Contents

1. Introduction and Roadmap
   Document Scope and Audience .................................................. 1-1
   Guide to This Document .......................................................... 1-2
   Related Documentation ............................................................. 1-3
   Samples for the Web Services Developer ..................................... 1-4
      Downloading Examples Described in this Guide .............................. 1-4
      Avitek Medical Records Application (MedRec) and Tutorials ............ 1-5
      Web Services Examples in the WebLogic Server Distribution ............ 1-5
      Additional Web Services Examples Available for Download .............. 1-5
   Release-Specific WebLogic Web Services Information ....................... 1-5
   Differences Between 8.1 and 9.X WebLogic Web Services ................... 1-6
   Summary of WebLogic Web Services Features ................................ 1-6

2. Understanding WebLogic Web Services
   What Are Web Services? ............................................................ 2-1
   Why Use Web Services? ............................................................ 2-2
   Anatomy of a WebLogic Web Service ........................................... 2-3
   Roadmap of Common Web Service Development Tasks ....................... 2-4
   Standards Supported by WebLogic Web Services .............................. 2-6
      BEA Implementation of Web Service Specifications ......................... 2-8
      Web Services Metadata for the Java Platform (JSR-181) .................. 2-8
      Enterprise Web Services 1.1 .................................................. 2-9
3. Common Web Services Use Cases and Examples
   Creating a Simple HelloWorld Web Service ............................................. 3-2
   Creating a Web Service With User-Defined Data Types .............................. 3-7
   Creating a Web Service from a WSDL File ............................................. 3-14
   Invoking a Web Service from a Stand-alone JAX-RPC Java Client ..................... 3-23
   Invoking a Web Service from a WebLogic Web Service .............................. 3-29

4. Iterative Development of WebLogic Web Services
   Overview of the WebLogic Web Service Programming Model .......................... 4-2
   Configuring Your Domain For Web Services Features ................................... 4-2
   Iterative Development of WebLogic Web Services Starting From Java: Main Steps .... 4-3
   Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps 4-5
   Creating the Basic Ant build.xml File .................................................... 4-7
5. Programming the JWS File

Overview of JWS Files and JWS Annotations .................................................. 5-1
Programming the JWS File: Java Requirements ................................................. 5-2
Programming the JWS File: Typical Steps ......................................................... 5-3
  Example of a JWS File ..................................................................................... 5-4
  Specifying That the JWS File Implements a Web Service ............................. 5-6
  Specifying the Mapping of the Web Service to the SOAP Message Protocol ... 5-6
  Specifying the Context Path and Service URI of the Web Service ................ 5-7
  Specifying That a JWS Method Be Exposed as a Public Operation ............... 5-8
  Customizing the Mapping Between Operation Parameters and WSDL Parts .... 5-9
  Customizing the Mapping Between the Operation Return Value and a WSDL Part 5-10
Accessing Runtime Information about a Web Service Using the JwsContext ...... 5-11
  Guidelines for Accessing the Web Service Context ..................................... 5-11
  Methods of the JwsContext ........................................................................... 5-13
Should You Implement a Stateless Session EJB? ................................. 5-16
Programming Guidelines When Implementing an EJB in Your JWS File. .... 5-17
Example of a JWS File That Implements an EJB ................................. 5-18
Programming the User-Defined Java Data Type ............................... 5-19
Throwing Exceptions ............................................................... 5-21
Invoking Another Web Service from the JWS File ............................ 5-24
Programming Additional Miscellaneous Features Using JWS Annotations and APIs . 5-24
  Streaming SOAP Attachments ................................................. 5-24
  Using SOAP 1.2 ................................................................ 5-25
  Specifying that Operations Run Inside of a Transaction ................. 5-26
  Getting the HttpServletRequest/Response Object ................... 5-27
JWS Programming Best Practices ................................................. 5-29

6. Advanced JWS Programming: Implementing Asynchronous Features

Using Web Service Reliable Messaging ......................................... 6-1
  Use of WS-Policy Files for Web Service Reliable Messaging Configuration . 6-2
  Using Web Service Reliable Messaging: Main Steps ..................... 6-4
Configuring the Destination WebLogic Server Instance ..................... 6-6
Configuring the Source WebLogic Server Instance .......................... 6-8
Creating the Web Service Reliable Messaging WS-Policy File ............. 6-8
Programming Guidelines for the Reliable JWS File ......................... 6-10
Programming Guidelines for the JWS File That Invokes a Reliable Web Service . 6-15
WsrmUtils Utility Class .......................................................... 6-17
Updating the build.xml File for a Client of a Reliable Web Service .... 6-18
Client Considerations When Redeploying a Reliable Web Service ....... 6-19
Invoking a Web Service Using Asynchronous Request-Response ........ 6-19
  Using Asynchronous Request-Response: Main Steps .................... 6-20
Writing the Asynchronous JWS File ........................................... 6-21
Updating the build.xml File When Using Asynchronous Request-Response .... 6-26
Disabling The Internal Asynchronous Service .................................. 6-27
Using Callbacks to Notify Clients of Events .................................. 6-27
Callback Implementation Overview and Terminology ......................... 6-28
Programming Callbacks: Main Steps ......................................... 6-29
Programming Guidelines for Target Web Service .......................... 6-31
Programming Guidelines for the Callback Client Web Service ............. 6-32
Programming Guidelines for the Callback Interface ........................ 6-35
Updating the build.xml File for the Client Web Service ...................... 6-36
Creating Conversational Web Services ....................................... 6-37
Creating a Conversational Web Service: Main Steps ......................... 6-39
Programming Guidelines for the Conversational JWS File .................. 6-40
Programming Guidelines for the JWS File That Invokes a Conversational Web Service ............................................. 6-44
ConversationUtils Utility Class ............................................ 6-47
Updating the build.xml File for a Client of a Conversational Web Service . 6-47
Updating a Stand-Alone Java Client to Invoke a Conversational Web Service .... 6-48
Client Considerations When Redeploying a Conversational Web Service .... 6-50
Creating Buffered Web Services ........................................... 6-50
Creating a Buffered Web Service: Main Steps ............................. 6-50
Configuring the Host WebLogic Server Instance for the Buffered Web Service ... 6-52
Programming Guidelines for the Buffered JWS File ......................... 6-53
Programming the JWS File That Invokes the Buffered Web Service .......... 6-55
Updating the build.xml File for a Client of the Buffered Web Service ........ 6-56
Using the Asynchronous Features Together ................................ 6-57
Example of a JWS File That Implements a Reliable Conversational Web Service . 6-58
Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service .................................................. 6-59
Using Reliable Messaging or Asynchronous Request Response With a Proxy Server .............................................................. 6-61

7. Advanced JWS Programming: JMS Transport and SOAP Message Handlers
   Using JMS Transport as the Connection Protocol ........................................ 7-1
   Using JMS Transport Starting From Java: Main Steps ............................. 7-2
   Using JMS Transport Starting From WSDL: Main Steps ....................... 7-4
   Using the @WLJmsTransport JWS Annotation ....................................... 7-6
   Using the <WLJmsTransport> Child Element of the jwsc Ant Task .......... 7-7
   Updating the WSDL to Use JMS Transport .......................................... 7-8
   Invoking a WebLogic Web Service Using JMS Transport ....................... 7-9
   Creating and Using SOAP Message Handlers ....................................... 7-12
   Adding SOAP Message Handlers to a Web Service: Main Steps .............. 7-15
   Designing the SOAP Message Handlers and Handler Chains ................... 7-15
   Creating the GenericHandler Class .................................................... 7-18
   Configuring Handlers in the JWS File ................................................. 7-26
   Creating the Handler Chain Configuration File .................................... 7-31
   Compiling and Rebuilding the Web Service ........................................... 7-32

8. Data Types and Data Binding
   Overview of Data Types and Data Binding ........................................... 8-1
   Supported Built-In Data Types ......................................................... 8-2
   XML-to-Java Mapping for Built-In Data Types ...................................... 8-2
   Java-to-XML Mapping for Built-In Data Types ...................................... 8-4
   Supported User-Defined Data Types ............................................... 8-6
   Supported XML User-Defined Data Types ......................................... 8-6
9. Invoking Web Services

Overview of Web Services Invocation ........................................ 9-1
Types of Client Applications .................................................. 9-2
JAX-RPC .................................................................................. 9-2
The clientgen Ant Task .............................................................. 9-3
Examples of Clients That Invoke Web Services ......................... 9-4
Invoking a Web Service from a Stand-alone Client: Main Steps .... 9-4
Using the clientgen Ant Task To Generate Client Artifacts .......... 9-5
Getting Information About a Web Service .................................. 9-6
Writing the Java Client Application Code to Invoke a Web Service . 9-8
Compiling and Running the Client Application ......................... 9-9
Sample Ant Build File for a Stand-Alone Java Client ................. 9-11
Invoking a Web Service from Another Web Service ................. 9-12
Sample build.xml File for a Web Service Client ....................... 9-14
Sample JWS File That Invokes a Web Service .......................... 9-16
Using a Proxy Server When Invoking a Web Service ................. 9-18
Using the HttpTransportInfo API to Specify the Proxy Server ... 9-18
Using System Properties to Specify the Proxy Server ............... 9-20
Client Considerations When Redeploying a Web Service .......... 9-21
Creating and Using Client-Side SOAP Message Handlers .......... 9-22
Using Client-Side SOAP Message Handlers: Main Steps .......... 9-23
Example of a Client-Side Handler Class .................................. 9-24
Creating the Client-Side SOAP Handler Configuration File ...... 9-25
XML Schema for the Client-Side Handler Configuration File ...... 9-26
Specifying the Client-Side SOAP Handler Configuration File to clientgen .... 9-27
Using a Client-Side Security WS-Policy File .............................. 9-27
10. Configuring Security

Overview of Web Services Security ........................................ 10-1
What Type of Security Should You Configure? ................................ 10-2
Configuring Message-Level Security (Digital Signatures and Encryption) .... 10-3
  Main Use Cases of Message-Level Security .................................. 10-4
  Using WS-Policy Files for Message-Level Security Configuration ....... 10-4
  WebLogic Server WS-Policy Files ........................................... 10-5
  Abstract and Concrete WS-Policy Files ..................................... 10-13
Configuring Simple Message-Level Security: Main Steps ...................... 10-14
  Ensuring That WebLogic Server Can Validate the Client’s Certificate ...... 10-17
  Updating the JWS File with @Policy and @Policies Annotations ........... 10-17
  Using Key Pairs Other Than the Out-Of-The-Box SSL Pair .................. 10-20
  Setting the SOAP Message Expiration ..................................... 10-22
  Creating and Using a Custom WS-Policy File ................................ 10-23
  Configuring and Using Security Contexts and Derived Keys (WS-SecureConversation) .......................................................... 10-27
  Associating WS-Policy Files at Runtime Using the Administration Console ... 10-32
  Using Security Assertion Markup Language (SAML) Tokens For Identity .... 10-32
  Using Only X.509 Certificate Tokens for Identity ............................ 10-36
  Using a Password Digest In the SOAP Message Rather Than Plaintext ...... 10-38
  Associating a Web Service with a Security Configuration Other Than the Default10-40
  Using System Properties to Debug Message-Level Security .................. 10-40
Updating a Client Application to Invoke a Message-Secured Web Service ...... 10-41
Invoking a Message-Secured Web Service From a Client Running in a WebLogic Server Instance ................................................................. 10-44
Configuring Transport-Level Security .................................................. 10-45
Configuring Two-Way SSL for a Client Application ............................... 10-47
Additional Web Services SSL Examples ............................................... 10-48
Configuring Access Control Security: Main Steps .................................. 10-48
Updating the JWS File With the Security-Related Annotations ............... 10-50
Updating the JWS File With the @RunAs Annotation ............................. 10-52
Setting the Username and Password When Creating the JAX-RPC Service Object .......................... 10-53

11. Administering Web Services
    Overview of WebLogic Web Services Administration Tasks .................. 11-1
    Administration Tools ....................................................................... 11-2
    Using the Administration Console .................................................. 11-3
        Invoking the Administration Console ......................................... 11-4
        How Web Services Are Displayed In the Administration Console ...... 11-5
        Creating a Web Services Security Configuration ............................ 11-7
    Using the WebLogic Scripting Tool .................................................. 11-8
    Using WebLogic Ant Tasks .............................................................. 11-8
    Using the Java Management Extensions (JMX) ................................. 11-8
    Using the J2EE Deployment API ...................................................... 11-9
    Using Work Managers to Prioritize Web Services Work and Reduce Stuck Execute Threads ......................... 11-10

12. Publishing and Finding Web Services Using UDDI
    Overview of UDDI ........................................................................... 12-1
    UDDI and Web Services .................................................................. 12-2
    UDDI and Business Registry ............................................................ 12-2
13. Upgrading WebLogic Web Services From Previous Releases to 9.2

Upgrading a 9.0 or 9.1 WebLogic Web Service to 9.2 ................................. 13-1
Upgrading an 8.1 WebLogic Web Service to 9.2 ......................................... 13-2
  Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 9.2: Main Steps. 13-3
  Upgrading an 8.1 EJB-Implemented WebLogic Web Service to 9.2: Main Steps .. 13-9
Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes .... 13-19

A. Ant Task Reference

Overview of WebLogic Web Services Ant Tasks ................................. A-1
List of Web Services Ant Tasks .................................................... A-2
Using the Web Services Ant Tasks ............................................. A-2
Setting the Classpath for the WebLogic Ant Tasks ............................ A-4
Differences in Operating System Case Sensitivity When Manipulating WSDL and XML Schema Files ........................................... A-5
clientgen ................................................................. A-5
jwsc ............................................................... A-17
wsdle ................................................................. A-53

B. JWS Annotation Reference

Overview of JWS Annotation Tags ........................................... B-1
Standard JSR-181 JWS Annotations Reference ......................... B-4
  javax.jws.WebService ............................................... B-4
  javax.jws.WebMethod .............................................. B-6
  javax.jws.Oneway .................................................. B-6
  javax.jws.WebParam ............................................... B-7
  javax.jws.WebResult .............................................. B-8
  javax.jws.HandlerChain .......................................... B-9
  javax.jws.soap.SOAPBinding ..................................... B-11
  javax.jws.soap.SOAPMessageHandler ............................. B-13
  javax.jws.soap.InitParam ...................................... B-14
  javax.jws.soap.SOAPMessageHandlers .......................... B-14
WebLogic-Specific JWS Annotations Reference ......................... B-16
  weblogic.jws.AsyncFailure ..................................... B-17
  weblogic.jws.AsyncResponse .................................... B-20
  weblogic.jws.Binding ........................................... B-23
  weblogic.jws.BufferQueue ..................................... B-25
  weblogic.jws.Callback .......................................... B-26
  weblogic.jws.CallbackMethod .................................. B-28
  weblogic.jws.CallbackService .................................. B-29
  weblogic.jws.Context ........................................... B-31
weblogic.jws.Conversation ....................................................... B-33
weblogic.jws.Conversational .................................................... B-35
weblogic.jws.MessageBuffer .................................................... B-38
weblogic.jws.Policies .............................................................. B-40
weblogic.jws.Policy ................................................................. B-41
weblogic.jws.ReliabilityBuffer .................................................. B-43
weblogic.jws.ReliabilityErrorHandler ....................................... B-45
weblogic.jws.ServiceClient ..................................................... B-47
weblogic.jws.StreamAttachments .............................................. B-51
weblogic.jws.Transactional ...................................................... B-52
weblogic.jws.Types ............................................................... B-54
weblogic.jws.WildcardBinding .................................................. B-56
weblogic.jws.WildcardBindings ............................................... B-57
weblogic.jws.WLHttpTransport ................................................ B-57
weblogic.jws.WLHttpsTransport .............................................. B-59
weblogic.jws.WLJmsTransport .................................................. B-61
weblogic.jws.WSDL ............................................................... B-62
weblogic.jws.security.CallbackRolesAllowed ............................... B-64
weblogic.jws.security.RolesAllowed ....................................... B-65
weblogic.jws.security.RolesReferenced .................................... B-66
weblogic.jws.security.RunAs .................................................... B-67
weblogic.jws.security.SecurityRole .......................................... B-68
weblogic.jws.security.SecurityRoleRef ..................................... B-70
weblogic.jws.security.UserDataConstraint ................................ B-71
weblogic.jws.security.WssConfiguration .................................. B-73
weblogic.jws.soap.SOAPBinding .............................................. B-74
weblogic.jws.security.SecurityRoles (deprecated) ....................... B-77
weblogic.jws.security.SecurityIdentity (deprecated) ..................... B-79
C. Web Service Reliable Messaging Policy Assertion Reference

