2-13.

- 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) 1.1" on page 2-14.

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

WebLogic Web Services conform to the WS-Addressing (August 2004 Member Submission) and WS-ReliableMessaging 1.0 (February 2005 Member Submission) specifications when asynchronous features such as callbacks, addressing, conversations, and Web Service 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 specifications that define Web Service standards are written so as to allow for broad use of the specification throughout the industry. Thus the BEA 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. BEA complies with the Basic Profile 1.1 specification from the Web Services Interoperability Organization and considers it to be the baseline for Web Services interoperability. This guide does not necessarily document all of the Basic Profile 1.1 requirements. This guide does, however, document features that are beyond the requirements of the Basic Profile 1.1.

Web Services Metadata for the Java Platform (JSR-181) 2.0

Although it is possible to program a WebLogic Web Service manually by coding the standard JSR-109 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 5.0 metadata annotations feature 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-109 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 5.0 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* 2.0 specification (JSR-181) as well as a set of WebLogic-specific ones.

Enterprise Web Services 1.2

The *Implementing Enterprise Web Services* 1.2 specification (JSR-109) defines the programming model and runtime architecture for implementing Web Services in Java that run on a Java EE application server, such as WebLogic Server. In particular, it specifies that programmers implement Java EE Web Services using one of two components:

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

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

Java API for XML-Based Web Services (JAX-WS) 2.0

The Java API for XML Web Services (JAX-WS) is the centerpiece of a newly rearchitected API stack for Web services, the so-called "integrated stack" that includes JAX-WS 2.0, JAXB 2.0, and SAAJ 1.3. The integrated stack represents a logical rearchitecture of Web services functionality in the Java WSDP. JAX-WS is designed to take the place of JAX-RPC in Web services and Web applications.

Java Architecture for XML Binding (JAXB) 2.0

Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in Java code. This makes it easy for you to incorporate XML data and processing functions in applications based on Java technology without having to know much about XML itself.

Note: You can use JAXB *only* with JAX-WS 2.0 based Web Services. Because most of this document describes how to create JAX-RPC 1.1-based Web Services, it is assumed that you are using WebLogic's own data binding features, as described in Chapter 7, "Data Types and Data Binding."

SOAP 1.1 and 1.2

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 versions 1.1 and 1.2 of the SOAP 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 1.1 request for stock trading information embedded inside an HTTP request:

```
POST /StockQuote HTTP/1.1
Host: www.sample.com:7001
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>
```

By default, WebLogic Web Services use version 1.1 of SOAP; if you want your Web Service to use version 1.2, specify the weblogic.jws.Binding JWS annotation in the JWS file that implements your service.

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

SAAJ

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) 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>
```

The WSDL specification includes optional extension elements that specify different types of bindings that can be used when invoking the Web Service. The WebLogic Web Services runtime:

- Fully supports SOAP bindings, which means that if a WSDL file includes a SOAP binding, the WebLogic Web Services will use SOAP as the format and protocol of the messages used to invoke the Web Service.
- Ignores HTTP GET and POST bindings, which means that if a WSDL file includes this extension, the WebLogic Web Services runtime skips over the element when parsing the WSDL.
- Partially supports MIME bindings, which means that if a WSDL file includes this extension, the WebLogic Web Services runtime parses the element, but does not actually create MIME bindings when constructing a message due to a Web Service invoke.

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 7-2
- “Supported User-Defined Data Types” on page 7-6

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.

Web Services Security (WS-Security) 1.1

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 March 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).

WebLogic Web Services also implement the following token profiles:

- Web Services Security: Username Token Profile
- Web Services Security: X.509 Certificate Token Profile
- Web Services Security: SAML Token Profile 1.0

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 (August 2004 Member Submission)

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 Web Service reliable messaging) use addressing in their implementation.

See Web Services Addressing (WS-Addressing).

WS-Policy 1.2 (April 2006 Member Submission)

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).

WS-SecurityPolicy 1.2 (June 2006 Draft)

WS-SecurityPolicy defines a set of security policy assertions for use with the WS-Policy framework to describe how messages are to be secured in the context of WS-Security, WS-Trust and WS-SecureConversation.

See Web Services Security Policy (WS-SecurityPolicy).

WS-ReliableMessaging 1.0 (February 2005 Member Submission)

The WS-ReliableMessaging specification 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).

WS-Trust 1.3 (February 2005 Member Submission)

The WS-Trust specification defines extensions that build on Web Services Security (WS-Security) 1.1 to provide a framework for requesting and issuing security tokens, and to broker trust relationships.

See Web Services Trust Language (WS-Trust).

WS-SecureConversation 1.3 (February 2005 Member Submission)

The WS-SecureConversation specification defines extensions that build on Web Services Security (WS-Security) 1.1 and WS-Trust 1.3 (February 2005 Member Submission) to provide secure communication across one or more messages. Specifically, this specification defines mechanisms for establishing and sharing security contexts, and deriving keys from established security contexts (or any shared secret).

See Web Services Secure Conversation Language (WS-SecureConversation).

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 describe 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 either in your own development environment to recreate the example, or by following the instructions to create and run the example outside of an already setup development environment.

WARNING: Although both JAX-RPC 1.1 and JAX-WS 2.0 are supported in this release of WebLogic Server, this document concentrates almost exclusively on describing how to create JAX-RPC style Web Services. This is because, in this release, all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and WebLogic value-added features (such as asynchronous request-response and callbacks) work *only* with JAX-RPC style Web Services. Therefore, unless otherwise stated, you should assume that all descriptions and examples are for JAX-RPC Web Services.

For specific information about creating JAX-WS Web Services, see Chapter 6, "Implementing a JAX-WS 2.0 Web Service."

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:

- Iterative Development of WebLogic Web Services
- Programming the JWS File
- WebLogic Web Services: Advanced Programming
- Invoking Web Services
- Ant Task Reference

Creating a Simple HelloWorld Web Service

This section describes how to create a very simple Web Service that contains a single operation. The *JWS file* that implements the Web Service uses just the one required *JWS annotation*: `@WebService`. 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 following example shows how to create a Web Service called `HelloWorldService` that includes a single operation, `sayHelloWorld`. For simplicity, the operation does nothing other than return the inputted String value.

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 `bin` subdirectory of 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 project directory:

   ```
   prompt> mkdir /myExamples/hello_world
   ```

3. Create a `src` directory under the project directory, as well as sub-directories that correspond to the package name of the JWS file (shown later in this procedure):

   ```
   prompt> cd /myExamples/hello_world
   prompt> mkdir src/examples/webservices/hello_world
   ```

4. Create the JWS file that implements the Web Service by opening your favorite Java IDE or text editor and creating a Java file called `HelloWorldImpl.java` using the Java code specified in "Sample HelloWorldImpl.java JWS File" on page 3-5.

   The sample JWS file shows a Java class called `HelloWorldImpl` that contains a single public method, `sayHelloWorld(String)`. The `@WebService` annotation specifies that the Java class implements a Web Service called `HelloWorldService`. By default, all public methods are exposed as operations.

5. Save the `HelloWorldImpl.java` file in the `src/examples/webservices/hello_world` directory.

6. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the fully Java classname of the `jwsc` task:

   ```
   <project name="webservices-hello_world" default="all">

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

   </project>
   ```

   See "Sample Ant Build File for HelloWorldImpl.java" on page 3-5 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

7. Add the following call to the `jwsc` Ant task to the `build.xml` file, wrapped inside of the `build-service` target:

   ```
   <target name="build-service">

     <jwsc
       srcdir="src"
       destdir="output/helloWorldEar">

      <jws file="examples/webservices/hello_world/HelloWorldImpl.java" />

     </jwsc>

   </target>
   ```

   The `jwsc` WebLogic Web Service Ant task generates the supporting artifacts (such as the deployment descriptors, serialization classes for any user-defined data types, the WSDL file, and so on), compiles the user-created and generated Java code, and archives all the artifacts into an Enterprise Application EAR file that you later deploy to WebLogic Server.

8. Execute the `jwsc` Ant task by specifying the `build-service` target at the command line:

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

See the `output/helloWorldEar` directory to view the files and artifacts generated by the `jwsc` Ant task.

9. Start the WebLogic Server instance to which the Web Service will be deployed.

10. Deploy the Web Service, packaged in an Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `helloWorldEar` Enterprise application, located in the `output` directory.

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

    ```
    <taskdef name="wldeploy"
             classname="weblogic.ant.taskdefs.management.WLDeploy"/>

    <target name="deploy">

      <wldeploy action="deploy"
                name="helloWorldEar" source="output/helloWorldEar"
                user="${wls.username}" password="${wls.password}"
                verbose="true"
                adminurl="t3://${wls.hostname}:${wls.port}"
                targets="${wls.server.name}" />

    </target>
    ```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

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

    ```
    prompt> ant deploy
    ```

11. Test that the Web Service is deployed correctly by invoking its WSDL in your browser:

    `http://`*`host`*`:`*`port`*`/HelloWorldImpl/HelloWorldImpl?WSDL`

    You construct this URL by specifying the values of the `contextPath` and `serviceUri` attributes of the `WLHttpTransport` JWS annotation; however, because the JWS file in this use case does not include the `WLHttpTransport` annotation, specify the default values for the two attributes: the name of the Java class in the JWS file. Use the hostname and port relevant to your WebLogic Server instance.

See "Invoking a Web Service from a Stand-alone JAX-RPC Java Client" on page 3-23 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

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

Sample HelloWorldImpl.java JWS File

```
package examples.webservices.hello_world;

// Import the @WebService annotation

import javax.jws.WebService;

@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHelloWorld
 *
 * @author Copyright (c) 2005 by BEA Systems.  All rights reserved.
 */

public class HelloWorldImpl {

  // By default, all public methods are exposed as Web Services operation

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

Sample Ant Build File for HelloWorldImpl.java

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-hello_world" default="all">

  <!-- set global properties for this build -->

  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />

  <property name="ear.deployed.name" value="helloWorldEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/helloWorldEar" />
 <property name="clientclass-dir" value="${example-output}/clientclasses"
/>
```

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

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

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

<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy"/>
<target name="all" depends="clean,build-service,deploy,client" />

<target name="clean" depends="undeploy">
  <delete dir="${example-output}"/>
</target>

<target name="build-service">

  <jwsc
    srcdir="src"
    destdir="${ear-dir}">

    <jws file="examples/webservices/hello_world/HelloWorldImpl.java" />

  </jwsc>

</target>

<target name="deploy">
  <wldeploy action="deploy" name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>

<target name="undeploy">
  <wldeploy action="undeploy" name="${ear.deployed.name}"
    failonerror="false"
    user="${wls.username}" password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
```

```
      targets="${wls.server.name}" />
  </target>

  <target name="client">

    <clientgen

wsdl="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldImpl?WSD
L"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.hello_world.client"/>

    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/hello_world/client/**/*.java"/>

  </target>

  <target name="run">
    <java classname="examples.webservices.hello_world.client.Main"
          fork="true" failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg
 line="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldImpl"
/>
    </java> </target>

</project>
```

Creating a Web Service With User-Defined Data Types

The preceding use case uses only a simple data type, `String`, as the parameter and return value of the Web Service operation. This next example shows how to create a 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 to 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 a Simple HelloWorld Web Service" 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 HelloWorld example.

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

   ```
   prompt> mkdir /myExamples/complex
   ```

3. Create a `src` directory under the project directory, as well as sub-directories that correspond to the package name of the JWS file (shown later in this procedure):

   ```
   prompt> cd /myExamples/complex
   prompt> mkdir src/examples/webservices/complex
   ```

4. 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 project directory, using the Java code specified in "Sample BasicStruct JavaBean" on page 3-10.
5. Save the `BasicStruct.java` file in the `src/examples/webservices/complex` sub-directory of the project directory.
6. Create the JWS file that implements the Web Service using the Java code specified in "Sample ComplexImpl.java JWS File" on page 3-11.

   The sample JWS file uses more JWS annotations than in the preceding example: `@WebMethod` to specify explicitly that a method should be exposed as a Web Service operation and to change its operation name from the default method name `echoStruct` to `echoComplexType`; `@WebParam` and `@WebResult` to configure the parameters and return values; `@SOAPBinding` to specify the type of Web Service; and `@WLHttpTransport` to specify the URI used to invoke the Web Service. The `ComplexImpl.java` JWS file also imports the `examples.webservice.complex.BasicStruct` class and then uses the `BasicStruct` user-defined data type as both a parameter and return value of the `echoStruct()` method.

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

7. Save the `ComplexImpl.java` file in the `src/examples/webservices/complex` sub-directory of the project directory.
8. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the fully Java classname of the `jwsc` task:

```
<project name="webservices-complex" default="all">
  <taskdef name="jwsc"
           classname="weblogic.wsee.tools.anttasks.JwscTask" />
</project>
```

   See "Sample Ant Build File for ComplexImpl.java JWS File" on page 3-12 for a full sample `build.xml` file.

9. Add the following call to the `jwsc` Ant task to the `build.xml` file, wrapped inside of the `build-service` target:

```
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="output/ComplexServiceEar" >
    <jws file="examples/webservices/complex/ComplexImpl.java" />
  </jwsc>
 </target>
```

10. Execute the `jwsc` Ant task:

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

    See the `output/ComplexServiceEar` directory to view the files and artifacts generated by the `jwsc` Ant task.

11. Start the WebLogic Server instance to which the Web Service will be deployed.
12. Deploy the Web Service, packaged in the `ComplexServiceEar` Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task.
13. Test that the Web Service is deployed correctly by invoking its WSDL in your browser:

    `http://`*`host`*`:`*`port`*`/complex/ComplexService?WSDL`

See "Invoking a Web Service from a Stand-alone JAX-RPC Java Client" on page 3-23 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

Sample BasicStruct JavaBean

```
package examples.webservices.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;

  // 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 ComplexImpl.java JWS File

```
package examples.webservices.complex;

// Import the standard JWS annotation interfaces

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

// Import the WebLogic-specific JWS annotation interface

import weblogic.jws.WLHttpTransport;

// Import the BasicStruct JavaBean

import examples.webservices.complex.BasicStruct;

// Standard JWS annotation that specifies that the portType 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 this is a document-literal-wrapped
// Web Service

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

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

@WLHttpTransport(contextPath="complex", serviceUri="ComplexService",
                 portName="ComplexServicePort")

/**
 * This JWS file forms the basis of a WebLogic Web Service.  The Web Services
 * has two public operations:
 *
 *   - echoInt(int)
 *   - echoComplexType(BasicStruct)
 *
 * The Web Service is defined as a "document-literal" service, which means
 * that the SOAP messages have a single part referencing an XML Schema element
 * that defines the entire body.
 *
```

```
 * @author Copyright (c) 2005 by BEA Systems. All Rights Reserved.
 */

public class ComplexImpl {

  // 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: echoInt.
  //
  // The WebResult annotation specifies that the name of the result of the
  // operation in the generated WSDL is "IntegerOutput", rather than the
  // default name "return".   The WebParam annotation specifies that the input
  // parameter name in the WSDL file is "IntegerInput" rather than the Java
  // name of the parameter, "input".

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

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

  // Standard JWS annotation to expose method "echoStruct" as a public operation
  // called "echoComplexType"
  // The WebResult annotation specifies that the name of the result of the
  // operation in the generated WSDL is "EchoStructReturnMessage",
  // rather than the default name "return".

  @WebMethod(operationName="echoComplexType")
  @WebResult(name="EchoStructReturnMessage",
             targetNamespace="http://example.org/complex")
  public BasicStruct echoStruct(BasicStruct struct)

  {
    System.out.println("echoComplexType called");
    return struct;
  }
}
```

Sample Ant Build File for ComplexImpl.java JWS File

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-complex" default="all">

  <!-- set global properties for this build -->

  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />

  <property name="ear.deployed.name" value="complexServiceEAR" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/complexServiceEar" />
 <property name="clientclass-dir" value="${example-output}/clientclass" />

  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

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

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

  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>

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

  <target name="clean" depends="undeploy">
    <delete dir="${example-output}"/>
  </target>

  <target name="build-service">

    <jwsc
      srcdir="src"
      destdir="${ear-dir}"
      keepGenerated="true"
      >
      <jws file="examples/webservices/complex/ComplexImpl.java" />
    </jwsc>
```

```
</target>

<target name="deploy">
  <wldeploy action="deploy"
    name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}"/>
</target>

<target name="undeploy">
  <wldeploy action="undeploy" failonerror="false"
    name="${ear.deployed.name}"
    user="${wls.username}" password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}"/>
</target>

<target name="client">

  <clientgen
   wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
    destDir="${clientclass-dir}"
    packageName="examples.webservices.complex.client"/>

  <javac
    srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
    includes="**/*.java"/>

  <javac
    srcdir="src" destdir="${clientclass-dir}"
    includes="examples/webservices/complex/client/**/*.java"/>
</target>

<target name="run" >
  <java fork="true"
        classname="examples.webservices.complex.client.Main"
        failonerror="true" >
    <classpath refid="client.class.path"/>
   <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
/>
```

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

Creating a Web Service from a WSDL File

Another typical use case of creating a Web Service is to start from an existing WSDL file, often referred to as the *golden WSDL*. A WSDL file is 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 artifacts that implement the Web Service so that it can be deployed to WebLogic Server. These artifacts include:

- The JWS interface file that represents the Java implementation of your Web Service.
- Data binding artifacts used by WebLogic Server to convert between the XML and Java representations of the Web Service parameters and return values.
- A JWS file that contains a partial implementation of the generated JWS interface.
- Optional Javadocs for the generated JWS interface.

You use the `wsdlc` Ant task to generate these artifacts. Typically you run this Ant task one time to generate a JAR file that contains the generated JWS interface file and data binding artifacts, then code the generated JWS file that implements the interface, adding the business logic of your Web Service. In particular, you add Java code to the methods that implement the Web Service operations so that the operations behave as needed and add additional JWS annotations.

WARNING: The only file generated by the `wsdlc` Ant task that you update is the JWS implementation file; you never need to update the JAR file that contains the JWS interface and data binding artifacts.

After you have coded the JWS implementation file, you run the `jwsc` Ant task to generate the deployable Web Service, using the same steps as described in the preceding sections. The only difference is that you use the `compiledWsdl` attribute to specify the JAR file (containing the JWS interface file and data binding artifacts) generated by the `wsdlc` Ant task.

The following simple example shows how to create a Web Service from the WSDL file shown in "Sample WSDL File" on page 3-19. 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 `bin` subdirectory of 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 working directory:

   ```
   prompt> mkdir /myExamples/wsdlc
   ```

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 `/myExamples/wsdlc/wsdl_files` directory. See "Sample WSDL File" on page 3-19 for a full listing of the file.

4. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the fully Java classname of the `wsdlc` task:

   ```
   <project name="webservices-wsdlc" default="all">

     <taskdef name="wsdlc"
              classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>

   </project>
   ```

   See "Sample Ant Build File for TemperatureService" on page 3-21 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

5. Add the following call to the `wsdlc` Ant task to the `build.xml` file, wrapped inside of the `generate-from-wsdl` target:

   ```
   <target name="generate-from-wsdl">

     <wsdlc
         srcWsdl="wsdl_files/TemperatureService.wsdl"
         destJwsDir="output/compiledWsdl"
         destImplDir="output/impl"
         packageName="examples.webservices.wsdlc" />

   </target>
   ```

   The `wsdlc` task in the examples generates the JAR file that contains the JWS interface and data binding artifacts into the `output/compiledWsdl` directory under the current directory. It also generates a partial implementation file (`TemperaturePortTypeImpl.java`) of the JWS interface into the `output/impl/examples/webservices/wsdlc` directory (which is a combination of the

output directory specified by `destImplDir` and the directory hierarchy specified by the package name). All generated JWS files will be packaged in the `examples.webservices.wsdlc` package.

6. Execute the `wsdlc` Ant task by specifying the `generate-from-wsdl` target at the command line:

   ```
   prompt> ant generate-from-wsdl
   ```

   See the `output` directory if you want to examine the artifacts and files generated by the `wsdlc` Ant task.

7. Update the generated `output/impl/examples/webservices/wsdlc/TemperaturePortTypeImpl.java` JWS implementation file using your favorite Java IDE or text editor to add Java code to the methods so that they behave as you want. See "Sample TemperaturePortType Java Implementation File" on page 3-20 for an example; the added Java code is in bold. The generated JWS implementation file automatically includes values for the `@WebService` and `@WLHttpTransport` JWS annotations that correspond to the values in the original WSDL file.

   WARNING: There are restrictions on the JWS annotations that you can add to the JWS implementation file in the "starting from WSDL" use case. See wsdlc for details.

   For simplicity, the sample `getTemp()` method in `TemperaturePortTypeImpl.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.

8. Copy the updated `TemperaturePortTypeImpl.java` file into a permanent directory, such as a `src` directory under the project directory; remember to create child directories that correspond to the package name:

```
prompt> cd /examples/wsdlc
prompt> mkdir src/examples/webservices/wsdlc
prompt> cp output/impl/examples/webservices/wsdlc/TemperaturePortTypeImpl.java
\
            src/examples/webservices/wsdlc/TemperaturePortTypeImpl.java
```

9. Add a `build-service` target to the `build.xml` file that executes the `jwsc` Ant task against the updated JWS implementation class. Use the `compiledWsdl` attribute of `jwsc` to specify the name of the JAR file generated by the `wsdlc` Ant task:

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

```
<target name="build-service">

 <jwsc
    srcdir="src"
    destdir="${ear-dir}">
    <jws
       file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
       compiledWsdl="output/compiledWsdl/TemperatureService_wsdl.jar"
    />
 </jwsc>

</target>
```

10. Execute the `build-service` target to generate a deployable Web Service:

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

    You can iteratively keep rerunning this target if you want to update the JWS file bit by bit.

11. Start the WebLogic Server instance to which the Web Service will be deployed.

12. Deploy the Web Service, packaged in an Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `wsdlcEar` Enterprise application, located in the `output` directory.

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

    ```
    <taskdef name="wldeploy"
             classname="weblogic.ant.taskdefs.management.WLDeploy"/>

    <target name="deploy">

      <wldeploy action="deploy" name="wsdlcEar"
        source="output/wsdlcEar" user="${wls.username}"
        password="${wls.password}" verbose="true"
        adminurl="t3://${wls.hostname}:${wls.port}"
        targets="${wls.server.name}" />

    </target>
    ```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

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

    ```
    prompt> ant deploy
    ```

13. Test that the Web Service is deployed correctly by invoking its WSDL in your browser:

    `http://`*`host`*`:`*`port`*`/temp/TemperatureService?WSDL`

    The context path and service URI section of the preceding URL are specified by the original golden WSDL. Use the hostname and port relevant to your WebLogic Server

instance. Note that the deployed and original WSDL files are the same, except for the host and port of the endpoint address.

See "Invoking a Web Service from a Stand-alone JAX-RPC Java Client" on page 3-23 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

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

Sample WSDL File

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

<definitions
   name="TemperatureService"
   targetNamespace="http://www.bea.com/wls90"
   xmlns:tns="http://www.bea.com/wls90"
   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="zip" 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="literal"
                                 namespace="http://www.bea.com/wls90" />
                        </input>
                        <output>
                                <soap:body use="literal"
```

```
                                namespace="http://www.bea.com/wls90" />
                        </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://localhost:7001/temp/TemperatureService"/>
            </port>
        </service>

</definitions>
```

Sample TemperaturePortType Java Implementation File

```
package examples.webservices.wsdlc;

import javax.jws.WebService;
import weblogic.jws.*;

/**
 * TemperaturePortTypeImpl class implements web service endpoint interface
TemperaturePortType */

@WebService(
  serviceName="TemperatureService",
  endpointInterface="examples.webservices.wsdlc.TemperaturePortType")

@WLHttpTransport(
  contextPath="temp",
  serviceUri="TemperatureService",
  portName="TemperaturePort")
public class TemperaturePortTypeImpl implements TemperaturePortType {

  public TemperaturePortTypeImpl() {

  }

  public float getTemp(java.lang.String zip)

  {
     return 1.234f;
```

```
  }
}
```

Sample Ant Build File for TemperatureService

The following `build.xml` file uses properties to simplify the file.

```
<project default="all">

  <!-- set global properties for this build -->

  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />

  <property name="ear.deployed.name" value="wsdlcEar" />
  <property name="example-output" value="output" />
 <property name="compiledWsdl-dir" value="${example-output}/compiledWsdl"
/>
  <property name="impl-dir" value="${example-output}/impl" />
  <property name="ear-dir" value="${example-output}/wsdlcEar" />
 <property name="clientclass-dir" value="${example-output}/clientclasses"
/>

  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

  <taskdef name="wsdlc"
           classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>

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

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

  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>
```

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

<target name="clean" depends="undeploy">
  <delete dir="${example-output}"/>
</target>

<target name="generate-from-wsdl">

  <wsdlc
      srcWsdl="wsdl_files/TemperatureService.wsdl"
      destJwsDir="${compiledWsdl-dir}"
      destImplDir="${impl-dir}"
      packageName="examples.webservices.wsdlc" />

</target>

<target name="build-service">

  <jwsc
    srcdir="src"
    destdir="${ear-dir}">

    <jws file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
       compiledWsdl="${compiledWsdl-dir}/TemperatureService_wsdl.jar" />

  </jwsc>

</target>

<target name="deploy">
  <wldeploy action="deploy" name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>

<target name="undeploy">
  <wldeploy action="undeploy" name="${ear.deployed.name}"
    failonerror="false"
    user="${wls.username}" password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
```

```
      targets="${wls.server.name}" />
  </target>

  <target name="client">

    <clientgen

wsdl="http://${wls.hostname}:${wls.port}/temp/TemperatureService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.wsdlc.client"/>

    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/wsdlc/client/**/*.java"/>

  </target>

  <target name="run">
    <java classname="examples.webservices.wsdlc.client.TemperatureClient"
         fork="true" failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg
        line="http://${wls.hostname}:${wls.port}/temp/TemperatureService"
/>
    </java>

  </target>

</project>
```

Invoking a Web Service from a Stand-alone JAX-RPC Java Client

When you 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.

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:

- Java source code for the JAX-RPC `Stub` and `Service` interface implementations for the particular Web Service you want to invoke.
- Java classes for any user-defined XML Schema data types included in the WSDL file.
- 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.
- Client-side copy of the WSDL file.

The following example shows how to create a Java client application that invokes the `echoComplexType` operation of the `ComplexService` WebLogic Web Service described in "Creating a Web Service With User-Defined Data Types" on page 3-7. The `echoComplexType` operation takes as both a parameter and return type the `BasicStruct` user-defined data type. It is assumed in this procedure that you have already created and deployed the `ComplexService` Web Service.

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 `bin` subdirectory of 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 project directory:

   ```
   prompt> mkdir /myExamples/simple_client
   ```

3. Create a `src` directory under the project directory, as well as sub-directories that correspond to the package name of the Java client application (shown later on in this procedure):

   ```
   prompt> cd /myExamples/simple_client
   prompt> mkdir src/examples/webservices/simple_client
   ```

4. Create a standard Ant `build.xml` file in the project directory and add a `taskdef` Ant task to specify the fully Java classname of the `clientgen` task:

   ```
   <project name="webservices-simple_client" default="all">

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

   </project>
   ```

See "Sample Ant Build File For Building Stand-alone Client Application" on page 3-28 for a full sample `build.xml` file The full `build.xml` file uses properties, such as `${clientclass-dir}`, rather than always using the hard-coded name output directory for client classes.

5. Add the following calls to the `clientgen` and `javac` Ant tasks to the `build.xml` file, wrapped inside of the `build-client` target:

```
<target name="build-client">

  <clientgen
   wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
    destDir="output/clientclass"
    packageName="examples.webservices.simple_client"/>

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

  <javac
     srcdir="src" destdir="output/clientclass"
     includes="examples/webservices/simple_client/*.java"/>

</target>
```

   The `clientgen` Ant task uses the WSDL of the deployed `ComplexService` Web Service to generate the needed artifacts and puts them into the `output/clientclass` directory, using the specified package name. Replace the variables with the actual hostname and port of your WebLogic Server instance that is hosting the Web Service.

   The `clientgen` Ant task also automatically generates the `examples.webservices.complex.BasicStruct` JavaBean class, which is the Java representation of the user-defined data type specified in the WSDL.

   The `build-client` target also specifies the standard `javac` Ant task, in addition to `clientgen`, to compile all the Java code, including the stand-alone Java program described in the next step, into class files.

   The `clientgen` Ant task also provides the `destFile` attribute if you want the Ant task to automatically compile the generated Java code and package all artifacts into a JAR file. For details and an example, see clientgen.

6. Create the Java client application file that invokes the `echoComplexType` 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 3-27.

The `Main` client application takes a single argument: the WSDL URL of the Web Service. The application then follows standard JAX-RPC guidelines to invoke an operation of the Web Service using the Web Service-specific implementation 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 8, "Invoking Web Services."

7. Save the `Main.java` file in the `src/examples/webservices/simple_client` sub-directory of the main project directory.

8. Execute the `clientgen` and `javac` Ant tasks by specifying the `build-client` target at the command line:

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

   See the `output/clientclass` directory to view the files and artifacts generated by the `clientgen` Ant task.

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

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

   <target name="run" >

     <java fork="true"
           classname="examples.webservices.simple_client.Main"
           failonerror="true" >
       <classpath refid="client.class.path"/>
       <arg
   line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
       />
     </java>

   </target>
   ```

   The `run` target invokes the `Main` application, passing it the WSDL URL of the deployed Web Service as its single argument. The `classpath` element adds the `clientclass` directory to the CLASSPATH, using the reference created with the `<path>` task.

10. Execute the `run` target to invoke the `echoComplexType` operation:

    ```
    prompt> ant run
    ```

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

```
    run:
         [java] echoComplexType called. Result: 999, Hello Struct
```

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.webservices.simple_client;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;

// import the BasicStruct class, used as a param and return value of the
// echoComplexType operation.  The class is generated automatically by
// the clientgen Ant task.

import examples.webservices.complex.BasicStruct;

/**
 * This is a simple stand-alone client application that invokes the
 * the echoComplexType operation of the ComplexService Web service.
 *
 * @author Copyright (c) 2005 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] + "?WSDL" );
    ComplexPortType port = service.getComplexServicePort();

    BasicStruct in = new BasicStruct();

    in.setIntValue(999);
    in.setStringValue("Hello Struct");

    BasicStruct result = port.echoComplexType(in);
   System.out.println("echoComplexType called. Result: " + result.getIntValue()
+ ", " + result.getStringValue());
  }
}
```

Sample Ant Build File For Building Stand-alone Client Application

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-simple_client" default="all">

  <!-- set global properties for this build -->

  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />

  <property name="example-output" value="output" />
 <property name="clientclass-dir" value="${example-output}/clientclass" />

  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

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

  <target name="clean" >
    <delete dir="${clientclass-dir}"/>
  </target>

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

  <target name="build-client">

    <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.simple_client"/>

    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/simple_client/*.java"/>
  </target>
```

```
  <target name="run" >
    <java fork="true"
          classname="examples.webservices.simple_client.Main"
          failonerror="true" >
      <classpath refid="client.class.path"/>
     <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
 />
    </java>
  </target>

</project>
```

Invoking a Web Service from a WebLogic Web Service

You can also invoke a Web Service (WebLogic, .NET, and so on) from within a deployed WebLogic Web Service, rather than from a stand-alone client.

The procedure is similar to that described in "Invoking a Web Service from a Stand-alone JAX-RPC Java Client" on page 3-23 except that instead of running the `clientgen` Ant task to generate the client stubs, you use the `<clientgen>` child element of `<jws>`, inside of the `jwsc` Ant task, instead. The `jwsc` Ant task automatically packages the generated client stubs in the invoking Web Service WAR file so that the Web Service has immediate access to them. You then follow standard JAX-RPC programming guidelines in the JWS file that implements the Web Service that invokes the other Web Service.

The following example shows how to write a JWS file that invokes the `echoComplexType` operation of the `ComplexService` Web Service described in "Creating a Web Service With User-Defined Data Types" on page 3-7; it is assumed that you have successfully deployed the `ComplexService` Web Service.

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 `bin` subdirectory of 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 project directory:

   ```
   prompt> mkdir /myExamples/service_to_service
   ```

3. Create a `src` directory under the project directory, as well as sub-directories that correspond to the package name of the JWS and client application files (shown later on in this procedure):

```
prompt> cd /myExamples/service_to_service
prompt> mkdir src/examples/webservices/service_to_service
```

4. Create the JWS file that implements the Web Service that invokes the `ComplexService` Web Service. Open your favorite Java IDE or text editor and create a Java file called `ClientServiceImpl.java` using the Java code specified in "Sample ClientServiceImpl.java JWS File" on page 3-32.

   The sample JWS file shows a Java class called `ClientServiceImpl` that contains a single public method, `callComplexService()`. The Java class imports the JAX-RPC stubs, generated later on by the `jwsc` Ant task, as well as the `BasicStruct` JavaBean (also generated by `clientgen`), which is data type of the parameter and return value of the `echoComplexType` operation of the `ComplexService` Web Service.

   The `ClientServiceImpl` Java class defines one method, `callComplexService()`, which takes two parameters: a `BasicStruct` which is passed on to the `echoComplexType` operation of the `ComplexService` Web Service, and the URL of the `ComplexService` Web Service. The method then uses the standard JAX-RPC APIs to get the `Service` and `PortType` of the `ComplexService`, using the stubs generated by `jwsc`, and then invokes the `echoComplexType` operation.

5. Save the `ClientServiceImpl.java` file in the `src/examples/webservices/service_to_service` directory.

6. Create a standard Ant `build.xml` file in the project directory and add the following task:

   ```
   <project name="webservices-service_to_service" default="all">

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

   </project>
   ```

   The `taskdef` task defines the full classname of the `jwsc` Ant task.

   See "Sample Ant Build File For Building ClientService" on page 3-33 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `deploy`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

7. Add the following call to the `jwsc` Ant task to the `build.xml` file, wrapped inside of the `build-service` target:

   ```
   <target name="build-service">

     <jwsc
       srcdir="src"
   ```

```
    destdir="output/ClientServiceEar" >
    <jws

file="examples/webservices/service_to_service/ClientServiceImpl.java">
      <clientgen

wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
        packageName="examples.webservices.service_to_service" />
    </jws>
  </jwsc>
</target>
```

In the preceding example, the `<clientgen>` child element of the `<jws>` element of the `jwsc` Ant task specifies that, in addition to compiling the JWS file, `jwsc` should also generate and compile the client artifacts needed to invoke the Web Service described by the WSDL file.

8. Execute the `jwsc` Ant task by specifying the `build-service` target at the command line:

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

9. Start the WebLogic Server instance to which you will deploy the Web Service.

10. Deploy the Web Service, packaged in an Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `ClientServiceEar` Enterprise application, located in the `output` directory.

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

    ```
    <taskdef name="wldeploy"
             classname="weblogic.ant.taskdefs.management.WLDeploy"/>

    <target name="deploy">

      <wldeploy action="deploy" name="ClientServiceEar"
        source="ClientServiceEar" user="${wls.username}"
        password="${wls.password}" verbose="true"
        adminurl="t3://${wls.hostname}:${wls.port}"
        targets="${wls.server.name}" />

    </target>
    ```

    Substitute the values for `wls.username`, `wls.password`, `wls.hostname`, `wls.port`, and `wls.server.name` that correspond to your WebLogic Server instance.

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

    ```
    prompt> ant deploy
    ```

11. Test that the Web Service is deployed correctly by invoking its WSDL in your browser:

```
http://host:port/ClientService/ClientService?WSDL
```

See "Invoking a Web Service from a Stand-alone JAX-RPC Java Client" on page 3-23 for an example of creating a JAX-RPC Java client application that invokes a Web Service.

Sample ClientServiceImpl.java JWS File

```
package examples.webservices.service_to_service;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

import javax.jws.WebService;
import javax.jws.WebMethod;

import weblogic.jws.WLHttpTransport;

// Import the BasicStruct data type, generated by clientgen and used
// by the ComplexService Web Service

import examples.webservices.complex.BasicStruct;

// Import the JAX-RPC Stubs for invoking the ComplexService Web Service.
// Stubs generated by clientgen

import examples.webservices.service_to_service.ComplexPortType;
import examples.webservices.service_to_service.ComplexService_Impl;
import examples.webservices.service_to_service.ComplexService;

@WebService(name="ClientPortType", serviceName="ClientService",
            targetNamespace="http://examples.org")

@WLHttpTransport(contextPath="ClientService", serviceUri="ClientService",
                 portName="ClientServicePort")

public class ClientServiceImpl {

  @WebMethod()
  public String callComplexService(BasicStruct input, String serviceUrl)
      throws ServiceException, RemoteException
  {

    // Create service and port stubs to invoke ComplexService
    ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
    ComplexPortType port = service.getComplexServicePort();

    // Invoke the echoComplexType operation of ComplexService
    BasicStruct result = port.echoComplexType(input);
    System.out.println("Invoked ComplexPortType.echoComplexType." );
```

```
   return "Invoke went okay!  Here's the result: '" + result.getIntValue() + ",
" + result.getStringValue() + "'";

  }
}
```

Sample Ant Build File For Building ClientService

The following `build.xml` file uses properties to simplify the file.

```
<project name="webservices-service_to_service" default="all">

  <!-- set global properties for this build -->

  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />

  <property name="ear.deployed.name" value="ClientServiceEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/ClientServiceEar" />
  <property name="clientclass-dir" value="${example-output}/clientclasses" />

  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

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

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

  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>

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

  <target name="clean" depends="undeploy">
    <delete dir="${example-output}"/>
  </target>

  <target name="build-service">

    <jwsc
        srcdir="src"
        destdir="${ear-dir}" >
```

```
        <jws
         file="examples/webservices/service_to_service/ClientServiceImpl.java">
          <clientgen

wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
                packageName="examples.webservices.service_to_service" />
        </jws>

    </jwsc>

  </target>

  <target name="deploy">
    <wldeploy action="deploy" name="${ear.deployed.name}"
      source="${ear-dir}" user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>

  <target name="undeploy">
    <wldeploy action="undeploy" name="${ear.deployed.name}"
      failonerror="false"
      user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>

  <target name="client">

    <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/ClientService/ClientService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.service_to_service.client"/>

    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/service_to_service/client/**/*.java"/>

  </target>

  <target name="run">
    <java classname="examples.webservices.service_to_service.client.Main"
          fork="true"
          failonerror="true" >
          <classpath refid="client.class.path"/>
```