Overview of a WS-Policy File That Contains Web Service Reliable Messaging Assertions . C-1
Graphical Representation .................................................. C-2
Example of a WS-Policy File With Web Service Reliable Messaging Assertions . C-3
Element Description .......................................................... C-3
  beapolicy:Expires .......................................................... C-3
  beapolicy:QOS ............................................................... C-4
  wsrn:AcknowledgementInterval ....................................... C-5
  wsrn:BaseRetransmissionInterval ..................................... C-6
  wsrn:ExponentialBackoff ............................................... C-6
  wsrn:InactivityTimeout ................................................. C-7
  wsrn:RMAssertion ....................................................... C-7

D. Security Policy Assertion Reference

Overview of a WS-Policy File That Contains Security Assertions ....... D-1
Graphical Representation .................................................. D-2
Example of a Policy File With Security Elements ..................... D-5
Element Description .......................................................... D-6
  CanonicalizationAlgorithm ............................................. D-6
  Claims ...................................................................... D-6
  Confidentiality ........................................................... D-7
  ConfirmationMethod ...................................................... D-7
  DigestAlgorithm .......................................................... D-9
  EncryptionAlgorithm .................................................... D-10
  Identity ................................................................. D-10
  Integrity ................................................................. D-11
  KeyInfo ................................................................. D-11
exposed .............................................................. E-4
login-config ..................................................... E-5
mbean-name ..................................................... E-5
port-component ................................................ E-5
port-component-name ....................................... E-6
service-endpoint-address ................................. E-6
transport-guarantee ......................................... E-6
weblogic-webservices ....................................... E-7
webservice-contextpath ..................................... E-7
webservice-description ..................................... E-7
webservice-description-name ....................... E-8
webservice-security ........................................ E-8
webservice-serviceuri .................................... E-8
wsdl ............................................................. E-9
wsdl-publish-file ............................................ E-9
Introduction and Roadmap

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

- “Document Scope and Audience” on page 1-1
- “Guide to This Document” on page 1-2
- “Related Documentation” on page 1-3
- “Samples for the Web Services Developer” on page 1-4
- “Release-Specific WebLogic Web Services Information” on page 1-5
- “Differences Between 8.1 and 9.X WebLogic Web Services” on page 1-6
- “Summary of WebLogic Web Services Features” on page 1-6

Document Scope and Audience

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

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

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

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

Guide to This Document

This document is organized as follows:

- This chapter, Chapter 1, “Introduction and Roadmap,” introduces the organization of this guide and the features of WebLogic Web Services.
- Chapter 2, “Understanding WebLogic Web Services,” provides an overview of how WebLogic Web Services are implemented, why they are useful, and the standard specifications that they implement or to which they conform.
- Chapter 3, “Common Web Services Use Cases and Examples,” provides a set of common use case and examples of programming WebLogic Web Services, along with step by step instructions on reproducing the example in your own environment.
- Chapter 4, “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 6, “Advanced JWS Programming: Implementing Asynchronous Features,” discusses how to implement the following advanced features for your Web Service: Web Service reliable messaging, conversational Web Services, buffering, asynchronous request-response, and SOAP message handlers for intercepting the request and response SOAP message.
- Chapter 8, “Data Types and Data Binding,” discusses the built-in and user-defined XML Schema and Java data types that are supported by WebLogic Web Services.
Chapter 9, “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 10, “Configuring Security,” provides information about configuring different types of security for a WebLogic Web Service: message-level (digital signatures and encryption), transport-level (SSL), and access control.

Chapter 11, “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 12, “Publishing and Finding Web Services Using UDDI,” describes how to use UDDI to publish and find Web Services.

Chapter 13, “Upgrading WebLogic Web Services From Previous Releases to 9.2,” describes how to upgrade an 8.1, 9.0, and 9.1 Web Service to run on the new 9.2 Web Services runtime environment.


Appendix B, “JWS Annotation Reference,” provides reference information about the JWS annotations (both standard JSR-181 and WebLogic-specific) that you can use in the JWS file that implements your Web Service.


Related Documentation
This document contains Web Service-specific design and development information.
For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- **Developing WebLogic Server Applications** is a guide to developing WebLogic Server components (such as Web applications and EJBs) and applications.
- **Developing Web Applications, 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.

**Downloading Examples Described in this Guide**

Many of the samples described in this guide are available for download from the dev2dev CodeShare site. Each example is self-contained and requires only that you install WebLogic Server, create a domain, and start a server instance. All needed files, such as the JWS file that
implements the sample Web Service, the Java client to invoke the Web Service, and the Ant build.xml file to build, deploy, and run the example are included in the ZIP file.

**Avitek Medical Records Application (MedRec) and Tutorials**

MedRec is an end-to-end sample J2EE 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 J2EE 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.
Differences Between 8.1 and 9.X WebLogic Web Services

Web Services is one of the most important themes of J2EE 1.4, and thus of WebLogic Server 9.X. J2EE 1.4 introduces a standard Java component model for authoring Web Services with the inclusion of new specifications such as Implementing Enterprise Web Services (JSR-921) and Java API for XML Registries (JAX-R), as well as the updated JAX-RPC and SAAJ specifications. Because the implementation of Web Services is now a J2EE standard, there have been many changes between 8.1 and 9.X WebLogic Web Services.

In particular, the programming model used to create WebLogic Web Services has changed to take advantage of the powerful new metadata annotations feature introduced in Version 5.0 of the JDK. In 9.X you use JWS metadata annotations to annotate a Java file with information that specifies the shape and behavior of the Web Service. These JWS annotations include both the standard ones defined by the Web Services Metadata for the Java Platform specification (JSR-181), as well as additional WebLogic-specific ones. This JWS-based programming model is similar to that of WebLogic Workshop 8.1, although in 8.1 the metadata was specified via Javadoc tags. The WebLogic Web Services programming model in 8.1, by contrast, used the many attributes of the Web Service Ant tasks, such as servicegen, to specify the shape and behavior of the Web Service. Occasionally programmers had to update the deployment descriptor file (webservices.xml) manually to specify characteristics of the Web Service. The new programming model makes implementing Web Services much easier and quicker.

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

Additionally, the runtime environment upon which WebLogic Web Services 9.X run has been completely rewritten to support the Implementing Enterprise Web Services, Version 1.1 (JSR-921), specification. This means that Web Services created in 9.X are internally implemented differently from those created in 8.1 and both run in completely different runtime environments. The 8.1 runtime environment has been deprecated, although it will continue to be supported for a limited number of future WebLogic Server releases. This means that even though 8.1 WebLogic Web Services run correctly on WebLogic Server 9.X, this may not always be true and BEA recommends that you upgrade the 8.1 Web Services to run in the 9.X runtime environment.

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

Summary of WebLogic Web Services Features

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

- Programming model based on the new **JDK 5.0 metadata annotations** feature. The Web Services programming model uses JWS annotations, defined by the **Web Services Metadata for the Java Platform specification (JSR-181)**.
  
  See Chapter 5, “Programming the JWS File.”

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

- 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” on page 6-1
  - “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19
  - “Using Callbacks to Notify Clients of Events” on page 6-27
  - “Creating Conversational Web Services” on page 6-37
  - “Creating Buffered Web Services” on page 6-50
  - “Using JMS Transport as the Connection Protocol” on page 7-1

- 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)” on page 10-3.

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

- Data binding between built-in and user-defined XML and Java data types.
  
  See Chapter 8, “Data Types and Data Binding.”

- SOAP message handlers that intercept the request and response SOAP message from an invoke of a Web Service.
Introduction and Roadmap

See “Creating and Using SOAP Message Handlers” on page 7-12.

- Ant tasks that handle JWS files, generate a Web Service from a WSDL file, and create the
  JAX-RPC client classes needed to invoke a Web Service.
  See Appendix A, “Ant Task Reference.”

- Implementation of and conformance with standard Web Services specifications.
  See “Standards Supported by WebLogic Web Services” on page 2-6.
Understanding WebLogic Web Services

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

- “What Are Web Services?” on page 2-1
- “Why Use Web Services?” on page 2-2
- “Anatomy of a WebLogic Web Service” on page 2-3
- “Roadmap of Common Web Service Development Tasks” on page 2-4
- “Standards Supported by WebLogic Web Services” on page 2-6

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, J2EE applications, legacy applications, and so on. They are written in Java, C++, Perl, and other programming languages. Application interoperability is the goal of Web Services and depends upon the service provider's adherence to published industry standards.
Anatomy of a WebLogic Web Service

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

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

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

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

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

This programing model takes advantage of the new JDK 5.0 metadata annotations feature 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 (JSR-181) as well as a set of WebLogic-specific ones.

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

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 \texttt{jwsc} Ant task packages the Web service in a standard Web application WAR file with all the standard WAR artifacts, such as the \texttt{web.xml} and \texttt{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 \texttt{webservices.xml} and \texttt{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 \texttt{jwsc} against more than one JWS file, you can choose whether \texttt{jwsc} packages the Web Services in a single WAR file, or whether \texttt{jwsc} packages each Web Service in a separate WAR file. In either case, \texttt{jwsc} generates a single Enterprise Application.

If you explicitly implement \texttt{javax.ejb.SessionBean} in your JWS file, then the \texttt{jwsc} Ant task packages the Web Service in a standard EJB JAR file with all the usual artifacts, such as the \texttt{ejb-jar.xml} and \texttt{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>
<thead>
<tr>
<th>Major Task</th>
<th>Subtasks and Additional Information</th>
</tr>
</thead>
</table>
| 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 2-6  
| | “Creating a Simple HelloWorld Web Service” on page 3-2  
| | “Common Web Services Use Cases and Examples” on page 3-1  |
| Iteratively develop a basic WebLogic Web Service. | “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  
| | “Integrating Web Services Into the WebLogic Split Development Directory Environment” on page 4-21  
| | “Programming the JWS File” on page 5-1  
| | “Supported Built-In Data Types” on page 8-2  
| | “Supported User-Defined Data Types” on page 8-6  
| | “Programming the User-Defined Java Data Type” on page 5-19  
| | “Throwing Exceptions” on page 5-21  
| | “Accessing Runtime Information about a Web Service Using the JwsContext” on page 5-11  
| | “Should You Implement a Stateless Session EJB?” on page 5-16  
| | “Creating the Basic Ant build.xml File” on page 4-7  
| | “Running the jwsc WebLogic Web Services Ant Task” on page 4-7  |
| 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 4-17  |
Table 2-1  Web Services Tasks

<table>
<thead>
<tr>
<th>Major Task</th>
<th>Subtasks and Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke the Web Service.</td>
<td>“Invoking a Web Service from a Stand-alone Client: Main Steps” on page 9-4</td>
</tr>
<tr>
<td></td>
<td>“Invoking a Web Service from Another Web Service” on page 9-12</td>
</tr>
<tr>
<td></td>
<td>“Invoking a Web Service Using Asynchronous Request-Response” on page 6-19</td>
</tr>
<tr>
<td></td>
<td>“Creating and Using Client-Side SOAP Message Handlers” on page 9-22</td>
</tr>
<tr>
<td></td>
<td>“Using a Client-Side Security WS-Policy File” on page 9-27</td>
</tr>
<tr>
<td>Add advanced features to the Web Service.</td>
<td>“Using Web Service Reliable Messaging” on page 6-1</td>
</tr>
<tr>
<td></td>
<td>“Using Callbacks to Notify Clients of Events” on page 6-27</td>
</tr>
<tr>
<td></td>
<td>“Creating Conversational Web Services” on page 6-37</td>
</tr>
<tr>
<td></td>
<td>“Creating Buffered Web Services” on page 6-50</td>
</tr>
<tr>
<td></td>
<td>“Using JMS Transport as the Connection Protocol” on page 7-1</td>
</tr>
<tr>
<td></td>
<td>“Creating and Using SOAP Message Handlers” on page 7-12</td>
</tr>
<tr>
<td>Secure the Web Service.</td>
<td>“Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3</td>
</tr>
<tr>
<td></td>
<td>“Configuring Transport-Level Security” on page 10-45</td>
</tr>
<tr>
<td></td>
<td>“Configuring Access Control Security: Main Steps” on page 10-48</td>
</tr>
<tr>
<td>Upgrade an 8.1, 9.0, or 9.1 WebLogic Web Service to run in the 9.2 runtime.</td>
<td>“Upgrading a 9.0 or 9.1 WebLogic Web Service to 9.2” on page 13-1</td>
</tr>
<tr>
<td></td>
<td>“Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 9.2: Main Steps” on page 13-3</td>
</tr>
<tr>
<td></td>
<td>“Upgrading an 8.1 EJB-Implemented WebLogic Web Service to 9.2: Main Steps” on page 13-9</td>
</tr>
</tbody>
</table>

Standards Supported by WebLogic Web Services

A Web Service requires the following standard specification implementations or conformance:
Standards Supported by WebLogic Web Services

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

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

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

  WebLogic Web Services are hosted by WebLogic Server and are implemented using standard J2EE components, as defined by the *Implementing Enterprise Web Services* 1.1 specification (JSR-921, which is the 1.1 maintenance release of JSR-109). See “Enterprise Web Services 1.1” on page 2-9.

- 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 2-9.

  WebLogic Web Services implement the SOAP with Attachments API for Java 1.2 specification to access any attachments to the SOAP message. See “SAAJ 1.2” on page 2-10.

- A standard for describing the Web Service to clients so they can invoke it.

  WebLogic Web Services use Web Services Description Language (WSDL) 1.1, an XML-based specification, to describe themselves. See “WSDL 1.1” on page 2-10.

  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 and WS-PolicyAttachment specifications when using policies to describe their reliable messaging and security (digital signatures and encryption) functionality. See “WS-Policy 1.0” on page 2-15 and “WS-PolicyAttachment 1.0” on page 2-15.