```
          <arg
line="http://${wls.hostname}:${wls.port}/ClientService/ClientService"/>
    </java>
  </target>
</project>
```

CHAPTER 4

Iterative Development of WebLogic Web Services

The following sections describe the iterative development process for WebLogic Web Services:

- "Integrating Web Services Into the WebLogic Split Development Directory Environment" on page 4-22

WARNING: Although both JAX-RPC 1.1 and JAX-WS 2.0 are supported in this release of WebLogic Server, this document concentrates almost exclusively on describing how to create JAX-RPC style Web Services. This is because, in this release, all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and WebLogic value-added features (such as asynchronous request-response and callbacks) work *only* with JAX-RPC style Web Services. Therefore, unless otherwise stated, you should assume that all descriptions and examples are for JAX-RPC Web Services.

For specific information about creating JAX-WS Web Services, see Chapter 6, "Implementing a JAX-WS 2.0 Web Service."

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 metadata feature, introduced in 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. For additional detailed information about this programming model, see "Anatomy of a WebLogic Web Service" on page 2-3.

The following sections describe the high-level steps for iteratively developing a Web Service, either starting from Java or starting from an existing WSDL file:

- "Iterative Development of WebLogic Web Services Starting From Java: Main Steps" on page 4-3
- "Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps" on page 4-5

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 process. 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.

Configuring Your Domain For Web Services Features

After you have created a WebLogic Server domain, you can use the Configuration Wizard to update the domain, using a Web Services-specific extension template, so that the resources required by certain WebLogic Web Services features are automatically configured. Although use of this extension template is not required, it makes the configuration of JMS and JDBC resources much easier.

The Web Services extension template automatically configures the resources required for the following features:

- Web Services Reliable Messaging
- Buffering
- JMS Transport

To update your domain so that it is automatically configured for these Web Services features:

1. Start the Configuration Wizard.
2. In the Welcome window, select Extend an Existing WebLogic Domain.
3. Click Next.
4. Select the domain to which you want to apply the extension template.
5. Click Next.
6. Select Extend My Domain Using an Existing Extension Template.
7. Enter the following value in the Template Location text box:

   WL_HOME`/common/templates/applications/wls_webservice.jar`

   where *WL_HOME* refers to the main WebLogic Server directory, such as `/bea_home/wlserver_10.0`.
8. Click Next.
9. If you want to further configure the JMS and JDBC resources, select Yes. This is not typical.

   Otherwise, click Next.
10. Verify that you are extending the correct domain, then click Extend.
11. Click Done to exit.

For detailed instructions about using the Configuration Wizard to create and update WebLogic Server domains, see Creating WebLogic Domains Using the Configuration Wizard.

Iterative Development of WebLogic Web Services Starting From Java: Main Steps

This section describes the general procedure for iteratively developing WebLogic Web Services starting from Java, if effect, coding the JWS file from scratch and later generating the WSDL file that describes the service. See Chapter 3, "Common Web Services Use Cases and Examples," for specific examples of this process. The following procedure is just a recommendation; if you have already set up your own development environment, you can use this procedure as a guide for updating your existing environment to develop WebLogic Web Services.

This procedure 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 4-21 for details.

To iteratively develop a WebLogic Web Service starting from Java, 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 the `bin` subdirectory of 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 project directory that will contain the JWS file, Java source for any user-defined data types, and the Ant `build.xml` file. You can name this directory anything you want.
3. In the project 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 5-19.
5. In the project directory, create a basic Ant build file called `build.xml`.

   See "Creating the Basic Ant build.xml File" on page 4-7.
6. Run the `jwsc` Ant task against the JWS file to generate source code, data binding artifacts, deployment descriptors, and so on, into an output directory. The `jwsc` Ant task generates an

Enterprise Application directory structure at this output directory; later you deploy this exploded directory to WebLogic Server as part of the iterative development process.

See "Running the jwsc WebLogic Web Services Ant Task" on page 4-7.

7. Deploy the Web Service to WebLogic Server.

   See "Deploying and Undeploying WebLogic Web Services" on page 4-15.

8. Invoke the WSDL of the Web Service to ensure that it was deployed correctly.

   See "Browsing to the WSDL of the Web Service" on page 4-17.

9. Test the Web Service using the WebLogic Web Services test client.

   See "Testing the Web Service" on page 4-20.

10. 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 4-15, 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.

Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps

This section describes the general procedure for iteratively developing WebLogic Web Services based on an existing WSDL file. See Chapter 3, "Common Web Services Use Cases and Examples," for a specific example of this process. The procedure is just a recommendation; if you have already set up your own development environment, you can use this procedure as a guide for updating your existing environment to develop WebLogic Web Services.

This procedure 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 4-21 for details.

It is assumed in this procedure that you already have an existing WSDL file.

To iteratively develop a WebLogic Web Service starting from WSDL, 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 the `bin` subdirectory of 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 project directory that will contain the generated artifacts and the Ant `build.xml` file. You can name this directory anything you want.
3. In the project directory, create a basic Ant build file called `build.xml`.

   See "Creating the Basic Ant build.xml File" on page 4-7.
4. Put your WSDL file in a directory that the `build.xml` Ant build file is able to read. For example, you can put the WSDL file in a `wsdl_files` child directory of the project directory.
5. Run the `wsdlc` Ant task against the WSDL file to generate the JWS interface, the stubbed-out JWS class file, JavaBeans that represent the XML Schema data types, and so on, into output directories.

   See "Running the wsdlc WebLogic Web Services Ant Task" on page 4-11.
6. Update the stubbed-out JWS file generated by the `wsdlc` Ant task, adding the business code to make the Web Service work as you want.

   See "Updating the Stubbed-Out JWS Implementation Class File Generated By wsdlc" on page 4-13.
7. Run the `jwsc` Ant task, specifying the artifacts generated by the `wsdlc` Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web Service.

   See "Running the jwsc WebLogic Web Services Ant Task" on page 4-7.
8. Deploy the Web Service to WebLogic Server.

   See "Deploying and Undeploying WebLogic Web Services" on page 4-15.
9. Invoke the deployed WSDL of the Web Service to test that the service was deployed correctly.

   The URL used to invoke the WSDL of the deployed Web Service is essentially the same as the value of the `location` attribute of the `<address>` element in the original WSDL (except for the host and port values which now correspond to the host and port of the WebLogic Server instance to which you deployed the service.) This is because the `wsdlc` Ant task generated values for the `contextPath` and `serviceURI` of the `@WLHttpTransport` annotation in the JWS implementation file so that together they create the same URI as the endpoint address specified in the original WSDL.

See either the original WSDL or "Browsing to the WSDL of the Web Service" on page 4-17 for information about invoking the deployed WSDL.

10. Test the Web Service using the WebLogic Web Services test client.

    See "Testing the Web Service" on page 4-20.

11. To make changes to the Web Service, update the generated JWS file, undeploy the Web Service as described in "Deploying and Undeploying WebLogic Web Services" on page 4-15, 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 a `<project>` root element 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 execute various stages of the Web Services development process, such as running the `jwsc` Ant task to process a JWS file and deploying the Web Service to WebLogic Server.

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

```
<project default="all">

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

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

  <target name="build-service">
     <!--add jwsc and related tasks 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. The JWS file can be either one you coded yourself from scratch or one generated by the `wsdlc` Ant task. The `jwsc`-generated artifacts include:

- Java source files that implement a standard JSR-109 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.

If you are running the `jwsc` Ant task against a JWS file generated by the `wsdlc` Ant task, the `jwsc` task does not generate these artifacts, because the `wsdlc` Ant task already generated them for you and packaged them into a JAR file In this case, you use an attribute of the `jwsc` Ant task to specify this `wsdlc`-generated JAR file.

After generating all the required artifacts, the `jwsc` Ant task compiles the Java files (including your JWS file), packages the compiled classes and generated artifacts into a deployable JAR archive file, and finally creates an exploded Enterprise Application directory that contains the JAR file.

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.JwscTask" />

<target name="build-service">

    <jwsc
      srcdir="src_directory"
      destdir="ear_directory"
      >
      <jws file="JWS_file"
```

```
            compiledWsdl="WSDLC_Generated_JAR" />
    </jwsc>
  </target>
```

where

- *ear_directory* refers to an Enterprise Application directory that will contain all the generated artifacts.
- *src_directory* refers to the top-level directory that contains subdirectories that correspond to the package name of your JWS file.
- *JWS_file* refers to the full pathname of your JWS file, relative to the value of the `src_directory` attribute.
- *WSDLC_Generated_JAR* refers to the JAR file generated by the `wsdlc` Ant task that contains the JWS interface file and data binding artifacts that correspond to an existing WSDL file.

  Note: You specify this attribute only in the "starting from WSDL" use case; this procedure is described in "Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps" on page 4-5.

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

Only the `srcdir` and `destdir` attributes of the `jwsc` Ant task are required. This means that, by default, it is assumed that Java files referenced by the JWS file (such as JavaBeans input parameters or user-defined exceptions) are in the same package as the JWS file. If this is not the case, use the `sourcepath` attribute to specify the top-level directory of these other Java files. See jwsc for more information.

Examples of Using jwsc

The following `build.xml` excerpt shows an example of running the `jwsc` Ant task on a JWS file:

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

<target name="build-service">

    <jwsc
      srcdir="src"
      destdir="output/helloWorldEar">
```

```
      <jws
         file="examples/webservices/hello_world/HelloWorldImpl.java" />
    </jwsc>
</target>
```

In the example, the Enterprise Application will be generated, in exploded form, in `output/helloWorldEar`, relative to the current directory. The JWS file is called `HelloWorldImpl.java`, and is located in the `src/examples/webservices/hello_world` directory, relative to the current directory. This implies that the JWS file is in the package `examples.webservices.helloWorld`.

The following example is similar to the preceding one, except that it uses the `compiledWsdl` attribute to specify the JAR file that contains `wsdlc`-generated artifacts (for the "starting with WSDL" use case):

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

<target name="build-service">

  <jwsc
    srcdir="src"
    destdir="output/wsdlcEar">

    <jws

        file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"

       compiledWsdl="output/compiledWsdl/TemperatureService_wsdl.jar" />

  </jwsc>

</target>
```

In the preceding example, the `TemperaturePortTypeImpl.java` file is the stubbed-out JWS file that you previously updated to include the business logic to make your service work as you want. Because the `compiledWsdl` attribute is specified and points to a JAR file, the `jwsc` Ant task does not regenerate the artifacts that are included in the JAR.

To actually run this task, type at the command line the following:

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

Advanced Uses of jwsc

This section described two very simple examples of using the `jwsc` Ant task. The task, however, includes additional attributes and child elements that make the tool very powerful and useful. For example, you can use the tool to:

- Process multiple JWS files at once. You can choose to package each resulting Web Service into its own Web application WAR file, or group all of the Web Services into a single WAR file.
- Specify the transports (HTTP/HTTPS/JMS) that client applications can use when invoking the Web Service, possibly overriding any existing `@WLXXXTransport` annotations.
- Automatically generate the JAX-RPC client stubs of any other Web Service that is invoked within the JWS file.
- Update an existing Enterprise Application or Web application, rather than generate a completely new one.

See "jwsc" on page A-17 for complete documentation and examples about the `jwsc` Ant task.

Running the wsdlc WebLogic Web Services Ant Task

The `wsdlc` Ant task takes as input a WSDL file and generates artifacts that together partially implement a WebLogic Web Service. These artifacts include:

- The JWS interface file that represents the Java implementation of your Web Service.
- Data binding artifacts used by WebLogic Server to convert between the XML and Java representations of the Web Service parameters and return values.
- A JWS file that contains a stubbed-out implementation of the generated JWS interface.
- Optional Javadocs for the generated JWS interface.

The `wsdlc` Ant task packages the JWS interface file and data binding artifacts together into a JAR file that you later specify to the `jwsc` Ant task. You never need to update this JAR file; the only file you update is the JWS implementation class.

To run the `wsdlc` Ant task, add the following `taskdef` and `generate-from-wsdl` targets to the `build.xml` file:

```
<taskdef name="wsdlc"
        classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
```

```
<target name="generate-from-wsdl">
  <wsdlc
      srcWsdl="WSDL_file"
      destJwsDir="JWS_interface_directory"
      destImplDir="JWS_implementation_directory"
      packageName="Package_name" />
</target>
```

where

- *WSDL_file* refers to the name of the WSDL file from which you want to generate a partial implementation, including its absolute or relative pathname.
- *JWS_interface_directory* refers to the directory into which the JAR file that contains the JWS interface and data binding artifacts should be generated.

  The name of the generated JAR file is *WSDLFile*_wsdl.jar, where *WSDLFile* refers to the root name of the WSDL file. For example, if the name of the WSDL file you specify to the file attribute is MyService.wsdl, then the generated JAR file is MyService_wsdl.jar.
- *JWS_implementation_directory* refers to the top directory into which the stubbed-out JWS implementation file is generated. The file is generated into a sub-directory hierarchy corresponding to its package name.

  The name of the generated JWS file is *PortType*Impl.java, where *PortType* refers to the name attribute of the <portType> element in the WSDL file for which you are generating a Web Service. For example, if the port type name is MyServicePortType, then the JWS implementation file is called MyServicePortTypeImpl.java.
- *Package_name* refers to the package into which the generated JWS interface and implementation files should be generated. If you do not specify this attribute, the wsdlc Ant task generates a package name based on the targetNamespace of the WSDL.

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

Only the srcWsdl and destJwsDir attributes of the wsdlc Ant task are required. Typically, however, you also generate the stubbed-out JWS file to make your programming easier. BEA also recommends you explicitly specify the package name in case the targetNamespace of the WSDL file is not suitable to be converted into a readable package name.

The following build.xml excerpt shows an example of running the wsdlc Ant task against a WSDL file:

```
<taskdef name="wsdlc"
        classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>

<target name="generate-from-wsdl">

    <wsdlc
        srcWsdl="wsdl_files/TemperatureService.wsdl"
        destJwsDir="output/compiledWsdl"
        destImplDir="impl_output"
        packageName="examples.webservices.wsdlc" />

</target>
```

In the example, the existing WSDL file is called `TemperatureService.wsdl` and is located in the `wsdl_files` subdirectory of the directory that contains the `build.xml` file. The JAR file that will contain the JWS interface and data binding artifacts is generated to the `output/compiledWsdl` directory; the name of the JAR file is `TemperatureService_wsdl.jar`. The package name of the generated JWS files is `examples.webservices.wsdld`. The stubbed-out JWS file is generated into the `impl_output/examples/webservices/wsdlc` directory relative to the current directory. Assuming that the port type name in the WSDL file is `TemperaturePortType`, then the name of the JWS implementation file is `TemperaturePortTypeImpl.java`.

To actually run this task, type the following at the command line:

```
prompt> ant generate-from-wsdl
```

See "wsdlc" on page A-52 for additional attributes of the `wsdlc` Ant task.

Updating the Stubbed-Out JWS Implementation Class File Generated By wsdlc

The `wsdlc` Ant task generates the stubbed-out JWS implementation file into the directory specified by its `destImplDir` attribute; the name of the file is *`PortType`*`Impl.java`, where *`PortType`* is the name of the portType in the original WSDL. The class file includes everything you need to compile it into a Web Service, except for your own business logic in the methods that implement the operations.

The JWS class implements the JWS Web Service endpoint interface that corresponds to the WSDL file; the JWS interface is also generated by `wsdlc` and is located in the JAR file that contains other artifacts, such as the Java representations of XML Schema data types in the WSDL and so on. The public methods of the JWS class correspond to the operations in the WSDL file.

The wsdlc Ant task automatically includes the `@WebService` and `@WLHttpTransport` annotations in the JWS implementation class; the values of the attributes correspond to equivalent values in the WSDL. For example, the `serviceName` attribute of `@WebService` is the same as the `name` attribute of the `<service>` element in the WSDL file; the `contextPath` and `serviceUri` attributes of `@WLHttpTransport` together make up the endpoint address specified by the `location` attribute of the `<address>` element in the WSDL.

When you update the JWS file, you add Java code to the methods so that the corresponding Web Service operations works as you want. Typically, the generated JWS file contains comments where you should add code, such as:

```
//replace with your impl here
```

You can also add additional JWS annotations to the file, with the following restrictions:

- The *only* standard JWS annotations (in the `javax.jws.*` package) you can include in the JWS implementation file are `@WebService`, `@HandlerChain`, `@SOAPMessageHandler`, and `@SOAPMessageHandlers`. If you specify any other standard JWS annotations, the `jwsc` Ant task returns error when you try to compile the JWS file into a Web Service.
- You can specify *only* the `serviceName` and `endpointInterface` attributes of the `@WebService` annotation. Use the `serviceName` attribute to specify a different `<service>` WSDL element from the one that the `wsdlc` Ant task used, in the rare case that the WSDL file contains more than one `<service>` element. Use the `endpointInterface` attribute to specify the JWS interface generated by the `wsdlc` Ant task.
- You can specify any WebLogic-specific JWS annotation that you want.

After you have updated the JWS file, BEA recommends that you move it to an official source location, rather than leaving it in the `wsdlc` output location.

The following example shows the `wsdlc`-generated JWS implementation file from the WSDL shown in "Sample WSDL File" on page 3-19; the text in bold indicates where you would add Java code to implement the single operation (`getTemp`) of the Web Service:

```
package examples.webservices.wsdlc;

import javax.jws.WebService;
import weblogic.jws.*;

/**
 * TemperaturePortTypeImpl class implements web service endpoint interface
 * TemperaturePortType */
```

```
@WebService(
  serviceName="TemperatureService",
  endpointInterface="examples.webservices.wsdlc.TemperaturePortType")

@WLHttpTransport(
  contextPath="temp",
  serviceUri="TemperatureService",
  portName="TemperaturePort")

public class TemperaturePortTypeImpl implements TemperaturePortType {

  public TemperaturePortTypeImpl() {

  }

  public float getTemp(java.lang.String zipcode)

  {

    //replace with your impl here

     return 0;

  }

}
```

Deploying and Undeploying WebLogic Web Services

Because Web Services are packaged as Enterprise Applications, deploying a Web Service simply means deploying the corresponding EAR file or exploded directory.

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` Ant task. 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="Source" user="AdminUser"
      password="AdminPassword"
      adminurl="AdminServerURL"
      targets="ServerName"/>

</target>
```

where

- *DeploymentName* refers to the deployment name of the Enterprise Application, or the name that appears in the Administration Console under the list of deployments.
- *Source* refers to the name of the Enterprise Application EAR file or exploded directory that is being deployed. By default, the `jwsc` Ant task generates an exploded Enterprise Application directory.
- *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 Enterprise Application exploded directory, located in the `output/ComplexServiceEar` directory relative to the current directory, be deployed to the `myServer` WebLogic Server instance. Its deployed name is `ComplexServiceEar`.

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

```
    <wldeploy action="deploy"
      name="ComplexServiceEar"
      source="output/ComplexServiceEar" user="weblogic"
      password="weblogic" verbose="true"
      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="ComplexServiceEar"
    user="weblogic"
    password="weblogic" verbose="true"
    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 Enterprise application. For more information on the Administration Console, see the Online Help.

Browsing to the WSDL of the Web Service

You can display the WSDL of the Web Service in your browser to ensure that it has deployed correctly.

The following URL shows how to display the Web Service WSDL in your browser:

```
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 context root of the Web Service. There are many places to set the context root (the `contextPath` attribute of the `@WLHttpTransport` annotation, the `<WLHttpTransport>`, `<module>`, or`<jws>` element of `jwsc`) and certain methods take precedence over others. See "How to Determine the Final Context Root of a WebLogic Web Service" on page A-19 for a complete explanation.
- *serviceUri* refers to the value of the `serviceUri` attribute of the `@WLHttpTransport` JWS annotation of the JWS file that implements your Web Service or `<WLHttpTransport>` child element of the jwsc Ant task; the second takes precedence over the first.

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

```
...
@WLHttpTransport(contextPath="complex",
                 serviceUri="ComplexService",
                 portName="ComplexServicePort")
/**
 * This JWS file forms the basis of a WebLogic Web Service.
 *
 */
public class ComplexServiceImpl {
...
```

Further assume that you do *not* override the `contextPath` or `serviceURI` values by setting equivalent attributes for the `<WLHttpTransport>` element of the `jwsc` Ant task. Then the URL

to view the WSDL of the Web Service, assuming the service is running on a host called `ariel` at the default port number (`7001`), is:

```
http://ariel:7001/complex/ComplexService?WSDL
```

Configuring the Server Address Specified in the Dynamic WSDL

The WSDL of a deployed Web Service (also called *dynamic WSDL*) includes an `<address>` element that assigns an address (URI) to a particular Web Service port. For example, assume that the following WSDL snippet partially describes a deployed WebLogic Web Service called `ComplexService`:

```
<definitions name="ComplexServiceDefinitions"
             targetNamespace="http://example.org">

...

  <service name="ComplexService">
    <port binding="s0:ComplexServiceSoapBinding" name="ComplexServicePort">
      <s1:address location="http://myhost:7101/complex/ComplexService"/>
    </port>
  </service>

</definitions>
```

The preceding example shows that the `ComplexService` Web Service includes a port called `ComplexServicePort`, and this port has an address of `http://myhost:7101/complex/ComplexService`.

WebLogic Server determines the `complex/ComplexService` section of this address by examining the `contextPath` and `serviceURI` attributes of the `WL`*`XXX`*`Transport` annotations or `jwsc` elements, as described in "Browsing to the WSDL of the Web Service" on page 4-17. However, the method WebLogic Server uses to determine the protocol and host section of the address (`http://myhost:7101`, in the example) is more complicated, as described below. For clarity, this section uses the term *server address* to refer to the protocol and host section of the address.

The server address that WebLogic Server publishes in a dynamic WSDL of a deployed Web Service depends on whether the Web Service can be invoked using HTTP/S or JMS, whether you have configured a proxy server, whether the Web Service is deployed to a cluster, or whether the Web Service is actually a callback service. The following sections reflect these different configuration options, and provide links to procedural information about changing the

configuration to suit your needs. It is assumed in the sections that you use the WebLogic Server Administration Console to configure cluster and standalone servers.

Web Service is not a callback service and can be invoked using HTTP/S

1. If the Web Service is deployed to a cluster, and the cluster `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` are set, then WebLogic Server uses these values in the server address of the dynamic WSDL.

   See Configure HTTP Settings for a Cluster.

2. If the preceding cluster values are not set, but the `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` values are set for the *individual server* to which the Web Service is deployed, then WebLogic Server uses these values in the server address.

   See Configure HTTP Protocol.

3. If these values are not set for either the cluster or an individual server, then WebLogic Server uses the server address of the WSDL request in the dynamic WSDL as well.

Web Service is not a callback service and can be invoked using JMS Transport

1. If the Web Service is deployed to a cluster and the `Cluster Address` is set, then WebLogic Server uses this value in the server address of the dynamic WSDL.

   See Configure Clusters.

2. If the cluster address is not set, or the Web Service is deployed to a standalone server, and the `Listen Address` of the server to which the Web Service is deployed is set, then WebLogic Server uses this value in the server address.

   See Configure Listen Addresses.

Web Service is a callback service

1. If the callback service is deployed to a cluster, and the cluster `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` are set, then WebLogic Server uses these values in the server address of the dynamic WSDL.

   See Configure HTTP Settings for a Cluster.

2. If the callback service is deployed to either a cluster or a standalone server, and the preceding cluster values are not set, but the `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` values are set for the *individual server* to which the callback service is deployed, then WebLogic Server uses these values in the server address.

   See Configure HTTP Protocol.

3. If the callback service is deployed to a cluster, but none of the preceding values are set, but the `Cluster Address` is set, then WebLogic Server uses this value in the server address.

   See Configure Clusters.

4. If none of the preceding values are set, but the `Listen Address` of the server to which the callback service is deployed is set, then WebLogic Server uses this value in the server address.