- 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 2-12.

- 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) 1.0” on page 2-13.

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

WebLogic Web Services conform to the WS-Addressing 1.0 and WS-ReliableMessaging 1.0 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)**

Although it is possible to program a WebLogic Web Service manually by coding the standard JSR-921 EJB or Java class from scratch and generating its associated artifacts by hand (deployment descriptor files, WSDL, data binding artifacts for user-defined data types, and so on), the entire process can be difficult and tedious. For this reason, BEA recommends that you take advantage of the new JDK 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-921 Java class or EJB and automatically generate all the associated artifacts.
The Java Web Service (JWS) annotated file (called a JWS file for simplicity) is the core of your Web Service. It contains the Java code that determines how your Web Service behaves. A JWS file is an ordinary Java class file that uses JDK 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.

**Enterprise Web Services 1.1**

The Implementing Enterprise Web Services 1.1 specification (JSR-921) defines the programming model and runtime architecture for implementing Web Services in Java that run on a J2EE application server, such as WebLogic Server. In particular, it specifies that programmers implement J2EE Web Services using one of 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 J2EE Web Service packaging format, deployment model, and runtime services, all of which are implemented by WebLogic Web Services.

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

**SOAP 1.1 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:
Understanding WebLogic Web Services

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 1.2**

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

The single package in the API, javax.xml.soap, provides the primary abstraction for SOAP messages with MIME attachments. Attachments may be entire XML documents, XML fragments, images, text documents, or any other content with a valid MIME type. In addition, the package provides a simple client-side view of a request-response style of interaction with a Web Service.

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

**WSDL 1.1**

Web Services Description Language (WSDL) is an XML-based specification that describes a Web Service. A WSDL document describes Web Service operations, input and output parameters, and how a client application connects to the Web Service.

Developers of WebLogic Web Services do not need to create the WSDL files; you generate these files automatically as part of the WebLogic Web Services development process.
The following example, for informational purposes only, shows a WSDL file that describes the stock trading Web Service StockQuoteService that contains the method GetLastStockQuote:

```xml
<?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">
            <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 8-2
- “Supported User-Defined Data Types” on page 8-6

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

The following table briefly describes the core JAX-RPC interfaces and classes.
For detailed information on JAX-RPC, see http://java.sun.com/xml/jaxrpc/index.html.

**Web Services Security (WS-Security) 1.0**

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


**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 1.0

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, but Web Service programmers can also use the APIs that conform to this specification stand-alone if additional addressing functionality is needed.

See Web Services Addressing (WS-Addressing).

WS-Policy 1.0

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-PolicyAttachment 1.0

The Web Services Policy Framework (WS-Policy) specification defines an abstract model and an XML-based expression grammar for policies. This specification, Web Services Policy Attachment (WS-PolicyAttachment), defines two general-purpose mechanisms for associating such policies with the subjects to which they apply. This specification also defines how these general-purpose mechanisms can be used to associate WS-Policy with WSDL and UDDI descriptions.

See Web Services Policy Attachment (WS-PolicyAttachment).

WS-ReliableMessaging 1.0

The WS-ReliableMessaging specification (February 4, 2005) 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 1.0**

The WS-Trust specification (February 2005) defines extensions that build on Web Services Security (WS-Security) 1.0 to provide a framework for requesting and issuing security tokens, and to broker trust relationships.

See Web Services Trust Language (WS-Trust).

**WS-SecureConversation 1.0**

The WS-SecureConversation specification defines extensions that build on Web Services Security (WS-Security) 1.0 and WS-Trust 1.0 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 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:

- “Creating a Simple HelloWorld Web Service” on page 3-2
- “Creating a Web Service With User-Defined Data Types” on page 3-7
- “Creating a Web Service from a WSDL File” on page 3-14
- “Invoking a Web Service from a Stand-alone JAX-RPC Java Client” on page 3-23
- “Invoking a Web Service from a WebLogic Web Service” on page 3-29

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.

The use cases do not go into detail about the tools and technologies used in the examples. For detailed information about specific features, see the relevant topics in this guide, in particular:

- Chapter 4, “Iterative Development of WebLogic Web Services”
- Chapter 5, “Programming the JWS File”
- Chapter 6, “Advanced JWS Programming: Implementing Asynchronous Features”
- Chapter 9, “Invoking Web Services”
- Appendix A, “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 3-4.

   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

```java
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
 */
```
Creating a Simple HelloWorld Web Service

* @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" />

</project>
Common Web Services Use Cases and Examples

```
<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"/>
</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?WSDL"
        destDir="${clientclass-dir}"
        packageName="examples.webservices.hello_world.client"/>
```
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 3-9.

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 3-10.

   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" class="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:

```xml
<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:

```bash
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:

```plaintext
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

```java
package examples.webservices.complex;

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

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 examples.webservices.complex.BasicStruct;

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

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

@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".

Common Web Services Use Cases and Examples

```java
@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.

```xml
<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" />
```
Creating a Web Service With User-Defined Data Types

<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}"/>
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
Creating a Web Service from a WSDL File

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/wsd1c/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-wsd1c" default="all">
   <taskdef name="wsdlc"
            classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
   </project>

   See “Sample Ant Build File for TemperatureService” on page 3-20 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/wsd1c/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” on
page A-53 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/wsd1c
prompt> mkdir src/examples/webservices/wsd1c
prompt> cp output/impl/examples/webservices/wsd1c/TemperaturePortTypeImpl.java
  src/examples/webservices/wsd1c/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 wsd1c 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/wsd1c/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:**

    ```xml
    <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
<?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
```
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}"
Common Web Services Use Cases and Examples

```xml
destImplDir="${impl-dir}"
packageName="examples.webservices.wsdlc" />
</target>
<target name="build-service">
  <jwsc
  srccdir="src"
destdir="${ear-dir}"
  </jwsc
  <jws file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
       compiledWsd1="${compiledWsd1-dir}/TemperatureService_wsdl.jar" />
</target>
<target name="deploy">
  <wldeploy action="deploy" name="${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"/>
```
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:
   ```xml
   <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 3-27 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:
   ```xml
   <target name="build-client">
     <clientgen
       wsdll="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
   ```
Invoking a Web Service from a Stand-alone JAX-RPC Java Client

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” on page A-5.

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 3-26.

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 9, “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:

```xml
<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:

```java
[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

```java
package examples.webservices.simple_client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
```
I nv o k i ng a Web S e r v i c e fr o m a S ta nd - al o ne J AX -R P C J av a C l i e nt

// 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}"/>

Programming Web Services for WebLogic Server

3-27


Common Web Services Use Cases and Examples

```xml
<pathelement path="${java.class.path}"/>
</path>
<taskdef name="clientgen"
   classname="weblogic.wsee.tools.annotation.ClientGenTask"/>
<target name="clean">
   <delete dir="${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 3-31.

   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:

```xml
<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 3-32 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:

```xml
<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:

```xml
<taskdef name="wldeploy" class="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>
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

```java
package examples.webservices.service_to_service;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
import javax.jws.WebMethod;
```

---

**Programming Web Services for WebLogic Server** 3-31
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">
<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"
Common Web Services Use Cases and Examples

```
<target name="deploy">
  <wldeploy action="deploy" 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="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>
```
Iterative Development of WebLogic Web Services

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

- “Overview of the WebLogic Web Service Programming Model” on page 4-2
- “Configuring Your Domain For Web Services Features” on page 4-2
- “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
- “Creating the Basic Ant build.xml File” on page 4-7
- “Running the jwsc WebLogic Web Services Ant Task” on page 4-7
- “Running the wsdlc WebLogic Web Services Ant Task” on page 4-11
- “Updating the Stubbed-Out JWS Implementation Class File Generated By wsdlc” on page 4-13
- “Deploying and Undeploying WebLogic Web Services” on page 4-15
- “Browsing to the WSDL of the Web Service” on page 4-17
- “Configuring the Server Address Specified in the Dynamic WSDL” on page 4-18
- “Testing the Web Service” on page 4-20
Overview of the WebLogic Web Service Programming Model

The WebLogic Web Services programming model centers around JWS files (Java files that use JWS annotations to specify the shape and behavior of the Web Service) and Ant tasks that execute on the JWS file. JWS annotations are based on the new metadata feature of Version 5.0 of the JDK (specified by JSR-175), and include both the standard annotations defined by the Web Services Metadata for the Java Platform specification (JSR-181), as well as additional WebLogic-specific ones. 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:
Iterative Development of WebLogic Web Services Starting From Java: Main Steps

- 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.
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/weblogic92`.
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.


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.

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

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 9, “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 \texttt{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 \texttt{wsdlc} WebLogic Web Services Ant Task” on page 4-11.

6. Update the stubbed-out JWS file generated by the \texttt{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 \texttt{wsdlc}” on page 4-13.

7. Run the \texttt{jwsc} Ant task, specifying the artifacts generated by the \texttt{wsdlc} Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web Service.

   See “Running the \texttt{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 \texttt{location} attribute of the \texttt{<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 \texttt{wsdlc} Ant task generated values for the \texttt{contextPath} and \texttt{serviceURI} of the \texttt{@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 \texttt{jwsc} Ant task.

    See Chapter 9, “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:

```xml
<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-921 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:

```xml
<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="WSDL_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.
Running the jwsc WebLogic Web Services Ant Task

- **JWS_file** refers to the full pathname of your JWS file, relative to the value of the src_directory attribute.

- **WSDL_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” on page A-17 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:

```xml
<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):

```xml
<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.
Running the wsdlc WebLogic Web Services Ant Task

- 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:

```xml
<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.
Iterative Development of WebLogic Web Services

- **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 \texttt{WSDLFile\_wsdl.jar}, where \texttt{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 \texttt{MyService.wsdl}, then the generated JAR file is \texttt{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 \texttt{PortTypeImpl.java}, where \texttt{PortType} refers to the name attribute of the \texttt{<portType>} element in the WSDL file for which you are generating a Web Service. For example, if the port type name is \texttt{MyServicePortType}, then the JWS implementation file is called \texttt{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 \texttt{wsdlc} Ant task generates a package name based on the \texttt{targetNamespace} of the WSDL.

  The required \texttt{taskdef} element specifies the full class name of the \texttt{wsdlc} Ant task.

  Only the \texttt{srcWsdl} and \texttt{destJwsDir} attributes of the \texttt{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 \texttt{targetNamespace} of the WSDL file is not suitable to be converted into a readable package name.

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

  ```xml
  <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 \texttt{TemperatureService.wsdl} and is located in the \texttt{wsdl_files} subdirectory of the directory that contains the \texttt{build.xml} file. The JAR file that will contain the JWS interface and data binding artifacts is generated to the
Updating the Stubbed-Out JWS Implementation Class File Generated By wsdlc

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:

cmd> ant generate-from-wsdl

See “wsdlc” on page A-53 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 PortTypeImpl.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:

```java
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:

```xml
<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`.

```xml
<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:

```xml
<target name="undeploy">
  <wldeploy action="undeploy"
    name="ComplexServiceEar"
    user="weblogic"
    password="weblogic" verbose="true"
  />
</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.
Iterative Development of WebLogic Web Services

- 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...

```java
@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:

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

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

```

```xml
</definitions>
```

4-18 Programming Web Services for WebLogic Server
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 WLXXXTransport 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 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
Iterative Development of WebLogic Web Services

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.

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 J2EE applications and modules, such as EJBs and Web applications, and you want to update the single build.xml file to include Web Services development.

For detailed information about the WebLogic split development directory environment, see Creating a Split Development Directory for an Application and the splitdir/helloWorldEar example installed with WebLogic Server, located in the BEA_HOME/weblogic90/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:

   ```xml
   <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:

   ```xml
   <!-- 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:

   ```xml
   <target name="compile">
   ```
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:

```xml
<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.
Iterative Development of WebLogic Web Services
Overview of JWS Files and JWS Annotations

One way to program a WebLogic Web Service is to code the standard JSR-921 EJB or Java class from scratch and generate its associated artifacts manually (deployment descriptor files, WSDL file, data binding artifacts for user-defined data types, and so on). This process can be difficult 
and tedious. 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 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 4-3 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 Chapter 6, “Advanced JWS Programming: Implementing Asynchronous Features,” 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 both the standard and WebLogic-specific JWS annotations, see Appendix B, “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 5-4 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:
   ```java
   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:
   ```java
   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 5-6.

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 5-7.

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.

```java
package examples.webservices.simple;
```
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

import weblogic.jws.WLHttpTransport;

@WebService(name="SimplePortType", serviceName="SimpleService",
targetNamespace="http://example.org")
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
use=SOAPBinding.Use.LITERAL,
parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
@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 {

       @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:

```java
@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:

```java
@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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>SOAPBinding.Style.RPC</td>
<td>SOAPBinding.Style.DOCUMENT</td>
</tr>
<tr>
<td></td>
<td>SOAPBinding.Style.DOCUMENT</td>
<td></td>
</tr>
<tr>
<td>use</td>
<td>SOAPBinding.Use.LITERAL</td>
<td>SOAPBinding.Use.LITERAL</td>
</tr>
<tr>
<td></td>
<td>SOAPBinding.Use.ENCODED</td>
<td></td>
</tr>
<tr>
<td>parameterStyle</td>
<td>SOAPBinding.ParameterStyle.BARE</td>
<td>SOAPBinding.ParameterStyle.WRAP</td>
</tr>
<tr>
<td></td>
<td>PED</td>
<td>PED</td>
</tr>
</tbody>
</table>

**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 Appendix B, “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:

```java
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:

```java
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:

```java
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.
Programming the JWS File

- **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:

```java
public class Simple {

    @WebMethod()
    @WebResult(name="IntegerOutput",
               targetNamespace="http://example.org/docLiteralBare")
    @WebParam(name="IntegerInput",
               targetNamespace="http://example.org/docLiteralBare")
    public int echoInt(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” on page 6-37 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.

```java
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 {
```
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:
  ```java
  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):
  ```java
  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:
  ```java
  @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:
  ```java
  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

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isFinished()</td>
<td>boolean</td>
<td>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.</td>
</tr>
<tr>
<td>finishConversation()</td>
<td>void</td>
<td>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.</td>
</tr>
<tr>
<td>setMaxAge(java.util.Date)</td>
<td>void</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setMaxAge(String)</td>
<td>void</td>
<td>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(&quot;10 minutes&quot;) 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.</td>
</tr>
<tr>
<td>getMaxAge()</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>getCurrentAge()</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>resetIdleTime()</td>
<td>void</td>
<td>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.</td>
</tr>
</tbody>
</table>
# Accessing Runtime Information about a Web Service Using the JwsContext

## Programming Web Services for WebLogic Server

### Table 5-2 Methods of the JwsContext

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setMaxIdleTime(long)</code></td>
<td>void</td>
<td>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 <code>maxIdleTime</code> attribute of the <code>@Conversational</code> 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 <code>@Conversation</code> or <code>@Conversational</code> annotation.</td>
</tr>
</tbody>
</table>
| `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:  
  
  ```java
  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. |
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 5-18 for an example; the relevant sections are shown in bold:

- Import the standard J2EE EJB classes:
  ```java
  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.
  ```java
  import weblogic.ejbgen.Session;
  ```

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

```java
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 implements a stateless session EJB. The relevant code is shown in bold.

```java
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)
```
```java
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](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” in the documentation.
Types” on page 8-6. See “Supported Built-In Data Types” on page 8-2 for the full list of supported built-in data types.

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

```java
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 3-10:

```java
package examples.webservices.complex;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import weblogic.jws.WLHttpTransport;
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:

```java
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:

```bash
-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:

```java
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;
```

```java
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() {...}
}
```

```java
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() {
        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 9-12 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:

- “Streaming SOAP Attachments” on page 5-24
- “Using SOAP 1.2” on page 5-25
- “Specifying that Operations Run Inside of a Transaction” on page 5-26
- “Getting the HttpServletRequest/Response Object” on page 5-27

**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`” on page B-51 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):

```java
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):

```java
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” on page B-52 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):

```java
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 initied...");
    }

    public void destroy() {
    }

    public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws ServiceException {
    }

    public void handleResponse(HttpServletRequest request, HttpServletResponse response) throws ServiceException {
    }
}
```
Programming the JWS File

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

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

- Create a variable of data type ServletEndpointContext:
  ```java
  private ServletEndpointContext wsctx = null;
  ```

- Because the JWS file implements ServiceLifecycle, you must also implement the init and destroy lifecycle methods:
  ```java
  public void init(Object context) throws ServiceException {
      System.out.println("ServletEndpointContext inited...");
  }
```
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 `jws` 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 returns...
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.
Advanced JWS Programming:
Implementing Asynchronous Features

The following sections describe how to use JWS files to implement asynchronous features. The first four sections describe how to implement these features separately. Typically, however, programmers use these features together; see “Using the Asynchronous Features Together” on page 6-57 for more information.

- “Using Web Service Reliable Messaging” on page 6-1
- “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19
- “Using Callbacks to Notify Clients of Events” on page 6-27
- “Creating Conversational Web Services” on page 6-37
- “Creating Buffered Web Services” on page 6-50
- “Using the Asynchronous Features Together” on page 6-57
- “Using Reliable Messaging or Asynchronous Request Response With a Proxy Server” on page 6-61

Using Web Service Reliable Messaging

Web Service reliable messaging is a framework whereby an application running in one application server can reliably invoke a Web Service running on another application server, assuming that both servers implement the WS-ReliableMessaging specification. Reliable is defined as the ability to guarantee message delivery between the two Web Services.
Note: Web Services reliable messaging works between any two application servers that implement the WS-ReliableMessaging specification. In this document, however, it is assumed that the two application servers are WebLogic Server instances.

WebLogic Web Services conform to the WS-ReliableMessaging specification (February 2005), which describes how two Web Services running on different application servers can communicate reliably in the presence of failures in software components, systems, or networks. In particular, the specification describes an interoperable protocol in which a message sent from a source endpoint (or client Web Service) to a destination endpoint (or Web Service whose operations can be invoked reliably) is guaranteed either to be delivered, according to one or more delivery assurances, or to raise an error.

A reliable WebLogic Web Service provides the following delivery assurances:

- AtMostOnce—Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
- AtLeastOnce—Every message is delivered at least once. It is possible that some messages are delivered more than once.
- ExactlyOnce—Every message is delivered exactly once, without duplication.
- InOrder—Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

See the WS-ReliableMessaging specification for detailed documentation about the architecture of Web Service reliable messaging. “Using Web Service Reliable Messaging: Main Steps” on page 6-4 describes how to create the reliable and client Web Services and how to configure the two WebLogic Server instances to which the Web Services are deployed.

Note: Web Services reliable messaging is not supported with the JMS transport feature.

Use of WS-Policy Files for Web Service Reliable Messaging Configuration

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

These WS-Policy files are XML files that describe features such as the version of the supported WS-ReliableMessaging specification, the source endpoint’s retransmission interval, the destination endpoint’s acknowledgment interval, and so on.
You specify the names of the WS-Policy files that are attached to your Web Service using the @Policy JWS annotation in your JWS file. Use the @Policies annotation to group together multiple @Policy annotations. For reliable messaging, you specify these annotations only at the class level.

WebLogic Server includes two simple WS-Policy files that you can specify in your JWS file if you do not want to create your own WS-Policy files:

- **DefaultReliability.xml**—Specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmisstion interval of 3 seconds. See “DefaultReliability.xml WS-Policy File” on page 6-3 for the actual WS-Policy file.

- **LongRunningReliability.xml**—Similar to the preceding default reliable messaging WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours.) See “LongRunningReliability.xml WS-Policy File” on page 6-4 for the actual WS-Policy file.

You cannot change these pre-packaged files, so if their values do not suit your needs, you must create your own WS-Policy file.


**DefaultReliability.xml WS-Policy File**

```xml
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
  xmlns:beapolicy="http://www.bea.com/wsrm/policy"
>
  <wsrm:RMAssertion >
    <wsrm:InactivityTimeout
      Milliseconds="600000" />
    <wsrm:BaseRetransmissionInterval
      Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
</wsp:Policy
```
Advanced JWS Programming: Implementing Asynchronous Features

<wsrm:AcknowledgementInterval
    Milliseconds="200" />
<beapolicy:Expires Expires="P1D" optional="true"/>
</wsrm:RMAssertion>
</wsp:Policy>

LongRunningReliability.xml WS-Policy File

<?xml version="1.0"?>
<wsp:Policy
    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
    xmlns:beapolicy="http://www.bea.com/wsrm/policy"
>
    <wsrm:RMAssertion >
        <wsrm:InactivityTimeout
            Milliseconds="86400000" />
        <wsrm:BaseRetransmissionInterval
            Milliseconds="3000" />
        <wsrm:ExponentialBackoff />
        <wsrm:AcknowledgementInterval
            Milliseconds="200" />
        <beapolicy:Expires Expires="P1M" optional="true"/>
    </wsrm:RMAssertion>
</wsp:Policy>

Using Web Service Reliable Messaging: Main Steps

Configuring reliable messaging for a WebLogic Web Service requires standard JMS tasks such as creating JMS servers and Store and Forward (SAF) agents, as well as Web Service-specific tasks, such as adding additional JWS annotations to your JWS file. Optionally, you create WS-Policy files that describe the reliable messaging capabilities of the reliable Web Service if you do not use the pre-packaged ones.

If you are using the WebLogic client APIs to invoke a reliable Web Service, the client application must run on WebLogic Server. Thus, configuration tasks must be performed on both the source WebLogic Server instance on which the Web Service that includes client code to invoke the
reliable Web Service reliably is deployed, as well as the destination WebLogic Server instance on which the reliable Web Service itself is deployed.

The following procedure describes how to create a reliable Web Service, as well as a client Web Service that in turn invokes an operation of the reliable Web Service reliably. The procedure shows how to create the JWS files that implement the two Web Services from scratch; if you want to update existing JWS files, use this procedure as a guide. The procedure also shows how to configure the source and destination WebLogic Server instances.

It is assumed that you have created a WebLogic Server instance where you have set up an Ant-based development environment and that you have a working build.xml file to which you can add targets for running the jwsc Ant task and deploying the generated reliable Web Service. It is further assumed that you have a similar setup for another WebLogic Server instance that hosts the client Web Service that invokes the Web Service reliably. For more information, see:

- Chapter 3, “Common Web Services Use Cases and Examples”
- Chapter 4, “Iterative Development of WebLogic Web Services”
- Chapter 5, “Programming the JWS File”
- Chapter 9, “Invoking Web Services”

1. Configure the destination WebLogic Server instance for Web Service reliable messaging.
   This is the WebLogic Server instance to which the reliable Web Service is deployed.
   See “Configuring the Destination WebLogic Server Instance” on page 6-6.

2. Configure the source WebLogic Server instance for Web Service reliable messaging.
   This is the WebLogic Server instance to which the client Web Service that invokes the reliable Web Service is deployed.
   See “Configuring the Source WebLogic Server Instance” on page 6-8.

3. Using your favorite XML or plain text editor, optionally create a WS-Policy file that describes the reliable messaging capabilities of the Web Service running on the destination WebLogic Server. This step is not required if you plan to use one of the two WS-Policy files that are included in WebLogic Server; see “Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 6-2 for more information.
   See “Creating the Web Service Reliable Messaging WS-Policy File” on page 6-8 for details about creating your own WS-Policy file.
4. Create a new JWS file, or update an existing one, which implements the reliable Web Service that will run on the destination WebLogic Server.

   See “Programming Guidelines for the Reliable JWS File” on page 6-10.

5. Update your `build.xml` file to include a call to the `jwsc` Ant task which will compile the reliable JWS file into a Web Service.

   See “Running the jwsc WebLogic Web Services Ant Task” on page 4-7 for general information about using the `jwsc` task.

6. Compile your destination JWS file by calling the appropriate target and deploy to the destination WebLogic Server. For example:

   `prompt> ant build-mainService deploy-mainService`

7. Create a new JWS file, or update an existing one, that implements the client Web Service that invokes the reliable Web Service. This service will be deployed to the source WebLogic Server.

   See “Programming Guidelines for the JWS File That Invokes a Reliable Web Service” on page 6-15.

8. Update the `build.xml` file that builds the client Web Service.

   See “Updating the build.xml File for a Client of a Reliable Web Service” on page 6-18.

9. Compile your client JWS file by calling the appropriate target and deploy to the source WebLogic Server. For example:

   `prompt> ant build-clientService deploy-clientService`

**Configuring the Destination WebLogic Server Instance**

Configuring the WebLogic Server instance on which the reliable Web Service is deployed involves configuring JMS and store and forward (SAF) resources.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see “Configuring Your Domain For Web Services Features” on page 4-2.

If, however, you prefer to configure the resources yourself, use the following high-level procedure which lists the tasks and then points to the Administration Console Online Help for details on performing the tasks.
1. Invoke the Administration Console for the domain that contains the destination WebLogic Server in your browser.
   See “Invoking the Administration Console” on page 11-4 for instructions on the URL that invokes the Administration Console.

2. Optionally create a persistent store (either file or JDBC) that will be used by the destination WebLogic Server to store internal Web Service reliable messaging information. You can use an existing one, or the default store that always exists, if you do not want to create a new one.
   See Create file stores.

3. Create a JMS Server. If a JMS server already exists, you can use it if you do not want to create a new one.
   See Create JMS servers.

4. Create a JMS module, and then define a JMS queue in the module. If a JMS module already exists, you can use it if you do not want to create a new one. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name.
   Take note of the JNDI name you define for the JMS queue because you will later use it when you program the JWS file that implements your reliable Web Service.
   See Create JMS modules and Create queues.

5. Create a store and forward (SAF) agent. You can use an existing one if you do not want to create a new one.
   When you create the SAF agent:
   - Set the Agent Type field to Both to enable both sending and receiving agents.
   - Be sure to target the SAF agent to your WebLogic Server instance by clicking Next on the first assistant page rather than Finish.
   See Create Store and Forward agents.

Cluster Considerations
If you are using the Web Service reliable messaging feature in a cluster, you must:

- Still create a local JMS queue, rather than a distributed queue, when creating the JMS queue in step 4 in “Configuring the Destination WebLogic Server Instance”.
- Explicitly target this JMS queue to each server in the cluster.
Configuring the Source WebLogic Server Instance

Configuring the WebLogic Server instance on which the client Web Service is deployed involves configuring JMS and store and forward (SAF) resources.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see “Configuring Your Domain For Web Services Features” on page 4-2.

If, however, you prefer to configure the resources yourself, use the following high-level procedure which lists the tasks and then points to the Administration Console Online Help for details on performing the tasks.

1. Invoke the Administration Console for the domain that contains the source WebLogic Server in your browser.
   
   See “Invoking the Administration Console” on page 11-4 for instructions on the URL that invokes the Administration Console.

2. Create a persistent store (file or JDBC) that will be used by the source WebLogic Server to store internal Web Service reliable messaging information. You can use an existing one if you do not want to create a new one.
   
   See Create file stores.

3. Create a JMS Server. You can use an existing one if you do not want to create a new one.
   
   See Create JMS servers.

4. Create a store and forward (SAF) agent. You can use an existing one if you do not want to create a new one.
   
   Be sure when you create the SAF agent that you set the Agent Type field to Both to enable both sending and receiving agents.
   
   See Create Store and Forward agents.

Creating the Web Service Reliable Messaging WS-Policy File

A WS-Policy file is an XML file that contains policy assertions that comply with the WS-Policy specification. In this case, the WS-Policy file contains Web Service reliable messaging policy assertions.
You can use one of the two default reliable messaging WS-Policy files included in WebLogic Server; these files are adequate for most use cases. However, because these files cannot be changed, if they do not suit your needs, you must create your own. See “Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 6-2 for a description of the included WS-Policy files. The remainder of this section describes how to create your own WS-Policy file.

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declarations for using Web Service reliable messaging policy assertions:

```
<wsp:Policy
    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
    xmlns:beapolicy="http://www.bea.com/wsrm/policy">
```

You wrap all Web Service reliable messaging policy assertions inside of a `<wsrm:RMAssertion>` element. The assertions that use the `wsrm:` namespace are standard ones defined by the WS-ReliableMessaging specification. The assertions that use the `beapolicy:` namespace are WebLogic-specific. See Appendix C, “Web Service Reliable Messaging Policy Assertion Reference,” for details.

All Web Service reliable messaging assertions are optional, so only set those whose default values are not adequate. The order in which the assertions appear is important. You can specify the following assertions; the order they appear in the following list is the order in which they should appear in your WS-Policy file:

- `<wsrm:InactivityTimeout>`—Number of milliseconds, specified with the `Milliseconds` attribute, which defines an inactivity interval. After this amount of time, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may consider the sequence to have terminated due to inactivity. The same is true for the source endpoint. By default, sequences never timeout.

- `<wsrm:BaseRetransmissionInterval>`—Interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message. Default value is set by the SAF agent on the source endpoint’s WebLogic Server instance.

- `<wsrm:ExponentialBackoff>`—Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm. This element has no attributes.

- `<wsrm:AcknowledgmentInterval>`—Maximum interval, in milliseconds, in which the destination endpoint must transmit a stand-alone acknowledgement. The default value is set by the SAF agent on the destination endpoint’s WebLogic Server instance.
Advanced JWS Programming: Implementing Asynchronous Features

- `<beapolicy:Expires>`—Amount of time after which the reliable Web Service expires and does not accept any new sequence messages. The default value is to never expire. This element has a single attribute, `Expires`, whose data type is an XML Schema duration type. For example, if you want to set the expiration time to one day, use the following:
  `<beapolicy:Expires Expires="P1D"/>

- `<beapolicy:QOS>`—Delivery assurance level, as described in “Using Web Service Reliable Messaging” on page 6-1. The element has one attribute, `QOS`, which you set to one of the following values: `AtMostOnce`, `AtLeastOnce`, or `ExactlyOnce`. You can also include the `InOrder` string to specify that the messages be in order. The default value is `ExactlyOnce InOrder`. This element is typically not set.