   See Configure Listen Addresses.

Web Service is invoked using a proxy server.

Although not required, BEA recommends that you explicitly set the `Frontend Host`, `FrontEnd HTTP Port`, and `Frontend HTTPS Port` of either the cluster or individual server to which the Web Service is deployed to point to the proxy server.

See Configure HTTP Settings for a Cluster or Configure HTTP Protocol.

Testing the Web Service

After you have deployed a WebLogic Web Service, you can use the Web Services Test Client, included in the WebLogic Administration Console, to test your service without writing code. You can quickly and easily test any Web Service, including those with complex types and those using advanced features of WebLogic Server such as conversations. The test client automatically maintains a full log of requests allowing you to return to previous call to view the results.

To test a deployed Web Service using the Administration Console, follow these steps:

1. Invoke the Administration Console 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`).

2. Follow the procedure described in Test a Web Service.

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 Java Platform, Enterprise Edition (Java EE) Version 5 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 and the `splitdir/helloWorldEar` example installed with WebLogic Server, located in the *`BEA_HOME`*`/wlserver_10.0/samples/server/examples/src/examples` 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/helloWorldEar`, then create a directory called `/src/helloWorldEar/helloWebService`:

   ```
   prompt> mkdir /src/helloWorldEar/helloWebService
   ```

2. Create a directory hierarchy under the `helloWebService` directory that corresponds to the package name of your JWS file.

   For example, if your JWS file is in the package `examples.splitdir.hello` package, then create a directory hierarchy `examples/splitdir/hello`:

   ```
   prompt> cd /src/helloWorldEar/helloWebService
   prompt> mkdir examples/splitdir/hello
   ```

3. Put your JWS file in the just-created Web Service subdirectory of your main project directory (`/src/helloWorldEar/helloWebService/examples/splitdir/hello` in this example.)

4. 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 4-7.

   The `jwsc` `srcdir` attribute should point to the top-level directory that contains the JWS file (`helloWebService` in this example). The `jwsc` `destdir` attribute should point to the same destination directory you specify for `wlcompile`, as shown in the following example:

   ```
   <target name="build.helloWebService">
   ```

```
    <jwsc
        srcdir="helloWebService"
        destdir="destination_dir"
        keepGenerated="yes" >

        <jws file="examples/splitdir/hello/HelloWorldImpl.java" />

    </jwsc>

  </target>
```

In the example, *destination_dir* refers to the destination directory that the other split development directory environment Ant tasks, such as `wlappc` and `wlcompile`, also use.

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-helloWebService,compile,appc" />
```

WARNING: When you actually build your Enterprise Application, be sure you run the `jwsc` Ant task *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-helloWebService` target *before* the `appc` target.

6. If you use the `wlcompile` and `wlappc` Ant tasks to compile and validate the entire Enterprise Application, be sure to exclude the Web Service source directory for both Ant tasks. This is because the `jwsc` Ant task already took care of compiling and packaging the Web Service. For example:

```
<target name=”compile”>

   <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
           excludes="appStartup,helloWebService">
   ...
   </wlcomplile>

...
</target>

<target name="appc">

   <wlappc source="${dest.dir}" deprecation="yes" debug="false"
           excludes="helloWebService"/>

</target>
```

7. Update the `application.xml` file in the `META-INF` project source directory, adding a `<web>` module and specifying the name of the WAR file generated by the `jwsc` Ant task.

For example, add the following to the `application.xml` file for the helloWorld Web Service:

```
<application>
...
  <module>
    <web>
      <web-uri>examples/splitdir/hello/HelloWorldImpl.war</web-uri>
      <context-root>/hello</context-root>
    </web>
  </module>
...
</application>
```

Caution: The `jwsc` Ant task always generates a Web Application WAR file from the JWS file that implements your Web Service, unless your JWS file explicitly implements `javax.ejb.SessionBean`. In that case you must add an `<ejb>` module element to the `application.xml` file instead.

Your split development directory environment is now updated to include Web Service development. When you rebuild and deploy 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 "Browsing to the WSDL of the Web Service" on page 4-17.

CHAPTER 5

Programming the JWS File

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

WARNING: Although both JAX-RPC 1.1 and JAX-WS 2.0 are supported in this release of WebLogic Server, this document concentrates almost exclusively on describing how to create JAX-RPC style Web Services. This is because, in this release, all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and WebLogic value-added features (such as asynchronous request-response and

callbacks) work *only* with JAX-RPC style Web Services. Therefore, unless otherwise stated, you should assume that all descriptions and examples are for JAX-RPC Web Services.

For specific information about creating JAX-WS Web Services, see Chapter 6, "Implementing a JAX-WS 2.0 Web Service."

Overview of JWS Files and JWS Annotations

One way to program a WebLogic Web Service is to code the standard JSR-109 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). This process can be difficult and tedious. BEA recommends that you take advantage of the metadata annotations feature, new in JDK 5.0, 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 5.0 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.

This topic is part of the iterative development procedure for creating a Web Service, described in "Iterative Development of WebLogic Web Services Starting From Java: Main Steps" on page 4-4 and "Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps" on page 4-5. It is assumed that you have created a JWS file and now want to add JWS annotations to it.

Programming the JWS File: Java Requirements

When you program your JWS file, you must follow a set of requirements, as specified by the JSR-181 specification (*Web Services Metadata for the Java Platform*). In particular, the 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 a service endpoint interface by using the `@WebService.endpointInterface` annotation. In this case, it is assumed that the service endpoint interface exists and you cannot specify any other JWS annotations in the JWS file other than `@WebService.endpointInterface` and `@WebService.serviceName`.
- If JWS file does not implement a service endpoint interface, all public methods other than those inherited from `java.lang.Object` will be exposed as Web Service operations. This behavior can be overridden by using the `@WebMethod` annotation to specify explicitly those public methods that are to be exposed. If a `@WebMethod` annotation is present, only the methods to which it is applied are exposed.

Programming the JWS File: Typical Steps

The following sections how to use standard (JSR-181) and WebLogic-specific annotations in your JWS file to program basic Web Service features. The annotations are used at different levels, or targets, in your JWS file. Some are used at the class-level to indicate that the annotation applies to the entire JWS file. Others are used at the method-level and yet others at the parameter level. The sections discuss the following basic JWS annotations:

- `@WebService` (standard)
- `@SOAPBinding` (standard)
- `@WLHttpTransport` (WebLogic-specific)
- `@WebMethod` (standard)
- `@Oneway` (standard)
- `@WebParam` (standard)
- `@WebResult` (standard)

See WebLogic Web Services: Advanced Programming for information on using other JWS annotations to program more advanced features, such as Web Service reliable messaging, conversations, SOAP message handlers, and so on.

For reference documentation about the WebLogic-specific JWS annotations, see JWS Annotation Reference.

The following procedure describes the typical basic steps when programming the JWS file that implements a Web Service. See "Example of a JWS File" on page 5-5 for a code example.

1. Import the standard JWS annotations that will be 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 required `@WebService` JWS annotation at the class level to specify that the Java class exposes a Web Service.

   See "Specifying That the JWS File Implements a Web Service" on page 5-6.

4. Optionally 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.

   Although this JWS annotation is not required, BEA recommends you explicitly specify it in your JWS file to clarify the type of SOAP bindings a client application uses to invoke the Web Service.

   See "Specifying the Mapping of the Web Service to the SOAP Message Protocol" on page 5-7.

5. Optionally 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.

   Although this JWS annotation is not required, BEA recommends you explicitly specify it in your JWS file so that it is clear what URL a client application uses to invoke the Web Service.

   See "Specifying the Context Path and Service URI of the Web Service" on page 5-8.

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

   See "Specifying That a JWS Method Be Exposed as a Public Operation" on page 5-8.

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

See “Customizing the Mapping Between Operation Parameters and WSDL Parts” on page 5-9.

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

   See “Customizing the Mapping Between the Operation Return Value and a WSDL Part” on page 5-10.

9. Add business Java code to the methods to make the WebService behave the way you want.

Example of a JWS File

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

```
package examples.webservices.simple;

// 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 porType name of the Web
// Service is "SimplePortType", the service name is "SimpleService", and the
// targetNamespace used in the generated WSDL is "http://example.org"

@WebService(name="SimplePortType", serviceName="SimpleService",
            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 document-literal-wrapped.

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

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

@WLHttpTransport(contextPath="simple", serviceUri="SimpleService",
                 portName="SimpleServicePort")
```

```
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 */

public class 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 JWS File Implements a Web Service

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

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

In the example, the name of the Web Service is `SimplePortType`, which will later map to the `wsdl:portType` element in the WSDL file generated by the `jwsc` Ant task. The service name is `SimpleService`, 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 attribute of the `@WebService` annotation:

- `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 message protocol; for this reason, your JWS file should 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 excerpt:

```
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```

In the example, the Web Service uses document-wrapped-style encodings and literal message formats, which are also the default formats if you do not specify the `@SOAPBinding` annotation.

You can also use the WebLogic-specific `@weblogic.jws.soap.SOAPBinding` annotation to specify the SOAP binding at the method level; the attributes are the same as the standard `@javax.jws.soap.SOAPBinding` annotation.

You 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` (either the standard or WebLogic-specific) 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 excerpt:

```
@WLHttpTransport(contextPath="simple", serviceUri="SimpleService",
                 portName="SimpleServicePort")
```

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 `SimpleServicePort`. The URL used to invoke the Web Service over HTTP includes a context path of `simple` and a service URI of `SimpleService`, as shown in the following example:

`http://`*`host`*`:`*`port`*`/simple/SimpleService`

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

Specifying That a JWS 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 SimpleImpl {

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

In the example, the `sayHello()` method of the `SimpleImpl` JWS file is exposed as a public operation of the Web Service. The `operationName` attribute specifies, however, that the public name of the operation in the WSDL file is `sayHelloOperation`. 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 OneWayImpl {

  @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 checked 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.

If none of the public methods in your JWS file are annotated with the `@WebMethod` annotation, then by default *all* public methods are exposed as Web Service operations.

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 SimpleImpl {

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

```
  {
      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 (`WebParam.Mode.IN`, `WebParam.Mode.OUT`, or `WebParam.Mode.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 operations 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 shown in the following code excerpt:

```
public class Simple {

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

```
                targetNamespace="http://example.org/docLiteralBare")
        int input)
    {
        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 `return`. 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.

Accessing Runtime Information about a Web Service Using the JwsContext

When a client application invokes a WebLogic Web Service that was implemented with a JWS file, WebLogic Server automatically creates a *context* that the Web Service can use to access, and sometimes change, runtime information about the service. Much of this information is related to conversations, such as whether the current conversation is finished, the current values of the conversational properties, changing conversational properties at runtime, and so on. (See Creating Conversational Web Services for information about conversations and how to implement them.) Some of the information accessible via the context is more generic, such as the protocol that was used to invoke the Web Service (HTTP/S or JMS), the SOAP headers that were in the SOAP message request, and so on.

You can use annotations and WebLogic Web Service APIs in your JWS file to access runtime context information, as described in the following sections.

Guidelines for Accessing the Web Service Context

The following example shows a simple JWS file that uses the context to determine the protocol that was used to invoke the Web Service; the code in bold is discussed in the programming guidelines described after the example.

```
package examples.webservices.jws_context;

import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Context;

import weblogic.wsee.jws.JwsContext;
import weblogic.wsee.jws.Protocol;

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

@WLHttpTransport(contextPath="contexts", serviceUri="JwsContext",
                 portName="JwsContextPort")

/**
 * Simple web service to show how to use the @Context annotation.
 */

public class JwsContextImpl {

  @Context
  private JwsContext ctx;

  @WebMethod()
  public String getProtocol() {

    Protocol protocol = ctx.getProtocol();

    System.out.println("protocol: " + protocol);
    return "This is the protocol: " + protocol;
  }
}
```

Use the following guidelines in your JWS file to access the runtime context of the Web Service, as shown in the code in bold in the preceding example:

- Import the `@weblogic.jws.Context` JWS annotation:

  ```
  import weblogic.jws.Context;
  ```

- Import the `weblogic.wsee.jws.JwsContext` API, as well as any other related APIs that you might use (the example also uses the `weblogic.wsee.jws.Protocol` API):

  ```
  import weblogic.wsee.jws.JwsContext;
  import weblogic.wsee.jws.Protocol;
  ```

  See the `weblogic.wsee.*` Javadocs for reference documentation about the context-related APIs.

- Annotate a private variable, of data type `weblogic.wsee.jws.JwsContext`, with the field-level `@Context` JWS annotation:

  ```
  @Context
  private JwsContext ctx;
  ```

  WebLogic Server automatically assigns the annotated variable (in this case, `ctx`) with a runtime implementation of `JwsContext` the first time the Web Service is invoked, which is how you can later use the variable without explicitly initializing it in your code.

- Use the methods of the `JwsContext` class to get, and sometimes change, runtime information about the Web Service. The following example shows how to get the protocol that was used to invoke the Web Service:

  ```
  Protocol protocol = ctx.getProtocol();
  ```

  See "Methods of the JwsContext" on page 5-13 for the full list of available methods.

Methods of the JwsContext

The following table briefly describes the methods of the `JwsContext` that you can use in your JWS file to access runtime information about the Web Service. See `weblogic.wsee.*` Javadocs for detailed reference information about `JwsContext`, and other context-related APIs, as `Protocol` and `ServiceHandle`.

Table 5-2 Methods of the JwsContext

Method	Returns	Description
isFinished()	boolean	Returns a boolean value specifying whether the current conversation is finished, or if it is still continuing. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
finishConversation()	void	Finishes the current conversation. This method is equivalent to a client application invoking a method that has been annotated with the `@Conversation (Conversation.Phase.FINISH)` JWS annotation. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 5-2 Methods of the JwsContext

Method	Returns	Description
setMaxAge(java.util.Date)	void	Sets a new maximum age for the conversation to an absolute `Date`. If the date parameter is in the past, WebLogic Server immediately finishes the conversation. This method is equivalent to the `maxAge` attribute of the `@Conversational` annotation, which specifies the *default* maximum age of a conversation. Use this method to override this default value at runtime. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
setMaxAge(String)	void	Sets a new maximum age for the conversation by specifying a `String` duration, such as `1 day`. Valid values for the `String` parameter are a number and one of the following terms: • `seconds` • `minutes` • `hours` • `days` • `years` For example, to specify a maximum age of ten minutes, use the following syntax: `ctx.setMaxAge("10 minutes")` This method is equivalent to the `maxAge` attribute of the `@Conversational` annotation, which specifies the *default* maximum age of a conversation. Use this method to override this default value at runtime. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
getMaxAge()	long	Returns the maximum allowed age, in seconds, of a conversation. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 5-2 Methods of the JwsContext

Method	Returns	Description
getCurrentAge()	long	Returns the current age, in seconds, of the conversation. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
resetIdleTime()	void	Resets the timer which measures the number of seconds since the last activity for the current conversation. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
setMaxIdleTime(long)	void	Sets the number of seconds that the conversation can remain idle before WebLogic Server finishes it due to client inactivity. This method is equivalent to the `maxIdleTime` attribute of the `@Conversational` annotation, which specifies the *default* idle time of a conversation. Use this method to override this default value at runtime. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 5-2 Methods of the JwsContext

Method	Returns	Description
setMaxIdleTime(String)	void	Sets the number of seconds, specified as a `String`, that the conversation can remain idle before WebLogic Server finishes it due to client inactivity. Valid values for the `String` parameter are a number and one of the following terms: • `seconds` • `minutes` • `hours` • `days` • `years` For example, to specify a maximum idle time of ten minutes, use the following syntax: `ctx.setMaxIdleTime("10 minutes")` This method is equivalent to the `maxIdleTime` attribute of the `@Conversational` annotation, which specifies the *default* idle time of a conversation. Use this method to override this default value at runtime. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
getMaxIdleTime()	long	Returns the number of seconds that the conversation is allowed to remain idle before WebLogic Server finishes it due to client inactivity. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
getCurrentIdleTime()	long	Gets the number of seconds since the last client request, or since the conversation's maximum idle time was reset. Use this method only in conversational Web Services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
getCallerPrincipal()	java.security.Principal	Returns the security principal associated with the operation that was just invoked, assuming that basic authentication was performed.

Table 5-2 Methods of the JwsContext

Method	Returns	Description
isCallerInRole(String)	boolean	Returns `true` if the authenticated principal is within the specified security role.
getService()	weblogic.wsee.jws.ServiceHandle	Returns an instance of `ServiceHandle`, a WebLogic Web Service API, which you can query to gather additional information about the Web Service, such as the conversation ID (if the Web Service is conversational), the URL of the Web Service, and so on.
getLogger(String)	weblogic.wsee.jws.util.Logger	Gets an instance of the `Logger` class, which you can use to send messages from the Web Service to a log file.
getInputHeaders()	org.w3c.dom.Element[]	Returns an array of the SOAP headers associated with the SOAP request message of the current operation invoke.
setUnderstoodInputHeaders(boolean)	void	Indicates whether input headers should be understood.
getUnderstoodInputHeaders()	boolean	Returns the value that was most recently set by a call to setUnderstoodInputHeader.
setOutputHeaders(Element[])	void	Specifies an array of SOAP headers that should be associated with the outgoing SOAP response message sent back to the client application that initially invoked the current operation.
getProtocol()	weblogic.wsee.jws.Protocol	Returns the protocol (such as HTTP/S or JMS) used to invoke the current operation.

Should You Implement a Stateless Session EJB?

The `jwsc` Ant task always chooses a plain Java object as the underlying implementation of a Web Service when processing your JWS file.

Sometimes, however, you might want the underlying implementation of your Web Service to be a stateless session EJB so as to take advantage of all that EJBs have to offer, such as instance pooling, transactions, security, container-managed persistence, container-managed relationships, and data caching. If you decide you want an EJB implementation for your Web Service, then follow the programming guidelines in the following section.

Programming Guidelines When Implementing an EJB in Your JWS File

The general guideline is to always use EJBGen annotations in your JWS file to automatically generate, rather than manually create, the EJB Remote and Home interface classes and deployment descriptor files needed when implementing an EJB. EJBGen annotations work in the same way as JWS annotations: they follow the JDK 5.0 metadata syntax and greatly simplify your programming tasks.

For more information on EJBGen, see the EJBGen Reference section in Programming WebLogic Enterprise JavaBeans.

Follow these guidelines when explicitly implementing a stateless session EJB in your JWS file. See "Example of a JWS File That Implements an EJB" on page 5-19 for an example; the relevant sections are shown in bold:

- Import the standard Java Platform, Enterprise Edition (Java EE) Version 5 EJB classes:

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

- Import the EJBGen annotations, all of which are in the `weblogic.ejbgen` package. At a minimum you need to import the `@Session` annotation; if you want to use additional EJBGen annotations in your JWS file to specify the shape and behavior of the EJB, see the EJBGen reference guide for the name of the annotation you should import.

  ```
  import weblogic.ejbgen.Session;
  ```

- At a minimum, use the `@Session` annotation at the class level to specify the name of the EJB:

  ```
  @Session(ejbName="TransactionEJB")
  ```

  `@Session` is the only required EJBGen annotation when used in a JWS file. You can, if you want, use other EJBGen annotations to specify additional features of the EJB.

- Ensure that the JWS class implements `SessionBean`:

  ```
  public class TransactionImpl implements SessionBean {...
  ```

- You must also include the standard EJB methods `ejbCreate()`, `ejbActivate()` and so on, although you typically do not need to add code to these methods unless you want to change the default behavior of the EJB:

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

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

If you follow all these guidelines in your JWS file, the `jwsc` Ant task later compiles the Web Service into an EJB and packages it into an EJB JAR file inside of the Enterprise Application.