The following example shows a simple Web Service reliable messaging WS-Policy file:

```xml
<?xml version='1.0'?>
<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
>
   <wsrm:RMAssertion>
       <wsrm:InactivityTimeout Milliseconds="600000"/>
       <wsrm:BaseRetransmissionInterval Milliseconds="500"/>
       <wsrm:ExponentialBackoff/>
       <wsrm:AcknowledgementInterval Milliseconds="2000"/>
   </wsrm:RMAssertion>
</wsp:Policy>
```

Programming Guidelines for the Reliable JWS File

This section describes how to create the JWS file that implements the reliable Web Service. The following JWS annotations are used in the JWS file that implements a reliable Web Service:

- `@weblogic.jws.Policy`—Required. See “Using the @Policy Annotation” on page 6-12.
Using Web Service Reliable Messaging

- `@javax.jws.Oneway`—Required only if you are using Web Service reliable messaging on its own, without also using the asynchronous request-response feature. See “Using the @Oneway Annotation” on page 6-13 and “Using the Asynchronous Features Together” on page 6-57.

- `@weblogic.jws.BufferQueue`—Optional. See “Using the @BufferQueue Annotation” on page 6-14.

- `@weblogic.jws.ReliabilityBuffer`—Optional. See “Using the @ReliabilityBuffer Annotation” on page 6-14

The following example shows a simple JWS file that implements a reliable Web Service; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.reliable;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.ReliabilityBuffer;
import weblogic.jws.BufferQueue;

/**
 * Simple reliable Web Service.
 */
@WebService(name="ReliableHelloWorldPortType",
            serviceName="ReliableHelloWorldService")
@WLHttpTransport(contextPath="ReliableHelloWorld",
                  serviceUri="ReliableHelloWorld",
                  portName="ReliableHelloWorldServicePort")
@Policy(uri="ReliableHelloWorldPolicy.xml",
        direction=Policy.Direction.both,
        attachToWsd1=true)
@BufferQueue(name="webservices.reliable.queue")
public class ReliableHelloWorldImpl {
    @WebMethod()
    @Oneway()
    @ReliabilityBuffer(retryCount=10, retryDelay="10 seconds")
```
In the example, the `ReliableHelloWorldPolicy.xml` file is attached to the Web Service at the class level, which means that the policy file is applied to all public operations of the Web Service. The policy file is applied only to the request Web Service message (as required by the reliable messaging feature) and it is attached to the WSDL file.

The JMS queue that WebLogic Server uses internally to enable the Web Service reliable messaging has a JNDI name of `webservices.reliable.queue`, as specified by the `@BufferQueue` annotation.

The `helloWorld()` method has been marked with both the `@WebMethod` and `@Oneway` JWS annotations, which means it is a public operation called `helloWorld`. Because of the `@Policy` annotation, the operation can be invoked reliably. The Web Services runtime attempts to deliver reliable messages to the service a maximum of 10 times, at 10-second intervals, as described by the `@ReliabilityBuffer` annotation. The message may require re-delivery if, for example, the transaction is rolled back or otherwise does not commit.

### Using the `@Policy` Annotation

Use the `@Policy` annotation in your JWS file to specify that the Web Service has a WS-Policy file attached to it that contains reliable messaging assertions.

See “Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 6-2 for descriptions of the two WS-Policy files (`DefaultReliability.xml` and `LongRunningReliability.xml`) included in WebLogic Server that you can use instead of writing your own.

You must follow these requirements when using the `@Policy` annotation for Web Service reliable messaging:

- Specify the `@Policy` annotation *only* at the class-level.

- Because Web Service reliable messaging is applied to both the request and response SOAP message, set the `direction` attribute of the `@Policy` annotation only to its default value: `Policy.Direction.both`.

Use the `uri` attribute to specify the build-time location of the policy file, as follows:

- If you have created your own WS-Policy file, specify its location relative to the JWS file. For example:
Using Web Service Reliable Messaging

@Policy(uri="ReliableHelloWorldPolicy.xml",
    direction=Policy.Direction.both,
    attachToWsdl=true)

The example shows that the ReliableHelloWorldPolicy.xml file is located in the same
directory as the JWS file.

- To specify that the Web Service is going to use a WS-Policy file that is part of WebLogic
  Server, use the policy: prefix along with the name and path of the policy file. This
  syntax tells the jwsc Ant task at build-time not to look for an actual file on the file system,
  but rather, that the Web Service will retrieve the WS-Policy file from WebLogic Server at
  the time the service is deployed. Use this syntax when specifying one of the pre-packaged
  WS-Policy files or when specifying a WS-Policy file that is packaged in a shared J2EE
  library.

  **Note:** Shared J2EE libraries are useful when you want to share a WS-Policy file with
  multiple Web Services that are packaged in different Enterprise applications. As long
  as the WS-Policy file is located in the META-INF/policies or WEB-INF/policies
directory of the shared J2EE library, you can specify the policy file in the same way
  as if it were packaged in the same archive at the Web Service. See Creating Shared
  J2EE Libraries and Optional Packages at {DOCROOT}/programming/libraries.html
  for information on creating libraries and setting up your environment so the Web
  Service can find the policy files.

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

  @Policy(uri="http://someSite.com/policies/mypolicy.xml"
       direction=Policy.Direction.both,
       attachToWsdl=true)

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

**Using the @Oneway Annotation**

If you plan on invoking the reliable Web Service operation synchronously (or in other words, not
using the asynchronous request-response feature), then the implementing method is required to
be annotated with the @Oneway annotation to specify that the method is one-way. This means that
the method cannot return a value, but rather, must explicitly return void.
Conversely, if the method is *not* annotated with the **@Oneway** annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 and “Using the Asynchronous Features Together” on page 6-57.

**Using the **@BufferQueue** Annotation**

Use the **@BufferQueue** annotation to specify the JNDI name of the JMS queue which WebLogic Server uses to store reliable messages internally. The JNDI name is the one you configured when creating a JMS queue in step 4 in “Configuring the Destination WebLogic Server Instance”.

The **@BufferQueue** annotation is optional; if you do not specify it in your JWS file then WebLogic Server uses a queue with a JNDI name of `weblogic.wsee.DefaultQueue`. You must, however, still explicitly create a JMS queue with this JNDI name using the Administration Console.

**Using the **@ReliabilityBuffer** Annotation**

Use this annotation to specify the number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web Service implementation (default 3) and the amount of time that the server should wait in between retries (default 5 seconds).

Use the **retryCount** attribute to specify the number of retries and the **retryDelay** attribute to specify the wait time. The format of the **retryDelay** attribute is a number and then one of the following strings:

- seconds
- minutes
- hours
- days
- years

For example, to specify a retry count of 20 and a retry delay of two days, use the following syntax:

```
@ReliabilityBuffer(retryCount=20, retryDelay="2 days")
```
**Programming Guidelines for the JWS File That Invokes a Reliable Web Service**

If you are using the WebLogic client APIs, you must invoke a reliable Web Service from within a Web Service; you cannot invoke a reliable Web Service from a stand-alone client application.

The following example shows a simple JWS file for a Web Service that invokes a reliable operation from the service described in “Programming Guidelines for the Reliable JWS File” on page 6-10; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.reliable;
import java.rmi.RemoteException;
import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.ReliabilityErrorHandler;
import examples.webservices.reliable.ReliableHelloWorldPortType;
import weblogic.wsee.reliability.ReliabilityErrorContext;
import weblogic.wsee.reliability.ReliableDeliveryException;

@WebService(name="ReliableClientPortType",
            serviceName="ReliableClientService")
@WLHttpTransport(contextPath="ReliableClient",
                 serviceUri="ReliableClient",
                 portName="ReliableClientServicePort")

public class ReliableClientImpl  {

    @ServiceClient(
            serviceName="ReliableHelloWorldService",
            portName="ReliableHelloWorldServicePort")
    private ReliableHelloWorldPortType port;

    @WebMethod
    public void callHelloWorld(String input, String serviceUrl)
            throws RemoteException {
            port.helloWorld(input);

```
System.out.println(" Invoked the ReliableHelloWorld.helloWorld operation reliably.");

@ReliabilityErrorHandler(target="port")
public void onReliableMessageDeliveryError(ReliabilityErrorContext ctx) {
    ReliableDeliveryException fault = ctx.getFault();
    String message = null;
    if (fault != null) {
        message = ctx.getFault().getMessage();
    }
    String operation = ctx.getOperationName();
    System.out.println("Reliable operation " + operation + " may have not invoked. The error message is " + message);
}

Follow these guidelines when programming the JWS file that invokes a reliable Web Service; code snippets of the guidelines are shown in bold in the preceding example:

- Import the @ServiceClient and @ReliabitliyErrorHandler JWS annotations:
  import weblogic.jws.ServiceClient;
  import weblogic.jws.ReliabilityErrorHandler;

- Import the JAX-RPC stub, created later by the <clientgen> child element of the jwsc Ant task, of the port type of the reliable Web Service you want to invoke. The stub package is specified by the packageName attribute of <clientgen>, and the name of the stub is determined by the WSDL of the invoked Web Service.
  import examples.webservices.reliable.ReliableHelloWorldPortType;

- Import the WebLogic APIs that you will use in the method that handles the error that results when the client Web Service does not receive an acknowledgement of message receipt from the reliable Web Service:
  import weblogic.wsee.reliability.ReliabilityErrorContext;
  import weblogic.wsee.reliability.ReliableDeliveryException;

- In the body of the JWS file, use the @ServiceClient JWS annotation to specify the name and port of the reliable Web Service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.
  @ServiceClient(
    serviceName="ReliableHelloWorldService",
    portName="ReliableHelloWorldServicePort")
private ReliableHelloWorldPortType port;

• Using the stub you annotated with the @ServiceClient annotation, invoke the reliable operation:

        port.helloWorld(input);

Because the operation has been marked one-way, it does not return a value.

• Create a method that handles the error when the client Web Service does not receive an acknowledgement from the reliable Web Service that the latter has received a message and annotate this method with the @weblogic.jws.ReliabilityErrorHandler annotation:

        @ReliabilityErrorHandler(target="port")
        public void onReliableMessageDeliveryError(ReliabilityErrorContext ctx) {
            ReliableDeliveryException fault = ctx.getFault();
            String message = null;
            if (fault != null) {
                message = ctx.getFault().getMessage();
            }
            String operation = ctx.getOperationName();
            System.out.println("Reliable operation " + operation + " may have not invoked. The error message is " + message);
        }

This method takes ReliabilityErrorContext as its single parameter and returns void.

See “weblogic.jws.ReliabilityErrorHandler” on page B-45 for details about programming this error-handling method.

When programming the client Web Service, be sure you do not:

• Specify any reliable messaging annotations (other than @ReliabilityErrorHandler) or use any reliable messaging assertions in the associated WS-Policy files.

• Specify the wsdlLocation attribute of the @ServiceClient annotation. This is because the runtime retrieval of the specified WSDL might not succeed, thus it is better to let WebLogic Server use a local WSDL file instead.

**WsrmUtils Utility Class**

WebLogic Server provides a utility class for use with the Web Service Reliable Messaging feature. Use this class to perform common tasks such as set configuration options, get the sequence id, and terminate a reliable sequence. Some of these tasks are performed in the reliable Web Service, some are performed in the Web Service that invokes the reliable Web Service.

See weblogic.wsee.reliability.WsrmUtils for details.
Updating the build.xml File for a Client of a Reliable Web Service

To update a build.xml file to generate the JWS file that invokes the operation of a reliable Web Service, add taskdef and a build-reliable-client targets that look something like the following; see the description after the example for details:

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

<target name="build-reliable-client">
    <jwsc
        enableAsyncService="true"
        srcdir="src"
        destdir="${client-ear-dir}"
        >
        <jws file="examples/webservices/reliable/ReliableClientImpl.java">
            <clientgen
                wsdl="http://${wls.destination.host}:${wls.destination.port}/ReliableHelloWorld/ReliableHelloWorld?WSDL"
                packageName="examples.webservices.reliable" />
        </jws>
    </jwsc>
</target>
```

Use the taskdef Ant task to define the full class name of the jwsc Ant tasks.

Update the jwsc Ant task that compiles the client Web Service to include a <clientgen> child element of the <jws> element so as to generate and compile the JAX-RPC stubs for the deployed ReliableHelloWorld Web Service. The jwsc Ant task automatically packages them in the generated WAR file so that the client Web Service can immediately access the stubs. You do this because the ReliableClientImpl JWS file imports and uses one of the generated classes.
Client Considerations When Redeploying a Reliable Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated reliable 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 reliable Web Service, its work is considered complete when the existing reliable messaging sequence is explicitly ended by the client or because of a time-out.

For additional information about production redployment and Web Service clients, see “Client Considerations When Redeploying a Web Service” on page 9-21.

Invoking a Web Service Using Asynchronous Request-Response

When you invoke a Web Service synchronously, the invoking client application waits for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the Web Service might be adequate. However, because request processing can be delayed, it is often useful for the client application to continue its work and handle the response later on, or in other words, use the asynchronous request-response feature of WebLogic Web Services.

You invoke a Web Service asynchronously only from a client running in a WebLogic Web Service, never from a stand alone client application. The invoked Web Service does not change in any way, thus you can invoke any deployed Web Service (both WebLogic and non-WebLogic) asynchronously as long as the application server that hosts the Web Service supports the WS-Addressing specification.

When implementing asynchronous request-response in your client, rather than invoking the operation directly, you invoke an asynchronous flavor of the same operation. (This asynchronous flavor of the operation is automatically generated by the jwsc Ant task.) For example, rather than invoking an operation called 
\texttt{getQuote} directly, you would invoke \texttt{getQuoteAsync} instead. The asynchronous flavor of the operation always returns \texttt{void}, even if the original operation returns a value. You then include methods in your client that handle the asynchronous response.
or failures when it returns later on. You put any business logic that processes the return value of
the Web Service operation invoke or a potential failure in these methods. You use both naming
conventions and JWS annotations to specify these methods to the JWS compiler. For example, if
the asynchronous operation is called `getQuoteAsync`, then these methods might be called
`onGetQuoteAsyncResponse` and `onGetQuoteAsyncFailure`.

**Note:** For information about using asynchronous request-response with other asynchronous
features, such as Web Service reliable messaging or buffering, see “Using the
Asynchronous Features Together” on page 6-57. This section describes how to use the
asynchronous request-response feature on its own.

**Note:** The asynchronous request-response feature works only with HTTP; you cannot use it
with the HTTPS or JMS transport.

### Using Asynchronous Request-Response: Main Steps

The following procedure describes how to create a client Web Service that asynchronously
invokes an operation in a different Web Service. The procedure shows how to create the JWS file
that implements the client Web Service from scratch; if you want to update an existing JWS file,
use this procedure as a guide.

For clarity, it is assumed in the procedure that:

- The client Web Service is called `StockQuoteClientService`.
- The `StockQuoteClientService` service is going to invoke the `getQuote(String)`
  operation of the already-deployed `StockQuoteService` service whose WSDL is found at
  the following URL:


It is further assumed that you have set up an Ant-based development environment and that you
have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and
deploying the generated service. See Chapter 3, “Common Web Services Use Cases and
Examples,” Chapter 4, “Iterative Development of WebLogic Web Services,” and Chapter 5,
“Programming the JWS File.”

1. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that
   implements the `StockQuoteClientService` Web Service.
   See “Writing the Asynchronous JWS File” on page 6-21.

2. Update your `build.xml` file to compile the JWS file that implements the
   `StockQuoteClientService`. You will add a `<clientgen>` child element to the `jwsc` Ant
task so as to automatically generate the asynchronous flavor of the Web Service operations you are invoking.


3. Run the Ant target to build the StockQuoteClientService:
   `prompt> ant build-clientService`

4. Deploy the StockQuoteClientService Web Service as usual.
   See “Deploying and Undeploying WebLogic Web Services” on page 4-15.

When you invoke the StockQuoteClientService Web Service, which in turn invokes the StockQuoteService Web Service, the second invoke will be asynchronous rather than synchronous.

Writing the Asynchronous JWS File

The following example shows a simple JWS file that implements a Web Service called StockQuoteClient that has a single method, asyncOperation, that in turn asynchronously invokes the getQuote method of the StockQuote service. The Java code in bold is described “Coding Guidelines for Invoking a Web Service Asynchronously” on page 6-23. See “Example of a Synchronous Invoke” on page 6-25 to see how the asynchronous invoke differs from a synchronous invoke of the same operation.

```java
package examples.webservices.async_req_res;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;
import weblogic.wsee.async.AsyncPreCallContext;
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPostCallContext;
import javax.jws.WebService;
import javax.jws.WebMethod;
import examples.webservices.async_req_res.StockQuotePortType;
import java.rmi.RemoteException;
```
@WebService(name="StockQuoteClientPortType",
    serviceName="StockQuoteClientService",
    targetNamespace="http://examples.org/")

@WLHttpTransport(contextPath="asyncClient",
    serviceUri="StockQuoteClient",
    portName="StockQuoteClientServicePort")

/**
 * Client Web Service that invokes the StockQuote Service asynchronously.
 */

public class StockQuoteClientImpl {

    @ServiceClient(wsdlLocation="http://localhost:7001/async/StockQuote?WSDL",
            serviceName="StockQuoteService", portName="StockQuote")
    private StockQuotePortType port;

    @WebMethod
    public void asyncOperation (String symbol) throws RemoteException {
        AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
        apc.setProperty("symbol", symbol);
        try {
            port.getQuoteAsync(apc, symbol);
            System.out.println("in getQuote method of StockQuoteClient WS");
        } catch (RemoteException re) {
            System.out.println("RemoteException thrown");
            throw new RuntimeException(re);
        }
    }

    @AsyncResult(target="port", operation="getQuote")
    public void onGetQuoteAsyncResponse(AsyncPostCallContext apc, int quote) {
        System.out.println("-------------------");
        System.out.println("Got quote " + quote);
        System.out.println("-------------------");
    }

    @AsyncFailure(target="port", operation="getQuote")
    public void onGetQuoteAsyncFailure(AsyncPostCallContext apc, Throwable e) {
        System.out.println("-------------------");
        e.printStackTrace();
        System.out.println("-------------------");
    }
}

Advanced JWS Programming: Implementing Asynchronous Features

Programming Web Services for WebLogic Server
Coding Guidelines for Invoking a Web Service Asynchronously

The following guidelines for invoking an operation asynchronously correspond to the Java code shown in bold in the example described in “Writing the Asynchronous JWS File” on page 6-21. These guidelines are in addition to the standard ones for creating JWS files. See “Example of a Synchronous Invoke” on page 6-25 to see how the asynchronous invoke differs from a synchronous invoke of the same operation.

To invoke an operation asynchronously in your JWS file:

1. Import the following WebLogic-specific JWS annotations related to the asynchronous request-response feature:
   ```java
   import weblogic.jws.ServiceClient;
   import weblogic.jws.AsyncResponse;
   import weblogic.jws.AsyncFailure;
   ```

2. Import the JAX-RPC stub, created later by the jwsc Ant task, of the port type of the Web Service you want to invoke. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of jwsc, and the name of the stub is determined by the WSDL of the invoked Web Service.
   ```java
   import examples.webservices.async_req_res.StockQuotePortType;
   ```

3. Import the asynchronous pre- and post-call context WebLogic APIs:
   ```java
   import weblogic.wsee.async.AsyncCallContextFactory;
   import weblogic.wsee.async.AsyncPreCallContext;
   import weblogic.wsee.async.AsyncPostCallContext;
   ```

   The `AsyncPreCallContext` and `AsyncPostCallContext` APIs describe asynchronous contexts that you can use for a variety of reasons in your Web Service: to set a property in the pre-context so that the method that handles the asynchronous response can distinguish between different asynchronous calls; to set and get contextual variables, such as the name of the user invoking the operation, their password, and so on; to get the name of the JAX-RPC stub that invoked a method asynchronously; and to set a time-out interval on the context.

   See Javadocs for additional reference information about these APIs.

4. In the body of the JWS file, use the required `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the Web Service you will be invoking asynchronously. You specify this annotation at the field-level on a variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.
   ```java
   @ServiceClient(
   ```

---

**Invoking a Web Service Using Asynchronous Request-Response**

**Programming Web Services for WebLogic Server** 6-23
When you annotate a variable (in this case, `port`) with the `@ServiceClient` annotation, the Web Services runtime automatically initializes and instantiates the variable, preparing it so that it can be used to invoke another Web Service asynchronously.

- In the method of the JWS file which is going to invoke the `getQuote` operation asynchronously, get a pre-call asynchronous context using the context factory:

  ```java
  AsyncPreCallContext apc =
  AsyncCallContextFactory.getAsyncPreCallContext();
  ```

- Use the `setProperty` method of the pre-call context to create a property whose name and value is the same as the parameter to the `getQuote` method:

  ```java
  apc.setProperty("symbol", symbol);
  ```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the operation (in this case, `getQuote`). Instead of invoking it directly, however, invoke the asynchronous flavor of the operation, which has `Async` added on to the end of its name. The asynchronous flavor always returns `void`. Pass the asynchronous context as the first parameter:

  ```java
  port.getQuoteAsync(apc, symbol);
  ```

- For each operation you will be invoking asynchronously, create a method called `onOperationNameAsyncResponse`, where `OperationName` refers to the name of the operation, with initial letter always capitalized. The method must return `void`, and have two parameters: the post-call asynchronous context and the return value of the operation you are invoking. Annotate the method with the `@AsyncResponse` JWS annotation; use the `target` attribute to specify the variable whose datatype is the JAX-RPC stub and the `operation` attribute to specify the name of the operation you are invoking asynchronously. Inside the body of the method, put the business logic that processes the value returned by the operation.

  ```java
  @AsyncResponse(target="port", operation="getQuote")
  public void onGetQuoteAsyncResponse(AsyncPostCallContext apc, int quote) {
      System.out.println("-------------------");
      System.out.println("Got quote "+ quote);
      System.out.println("-------------------");
  }
  ```

- For each operation you will be invoking asynchronously, create a method called `onOperationNameAsyncFailure`, where `OperationName` refers to the name of the
operation, with initial letter capitalized. The method must return void, and have two parameters: the post-call asynchronous context and a Throwable object, the superclass of all exceptions to handle any type of exception thrown by the invoked operation. Annotate the method with the @AsyncFailure JWS annotation; use the target attribute to specify the variable whose datatype is the JAX-RPC stub and the operation attribute to specify the name of the operation you are invoking asynchronously. Inside the method, you can determine the exact nature of the exception and write appropriate Java code.

@AsyncFailure(target="port", operation="getQuote")
public void onGetQuoteAsyncFailure(AsyncPostCallContext apc, Throwable e) {
    System.out.println("-------------------");
    e.printStackTrace();
    System.out.println("-------------------");
}

Note: You are not required to use the @AsyncResponse and @AsyncFailure annotations, although it is a good practice because it clears up any ambiguity and makes your JWS file clean and understandable. However, in the rare use case where you want one of the onXXX methods to handle the asynchronous response or failure from two (or more) stubs that are invoking operations from two different Web Services that have the same name, then you should explicitly NOT use these annotations. Be sure that the name of the onXXX methods follow the correct naming conventions exactly, as described above.

Example of a Synchronous Invoke

The following example shows a JWS file that invokes the getQuote operation of the StockQuote Web Service synchronously. The example is shown only so you can compare it with the corresponding asynchronous invoke shown in “Writing the Asynchronous JWS File” on page 6-21.

package examples.webservices.async_req_res;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import javax.jws.WebService;
import javax.jws.WebMethod;
import java.rmi.RemoteException;
import java.rmi.RemoteException;
@WebService(name="SyncClientPortType",
    serviceName="SyncClientService",
    targetNamespace="http://examples.org/")
@WLHttpTransport(contextPath="syncClient",
  serviceUri="SyncClient",
  portName="SyncClientPort")

/**
 * Normal service-to-service client that invokes StockQuote service
 * synchronously.
 */

class SyncClientImpl {

  @ServiceClient(wsdlLocation="http://localhost:7001/async/StockQuote?WSDL",
      serviceName="StockQuoteService", portName="StockQuote")
  private StockQuotePortType port;

  @WebMethod
  public void nonAsyncOperation(String symbol) throws RemoteException {
    int quote = port.getQuote(symbol);
    System.out.println("-------------------");
    System.out.println("Got quote " + quote);
    System.out.println("-------------------");
  }
}

Updating the build.xml File When Using Asynchronous Request-Response

To update a build.xml file to generate the JWS file that invokes a Web Service operation asynchronously, add taskdefs and a build-clientService target that looks something like the following; see the description after the example for details:

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

<target name="build-clientService">
  <jwsc
    enableAsyncService="true"
    srcdir="src"
    destdir="${clientService-ear-dir}" />
  <jws
    file="examples/webservices/async_req_res/StockQuoteClientImpl.java" />
Using Callbacks to Notify Clients of Events

Use the taskdef Ant task to define the full class name of the jwsc Ant tasks. Update the jwsc Ant task that compiles the client Web Service to include a <clientgen> child element of the <jws> element so as to generate and compile the JAX-RPC stubs for the deployed StockQuote Web Service. The jwsc Ant task automatically packages them in the generated WAR file so that the client Web Service can immediately access the stubs. By default, the jwsc Ant task in this case generates both synchronous and asynchronous flavors of the Web Service operations in the JAX-RPC stubs. You do this because the StockQuoteClientImpl JWS file imports and uses one of the generated classes.

Disabling The Internal Asynchronous Service

By default, every WebLogic Server instance deploys an internal asynchronous Web Service that handles the asynchronous request-response feature. To specify that you do not want to deploy this internal service, start the WebLogic Server instance using the -Dweblogic.wsee.skip.async.response=true Java system property. One reason for disabling the asynchronous service is if you use a WebLogic Server instance as a Web proxy to a WebLogic cluster. In this case, asynchronous messages will never get to the cluster, as required, because the asynchronous service on the proxy server consumes them instead. For this reason, you must disable the asynchronous service on the proxy server using the system property.

For details on specifying Java system properties to configure WebLogic Server, see Specifying Java Options for a WebLogic Server Instance.

Using Callbacks to Notify Clients of Events

Callbacks notify a client of your Web Service that some event has occurred. For example, you can notify a client when the results of that client's request are ready, or when the client's request cannot be fulfilled.
When you expose method as a standard public operation in your JWS file (by using the 
@WebMethod annotation), the client sends a SOAP message to the Web Service to invoke the 
operation. When you add a callback to a Web Service, however, you define a message that the 
Web Service sends back to the client Web Service, notifying the client of an event that has 
ocurred. So exposing a method as a public operation and defining a callback are completely 
symmetrical processes, with opposite recipients.

WebLogic Server automatically routes the SOAP message from client invoke to the target Web 
Service. In order to receive callbacks, however, the client must be operating in an environment 
that provides the same services. This typically means the client is a Web Service running on a 
Web server. If the client does not meet these requirements, it is likely not capable of receiving 
callbacks from your Web Service.

The protocol and message format used for callbacks is always the same as the protocol and 
message format used by the conversation start method that initiated the current conversation. If 
you attempt to override the protocol or message format of a callback, an error is thrown.

Callback Implementation Overview and Terminology

To implement callbacks, you must create or update the following three Java files:

- **Callback interface**: Java interface file that defines the callback methods. You do not 
  explicitly implement this file yourself; rather, the jwsc Ant task automatically generates an 
  implementation of the interface. The implementation simply passes a message from the 
  target Web Service back to the client Web Service. The generated Web Service is deployed 
  to the same WebLogic Server that hosts the client Web Service.

  In the example in this section, the callback interface is called CallbackInterface. The 
  interface defines a single callback method called callbackOperation().

- **JWS file that implements the target Web Service**: The target Web Service includes one 
  or more standard operations that invoke a method defined in the callback interface; this 
  method in turn sends a message back to the client Web Service that originally invoked the 
  operation of the target Web Service.

  In the example, this Web Service is called TargetService and it defines a single standard 
  method called targetOperation().

- **JWS file that implements the client Web Service**: The client Web Service invokes an 
  operation of the target Web Service. This Web Service includes one or more methods that 
  specify what the client should do when it receives a callback message back from the target 
  Web Service via a callback method.
In the example, this Web Service is called `CallbackClient` and the method that is automatically invoked when it receives a callback is called `callbackHandler()`. The method that invokes `TargetService` in the standard way is called `clientOperation()`.

The following graphic shows the flow of messages:

1. The `clientOperation()` method of the `CallbackClient` Web Service, running in one WebLogic Server instance, explicitly invokes the `targetOperation()` operation of the `TargetService`. The `TargetService` service might be running in a separate WebLogic Server instance.

2. The implementation of the `TargetService.targetOperation()` method explicitly invokes the `callbackOperation()` operation of the `CallbackInterface`, which implements the callback service. The callback service is deployed to the WebLogic Server which hosts the client Web Service.

3. The `jwsc-generated` implementation of the `CallbackInterface.callbackOperation()` method simply sends a message back to the `CallbackClient` Web Service. The client Web Service includes a method `callbackHandler()` that handles this message.

### Programming Callbacks: Main Steps

The procedure in this section describes how to program and compile the three JWS files that are required to implement callbacks: the target Web Service, the client Web Service, and the callback interface. The procedure shows how to create the JWS files from scratch; if you want to update existing JWS files, you can also use this procedure as a guide.
It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web Services. For more information, see Chapter 3, “Common Web Services Use Cases and Examples,” Chapter 4, “Iterative Development of WebLogic Web Services,” and Chapter 5, “Programming the JWS File.”

1. Using your favorite IDE, create a new JWS file, or update an existing one, that implements the target Web Service.
   

   **Note:** The JWS file that implements the target Web Service invokes one or more callback methods of the callback interface. However, the step that describes how to program the callback interface comes later in this procedure. For this reason, programmers typically program the three JWS files at the same time, rather than linearly as implied by this procedure. The steps are listed in this order for clarity only.

2. Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the target JWS file into a Web Service.
   
   See “Running the jwsc WebLogic Web Services Ant Task” on page 4-7.

3. Run the Ant target to build the target Web Service. For example:

   ```bash
   prompt> ant build-mainService
   ```

4. Deploy the target Web Service as usual.
   
   See “Deploying and Undeploying WebLogic Web Services” on page 4-15.

5. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that implements the client Web Service. It is assumed that the client Web Service is deployed to a different WebLogic Server instance from the one that hosts the target Web Service.
   
   See “Programming Guidelines for the Callback Client Web Service” on page 6-32.

6. Create the callback JWS interface that implements the callback Web Service.
   
   See “Programming Guidelines for the Callback Interface” on page 6-35.

7. Update the `build.xml` file that builds the client Web Service. The `jwsc` Ant task that builds the client Web Service also implicitly generates the callback Web Service from the callback interface file.
   
   See “Updating the build.xml File for the Client Web Service” on page 6-36.

8. Run the Ant target to build the client and callback Web Services.
9. Deploy the client Web Service as usual. Because the callback service is packaged in the same EAR file as the client Web Service, it will also be deployed at the same time. See “Deploying and Undeploying WebLogic Web Services” on page 4-15.

**Programming Guidelines for Target Web Service**

The following example shows a simple JWS file that implements the target Web Service; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.callback;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Callback;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(name="CallbackPortType",
serviceName="TargetService",
targetNamespace="http://examples.org/")
@WLHttpTransport(contextPath="callback",
serviceUri="TargetService",
portName="TargetServicePort")

/**
 * callback service
 */

public class TargetServiceImpl {

@Callback
CallbackInterface callback;

@WebMethod
public void targetOperation (String message) {
    callback.callbackOperation (message);
}
}
```

Follow these guidelines when programming the JWS file that implements the target Web Service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the required JWS annotations:
import weblogic.jws.Callback;

- Use the @weblogic.jws.Callback JWS annotation to specify that a variable is a callback, which means that you can use the annotated variable to send callback events back to a client Web Service that invokes an operation of the TargetService Web Service. The data type of the variable is the callback interface, which in this case is called CallbackInterface.

```java
@Callback
CallbackInterface callback;
```

- In a method that implements an operation of the TargetService, use the annotated variable to invoke one of the callback methods of the callback interface, which in this case is called callbackOperation():

```java
callback.callbackOperation(message);
```

See Appendix B, “JWS Annotation Reference,” for additional information about the JWS annotations discussed in this section.