Example of a JWS File That Implements an EJB

The following example shows a simple JWS file that implement a stateless session EJB. The relevant code is shown in bold.

```
package examples.webservices.transactional;

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

import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Transactional;

import weblogic.ejbgen.Session;

@Session(ejbName="TransactionEJB")

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

@WLHttpTransport(contextPath="transactions", serviceUri="TransactionService",
                 portName="TransactionPort")

/**
 * This JWS file forms the basis of simple EJB-implemented WebLogic
 * Web Service with a single operation: sayHello.  The operation executes
 * as part of a transaction.
 *
 */

public class TransactionImpl implements SessionBean {

  @WebMethod()
  @Transactional(value=true)

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

  // 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) {}
}
```

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 representation 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 7-6. See "Supported Built-In Data Types" on page 7-2 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.webservices.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;

  // 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;
  }

}
```

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 ComplexImpl.java JWS File" on page 3-11:

```
package examples.webservices.complex;

// Import the standard JWS annotation interfaces

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

// Import the WebLogic-specific JWS annotation interface

import weblogic.jws.WLHttpTransport;

// Import the BasicStruct JavaBean

import examples.webservices.complex.BasicStruct;

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

...

public class ComplexImpl {

  @WebMethod(operationName="echoComplexType")
  public BasicStruct echoStruct(BasicStruct struct)

  {
    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 can be accessed in the JWS file by calling the static method `weblogic.wsee.util.WLSOAPFactory.createSOAPFactory()` which returns a `javax.xml.soap.SOAPFactory object`. Then at runtime, use the `-Djavax.xml.soap.SOAPFactory` flag to specify BEA's `SOAPFactory` implementation as shown:

```
-Djavax.xml.soap.SOAPFactory=weblogic.xml.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.webservices.soap_exceptions;

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 the @WebService annotation

import javax.jws.WebService;

// Import WLHttpTransport

import weblogic.jws.WLHttpTransport;

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

```
@WLHttpTransport(contextPath="exceptions",
                 serviceUri="SoapExceptionsService",
                 portName="SoapExceptionsServicePort")
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHelloWorld
 *
 * @author Copyright (c) 2005 by BEA Systems.  All rights reserved.
 */

public class SoapExceptionsImpl {

  public SoapExceptionsImpl() {

  }

  public void tirarSOAPException() {

    Detail detail = null;

    try {

      SOAPFactory soapFactory = SOAPFactory.newInstance();
      detail = soapFactory.createDetail();

    } catch (SOAPException e) {
        // do something
    }

    QName faultCode = null;
    String faultString = "the fault string";
    String faultActor = "the fault actor";
    throw new SOAPFaultException(faultCode, faultString, faultActor, detail);
  }
}
```

The preceding example uses the default implementation of `SOAPFactory`.

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.

Invoking Another Web Service from the JWS File

From within your JWS file you can invoke another Web Service, either one deployed on WebLogic Server or one deployed on some other application server, such as .NET. The steps to do this are similar to those described in "Invoking a Web Service from a Stand-alone JAX-RPC Java Client" on page 3-23, except that rather than running the `clientgen` Ant task to generate the client stubs, you include a `<clientgen>` child element of the `jwsc` Ant task that builds the invoking Web Service to generate the client stubs instead. You then use the standard JAX-RPC APIs in your JWS file the same as you do in a stand-alone client application.

See "Invoking a Web Service from Another Web Service" on page 8-12 for detailed instructions.

Programming Additional Miscellaneous Features Using JWS Annotations and APIs

The following sections describe additional miscellaneous features you can program by specifying particular JWS annotations in your JWS file or using WebLogic Web Services APIs:

Sending Binary Data Using MTOM/XOP

MTOM/XOP describes a method for optimizing the transmission of XML data of type `xs:base64Binary` in SOAP messages. When the transport protocol is HTTP, MIME attachments are used to carry that data while at the same time allowing both the sender and the receiver direct access to the XML data in the SOAP message without having to be aware that any MIME artifacts were used to marshal the `xs:base64Binary` data.

MTOM/XOP support is standard in JAX-WS 2.0 via the use of JWS annotations. For details, see the JAX-WS section in the Sun Developer Network. In this release of WebLogic Server, JAX-RPC-style Web Services also support it.

The MTOM specification does not require that, when MTOM is enabled, the Web Service runtime use XOP binary optimization when transmitting `base64binary` data. Rather, the

specification allows the runtime to choose to do so. This is because in certain cases the runtime may decide that it is more efficient to send `base64binary` data directly in the SOAP Message; an example of such a case is when transporting small amounts of data in which the overhead of conversion and transport consumes more resources than just inlining the data as is. The WebLogic Web Services implementation for MTOM for JAX-RPC service, however, *always* uses MTOM/XOP when MTOM is enabled.

Support for MTOM/XOP in WebLogic JAX-RPC Web Services is implemented using the pre-packaged WS-Policy file `Mtom.xml`. WS-Policy files follow the WS-Policy specification; this specification provides a general purpose model and XML syntax to describe and communicate the policies of a Web Service, in this case the use of MTOM/XOP to send binary data. The installation of the pre-packaged `Mtom.xml` WS-Policy file in the `types` section of the Web Service WSDL is as follows (provided for your information only; you cannot change this file):

```
<wsp:Policy wsu:Id="myService_policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <wsoma:OptimizedMimeSerialization

xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/optimizedmimeseriali
zation" />
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

When you deploy the compiled JWS file to WebLogic Server, the dynamic WSDL will automatically contain the following snippet that references the MTOM WS-Policy file; the snippet indicates that the Web Service uses MTOM/XOP:

```
<wsdl:binding name="BasicHttpBinding_IMtomTest"
                      type="i0:IMtomTest">
    <wsp:PolicyReference URI="#myService_policy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
```

You can associate the `Mtom.xml` WS-Policy file with a Web Service at development-time by specifying the `@Policy` metadata annotation in your JWS file. Be sure you also specify the `attachToWsdl=true` attribute to ensure that the dynamic WSDL includes the required reference to the `Mtom.xml` file; see the example below.

You can associate the `Mtom.xml` WS-Poligy file with a Web Service at deployment time by modifying the WSDL by adding the Policy to the types section just before deployment.

You can also attach the file at runtime using by the Administration Console; for details, see Associate a WS-Policy File With A Web Service. This section describes how to use the JWS annotation.

WARNING: In this release of WebLogic Server, the only supported Java data type when using MTOM/XOP is `byte[]`; other binary data types, such as `image`, are not supported.

In addition, this release of WebLogic Server does not support using MTOM with message-level security. In practical terms this means that you cannot specify the `Mtom.xml` WS-Policy file together with the `Encrypt.xml`, `Sign.xml`, `Wssc-dk.xml`, or `Wssc-sct.xml` WS-Policy files in the same JAX-RPC Web Service.

To send binary data using MTOM/XOP, follow these steps:

1. Use the WebLogic-specific `@weblogic.jws.Policy` annotation in your JWS file to specify that the pre-packaged `Mtom.xml` file should be applied to your Web Service, as shown in the following simple JWS file (relevant code shown in bold):

```
package examples.webservices.mtom;

import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Policy;

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

@WLHttpTransport(contextPath="mtom",
                 serviceUri="MtomService",
                 portName="MtomServicePort")

@Policy(uri="policy:Mtom.xml", attachToWsdl=true)

public class MtomImpl {

  @WebMethod
  public String echoBinaryAsString(byte[] bytes) {
    return new String(bytes);
  }
```

2. Use the Java `byte[]` data type in your Web Service operations as either a return value or input parameter whenever you want the resulting SOAP message to use MTOM/XOP to send or

receive the binary data. See the implementation of the `echoBinaryAsString` operation above for an example; this operation simply takes as input an array of `byte` and returns it as a `String`.

3. The WebLogic Web Services runtime has built in MTOM/XOP support which is enabled if the WSDL that the `clientgen` Ant task generates client-side artifacts for specifies MTOM/XOP support. In your client application itself, simply invoke the operations as usual, using `byte[]` as the relevant data type.

See the SOAP Message Transmission Optimization Mechanism specification for additional information about the MTOM/XOP feature itself as well as the version of the specification supported by WebLogic JAX-RPC Web Services.

Streaming SOAP Attachments

Using the `@weblogic.jws.StreamAttachments` JWS annotation, you can specify that a Web Service use a streaming API when reading inbound SOAP messages that include attachments, rather than the default behavior in which the service reads the entire message into memory. This feature increases the performance of Web Services whose SOAP messages are particular large.

See weblogic.jws.StreamAttachments for an example of specifying that attachments should be streamed.

Using SOAP 1.2

WebLogic Web Services use, by default, Version 1.1 of Simple Object Access Protocol (SOAP) as the message format when transmitting data and invocation calls between the Web Service and its client. To specify that the Web Service use Version 1.2 of SOAP, use the class-level `@weblogic.jws.Binding` annotation in your JWS file and set its single attribute to the value `Binding.Type.SOAP12`, as shown in the following example (relevant code shown in bold):

```
package examples.webservices.soap12;

import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Binding;

@WebService(name="SOAP12PortType",
            serviceName="SOAP12Service",
            targetNamespace="http://example.org")
```

```
@WLHttpTransport(contextPath="soap12",
                 serviceUri="SOAP12Service",
                 portName="SOAP12ServicePort")
@Binding(Binding.Type.SOAP12)
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello.  The class uses SOAP 1.2
 * as its binding.
 *
 */
public class SOAP12Impl {
  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

Other than set this annotation, you do not have to do anything else for the Web Service to use SOAP 1.2, including changing client applications that invoke the Web Service; the WebLogic Web Services runtime takes care of all the rest.

Specifying that Operations Run Inside of a Transaction

When a client application invokes a WebLogic Web Service operation, the operation invocation takes place outside the context of a transaction, by default. If you want the operation to run inside a transaction, specify the `@weblogic.jws.Transactional` annotation in your JWS file, and set the boolean `value` attribute to `true`, as shown in the following example (relevant code shown in bold):

```
package examples.webservices.transactional;

import javax.jws.WebMethod;
import javax.jws.WebService;

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

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

@WLHttpTransport(contextPath="transactionsPojo",
                 serviceUri="TransactionPojoService",
                 portName="TransactionPojoPort")

/**
 * This JWS file forms the basis of simple WebLogic
 * Web Service with a single operation: sayHello.  The operation executes
 * as part of a transaction.
 *
 */

public class TransactionPojoImpl {

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

If you want *all* operations of a Web Service to run inside of a transaction, specify the `@Transactional` annotation at the class-level. If you want only a subset of the operations to be transactional, specify the annotation at the method-level. If there is a conflict, the method-level value overrides the class-level.

See weblogic.jws.Transactional for information about additional attributes.

Getting the HttpServletRequest/Response Object

If your Web Service uses HTTP as its transport protocol, you can use the `weblogic.wsee.connection.transport.servlet.HttpTransportUtils` API to get the `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` objects from the JAX-RPC `ServletEndpointContext` object, as shown in the following example (relevant code shown in bold and explained after the example):

```
package examples.webservices.http_transport_utils;
```

```
import javax.xml.rpc.server.ServiceLifecycle;
import javax.xml.rpc.server.ServletEndpointContext;
import javax.xml.rpc.ServiceException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.WLHttpTransport;

import weblogic.wsee.connection.transport.servlet.HttpTransportUtils;

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

@WLHttpTransport(contextPath="servlet", serviceUri="HttpTransportUtils",
                 portName="HttpTransportUtilsPort")

public class HttpTransportUtilsImpl implements ServiceLifecycle {

  private ServletEndpointContext wsctx = null;

  public void init(Object context) throws ServiceException {
    System.out.println("ServletEndpointContext inited...");
    wsctx = (ServletEndpointContext)context;
  }

  public void destroy() {
    System.out.println("ServletEndpointContext destroyed...");
    wsctx = null;
  }

  @WebMethod()
  public String getServletRequestAndResponse() {

    HttpServletRequest request =
       HttpTransportUtils.getHttpServletRequest(wsctx.getMessageContext());
    HttpServletResponse response =
       HttpTransportUtils.getHttpServletResponse(wsctx.getMessageContext());

    System.out.println("HttpTransportUtils API used successfully.");
    return "HttpTransportUtils API used successfully";

  }

}
```

The important parts of the preceding example are as follows:

- Import the required JAX-RPC and Servlet classes:

```
import javax.xml.rpc.server.ServiceLifecycle;
import javax.xml.rpc.server.ServletEndpointContext;
import javax.xml.rpc.ServiceException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

- Import the WebLogic `HttpTransportUtils` class:

```
import weblogic.wsee.connection.transport.servlet.HttpTransportUtils;
```

- Because you will be querying the JAX-RPC message context, your JWS file must explicitly implement `ServiceLifecycle`:

```
public class HttpTransportUtilsImpl implements ServiceLifecycle
```

- Create a variable of data type `ServletEndpointContext`:

```
 private ServletEndpointContext wsctx = null;
```

- Because the JWS file implements `ServiceLifecycle`, you must also implement the `init` and `destroy` lifecycle methods:

```
  public void init(Object context) throws ServiceException {
    System.out.println("ServletEndpointContext inited...");
    wsctx = (ServletEndpointContext)context;
  }
  public void destroy() {
    System.out.println("ServletEndpointContext destroyed...");
    wsctx = null;
  }
```

- Finally, in the method that implements the Web Service operation, use the `ServletEndpointContext` object to get the `HttpServletRequest` and `HttpServletResponse` objects:

```
HttpServletRequest request =
  HttpTransportUtils.getHttpServletRequest(wsctx.getMessageContext());
HttpServletResponse response =
  HttpTransportUtils.getHttpServletResponse(wsctx.getMessageContext());
```

JWS Programming Best Practices

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

- When you create a document-literal-bare 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.

  Because of the nature of document-literal-bare 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, document-literal-wrapped Web Services are the most interoperable type of Web Service.

- 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 `return`, which is the default name of the returned value if you do not use the `@WebResult` annotation in your JWS file.

- 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

Implementing a JAX-WS 2.0 Web Service

The following topics describe why and how to implement a JAX-WS 2.0 Web Service:

Implementing a JAX-WS Web Service: Overview

Although both Java API for XML-Based RPC (JAX-RPC) 1.1 and Java API for XML Web Services (JAX-WS) 2.0 are supported in this release of WebLogic Server, the WebLogic Web Services documentation concentrates almost exclusively on describing how to create JAX-RPC-based Web Services. This is because all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and the WebLogic value-added features (such as asynchronous request-response and callbacks) work *only* with JAX-RPC style Web Services. For this reason, unless you specifically want to create a JAX-WS Web Service, it is assumed that you want to create a JAX-RPC service so that you can take full advantage of all features provided by WebLogic Server. This section, however, describes why and how to implement a JAX-WS 2.0 Web Service.

Reasons to implement a Web Service based on JAX-WS 2.0 include the following:

- JAX-WS 2.0 fully supports the Java Architecture for XML Binding (JAXB) 2.0 specification and thus provides *full* XML Schema support. JAXB provides a convenient way to bind an XML schema to a representation in Java code. This makes it easy for you

to incorporate XML data and processing functions in applications based on Java technology without having to know much about XML itself.

By contrast, the built-in and user-defined data types you can use in a JAX-RPC-style Web Service, although extensive, is limited to those described in "Data Types and Data Binding" on page 7-1.

- The JAX-WS 2.0 programming model is very similar to that of JAX-RPC 1.1 Web Services in that it uses metadata annotations described in the *Web Services Metadata for the Java Platform (JSR 181)* specification and then Ant tasks to compile the annotated Java file into a deployable enterprise application (EAR) file. However, the JAX-WS 2.0 programming model is more robust because it defines additional annotations, listed in the JAX-WS 2.0 specification, that you can use to customize the mapping from Java to XML schema/WSDL and to map Web Service operation parameter names to meaningful part/element names in the WSDL file.

- JAX-WS 2.0 defines two types of handlers: logical and protocol handlers. While protocol handlers have access to an entire message such as a SOAP message, logical handlers deal only with the payload of a message and are independent of the protocol being used. Handler chains can now be configured on a per-port, per-protocol, or per-service basis. A new framework of context objects has been added to allow client code to share information easily with handlers.

  By contrast, in JAX-RPC 1.1 Web Services you could program only SOAP handlers.

- JAX-WS 2.0 supports MTOM (Message Transmission and Optimization Mechanism). MTOM, together with XOP (XML Binary Optimized Packaging) defines how an XML binary data such as `xs:base64Binary` or `xs:hexBinary` can be optimally transmitted over the wire.

  Note: In this release of WebLogic Server, MTOM is also supported for JAX-RPC 1.1 style Web Services.

- The JAX-WS 2.0 specification defines standard and portable XML-based customizations. These customizations, or binding declarations, can customize almost all WSDL components that can be mapped to Java, such as the service endpoint interface class, method name, parameter name, exception class, etc. Using binding declarations you can also control certain features, such as asynchrony, provider, wrapper style, and additional headers.

- Finally, the JAX-RPC specification is not evolving, and the future of Web Services is in JAX-WS.

For additional documentation and examples about programming the preceding features in a JAX-WS Web Service, see the JAX-WS 2.0 User's Guide on java.net.

Implementing a JAX-WS Web Service: Guidelines

Implementing a JAX-WS Web Service is similar to implementing a JAX-RPC Web Service, and for the most part you can follow the procedures described in Chapter 4, "Iterative Development of WebLogic Web Services." There are, however, a few differences as described below:

- The set of metadata annotations you can use in your JWS file include those listed in the following specifications:
  - Web Services Metadata for the Java Platform (JSR 181)
  - Java API for XML Web Services (JAX-WS) 2.0 (JSR 224)
  - Java Architecture for XML Binding (JAXB) 2.0 (JSR 222)
  - Common Annotations for the Java Platform (JSR 250)

  See Java API for XML Web Services Annotations for additional information.
- You cannot use any of the WebLogic-specific annotations in your JWS file because they apply to JAX-RPC 1.1 Web Services *only*. The WebLogic-specific JWS anntoations are those described in JWS Annotation Reference.
- You cannot use any of the WebLogic Web Services APIs in a JAX-WS Web Service. An example of a WebLogic Web Service API is weblogic.wsee.reliability.
- You cannot use any of the features described by a WS-* specification with a JAX-WS Web Service. For example, you cannot make a JAX-WS Web Service reliable, as specified by WS-ReliableMessaging.
- When you use the `jwsc`, `wsdlc`, or `clientgen` Ant tasks, be sure to use the new `type` attribute to specify that the task should compile or generate client-side artifacts for a JAX-WS Web Service rather than the default JAX-RPC Web Service. See "Specifying a JAX-WS Web Service to the jwsc and clientgen Ant Tasks" on page 6-5 for an example.
- Depending on the type of Web Service you are generating (JAX-WS or JAX-RPC), some of the attributes of the jwsc, wsdlc, and clientgen Ant tasks may not apply. See the WebLogic Web Service Ant Task Reference for information about each attribute and whether it applies to JAX-WS Web Services, JAX-RPC, or both.

Simple Example of Implementing a JAX-WS Web Service

The following sections show a simple example of implementing a JAX-WS Web Service

Example of a JWS File That Implements a JAX-WS Web Service

The following Java file shows a simple example of implementing a JAX-WS Web Service.

The example uses the standard `@javax.jws.WebService` annotation to declare that it is a Web Service, and then the common `@javax.annotation.Resource` annotation to inject the Web Service context into the `context` variable. With the context one can get information about the Web Service; in this case, the principal user that invokes the service.

```
package examples.webservices.jaxws;

import javax.jws.WebService;

import javax.xml.ws.WebServiceContext;

import javax.annotation.Resource;

@WebService()

/**
 * This JWS file forms the basis of simple JAX-WS WebLogic Web Service
 *
 */

public class JaxWsImpl {

  @Resource
  private WebServiceContext context;

  public String sayHello(String message) {

    String principal = context.getUserPrincipal().getName();

    System.out.println("Hello! Here is the passed=in message: " + message + ".
And here is the user principal: " + principal + ".");

   return "Here is the message: '" + message + "'. And here is the user principal:
'" + principal + "'." ;

  }

}
```

Specifying a JAX-WS Web Service to the jwsc and clientgen Ant Tasks

The following excerpt from a `build.xml` Ant build file shows how to use the `type` attribute of the `jwsc` Ant task to specify that the task should generate a JAX-WS Web Service, rather than the default JAX-RPC service:

```
<jwsc
  srcdir="src"
  destdir="${ear-dir}">

  <jws file="examples/webservices/jaxws/JaxWsImpl.java"
       type="JAXWS"
  />

</jwsc>
```

Similarly, the following call to `clientgen` shows how to specify that the task should generate client-side artifacts used to invoke a JAX-WS Web Service:

```
<clientgen
 type="JAXWS"
 wsdl="http://${wls.hostname}:${wls.port}/JaxWsImpl/JaxWsImplService?WSDL"
 destDir="${clientclass-dir}"
 packageName="examples.webservices.jaxws.client"/>
```

Example of Invoking a JAX-WS Web Service

The following simple standalone Java client shows how to invoke the Web Service implemented in the preceding sections:

```
package examples.webservices.jaxws.client;

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

public class Main {
```

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

    JaxWsImplService service = new JaxWsImplService();
    JaxWsImpl port = service.getJaxWsImplPort();

    String result = null;
    result = port.sayHello("Hi there!");
    System.out.println( "Got result: " + result );
  }
}
```

CHAPTER 7

Data Types and Data Binding

The following sections provide information about supported data types (both built-in and user-defined) and data binding:

WARNING: Although both JAX-RPC 1.1 and JAX-WS 2.0 are supported in this release of WebLogic Server, this document concentrates almost exclusively on describing how to create JAX-RPC style Web Services. This is because, in this release, all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and WebLogic value-added features (such as asynchronous request-response and callbacks) work *only* with JAX-RPC style Web Services. Therefore, unless otherwise stated, you should assume that all descriptions and examples are for JAX-RPC Web Services. In the case of supported data types and data binding, for example, this document does *not* describe how to use JAXB.

For specific information about creating JAX-WS Web Services and JAXB, see Chapter 6, "Implementing a JAX-WS 2.0 Web Service."

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 Apache XmlBeans (in package `org.apache.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 representation is used in the JWS that implements the Web Service. The conversion of data between its XML and Java representations is called *data binding*.

WARNING: As of WebLogic Server 9.1, using XMLBeans 1.X data types (in other words, extensions of `com.bea.xml.XmlObject`) as parameters or return types of a WebLogic Web Service is deprecated. New applications should use XMLBeans 2.x data types.

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 `wsdl2c` Ant tasks that can automatically generate the data binding artifacts for most user-defined data types. See "Supported User-Defined Data Types" on page 7-6 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 7-5.

Table 7-1 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
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

Table 7-1 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
anyURI	java.net.URI
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[]
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

Table 7-1 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
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 7-8.

Table 7-2 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

Table 7-2 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Schema Data Type
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
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 `wsdlc` Ant tasks can automatically 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 7-2, 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 `wsdlc` 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 7-3 Supported User-Defined XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
`<xsd:complexType>` with elements of both simple and complex types.	JavaBean
`<xsd:complexType>` with simple content.	JavaBean
`<xsd:attribute>` in `<xsd:complexType>`	Property of a JavaBean
Derivation of new simple types by restriction of an existing simple type.	Equivalent Java data type of simple type.
Facets used with restriction element.	Facets not 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:any>`	`javax.xml.soap.SOAPElement` or `org.apache.xmlbeans.XmlObject`
`<xsd:any[]>`	`javax.xml.soap.SOAPElement[]` or `org.apache.xmlbeans.XmlObject[]`
`<xsd:union>`	Common parent type of union members.

Table 7-3 Supported User-Defined XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
<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 `jwsc` and `wsdlc` Ant tasks and their equivalent XML Schema data type.

Table 7-4 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 and multidimensional array of any supported data type (when used as a 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>

Table 7-4 Supported User-Defined Java Data Types

Java Data Type	Equivalent XML Schema Data Type
Apache XMLBeans (that are inherited from `org.apache.xmlbeans.XmlObject` only) **Note:** A Web Service that uses an Apache XMLBeans data type as a return type or parameter must be defined as document-literal-wrapped or document-literal-bare.	See Apache XMLBeans.
java.util.Collection	Literal Array
java.util.List	Literal Array
java.util.ArrayList	Literal Array
java.util.LinkedList	Literal Array
java.util.Vector	Literal Array
java.util.Stack	Literal Array
java.util.Set	Literal Array
java.util.TreeSet	Literal Array
java.utils.SortedSet	Literal Array
java.utils.HashSet	Literal Array

Note: The following user-defined Java data type, used as a parameter or return value of a WebLogic Web Service in Version 8.1, is no longer supported:

- JAX-RPC-style enumeration class

Additionally, generics are not supported when used as a parameter or return value. For example, the following Java method cannot be exposed as a public operation:

```
public ArrayList<String> echoGeneric(ArrayList<String> in) {
    return in;
}
```

CHAPTER 8

Invoking Web Services

The following sections describe how to invoke WebLogic Web Services:

Note: The following sections do not include information about invoking message-secured Web Services; for that topic, see the Web Services security guide, in particular Updating a Client Application to Invoke a Message-Secured Web Service.

WARNING: Although both JAX-RPC 1.1 and JAX-WS 2.0 are supported in this release of WebLogic Server, this document concentrates almost exclusively on describing how to create JAX-RPC style Web Services. This is because, in this release, all the WS-* specifications (such as WS-Security and WS-ReliableMessaging) and WebLogic value-added features (such as asynchronous request-response and

callbacks) work *only* with JAX-RPC style Web Services. Therefore, unless otherwise stated, you should assume that all descriptions and examples are for JAX-RPC Web Services.

For specific information about creating and invoking JAX-WS Web Services, see Chapter 6, "Implementing a JAX-WS 2.0 Web Service."

Overview of Web Services Invocation

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: In this context, a *client application* can be two types of clients: One is a stand-alone client that uses the WebLogic client classes to invoke a Web Service hosted on WebLogic Server or on other application servers. In this document, a *stand-alone client* is a client that has a runtime environment independent of WebLogic Server. The other type of client application that invokes a Web Service runs inside a Java Platform, Enterprise Edition (Java EE) Version 5 component deployed to WebLogic Server, such as an EJB or another Web Service.

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 stand-alone client application or one that runs as part of a WebLogic Server.

WARNING: You cannot use a dynamic client to invoke a Web Service operation that implements user-defined data types as parameters or return values. A dynamic client uses the JAX-RPC Call interface. Standard (static) clients use the Service and Stub JAX-RPC interfaces, which correctly invoke Web Services that implement user-defined data types.

Types of Client Applications

This section describes two different types of client applications:

- Stand-alone—A stand-alone client application, in its simplest form, is a Java program that has the `Main` public class that you invoke with the `java` command. It runs completely separately from WebLogic Server.

- A Java EE component deployed to WebLogic Server—In this type of client application, the Web Service invoke is part of the code that implements an EJB, servlet, or another Web Service. This type of client application, therefore, runs inside a WebLogic Server container.

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. WebLogic Server implements the JAX-RPC 1.1 specification.

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

Table 8-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Service	Main client interface.
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.

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

The clientgen Ant Task

The `clientgen` WebLogic Web Services Ant task generates, from an existing WSDL file, the client artifacts 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 Ant Task Reference.

WARNING: The fully qualified name of the `clientgen` Ant task supported in this release of WebLogic Server is `weblogic.wsee.tools.anttasks.ClientGenTask`. This is different from the `clientgen` Ant task supported in version 8.1 of WebLogic Server, which is `weblogic.webservice.clientgen`.

Although the 8.1 `clientgen` Ant task is still provided in this release of WebLogic Server, it is deprecated. If you want to generate the client artifacts to invoke a 9.X WebLogic Web Service, be sure you use the 9.X version of `clientgen` and not the 8.1 version. For example, if you have upgraded an 8.1 Web Service to 10.0, but your Ant scripts explicitly call the 8.1 `clientgen` Ant task by specifying its fully qualified name, then you must update your Ant scripts to call the 9.X `clientgen` instead.

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 Server 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 expand the **WebLogic Server Examples->Examples->API->Web Services** node.

Invoking a Web Service from a Stand-alone Client: Main Steps

In the following procedure it is assumed 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 4-7. For a full example of a `build.xml` file used in this section, see "Sample Ant Build File for a Stand-Alone Java Client" on page 8-11.

To create a Java stand-alone client application that invokes a Web Service:

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 8-5.

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 8-6.

4. Write the client application Java code that includes code for invoking the Web Service operation.

   See "Writing the Java Client Application Code to Invoke a Web Service" on page 8-8.

5. Compile and run your Java client application.

   See "Compiling and Running the Client Application" on page 8-9.

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">
```

```
    <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
      destDir="clientclasses"
      packageName="examples.webservices.simple_client"/>
</target>
```

Before you can 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` and `destDir` attributes of the `clientgen` Ant task to specify the WSDL file from which you want to create client-side artifacts and the directory into which these artifacts should be generated. The `packageName` attribute is optional; if you do not specify it, the clientgen task uses a package name based on the `targetNamespace` of the WSDL.

Note: The `clientgen` Ant task also provides the `destFile` attribute if you want the Ant task to automatically compile the generated Java code and package all artifacts into a JAR file. For details and an example, see clientgen.

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 that is the Java representation of the XML Schema data type defined in the WSDL. The JavaBean classes are generated into the `destDir` directory.

Note: The package of the Java user-defined data type is based on the XML Schema of the data type in the WSDL, which is different from the package name of the JAX-RPC stubs.

See "Sample Ant Build File for a Stand-Alone Java Client" on page 8-11 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.

The best way to get this information is to use the `clientgen` Ant task to generate the Web Service-specific JAX-RPC stubs and look at the generated `*.java` files. These files are generated into the directory specified by the `destDir` attribute, with subdirectories corresponding to either the value of the `packageName` attribute, or, if this attribute is not specified, to a package based on the `targetNamespace` of the WSDL.

- The *`ServiceName`*`.java` source file contains the `get`*`PortName`*`()` methods for getting the Web Service port, where *`ServiceName`* refers to the name of the Web Service and *`PortName`* refers to the name of the port. If the Web Service was implemented with a JWS file, the name of the Web Service is the value of the `serviceName` attribute of the `@WebService` JWS annotation and the name of the port is the value of the `portName` attribute of the `@WLHttpTransport` 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 the Web Service was implemented with a JWS file, the port type is the value of the `name` attribute of the `@WebService` JWS annotation.

You can also examine the actual WSDL of the Web Service; see "Browsing to the WSDL of the Web Service" on page 4-18 for details about the WSDL of a deployed WebLogic 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 to Invoke a Web Service

In the following code example, a stand-alone application invokes 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 and the Web Service-specific implementation of the `Service` interface, generated by `clientgen`, to invoke an operation of the Web Service.

The example also shows how to invoke an operation that has a user-defined data type (`examples.webservices.complex.BasicStruct`) as an input parameter and return value. The `clientgen` Ant task automatically generates the Java code for this user-defined data type.

```
package examples.webservices.simple_client;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;

// import the BasicStruct class, used as a param and return value of the
// echoComplexType operation.  The class is generated automatically by
// the clientgen Ant task.

import examples.webservices.complex.BasicStruct;

/**
 * This is a simple stand-alone client application that invokes the
 * the echoComplexType operation of the ComplexService Web service.
 *
 * @author Copyright (c) 2005 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] + "?WSDL");
    ComplexPortType port = service.getComplexServicePort();

    BasicStruct in = new BasicStruct();

    in.setIntValue(999);
    in.setStringValue("Hello Struct");
```

```
    BasicStruct result = port.echoComplexType(in);
   System.out.println("echoComplexType called. Result: " + result.getIntValue()
+ ", " + result.getStringValue());
  }
}
```

In the preceding example:

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

  ```
  ComplexService service = new ComplexService_Impl (args[0] + "?WSDL");
  ComplexPortType port = service.getComplexServicePort();
  ```

  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] + "?WSDL"`). The `getComplexServicePort()` method is used to return an instance of the `ComplexPortType` stub implementation.

- The following code shows how to invoke the `echoComplexType` operation of the `ComplexService` Web Service:

  ```
  BasicStruct result = port.echoComplexType(in);
  ```

  The `echoComplexType` operation returns the user-defined data type called `BasicStruct`.

The method of your application that invokes the Web Service operation must throw or catch `java.rmi.RemoteException` and `javax.xml.rpc.ServiceException`, both of which are thrown from the generated JAX-RPC stubs.

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 by the bold text in the following example

```
<target name="build-client">

  <clientgen
   wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
    destDir="clientclasses"
    packageName="examples.webservices.simple_client"/>

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

```
    <javac
      srcdir="src"
      destdir="clientclasses"
      includes="examples/webservices/simple_client/*.java"/>
</target>
```

In the example, the first `javac` task compiles the Java files in the `clientclasses` directory that were generated by `clientgen`, and the second `javac` task compiles the Java files in the `examples/webservices/simple_client` subdirectory of the current directory; where it is assumed your Java client application source is located.

In the preceding example, the `clientgen`-generated Java source files and the resulting compiled classes end up in the same directory (`clientclasses`). Although this might be adequate for proto-typing, it is often a best practice to keep source code (even generated code) in a different directory from the compiled classes. To do this, set the `destdir` for both `javac` tasks to a directory different from the `srcdir` directory. You must also copy the following `clientgen`-generated files from `clientgen`'s destination directory to `javac`'s destination directory, keeping the same sub-directory hierarchy in the destination:

packageName/ServiceName`_internaldd.xml`
packageName/ServiceName`_java_wsdl_mapping.xml`
packageName/ServiceName`_saved_wsdl.wsdl`

where *packageName* refers to the subdirectory hierarchy that corresponds to the package of the generated JAX-RPC stubs and *ServiceName* refers to the name of the Web Service.

To run the client application, add a `run` target to the `build.xml` that includes a call 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.webServices.simple_client.Main"
       failonerror="true" >
       <classpath refid="client.class.path"/>
```

```
        <arg
line="http://${wls.hostname}:${wls.port}/complex/ComplexService" />
    </java>

</target>
```

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

See "Sample Ant Build File for a Stand-Alone Java Client" on page 8-11 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`.

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 for a Stand-Alone Java Client

The following example shows a complete `build.xml` file for generating and compiling a stand-alone Java client. See "Using the clientgen Ant Task To Generate Client Artifacts" on page 8-5 and "Compiling and Running the Client Application" on page 8-9 for explanations of the sections in bold.

```
<project name="webservices-simple_client" default="all">

  <!-- set global properties for this build -->

  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />

  <property name="example-output" value="output" />
 <property name="clientclass-dir" value="${example-output}/clientclass" />

  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

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

```
  <target name="clean" >
    <delete dir="${clientclass-dir}"/>
  </target>

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

  <target name="build-client">

    <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.simple_client"/>

    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/simple_client/*.java"/>
  </target>

  <target name="run" >
    <java fork="true"
          classname="examples.webservices.simple_client.Main"
          failonerror="true" >
      <classpath refid="client.class.path"/>
     <arg  line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
      />
    </java>
  </target>

</project>
```

Invoking a Web Service from Another Web Service

Invoking a Web Service from within a WebLogic Web Service is similar to invoking one from a stand-alone Java application, as described in "Invoking a Web Service from a Stand-alone Client: Main Steps" on page 8-4. However, instead of using the `clientgen` Ant task to generate the JAX-RPC stubs of the Web Service to be invoked, you use the `<clientgen>` child element of the `<jws>` element, inside the `jwsc` Ant task that compiles the invoking Web Service. In the JWS file that invokes the other Web Service, however, you still use the same standard JAX-RPC APIs

to get `Service` and `PortType` instances to invoke the Web Service operations. This section describes the differences between invoking a Web Service from a client in a Java EE component and invoking from a stand-alone client.

It is assumed that you have read and understood "Invoking a Web Service from a Stand-alone Client: Main Steps" on page 8-4. It is also assumed 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` that builds a Web Service that you want to update to invoke another Web Service.

The following list describes the changes you must make to the `build.xml` file that builds your client Web Service, which will invoke another Web Service. See "Sample build.xml File for a Web Service Client" on page 8-14 for the full sample `build.xml` file:

- Add a `<clientgen>` child element to the `<jws>` element that specifies the JWS file that implements the Web Service that invokes another Web Service. Set the required `wsdl` attribute to the WSDL of the Web Service to be invoked. Set the required `packageName` attribute to the package into which you want the JAX-RPC client stubs to be generated.

The following bullets describe the changes you must make to the JWS file that implements the client Web Service; see "Sample JWS File That Invokes a Web Service" on page 8-16 for the full JWS file example.

- Import the files generated by the `<clientgen>` child element of the `jwsc` Ant task. These include the JAX-RPC stubs of the invoked Web Service, as well as the Java representation of any user-defined data types used as parameters or return values in the operations of the invoked Web Service.

  Note: The user-defined data types are generated into a package based on the XML Schema of the data type in the WSDL, *not* in the package specified by `clientgen`. The JAX-RPC stubs, however, use the package name specified by the `packageName` attribute of the `<clientgen>` element.

- Update the method that contains the invoke of the Web Service to either throw or catch both `java.rmi.RemoteException` and `javax.xml.rpc.ServiceException`.

- Get the Service and PortType JAX-RPC stubs and invoke the operation on the port as usual; see "Writing the Java Client Application Code to Invoke a Web Service" on page 8-8 for details.

Sample build.xml File for a Web Service Client

The following sample `build.xml` file shows how to create a Web Service that itself invokes another Web Service; the relevant sections that differ from the `build.xml` for building a simple Web Service that does not invoke another Web Service are shown in bold.

The `build-service` target in this case is very similar to a target that builds a simple Web Service; the only difference is that the `jwsc` Ant task that builds the invoking Web Service also includes a `<clientgen>` child element of the `<jws>` element so that `jwsc` also generates the required JAX-RPC client stubs.

```
<project name="webservices-service_to_service" default="all">

  <!-- set global properties for this build -->

  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />

  <property name="ear.deployed.name" value="ClientServiceEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/ClientServiceEar" />
  <property name="clientclass-dir" value="${example-output}/clientclasses" />

  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>

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

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

  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>

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

  <target name="clean" depends="undeploy">
    <delete dir="${example-output}"/>
  </target>

  <target name="build-service">
```

```
    <jwsc
        srcdir="src"
        destdir="${ear-dir}" >

        <jws
         file="examples/webservices/service_to_service/ClientServiceImpl.java">
          <clientgen

wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
                packageName="examples.webservices.service_to_service" />
        </jws>

    </jwsc>

  </target>

  <target name="deploy">
    <wldeploy action="deploy" name="${ear.deployed.name}"
      source="${ear-dir}" user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>

  <target name="undeploy">
    <wldeploy action="undeploy" name="${ear.deployed.name}"
      failonerror="false"
      user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>

  <target name="client">

    <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/ClientService/ClientService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.service_to_service.client"/>

    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/service_to_service/client/**/*.java"/>

  </target>
```

```
  <target name="run">
    <java classname="examples.webservices.service_to_service.client.Main"
          fork="true"
          failonerror="true" >
          <classpath refid="client.class.path"/>
          <arg

line="http://${wls.hostname}:${wls.port}/ClientService/ClientService"/>

    </java>

  </target>

</project>
```

Sample JWS File That Invokes a Web Service

The following sample JWS file, called `ClientServiceImpl.java`, implements a Web Service called `ClientService` that has an operation that in turn invokes the `echoComplexType` operation of a Web Service called `ComplexService`. This operation has a user-defined data type (`BasicStruct`) as both a parameter and a return value. The relevant code is shown in bold and described after the example.

```
package examples.webservices.service_to_service;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;

import javax.jws.WebService;
import javax.jws.WebMethod;

import weblogic.jws.WLHttpTransport;

// Import the BasicStruct data type, generated by clientgen and used
// by the ComplexService Web Service

import examples.webservices.complex.BasicStruct;

// Import the JAX-RPC Stubs for invoking the ComplexService Web Service.
// Stubs generated by clientgen

import examples.webservices.service_to_service.ComplexPortType;
import examples.webservices.service_to_service.ComplexService_Impl;
import examples.webservices.service_to_service.ComplexService;

@WebService(name="ClientPortType", serviceName="ClientService",
            targetNamespace="http://examples.org")
```

```
@WLHttpTransport(contextPath="ClientService", serviceUri="ClientService",
                 portName="ClientServicePort")

public class ClientServiceImpl {

  @WebMethod()
  public String callComplexService(BasicStruct input, String serviceUrl)
      throws ServiceException, RemoteException
  {

    // Create service and port stubs to invoke ComplexService
    ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
    ComplexPortType port = service.getComplexServicePort();

    // Invoke the echoComplexType operation of ComplexService
    BasicStruct result = port.echoComplexType(input);
    System.out.println("Invoked ComplexPortType.echoComplexType." );

   return "Invoke went okay!  Here's the result: '" + result.getIntValue() + ",
" + result.getStringValue() + "'";

  }
}
```

Follow these guidelines when programming the JWS file that invokes another Web Service; code snippets of the guidelines are shown in bold in the preceding example:

- Import any user-defined data types that are used by the invoked Web Service. In this example, the `ComplexService` uses the `BasicStruct` JavaBean:

  ```
  import examples.webservices.complex.BasicStruct;
  ```

- Import the JAX-RPC stubs of the `ComplexService` Web Service; the stubs are generated by the `<cliengen>` child element of `<jws>`:

  ```
  import examples.webservices.service_to_service.ComplexPortType;
  import examples.webservices.service_to_service.ComplexService_Impl;
  import examples.webservices.service_to_service.ComplexService;
  ```

- Ensure that your client Web Service throws or catches `ServiceException` and `RemoteException`:

  ```
  throws ServiceException, RemoteException
  ```

- Create the JAX-RPC Service and Port instances for the `ComplexService`:

  ```
  ComplexService service = new
       ComplexService_Impl(serviceUrl + "?WSDL");
  ComplexPortType port = service.getComplexServicePort();
  ```

- Invoke the `echoComplexType` operation of `ComplexService` using the port you just instantiated:

  ```
  BasicStruct result = port.echoComplexType(input);
  ```

Using a Stand-Alone Client JAR File When Invoking Web Services

It is assumed in this document that, when you invoke a Web Service using the client-side artifacts generated by the `clientgen` or `wsdlc` Ant tasks, you have the entire set of WebLogic Server classes in your CLASSPATH. If, however, your computer does *not* have WebLogic Server installed, you can still invoke a Web Service by using the stand-alone WebLogic Web Services client JAR file, as described in this section.