### Programming Guidelines for the Callback Client Web Service

The following example shows a simple JWS file for a client Web Service that invokes the target Web Service described in “Programming Guidelines for Target Web Service” on page 6-31; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.callback;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.CallbackMethod;
import weblogic.jws.security.CallbackRolesAllowed;
import weblogic.jws.security.SecurityRole;
import javax.jws.WebService;
import javax.jws.WebMethod;
import examples.webservices.callback.CallbackPortType;
import java.rmi.RemoteException;

import javax.jws.WebMethod;

import examples.webservices.callback.CallbackPortType;
import java.rmi.RemoteException;

@WebService(name="CallbackClientPortType",
            serviceName="CallbackClientService",
            targetNamespace="http://examples.org/"

```
Using Callbacks to Notify Clients of Events

@WLHttpTransport(contextPath="callbackClient", serviceUri="CallbackClient", portName="CallbackClientPort")

public class CallbackClientImpl {

@ServiceClient(
    serviceName="TargetService",
    portName="TargetServicePort")
@CallbackRolesAllowed(@SecurityRole(role="mgr", mapToPrincipals="joe"))
private CallbackPortType port;

@WebMethod
public void clientOperation (String message) {
    try {
        port.targetOperation(message);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

@CallbackMethod(target="port", operation="callbackOperation")
@CallbackRolesAllowed(@SecurityRole(role="engineer", mapToPrincipals="shackell"))
public void callbackHandler(String msg) {
    System.out.println (msg);
}

Follow these guidelines when programming the JWS file that invokes the target Web Service; code snippets of the guidelines are shown in bold in the preceding example:

- Import the required JWS annotations:
  
  ```java
  import weblogic.jws.ServiceClient;
  import weblogic.jws.CallbackMethod;
  ```

- Optionally import the security-related annotations if you want to specify the roles that are allowed to invoke the callback methods:

  ```java
  import weblogic.jws.security.CallbackRolesAllowed;
  import weblogic.jws.security.SecurityRole;
  ```

- Import the JAX-RPC stub of the port type of the target Web Service you want to invoke. The actual stub itself will be created later by the jwsc Ant task. The stub package is
specified by the `packageName` attribute of the `<clientgen>` child element of `<jws>`, and the name of the stub is determined by the WSDL of the invoked Web Service.

```java
import examples.webservices.callback.CallbackPortType;

In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the target Web Service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.

```java
@ServiceClient(
    serviceName="TargetService",
    portName="TargetServicePort")
@CallbackRolesAllowed(@SecurityRole(role="mgr",
    mapToPrincipals="joe"))
private CallbackPortType port;
```

The preceding code also shows how to use the optional `@CallbackRolesAllowed` annotation to specify the list of `@SecurityRoles` that are allowed to invoke the callback methods.

- Using the variable you annotated with the `@ServiceClient` annotation, invoke an operation of the target Web Service. This operation in turn will invoke a callback method of the callback interface:

```java
port.targetOperation(message);
```

- Create a method that will handle the callback message received from the callback service. You can name this method anything you want. However, its signature should exactly match the signature of the corresponding method in the callback interface.

Annotate the method with the `@CallbackMethod` annotation to specify that this method handles callback messages. Use the `target` attribute to specify the name of the JAX-RPC port for which you want to receive callbacks (in other words, the variable you previously annotated with `@ServiceClient`). Use the `operation` attribute to specify the name of the callback method in the callback interface from which this method will handle callback messages.

```java
@CallbackMethod(target="port", operation="callbackOperation")
@CallbackRolesAllowed(@SecurityRole(role="engineer",
    mapToPrincipals="shackell"))
public void callbackHandler(String msg) {
    System.out.println (msg);
}
```
The preceding code also shows how to use the optional \texttt{@CallbackRolesAllowed} annotation to further restrict the security roles that are allowed to invoke this particular callback method.

See Appendix B, “JWS Annotation Reference,” for additional information about the JWS annotations discussed in this section.

**Programming Guidelines for the Callback Interface**

The callback interface is also a JWS file that implements a Web Service, except for one big difference: instead of using the standard \texttt{@javax.jws.WebService} annotation to specify that it is a standard Web Service, you use the WebLogic-specific \texttt{@weblogic.jws.CallbackService} to specify that it is a callback service. The attributes of \texttt{@CallbackService} are a restricted subset of the attributes of \texttt{@WebService}.

Follow these restrictions on the allowed data types and JWS annotations when programming the JWS file that implements a callback service:

- You cannot use any WebLogic-specific JWS annotations other than \texttt{@weblogic.jws.CallbackService}.
- You can use all standard JWS annotations except for the following:
  - \texttt{javax.jws.HandlerChain}
  - \texttt{javax.jws.soap.SOAPMessageHandler}
  - \texttt{javax.jws.soap.SOAPMessageHandlers}
- You can use all supported data types as parameters or return values except \texttt{Holder} classes (user-defined data types that implement the \texttt{javax.xml.rpc.holders.Holder} interface).

The following example shows a simple callback interface file that implements a callback Web Service. The target Web Service, described in “Programming Guidelines for Target Web Service” on page 6-31, explicitly invokes a method in this interface. The jwsc-generated implementation of the callback interface then automatically sends a message back to the client Web Service that originally invoked the target Web Service; the client service is described in “Programming Guidelines for the Callback Client Web Service” on page 6-32. See the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.callback;

import weblogic.jws.CallbackService;

import weblogic.jws.CallbackService;
import javax.jws.Oneway;
import javax.jws.WebMethod;
```
Advanced JWS Programming: Implementing Asynchronous Features

@CallbackService
public interface CallbackInterface {
    @WebMethod
    @Oneway
    public void callbackOperation (String msg);
}

Follow these guidelines when programming the JWS interface file that implements the callback Web Service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the required JWS annotation:
  ```java
  import weblogic.jws.CallbackService;
  ```

- Annotate the interface declaration with the @CallbackService annotation to specify that the JWS file implements a callback service:
  ```java
  @CallbackService
  public interface CallbackInterface {
  ```

- Create a method that the target Web Service explicitly invokes; this is the method that automatically sends a message back to the client service that originally invoked the target Web Service. Because this is a Java interface file, you do not provide an implementation of this method. Rather, the WebLogic Web Services runtime generates an implementation of the method via the jwsc Ant task.
  ```java
  public void callbackOperation (String msg);
  ```

  **Note:** Although the example shows the callback method returning void and annotated with the @Oneway annotation, this is not a requirement.

See Appendix B, "JWS Annotation Reference," for additional information about the JWS annotations discussed in this section.

### Updating the build.xml File for the Client Web Service

When you run the jwsc Ant task against the JWS file that implements the client Web Service, the task implicitly also generates the callback Web Service, as described in this section.

You update a `build.xml` file to generate a client Web Service that invokes the target Web Service by adding `taskdefs` and a `build-clientService` target that looks something like the following example. See the description after the example for details.

```xml
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
```
Use the taskdef Ant task to define the full classname of the jwsc Ant tasks.

Update the jwsc Ant task that compiles the client Web Service to include a <clientgen> child element of the <jws> element so as to generate and compile the JAX-RPC stubs for the deployed TargetService Web Service. The jwsc Ant task automatically packages them in the generated WAR file so that the client Web Service can immediately access the stubs. You do this because the CallbackClientImpl JWS file imports and uses one of the generated classes.

Because the WSDL of the target Web Service includes an additional <service> element that describes the callback Web Service (which the target Web Service invokes), the <clientgen> child element of the jwsc Ant task also generates and compiles the callback Web Service and packages it in the same EAR file as the client Web Service.

Creating Conversational Web Services

A Web Service and the client application that invokes it may communicate multiple times to complete a single task. Also, multiple client applications might communicate with the same Web Service at the same time. Conversations provide a straightforward way to keep track of data between calls and to ensure that the Web Service always responds to the correct client.

Conversations meet two challenges inherent in persisting data across multiple communications:

- Conversations uniquely identify a two-way communication between one client application and one Web Service so that messages are always returned to the correct client. For
example, in a shopping cart application, a conversational Web Service keeps track of which shopping cart belongs to which customer. A conversational Web Service implements this by creating a unique conversation ID each time a new conversation is started with a client application.

- Conversations maintain state between calls to the Web Service; that is, they keep track of the data associated with a particular client application between its calls to the service. Conversations ensure that the data associated with a particular client is saved until it is no longer needed or the operation is complete. For example, in a shopping cart application, a conversational Web Service remembers which items are in the shopping cart while the customer continues shopping. Maintaining state is also needed to handle failure of the computer hosting the Web Service in the middle of a conversation; all state-related data is persisted to disk so that when the computer comes up it can continue the conversation with the client application.

WebLogic Server manages this unique ID and state by creating a conversation context each time a client application initiates a new conversation. The Web Service then uses the context to correlate calls to and from the service and to persist its state-related data.

Conversations between a client application and a Web Service have three distinct phases:

- **Start**—A client application initiates a conversation by invoking the start operation of the conversational Web Service. The Web Service in turn creates a new conversation context and an accompanying unique ID, and starts an internal timer to measure the idle time and the age of the conversation.

- **Continue**—After the client application has started the conversation, it invokes one or more continue operations to continue the conversation. The conversational Web Service uses the ID associated with the invoke to determine which client application it is conversing with, what state to persist, and which idle timer to reset. A typical continue operation would be one that requests more information from the client application, requests status, and so on.

- **Finish**—A client application explicitly invokes the finish operation when it has finished its conversation; the Web Service then marks any data or resources associated with the conversation as deleted.

Conversations typically occur between two WebLogic Web Services: one is marked conversational and defines the start, continue, and finish operations and the other Web Service uses the `@ServiceClient` annotation to specify that it is a client of the conversational Web Service. You can also invoke a conversational Web Service from a stand-alone Java client, although there are restrictions.

As with other WebLogic Web Service features, you use JWS annotations to specify that a Web Service is conversational.
WARNING: The client Web Service that invokes a conversational Web Service is not required to also be conversational. However, if the client is not conversational, there is a danger of multiple instances of this client accessing the same conversational Web Service stub and possibly corrupting the saved conversational state. If you believe this might true in your case, then specify that the client Web Service also be conversational. In this case you cannot use a stand-alone Java client, because there is no way to mark it as conversational using the WebLogic APIs.

Caution: A conversational Web Service on its own does not guarantee message delivery or that the messages are delivered in order, exactly once. If you require this kind of message delivery guarantee, you must also specify that the Web Service be reliable. See “Using Web Service Reliable Messaging” on page 6-1 and “Using the Asynchronous Features Together” on page 6-57.

Creating a Conversational Web Service: Main Steps

The following procedure describes how to create a conversational Web Service, as well as a client Web Service and stand-alone Java client application, both of which initiate and conduct a conversation. The procedure shows how to create the JWS files that implement the two Web Services from scratch. If you want to update existing JWS files, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working build.xml file to which you can add targets for running the jwsc Ant task and deploying the generated conversational Web Service. It is further assumed that you have a similar setup for the WebLogic Server instance that hosts the client Web Service that initiates the conversation. For more information, see Chapter 3, “Common Web Services Use Cases and Examples,” Chapter 4, “Iterative Development of WebLogic Web Services,” and Chapter 5, “Programming the JWS File.”

1. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that implements the conversational Web Service.

   See “Programming Guidelines for the Conversational JWS File” on page 6-40.

2. Update your build.xml file to include a call to the jwsc Ant task to compile the conversational JWS file into a Web Service.

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

3. Run the Ant target to build the conversational Web Service. For example:

   prompt> ant build-mainService
4. Deploy the Web Service as usual.
   See “Deploying and Undeploying WebLogic Web Services” on page 4-15.

5. If the client application is a stand-alone Java client, see “Updating a Stand-Alone Java Client to Invoke a Conversational Web Service” on page 6-48. If the client application is itself a Web Service, follow these steps:
   a. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that implements the client Web Service that initiates and conducts the conversation with the conversational Web Service. It is assumed that the client Web Service is deployed to a different WebLogic Server instance from the one that hosts the conversational Web Service.
      See “Programming Guidelines for the JWS File That Invokes a Conversational Web Service” on page 6-44.
   b. Update the build.xml file that builds the client Web Service.
      See “Updating the build.xml File for a Client of a Conversational Web Service” on page 6-47.
   c. Run the Ant target to build the client Web Service:
      ```
prompt> ant build-clientService
      ```
   d. Deploy the Web Service as usual.
      See “Deploying and Undeploying WebLogic Web Services” on page 4-15.

### Programming Guidelines for the Conversational JWS File

The following example shows a simple JWS file that implements a conversational Web Service; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.conversation;

import java.io.Serializable;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Conversation;
import weblogic.jws.Conversational;
import weblogic.jws.Context;
import weblogic.wsee.jws.JwsContext;
import weblogic.wsee.jws.ServiceHandle;
```

```java
```
import javax.jws.WebService;
import javax.jws.WebMethod;

@Conversational(maxIdleTime="10 minutes",
maxAge="1 day",
runAsStartUser=false,
singlePrincipal=false)
@WebService(name="ConversationalPortType",
serviceName="ConversationalService",
targetNamespace="http://examples.org/"
)
@WLHttpTransport(contextPath="conv",
serviceUri="ConversationalService",
portName="ConversationalServicePort")

/**
 * Conversational Web Service.
 */
public class ConversationalServiceImpl implements Serializable {

@Context
private JwsContext ctx;

public String status = "undefined";

@WebMethod
@Conversation (Conversation.Phase.START)
public String start() {
    ServiceHandle handle = ctx.getService();
    String convID = handle.getConversationID();
    status = "start";
    return "Starting conversation, with ID " + convID + " and status equal to " + status;
}

@WebMethod
@Conversation (Conversation.Phase.CONTINUE)
public String middle(String message) {
    status = "middle";
    return "Middle of conversation; the message is: " + message + " and status is " + status;
}

@WebMethod
@Conversation (Conversation.Phase.FINISH)
public String finish(String message) {

status = "finish";
return "End of conversation; the message is: " + message + " and status is " + status;
}
}

Follow these guidelines when programming the JWS file that implements a conversational Web Service. Code snippets of the guidelines are shown in bold in the preceding example.