The standalone client JAR file supports basic client-side functionality, such as:

- Use with client-side artifacts created by both the `clientgen` Ant tasks
- Processing SOAP messages
- Using client-side SOAP message handlers
- Using MTOM
- Invoking both JAX-RPC 1.1 and JAX-WS 2.0 Web Services
- Using SSL

The stand-alone client JAR file does *not*, however, support invoking Web Services that use the following advanced features:

- Web Services reliable SOAP messaging
- Message-level security (WS-Security)
- Conversations
- Asynchronous request-response
- Buffering
- JMS transport

To use the stand-alone WebLogic Web Services client JAR file with your client application, follow these steps:

1. Copy the file *WL_HOME*/server/lib/wseeclient.zip from the computer hosting WebLogic Server to the client computer, where *WL_HOME* refers to the WebLogic Server installation directory, such as /bea/wlserver_10.0.
2. Unzip the wseeclient.zip file into the appropriate directory. For example, you might unzip the file into a directory that contains other classes used by your client application.
3. Add the wseeclient.jar file (unzipped from the wseeclient.zip file) to your CLASSPATH.

   Note: Also be sure that your CLASSPATH includes the JAR file that contains the Ant classes (ant.jar). This JAR file is typically located in the lib directory of the Ant distribution.

Using a Proxy Server When Invoking a Web Service

You can use a proxy server to proxy requests from a client application to an application server (either WebLogic or non-WebLogic) that hosts the invoked Web Service. You typically use a proxy server when the application server is behind a firewall. There are two ways to specify the proxy server in your client application: programmatically using the WebLogic HttpTransportInfo API or using system properties.

For a complete example of using a proxy server when invoking a Web Service, see the example on the dev2dev Code Example site.

Using the HttpTransportInfo API to Specify the Proxy Server

You can programmatically specify within the Java client application itself the details of the proxy server that will proxy the Web Service invoke by using the standard java.net.* classes and the WebLogic-specific HttpTransportInfo API. You use the java.net classes to create a Proxy object that represents the proxy server, and then use the WebLogic API and properties to set the proxy server on the JAX-RPC stub, as shown in the following sample client that invokes the echo operation of the HttpProxySampleService Web Service. The code in bold is described after the example:

```
package dev2dev.proxy.client;

import javax.xml.rpc.Stub;

import java.net.Proxy;
import java.net.InetSocketAddress;

import weblogic.wsee.connection.transport.http.HttpTransportInfo;
```

```
/**
 * Sample client to invoke a service through a proxy server via
 * programmatic API
 */
public class HttpProxySampleClient {
  public static void main(String[] args) throws Throwable{
    assert args.length == 5;
    String endpoint = args[0];
    String proxyHost = args[1];
    String proxyPort = args[2];
    String user = args[3];
    String pass = args[4];

    //create service and port
    HttpProxySampleService service = new HttpProxySampleService_Impl();
    HttpProxySamplePortType port =
service.getHttpProxySamplePortTypeSoapPort();

    //set endpoint address
    ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);

    //set proxy server info
    Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
Integer.parseInt(proxyPort)));
    HttpTransportInfo info = new HttpTransportInfo();
    info.setProxy(p);

((Stub)port)._setProperty("weblogic.wsee.connection.transportinfo",info);

    //set proxy-authentication info

((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",user)
;

((Stub)port)._setProperty("weblogic.webservice.client.proxypassword",pass)
;

    //invoke
    String s = port.echo("Hello World!");
    System.out.println("echo: " + s);
```

```
  }
}
```

The sections of the preceding example to note are as follows:

- Import the required `java.net.*` classes:

  ```
  import java.net.Proxy;
  import java.net.InetSocketAddress;
  ```

- Import the WebLogic `HttpTransportInfo` API:

  ```
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

- Create a `Proxy` object that represents the proxy server:

  ```
  Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
  Integer.parseInt(proxyPort)));
  ```

  The `proxyHost` and `proxyPort` arguments refer to the host computer and port of the proxy server.

- Create an `HttpTransportInfo` object and use the setProxy() method to set the proxy server information:

  ```
  HttpTransportInfo info = new HttpTransportInfo();
  info.setProxy(p);
  ```

- Use the `weblogic.wsee.connection.transportinfo` WebLogic stub property to set the `HttpTransportInfo` object on the JAX-RPC stub:

  ```
  ((Stub)port)._setProperty("weblogic.wsee.connection.transportinfo",info
  );
  ```

- Use `weblogic.webservice.client.proxyusername` and `weblogic.webservice.client.proxypassword` WebLogic-specific stub properties to specify the username and password of a user who is authenticated to access the proxy server:

  ```
  ((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",us
  er);

  ((Stub)port)._setProperty("weblogic.webservice.client.proxypassword",pa
  ss);
  ```

  Alternatively, you can use the `setProxyUsername()` and `setProxyPassword()` methods of the `HttpTransportInfo` API to set the proxy username and password, as shown in the following example:

```
info.setProxyUsername("juliet".getBytes());
info.setProxyPassword("secret".getBytes());
```

Using System Properties to Specify the Proxy Server

When you use system properties to specify the proxy server, you write your client application in the standard way, and then specify the following system properties when you execute the client application:

- `proxySet=true`
- `proxyHost=`*`proxyHost`*
- `proxyPort=`*`proxyPort`*
- `weblogic.webservice.client.proxyusername=`*`proxyUsername`*
- `weblogic.webservice.client.proxypassword=`*`proxyPassword`*

where *`proxyHost`* is the name of the host computer on which the proxy server is running, *`proxyPort`* is the port to which the proxy server is listening, *`proxyUsername`* is the authenticated proxy server user and *`proxyPassword`* is the user's password.

The following excerpt from an Ant build script shows an example of setting these system properties when invoking a client application called `clients.InvokeMyService`:

```
<target name="run-client">
  <java fork="true"
        classname="clients.InvokeMyService"
        failonerror="true">
    <classpath refid="client.class.path"/>
    <arg line="${http-endpoint}"/>
    <jvmarg line=
     "-DproxySet=true
      -DproxyHost=${proxy-host}
      -DproxyPort=${proxy-port}
      -Dweblogic.webservice.client.proxyusername=${proxy-username}
      -Dweblogic.webservice.client.proxypassword=${proxy-passwd}"
    />
  </java>
</target>
```

Client Considerations When Redeploying a Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated WebLogic Web Service alongside an older version of the same Web Service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web Service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web Service. If the client is connected to a conversational or reliable Web Service, its work is considered complete when the existing conversation or reliable messaging sequence is explicitly ended by the client or because of a timeout.

You can continue using the old client application with the new version of the Web Service, as long as the following Web Service artifacts have not changed in the new version:

- the WSDL that describes the Web Service
- the WS-Policy files attached to the Web Service

If any of these artifacts have changed, you must regenerate the JAX-RPC stubs used by the client application by re-running the `clientgen` Ant task.

For example, if you change the signature of an operation in the new version of the Web Service, then the WSDL file that describes the new version of the Web Service will also change. In this case, you must regenerate the JAX-RPC stubs. If, however, you simply change the implementation of an operation, but do not change its public contract, then you can continue using the existing client application.

WebLogic Web Services Stub Properties

WebLogic Server provides a set of stub properties that you can set in the JAX-RPC Stub used to invoke a WebLogic Web Service. Use the `Stub._setProperty()` method to set the properties, as shown in the following example:

```
((Stub)port)._setProperty(WLStub.MARSHAL_FORCE_INCLUDE_XSI_TYPE,"true");
```

Most of the stub properties are defined in the `WLStub` class. See `weblogic.wsee.jaxrpc.WLStub` for details.

The following table describes additional stub properties not defined in the `WLStub` class.

Table 8-2 Additional Stub Properties

Stub Property	Description
weblogic.wsee.transport.connection.timeout	Specifies, in milliseconds, how long a client application that is attempting to invoke a Web Service waits to make a connection. After the specified time elapses, if a connection hasn't been made, the attempt times out.
weblogic.wsee.transport.read.timeout	Specifies, in milliseconds, how long a client application waits for a response from a Web Service it is invoking. After the specified time elapses, if a response hasn't arrived, the client times out.
weblogic.wsee.security.bst.serverVerifyCert	Specifies the certificate that the client application uses to validate the signed response from WebLogic Server. By default, WebLogic Server includes the certification used to validate in the response SOAP message itself; if this is not possible, then use this stub property to specify a different one. This stub property applies *only* to client applications that run inside of a WebLogic Server container, and not to stand-alone client applications. The value of the property is an object of data type `java.security.cert.X509Certificate`.

Table 8-2 Additional Stub Properties

Stub Property	Description
weblogic.wsee.security.bst.serverEncryptCert	Specifies the certificate that the client application uses to encrypt the request SOAP message sent to WebLogic Server. By default, the client application uses the public certificate published in the Web Service's WSDL; if this is not possible, then use this stub property to specify a different one. This stub property applies *only* to client applications that run inside of a WebLogic Server container, and not to stand-alone client applications. The value of the property is an object of data type `java.security.cert.X509Certificate`.

weblogic.wsee.marshal.forceIncludeXsiType	Specifies that the SOAP messages for a Web Service operation invoke should include the XML Schema data type of each parameter. By default, the SOAP messages do not include the data type of each parameter.
	If you set this property to `True`, the elements in the SOAP messages that describe operation parameters will include an `xsi:type` attribute to specify the data type of the parameter, as shown in the following example:
	`<soapenv:Envelope>`
	`...`
	`<maxResults xsi:type="xs:int">10</maxResults>`
	`...`
	By default (or if you set this property to `False`), the parameter element would look like the following example:
	`<soapenv:Envelope>`
	`...`
	`<maxResults>10</maxResults>`
	`...`
	Valid values for this property are `True` and `False`; default value is `False`.

Setting the Character Encoding For the Response SOAP Message

Use the `weblogic.wsee.jaxrpc.WLStub.CHARACTER_SET_ENCODING` WLStub property to set the character encoding of the response (outbound) SOAP message. You can set it to the following two values:

- UTF-8
- UTF-16

The following code snippet from a client application shows how to set the character encoding to UTF-16:

```
  Simple port = service.getSimpleSoapPort();
  ((Stub)
port)._setProperty(weblogic.wsee.jaxrpc.WLStub.CHARACTER_SET_ENCODING,
"UTF-16");
  port.invokeMethod();
```

See `weblogic.wsee.jaxrpc.WLStub` for additional WLStub properties you can set.

CHAPTER 9

Administering Web Services

The following sections describe how to administer WebLogic Web Services:

Overview of WebLogic Web Services Administration Tasks

When you use the `jwsc` Ant task to compile and package a WebLogic Web Service, the task packages it as part of an Enterprise Application. The Web Service itself is packaged inside the Enterprise application as a Web application WAR file, by default. However, if your JWS file explicitly implemented `javax.ejb.SessionBean`, then the Web Service is packaged as an EJB JAR file. Therefore, basic administration of Web Services is very similar to basic administration

of standard Java Platform, Enterprise Edition (Java EE) Version 5 applications and modules. These standard tasks include:

- Installing the Enterprise application that contains the Web Service.
- Starting and stopping the deployed Enterprise application.
- Configuring the Enterprise application and the archive file which implements the actual Web Service. You can configure general characteristics of the Enterprise application, such as the deployment order, or module-specific characteristics, such as session time-out for Web applications or transaction type for EJBs.
- Creating and updating the Enterprise application's deployment plan.
- Monitoring the Enterprise application.
- Testing the Enterprise application.

The following administrative tasks are specific to Web Services:

- Configuring the JMS resources used by Web Service reliable messaging and JMS transport
- Configuring the WS-Policy files associated with a Web Service endpoint or its operations.

  WARNING: If you used the `@Policy` annotation in your Web Service to specify an associated WS-Policy file at the time you programmed the JWS file, you cannot change this association at run-time using the Administration Console or other administrative tools. You can only associate a *new* WS-Policy file, or disassociate one you added at run-time.
- Viewing the SOAP handlers associated with the Web Service.
- Viewing the WSDL of the Web Service.
- Creating a Web Service security configuration.

Administration Tools

There are a variety of ways to administer Java EE 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 Java EE 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:

- Install an Enterprise application.
- Start and stop a deployed Enterprise application.
- Configure an Enterprise application.
- Configure Web applications.
- Configure EJBs.
- Create a deployment plan.
- Update a deployment plan.
- Test the modules in an Enterprise application.
- Configure JMS resources for Web Service reliable messaging.
- Associate the WS-Policy file with a Web Service.
- View the SOAP message handlers of a Web Service.
- View the WSDL of a Web Service.

- Create a Web Service security configuration

Invoking the Administration Console

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

`http://host:port/console`

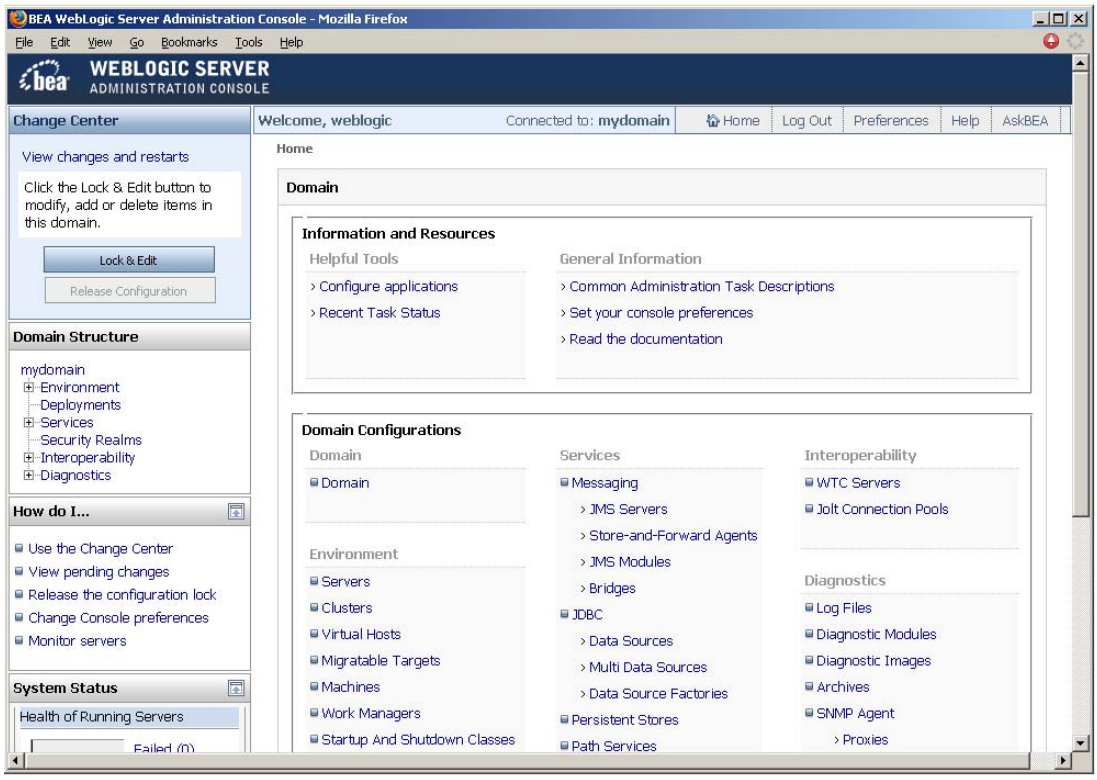
where

- *host* refers to the computer on which 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 9-1 WebLogic Server Administration Console Main Window



How Web Services Are Displayed In the Administration Console

Web Services are typically deployed to WebLogic Server as part of an Enterprise Application. The Enterprise Application can be either archived as an EAR, or be in exploded directory format. The Web Service itself is almost always packaged as a Web Application; the only exception is if your JWS file explicitly implements `javax.ejb.SessionBean`, in which case it is packaged as an EJB. The Web Service can be in archived format (WAR or EJB JAR file, respectively) or as an exploded directory.

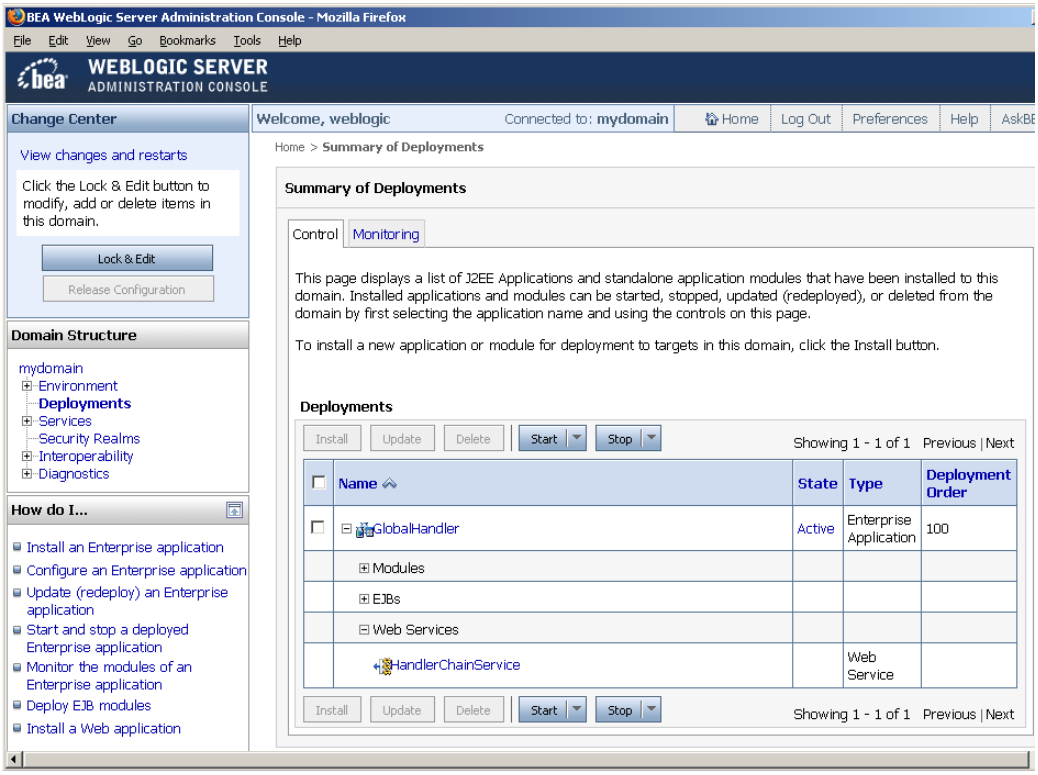
It is not required that a Web Service be installed as part of an Enterprise application; it can be installed as just the Web Application or EJB. However, BEA recommends that users install the

Web Service as part of an Enterprise application. The WebLogic Ant task used to create a Web Service, `jwsc`, always packages the generated Web Service into an Enterprise application.

To view and update the Web Service-specific configuration information about a Web Service using the Administration Console, click on the Deployments node in the left pane and, in the Deployments table that appears in the right pane, find the Enterprise application in which the Web Service is packaged. Expand the application by clicking the + node; the Web Services in the application are listed under the **Web Services** category. Click on the name of the Web Service to view or update its configuration.

The following figure shows how the `HandlerChainService` Web Service, packaged inside the `GlobalHandler` Enterprise application, is displayed in the **Deployments** table of the Administration Console.

Figure 9-2 Web Service Displayed in Deployments Table of Administration Console



Creating a Web Services Security Configuration

When a deployed WebLogic Web Service has been configured to use message-level security (encryption and digital signatures, as described by the WS-Security specification), the Web Services runtime determines whether a Web Service security configuration is also associated with the service. This security configuration specifies information such as whether to use an X.509 certificate for identity, whether to use password digests, the keystore to be used for encryption, and so on. A single security configuration can be associated with many Web Services.

Because Web Services security configurations are domain-wide, you create them from the ***domainName*** **> WebService Security** tab of the Administration Console, rather than the **Deployments** tab. The following figure shows the location of this tab.

Figure 9-3 Web Service Security Configuration in 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 Java EE 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.

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:

- 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)

A managed bean (MBean) is a Java bean that provides a Java Management Extensions (JMX) interface. JMX is the Java EE solution for monitoring and managing resources on a network. Like SNMP and other management standards, JMX is a public specification and many vendors of commonly used monitoring products support it.

BEA WebLogic Server provides a set of MBeans that you can use to configure, monitor, and manage WebLogic Server resources through JMX. WebLogic Web Services also have their own set of MBeans that you can use to perform some Web Service administrative tasks.

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:

- `WebserviceSecurityConfigurationMBean`

- WebserviceCredentialProviderMBean
- WebserviceSecurityMBean
- WebserviceSecurityTokenMBean
- WebserviceTimestampMBean
- WebserviceTokenHandlerMBean

The runtime Web Services MBeans are:

- WseeRuntimeMBean
- WseeHandlerRuntimeMBean
- WseePortRuntimeMBean
- WseeOperationRuntimeMBean
- WseePolicyRuntimeMBean

For more information on JMX, see:

- Understanding WebLogic Server MBeans
- Accessing WebLogic Server MBeans with JMX
- Managing a Domain's Configuration with JMX
- WebLogic Server MBean Reference.

Using the Java EE Deployment API

In Java EE 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 Java EE Deployment architecture, which in turn defines the contracts that enable tools or application programmers to configure and deploy applications on any Java EE platform product. The contracts define a uniform model between tools and Java EE 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 Java EE deployment tools in order to deploy an application on many different Java EE platform products.

See Deploying Applications to WebLogic Server for more information.

Using Work Managers to Prioritize Web Services Work and Reduce Stuck Execute Threads

After a connection has been established between a client application and a Web Service, the interactions between the two are ideally smooth and quick, whereby the client makes requests and the service responds in a prompt and timely manner. Sometimes, however, a client application might take a long time to make a new request, during which the Web Service waits to respond, possibly for the life of the WebLogic Server instance; this is often referred to as a *stuck execute thread*. If, at any given moment, WebLogic Server has a lot of stuck execute threads, the overall performance of the server might degrade.

If a particular Web Service gets into this state fairly often, you can specify how the service prioritizes the execution of its work by configuring a Work Manager and applying it to the service. For example, you can configure a *response time request class* (a specific type of Work Manager component) that specifies a response time goal for the Web Service.

The following shows an example of how to define a response time request class in a deployment descriptor:

```
<work-manager>
    <name>responsetime_workmanager</name>
        <response-time-request-class>
            <name>my_response_time</name>
            <goal-ms>2000</goal-ms>
        </response-time-request-class>
</work-manager>
```

You can configure the response time request class using the Administration Console, as described in "Work Manager: Response Time: Configuration" in the *Administration Console Online Help*.

For more information about Work Managers in general and how to configure them for your Web Service, see "Using Work Managers to Optimize Scheduled Work" in *Configuring WebLogic Server Environments*.

CHAPTER 10

Upgrading WebLogic Web Services From Previous Releases to 10.0

The following sections describe how to upgrade a pre-10.0 WebLogic Server Web Service to run in the 10.0 Web Service runtime environment:

Upgrading a 9.2 WebLogic Web Service to 10.0

You do not need to do anything to upgrade a 9.2 WebLogic Web Service to 10.0; you can redeploy it to WebLogic Server 10.0 without making any changes or recompiling it.

Upgrading a 9.0 or 9.1 WebLogic Web Service to 10.0

If your 9.0/9.1 Web Service used any of the following features, then you must recompile the Web Service before you redeploy it to WebLogic Server 10.0:

- Conversations
- `@weblogic.jws.Context` JWS annotation
- `weblogic.wsee.jws.JwsContext` API

To recompile, simply rerun the `jwsc` Ant task against the JWS file that implements your Web Service.

If your 9.0/9.1 Web Service did not use these features, then you can redeploy it to WebLogic Server 10.0 without making any changes or recompiling it.

Upgrading an 8.1 WebLogic Web Service to 10.0

This section describes how to upgrade an 8.1 WebLogic Web Service to use the new Version 10.0 Web Services runtime environment. The 10.0 runtime is based on the *Implementing Enterprise Web Services* 1.2 specification (JSR-109). The 10.0 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 10.0 of WebLogic Server because the 8.1 Web Services runtime is still supported in 10.0, 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 10.0.

Upgrading your 8.1 Web Service includes the following high-level tasks; the procedures in later sections go into more detail:

- Update the 8.1 Java source code of the Java class or stateless session EJB that implements the Web Service so that the source code uses JWS annotations.

  Version 10.0 WebLogic Web Services are implemented using JWS files, which are Java files that contains JWS annotations. The `jwsc` Ant task always implements the Web Service as a plain Java file unless you explicitly implement `javax.ejb.SessionBean` in your JWS file. This latter case is not typical. This programming model differs from that of 8.1, where you were required to specify the type of backend component (Java class or EJB).

- Update the Ant build script that builds the Web Service to call the 10.0 WebLogic Web Service Ant task `jwsc` instead of the 8.1 `servicegen` task.

In the sections that follow it is assumed 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 Services and other Java Platform, Enterprise Edition (Java EE) Version 5 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, because the 10.0 WebLogic Web Services runtime does not support JMS-implemented services.
- 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 10.0 Web Service. For details on how to write a client application that invokes a 10.0 Web Service, see Chapter 8, "Invoking Web Services."

Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 10.0: Main Steps

To upgrade an 8.1 Java class-implemented Web Service to use the 10.0 WebLogic Web Services runtime:

1. Open a command window and set your WebLogic Server 10.0 environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your 10.0 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 project directory:

   ```
   prompt> mkdir /myExamples/upgrade_pojo
   ```

3. Create an `src` directory under the project directory, as well as sub-directories that correspond to the package name of the new 10.0 JWS file (shown later in this procedure) that corresponds to the old 8.1 Java class:

   ```
   prompt> cd /myExamples/upgrade_pojo
   prompt> mkdir src/examples/webservices/upgrade_pojo
   ```

4. Copy the old Java class that implements the 8.1 Web Service to the `src/examples/webservices/upgrade_pojo` directory of the working directory. Rename the file, if desired.

5. Edit the Java file, as described in the following steps. See the old and new sample Java files in "Example of an 8.1 Java File and the Corresponding 10.0 JWS File" on page 10-5 for specific examples.

   a. If needed, change the package name and class name of the Java file to reflect the new 10.0 source environment.

b. Add `import` statements to import both the standard and WebLogic-specific JWS annotations.

c. Add, at a minimum, the following JWS annotation:

– The standard `@WebService` annotation at the Java class level to specify that the JWS file implements a Web Service.

BEA recommends you also add the following annotations:

– The standard `@SOAPBinding` annotation at the class-level to specify the type of Web Service, such as document-literal-wrapped or RPC-encoded.

– The WebLogic-specific `@WLHttpTransport` annotation at the class-level to specify the context and service URIs that are used in the URL that invokes the deployed Web Service.

– The standard `@WebMethod` annotation at the method-level 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 10.0. In 8.1, many of these features were configured with attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes" on page 10-20 for a table that lists equivalent JWS annotation, if available, for features you enabled in 8.1 using `servicegen` attributes.

6. Copy the old `build.xml` file that built the 8.1 Web Service to the 10.0 working directory.

7. 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 the `jwsc` Ant task call to compile the new 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, "Iterative Development of WebLogic Web Services."

See "Example of an 8.1 and Updated 10.0 Ant Build File for Java Class-Implemented Web Services" on page 10-7 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 the tasks needed to build the 10.0 Web Service, as described in the following steps.

c. Add the `jwsc` task to the build file. Set the `srdir` attribute to the src directory (`/myExamples/upgrade_pojo/src`, in this example) and the `destdir` attribute to the root Enterprise application directory you created in the preceding step.

   Set the `file` attribute of the `<jws>` child element to the name of the new JWS file, created earlier in this procedure.

   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 10.0. In 8.1, many of these features were configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes" on page 10-20 for a table that describes if there is an equivalent `jwsc` attribute for features you enabled using `servicegen` attributes.

8. Execute the `build-service` Ant target. Assuming all the tasks complete successfully, the resulting Enterprise application contains your upgraded 10.0 Web Service.

See "Deploying and Undeploying WebLogic Web Services" on page 4-15 and "Browsing to the WSDL of the Web Service" on page 4-18 for additional information about deploying and testing your Web Service.

Based on the sample Java code shown in the following sections, the URL to invoke the WSDL of the upgraded 10.0 Web Service is:

`http://host:port/upgradePOJO/HelloWorld?WSDL`

Example of an 8.1 Java File and the Corresponding 10.0 JWS 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) 2005 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 10.0 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 10.0 Ant Build File for Java Class-Implemented Web Services" on page 10-7.

WARNING: Because the following JWS file uses WebLogic-specific JWS annotations, the generated Web Service will be based on JAX-RPC 1.1 rather than JAX-WS 2.0.

```
package examples.webservices.upgrade_pojo;

// Import standard JWS annotations

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

// Import WebLogic JWS anntoation

import weblogic.jws.WLHttpTransport;

/**
 * Simple Java class that implements the HelloWorld92 Web service.   It takes
 * as input an integer and a String, and returns a message that includes these
 * two parameters.
 *
 * @author Copyright (c) 2005 by BEA Systems. All Rights Reserved.
 */

@WebService(name="HelloWorld92PortType", serviceName="HelloWorld",
            targetNamespace="http://example.org")
```

```
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

@WLHttpTransport(contextPath="upgradePOJO", serviceUri="HelloWorld",
                 portName="HelloWorld92Port")

public class HelloWorld92Impl {

  /**
   * 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 10.0 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; in the example, the Java file that implements the 8.1 Web Service has already been compiled into the `examples.javaclass.HelloWorld81` class:

```
<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 10.0 Web Service is shown below, with the relevant tasks discussed in this section in bold. In the example, the new JWS file that implements the 10.0 Web Service is called `HelloWorld92Impl.java`:

```
<project name="webservices-upgrade_pojo" default="all">

  <!-- set global properties for this build -->

  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />

  <property name="ear.deployed.name" value="upgradePOJOEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/upgradePOJOEar" />
```

```
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all" depends="clean,build-service,deploy" />
  <target name="clean" depends="undeploy">
    <delete dir="${example-output}"/>
  </target>
  <target name="build-service">
    <jwsc
      srcdir="src"
      destdir="${ear-dir}">
     <jws file="examples/webservices/upgrade_pojo/HelloWorld92Impl.java" />
    </jwsc>
  </target>
  <target name="deploy">
    <wldeploy action="deploy" name="${ear.deployed.name}"
      source="${ear-dir}" user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>
  <target name="undeploy">
    <wldeploy action="undeploy" name="${ear.deployed.name}"
      failonerror="false"
      user="${wls.username}" password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>
</project>
```

Upgrading an 8.1 EJB-Implemented WebLogic Web Service to 10.0: Main Steps

The following procedure describes how to upgrade an 8.1 EJB-implemented Web Service to use the 10.0 WebLogic Web Services runtime.

The 10.0 Web Services programming model is quite different from the 8.1 model in that it hides the underlying implementation of the Web Service. Rather than specifying up front that you want the Web Service to be implemented by a Java class or an EJB, the `jwsc` Ant task always picks a plain Java class implementation, unless you have explicitly implemented `javax.ejb.SessionBean` in the JWS file, which is not typical. For this reason, the following procedure does not show how to import EJB classes or use EJBGen, even though the 8.1 Web Service was explicitly implemented with an EJB. Instead, the procedure shows how to create a standard JWS file that is the 10.0 equivalent to the 8.1 EJB-implemented Web Service.

1. Open a command window and set your 10.0 WebLogic Server environment by executing the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your 10.0 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 project directory:

   ```
   prompt> mkdir /myExamples/upgrade_ejb
   ```

3. Create a `src` directory under the project directory, as well as sub-directories that correspond to the package name of the new 10.0 JWS file (shown later on in this procedure) that corresponds to your 8.1 EJB implementation:

   ```
   prompt> cd /myExamples/upgrade_ejb
   prompt> mkdir src/examples/webservices/upgrade_ejb
   ```

4. Copy the 8.1 EJB Bean file that implemented `javax.ejb.SessionBean` to the `src/examples/webservices/upgrade_ejb` directory of the working directory. Rename the file, if desired.

   Note: You do not need to copy over the 8.1 Home and Remote EJB files.

5. Edit the EJB Bean file, as described in the following steps. See the old and new sample Java files in "Example of 8.1 EJB Files and the Corresponding 10.0 JWS File" on page 10-13 for specific examples.

a. If needed, change the package name and class name of the Java file to reflect the new 10.0 source environment.

b. Optionally remove the `import` statements that import the EJB classes (`javax.ejb.*`). These classes are no longer needed in the upgraded JWS file.

c. Add `import` statements to import both the standard and WebLogic-specific JWS annotations.

d. Ensure that the JWS file does *not* implement `javax.ejb.SessionBean` anymore by removing the `implements SessionBean` code from the class declaration.

e. Remove all the EJB-specific methods:

- `ejbActivate()`
- `ejbRemove()`
- `ejbPassivate()`
- `ejbCreate()`

f. Add, at a minimum, the following JWS annotation:

- The standard `@WebService` annotation at the Java class level to specify that the JWS file implements a Web Service.

  BEA recommends you also add the following annotations:

- The standard `@SOAPBinding` annotation at the class-level to specify the type of Web Service, such as document-literal-wrapped or RPC-encoded.
- The WebLogic-specific `@WLHttpTransport` annotation at the class-level to specify the context and service URIs that are used in the URL that invokes the deployed Web Service.
- The standard `@WebMethod` annotation at the method-level 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.

g. 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 10.0. In 8.1, many of these features were configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes" on page 10-20 for a table that lists equivalent JWS annotation, if available, for features you enabled in 8.1 using `servicegen` attributes.

6. Copy the old `build.xml` file that built the 8.1 Web Service to the 10.0 working directory.

7. 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 the `jwsc` Ant task call to compile the new 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, "Iterative Development of WebLogic Web Services."

   See "Example of an 8.1 and Updated 10.0 Ant Build File for an 8.1 EJB-Implemented Web Service" on page 10-17 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 the tasks needed to build the 10.0 Web Service, as described in the following steps.

   c. Add the `jwsc` task to the build file. Set the `srdir` attribute to the src directory (`/myExamples/upgrade_ejb/src`, in this example) and the `destdir` attribute to the root Enterprise application directory you created in the preceding step.

      Set the `file` attribute of the `<jws>` child element to the name of the new JWS file, created earlier in this procedure.

      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 10.0. In 8.1, many of these features were configured using attributes of `servicegen`. See "Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes" on page 10-20 for a table that indicates whether there is an equivalent `jwsc` attribute for features you enabled using `servicegen` attributes.

8. Execute the `build-service` Ant target. Assuming all tasks complete successfully, the resulting Enterprise application contains your upgraded 10.0 Web Service.

See "Deploying and Undeploying WebLogic Web Services" on page 4-15 and "Browsing to the WSDL of the Web Service" on page 4-18 for additional information about deploying and testing your Web Service.

Based on the sample Java code shown in the following sections, the URL to invoke the WSDL of the upgraded 10.0 Web Service is:

`http://host:port/upgradeEJB/HelloWorldService?WSDL`

Example of 8.1 EJB Files and the Corresponding 10.0 JWS File

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 10.0 JWS file is shown in "Equivalent 10.0 JWS File" on page 10-16. The differences between the 8.1 and 10.0 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 10.0 Ant Build File for an 8.1 EJB-Implemented Web Service" on page 10-17.

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) 2005 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) 2005 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) 2005 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 10.0 JWS File

The differences between the 8.1 and 10.0 files are shown in bold. The value of some JWS annotations are taken from attributes of the 8.1 `servicegen` Ant task shown in "Example of an 8.1 and Updated 10.0 Ant Build File for an 8.1 EJB-Implemented Web Service" on page 10-17.

WARNING: Because the following JWS file uses WebLogic-specific JWS annotations, the generated Web Service will be based on JAX-RPC 1.1 rather than JAX-WS 2.0.

```
package examples.webservices.upgrade_ejb;

// Import the standard JWS annotations

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

// Import the WebLogic specific annotation

import weblogic.jws.WLHttpTransport;

// Class-level annotations

@WebService(name="HelloWorld92PortType", serviceName="HelloWorldService",
            targetNamespace="http://example.org")

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)

@WLHttpTransport(contextPath="upgradeEJB", serviceUri="HelloWorldService",
                 portName="HelloWorld92Port")

/**
 * HelloWorld92Impl is the JWS equivalent of the HelloWorld81 EJB that
 * implemented the 8.1 Web Service.  It has a single method,
 * sayHello(), that takes an integer and a String and returns a String.
 * <p>
```

```
 * @author Copyright (c) 2005 by BEA Systems. All Rights Reserved.
 */

public class HelloWorld92Impl {

  /** 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 HelloWorld92 Web Service 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 10.0 Ant Build File for an 8.1 EJB-Implemented Web Service

The following simple `build.xml` file shows the 8.1 way to build an EJB-implemented WebLogic Web Service using the `servicegen` Ant task. Following this example is an equivalent `build.xml` file that calls the `jwsc` Ant task to build a 10.0 Web Service.

```
<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="HelloWorldService"
       serviceURI="/HelloWorldService"
       generateTypes="True"
       expandMethods="True">
     </service>
   </servicegen>
 </target>

</project>
```

An equivalent `build.xml` file that calls the `jwsc` Ant task to build a 10.0 Web Service is shown below, with the relevant tasks discussed in this section in bold:

```
<project name="webservices-upgrade_ejb" default="all">

  <!-- set global properties for this build -->
```

```
<property name="wls.username" value="weblogic" />
<property name="wls.password" value="weblogic" />
<property name="wls.hostname" value="localhost" />
<property name="wls.port" value="7001" />
<property name="wls.server.name" value="myserver" />

<property name="ear.deployed.name" value="upgradeEJB" />
<property name="example-output" value="output" />
<property name="ear-dir" value="${example-output}/upgradeEJB" />

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

<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy"/>

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

<target name="clean" depends="undeploy">
  <delete dir="${example-output}"/>
</target>

<target name="build-service">

  <jwsc
    srcdir="src"
    destdir="${ear-dir}">

   <jws file="examples/webservices/upgrade_ejb/HelloWorld92Impl.java" />

  </jwsc>

</target>

<target name="deploy">
  <wldeploy action="deploy" name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>

<target name="undeploy">
  <wldeploy action="undeploy" name="${ear.deployed.name}"
    failonerror="false"
```

```
      user="${wls.username}" password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>
</project>
```

Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

The following table maps the attributes of the 8.1 `servicegen` Ant task to their equivalent 10.0 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>`)

See JWS Annotation Reference, and jwsc for more information about the 10.0 JWS annotations and `jwsc` Ant task.

.

Table 10-1 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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service.
`destEAR`	`destdir` attribute of the `jwsc` Ant task.
`keepGenerated`	`keepGenerated` attribute of the `jwsc` Ant task.
`mergeWithExistingWS`	No equivalent.
`overwrite`	No equivalent.
`warName`	`name` attribute of the `<jws>` child element of the `jwsc` Ant task.

Table 10-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`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 `file` attribute of the `<jws>` child element of the `jwsc` Ant task that specifies the name of the JWS file.
`excludeEJBs` (attribute of the `service` child element)	No equivalent.
`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 `file` attribute of the `<jws>` child element of the `jwsc` Ant task that specifies the name of the JWS file.
`JMSAction` (attribute of the `service` child element)	No equivalent because JMS-implemented Web Services are not supported in the 10.0 release.

Table 10-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`JMSConnectionFactory` (attribute of the `service` child element)	No equivalent because JMS-implemented Web Services are not supported in the 10.0 release.
`JMSDestination` (attribute of the `service` child element)	No equivalent because JMS-implemented Web Services are not supported in the 10.0 release.
`JMSDestinationType` (attribute of the `service` child element)	No equivalent because JMS-implemented Web Services are not supported in the 10.0 release.
`JMSMessageType` (attribute of the `service` child element)	No equivalent because JMS-implemented Web Services are not supported in the 10.0 release.
`JMSOperationName` (attribute of the `service` child element)	No equivalent because JMS-implemented Web Services are not supported in the 10.0 release.
`protocol` (attribute of the `service` child element)	One of the following WebLogic-specific annotations: • `@WLHttpTransport` • `@WLJmsTransport` **Note:** Because these are WebLogic-specific annotations, you can use them to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service.
`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` or `@WLJmsTransport` annotations. **Note:** Because these are WebLogic-specific annotations, you can use them to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service.

Table 10-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`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)	`value` attribute of the WebLogic-specific `@weblogic.jws.Binding` JWS annotation **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service.
`clientJarName` (attribute of `client` child element)	No equivalent.
`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.

Table 10-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`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 Web Service reliable messaging policy assertions. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Using Web Service Reliable Messaging.
`persistDuration` (attribute of the `reliability` child element)	No direct equivalent. Use WebLogic-specific `@Policy` attribute to specify a WS-Policy file that contains Web Service reliable messaging policy assertions. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Using Web Service Reliable Messaging.
`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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Configuring Message-Level Security (Digital Signatures and Encryption).

Table 10-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

servicegen or Child Element of servicegen Attribute	Equivalent JWS Annotation or jwsc Attribute
`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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Configuring Message-Level Security (Digital Signatures and Encryption).
`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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Configuring Message-Level Security (Digital Signatures and Encryption).
`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).

Table 10-1 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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Configuring Message-Level Security (Digital Signatures and Encryption).
`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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Configuring Message-Level Security (Digital Signatures and Encryption).
`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. **Note:** Because this is a WebLogic-specific annotation, you can use it to generate *only* a JAX-RPC 1.1-based Web Service, and not a JAX-WS 2.0 Web Service. See Configuring Message-Level Security (Digital Signatures and Encryption).