- **Conversational Web Services must implement** `java.io.Serializable`, so you must first import the class into your JWS file:
  ```java
  import java.io.Serializable;
  ```

- **Import the conversational JWS annotations**:
  ```java
  import weblogic.jws.Conversation;
  import weblogic.jws.Conversational;
  ```

- **If you want to access runtime information about the conversational Web Service, import the @Context annotation and context APIs**:
  ```java
  import weblogic.jws.Context;
  import weblogic.wsee.jws.JwsContext;
  import weblogic.wsee.jws.ServiceHandle;
  ```
  See “Accessing Runtime Information about a Web Service Using the JwsContext” on page 5-11 for more information about the runtime Web Service context.

- **Use the class-level @Conversational annotation to specify that the Web Service is conversational. Although this annotation is optional (assuming you are specifying the @Conversation method-level annotation), it is a best practice to always use it in your JWS file to clearly specify that your Web Service is conversational.**

Specify any of the following optional attributes: `maxIdleTime` is the maximum amount of time that the Web Service can be idle before WebLogic Server finishes the conversation; `maxAge` is the maximum age of the conversation; `runAsStartUser` indicates whether the continue and finish phases of an existing conversation are run as the user who started the conversation; and `singlePrincipal` indicates whether users other than the one who started a conversation are allowed to execute the continue and finish phases of the conversation.

```java
@Conversational(maxIdleTime="10 minutes",
maxAge="1 day",
runAsStartUser=false,
singlePrincipal=false )
```
If a JWS file includes the `@Conversational` annotation, all operations of the Web Service are conversational. The default phase of an operation, if it does not have an explicit `@Conversation` annotation, is continue. However, because a conversational Web Service is required to include at least one start and one finish operation, you must use the method-level `@Conversation` annotation to specify which methods implement these operations.

See “weblogic.jws.Conversational” on page B-35 for additional information and default values for the attributes.

- Your JWS file must implement `java.io.Serializable`:
  ```java
  public class ConversationalServiceImpl implements Serializable {
  ...
  ```

- To access runtime information about the Web Service, annotate a private class variable, of data type `weblogic.wsee.jws.JwsContext`, with the field-level `@Context JWS` annotation:
  ```java
  @Context
  private JwsContext ctx;
  ```

- Use the `@Conversation` annotation to specify the methods that implement the start, continue, and finish phases of your conversation. A conversation is required to have at least one start and one finish operation; the continue operation is optional. Use the following parameters to the annotation to specify the phase: `Conversation.Phase.START`, `Conversation.Phase.CONTINUE`, or `Conversation.Phase.FINISH`. The following example shows how to specify the start operation:
  ```java
  @WebMethod
  @Conversation (Conversation.Phase.START)
  public String start() {...
  ```

If you mark just one method of the JWS file with the `@Conversation` annotation, then the entire Web Service becomes conversational and each operation is considered part of the conversation; this is true even if you have not used the optional class-level `@Conversational` annotation in your JWS file. Any methods not explicitly annotated with `@Conversation` are, by default, continue operations. This means that, for example, if a client application invokes one of these continue methods without having previously invoked a start operation, the Web Service returns a runtime error.

Finally, if you plan to invoke the conversational Web Service from a stand-alone Java client, the start operation is required to be request-response, or in other words, it cannot be annotated with the `@Oneway JWS` annotation. The operation can return `void`. If you are going to invoke the Web Service only from client applications that run in WebLogic Server, then this requirement does not apply.
See “weblogic.jws.Conversation” on page B-33 for additional information.

- Use the JwsContext instance to get runtime information about the Web Service.

For example, the following code in the start operation gets the ID that WebLogic Server assigns to the new conversation:

```java
ServiceHandle handle = ctx.getService();
String convID = handle.getConversationID();
```

See “Accessing Runtime Information about a Web Service Using the JwsContext” on page 5-11 for detailed information on using the context-related APIs.

### Programming Guidelines for the JWS File That Invokes a Conversational Web Service

The following example shows a simple JWS file for a Web Service that invokes the conversational Web Service described in “Programming Guidelines for the Conversational JWS File” on page 6-40; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.conversation;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.wsee.conversation.ConversationUtils;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.rpc.Stub;
import examples.webservices.conversation.ConversationalPortType;
import java.rmi.RemoteException;

@WebService(name="ConversationalClientPortType",
            serviceName="ConversationalClientService",
            targetNamespace="http://examples.org/")
@WLHttpTransport(contextPath="convClient",
            serviceUri="ConversationalClient",
            portName="ConversationalClientPort")
/**
 * client that has a conversation with the ConversationalService.
 */
```
public class ConversationalClientImpl {

@ServiceClient(
    serviceName="ConversationalService",
    portName="ConversationalServicePort")

private ConversationalPortType port;

@WebMethod
public void runConversation(String message) {
    try {
        // Invoke start operation
        String result = port.start();
        System.out.println("start method executed.");
        System.out.println("The message is: " + result);

        // Invoke continue operation
        result = port.middle(message);
        System.out.println("middle method executed.");
        System.out.println("The message is: " + result);

        // Invoke finish operation
        result = port.finish(message);
        System.out.println("finish method executed.");
        System.out.println("The message is: " + result);
        ConversationUtils.renewStub((Stub)port);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

Follow these guidelines when programming the JWS file that invokes a conversational Web Service; code snippets of the guidelines are shown in bold in the preceding example:

- Import the @ServiceClient JWS annotation:
  import weblogic.jws.ServiceClient;

- Optionally import the WebLogic utility class for further configuring a conversation:
  import weblogic.wsee.conversation.ConversationUtils;
• Import the JAX-RPC stub of the port type of the conversational Web Service you want to invoke. The actual stub itself will be created later by the jwsc Ant task. The stub package is specified by the packageName attribute of the <clientgen> child element of <jws>, and the name of the stub is determined by the WSDL of the invoked Web Service.

```java
import examples.webservices.conversation.ConversationalPortType;
```

• In the body of the JWS file, use the @ServiceClient JWS annotation to specify the WSDL, name, and port of the conversational Web Service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.

```java
@ServiceClient(
    serviceName="ConversationalService",
    portName="ConversationalServicePort")
private ConversationalPortType port;
```

• Using the stub you annotated with the @ServiceClient annotation, invoke the start operation of the conversational Web Service to start the conversation. You can invoke the start method from any location in the JWS file (constructor, method, and so on):

```java
String result = port.start();
```

• Optionally invoke the continue methods to continue the conversation. Be sure you use the same stub instance so that you continue the same conversation you started:

```java
result = port.middle(message);
```

• Once the conversation is completed, invoke the finish operation so that the conversational Web Service can free up the resources it used for the current conversation:

```java
result = port.finish(message);
```

• If you want to reuse the Web Service conversation stub to start a new conversation, you must explicitly renew the stub using the renewStub() method of the weblogic.wsee.conversation.ConversationUtils utility class:

```java
ConversationUtils.renewStub((Stub)port);
```

**WARNING:** The client Web Service that invokes a conversational Web Service is not required to also be conversational. However, if the client is not conversational, there is a danger of multiple instances of this client accessing the same conversational Web Service stub and possibly corrupting the saved conversational state. If you believe this might true in your case, then specify that the client Web Service also be conversational.
ConversationUtils Utility Class

WebLogic Server provides a utility class for use with the conversation feature. Use this class to perform common tasks such as getting and setting the conversation ID and setting configuration options. Some of these tasks are performed in the conversational Web Service, some are performed in the client that invokes the conversational Web Service. See “Programming Guidelines for the JWS File That Invokes a Conversational Web Service” on page 6-44 for an example of using this class.

See weblogic.wsee.conversation.ConversationUtils for details.

Updating the build.xml File for a Client of a Conversational Web Service

You update a build.xml file to generate the JWS file that invokes a conversational Web Service by adding taskdefs and a build-clientService target that looks something like the following example. See the description after the example for details.

```xml
<taskdef name="jwsc" classpath="weblogic.wsee.tools.anttasks.Jwstask" />
<target name="build-clientService">
  <jwsc enableAsyncService="true" srcdir="src" destdir="${clientService-ear-dir}">
    <jws file="examples/webservices/conversation/ConversationalClientImpl.java">
      <clientgen wsdl="http://${wls.hostname}:${wls.port}/conv/ConversationalService?WSDL" packageName="examples.webservices.conversation"/>
    </jws>
  </jwsc>
</target>
```

Use the taskdef Ant task to define the full classname of the jwsc Ant tasks.
Update the `jwsc` Ant task that compiles the client Web Service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `ConversationalService` Web Service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web Service can immediately access the stubs. You do this because the `ConversationalClientImpl` JWS file imports and uses one of the generated classes.

**Updating a Stand-Alone Java Client to Invoke a Conversational Web Service**

The following example shows a simple stand-alone Java client that invokes the conversational Web Service described in “Programming Guidelines for the Conversational JWS File” on page 6-40. See the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.conv_standalone.client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
import weblogic.wsee.jaxrpc.WLStub;

/**
 * stand-alone client that invokes and converses with ConversationlService.
 */
public class Main {
    public static void main(String[] args)
        throws ServiceException, RemoteException{
        ConversationalService service = new ConversationalService_Impl(args[0] + "?WSDL");
        ConversationalPortType port = service.getConversationalServicePort();

        // Set property on stub to specify that client is invoking a Web Service
        // that uses advanced features; this property is automatically set if
        // the client runs in a WebLogic Server instance.
        Stub stub = (Stub)port;
        stub._setProperty(WLStub.COMPLEX, "true");

        // Invoke start operation to begin the conversation
        String result = port.start();
    }
}
Follow these guidelines when programming the stand-alone Java client that invokes a conversational Web Service. Code snippets of the guidelines are shown in bold in the preceding example.

- **Import the** `weblogic.wsee.jaxrpc.WLStub` class:
  ```java
  import weblogic.wsee.jaxrpc.WLStub;
  ```

- **Set the** `WLStub.Complex` property on the JAX-RPC stub of the `ConversationalService` using the `_setProperty` method:
  ```java
  Stub stub = (Stub)port;
  stub._setProperty(WLStub.COMPLEX, "true");
  ```
  This property specifies to the Web Services runtime that the client is going to invoke an advanced Web Service, in this case a conversational one. This property is automatically set when invoking a conversational Web Service from another WebLogic Web Service.

- **Invoke the start operation of the conversational Web Service to start the conversation:**
  ```java
  String result = port.start();
  ```

- **Optionally invoke the continue methods to continue the conversation:**
  ```java
  result = port.middle(message);
  ```

- **Once the conversation is completed, invoke the finish operation so that the conversational Web Service can free up the resources it used for the current conversation:**
  ```java
  result = port.finish(message);
  ```
Client Considerations When Redeploying a Conversational Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated conversational 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 Web Service, its work is considered complete when the existing conversation is explicitly ended by the client or because of a timeout.

For additional information about production redeployment and Web Service clients, see “Client Considerations When Redeploying a Web Service” on page 9-21.

Creating Buffered Web Services

When a buffered operation is invoked by a client, the method operation goes on a JMS queue and WebLogic Server deals with it asynchronously. As with Web Service reliable messaging, if WebLogic Server goes down while the method invocation is still in the queue, it will be dealt with as soon as WebLogic Server is restarted. When a client invokes the buffered Web Service, the client does not wait for a response from the invoke, and the execution of the client can continue.

Creating a Buffered Web Service: Main Steps

The following procedure describes how to create a buffered Web Service and a client Web Service that invokes an operation of the buffered Web Service. The procedure shows how to create the JWS files that implement the two Web Services from scratch. If you want to update existing JWS files, use this procedure as a guide. The procedure also shows how to configure the WebLogic Server instance that hosts the buffered Web Service.

**Note:** Unless you are also using the asynchronous request-response feature, you do not need to invoke a buffered Web Service from another Web Service, you can also invoke it from a stand-alone Java application.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the generated buffered Web Service. It is further assumed that you have a similar setup.
for the WebLogic Server instance that hosts the client Web Service that invokes the buffered Web Service. For more information, see:

- Chapter 3, “Common Web Services Use Cases and Examples”
- Chapter 4, “Iterative Development of WebLogic Web Services”
- Chapter 5, “Programming the JWS File”
- Chapter 9, “Invoking Web Services”

1. Configure the WebLogic Server instance that hosts the buffered Web Service.
   
   See “Configuring the Host WebLogic Server Instance for the Buffered Web Service” on page 6-52.

2. Create a new JWS file, or update an existing one, that will implement the buffered Web Service.
   
   See “Programming Guidelines for the Buffered JWS File” on page 6-53.

3. Update the build.xml file to include a call to the jwsc Ant task to compile the JWS file into a buffered Web Service; for example:
   
   ```xml
   <jwsc
     srcdir="src"
     destdir="${service-ear-dir}"
     >
     <jws
       file="examples/webservices_ASYNC_buffered/AsyncBufferedImpl.java"
     />
   </jwsc>
   ```
   
   See “Running the jwsc WebLogic Web Services Ant Task” on page 4-7 for general information about using the jwsc task.

4. Recompile your destination JWS file by calling the appropriate target and deploying the Web Service to WebLogic Server. For example:
   
   ```bash
   prompt> ant build-mainService deploy-mainService
   ```

5. Create a new JWS file, or update an existing one, that implements the client Web Service that invokes the buffered Web Service.
   
   See “Programming the JWS File That Invokes the Buffered Web Service” on page 6-55.

6. Update the build.xml file that builds the client Web Service.
   
   See “Updating the build.xml File for a Client of the Buffered Web Service” on page 6-56.
7. Recompile your client JWS file by calling the appropriate target, then redeploy the Web Service to the client WebLogic Server. For example:

   prompt> ant build-clientService deploy-clientService

**Configuring the Host WebLogic Server Instance for the Buffered Web Service**

Configuring the WebLogic Server instance on which the buffered Web Service is deployed involves configuring JMS resources, such as JMS servers and modules, that are used internally by the Web Services runtime.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see “Configuring Your Domain For Web Services Features” on page 4-2.

If, however, you prefer to configure the resources yourself, use the following high-level procedure which lists the tasks and then points to the Administration Console Online Help for details on performing the tasks.

1. Invoke the Administration Console for the domain that contains the WebLogic Server instance that hosts the buffered Web Service in your browser.

   See “Invoking the Administration Console” on page 11-4 for instructions on the URL that invokes the Administration Console.

2. Create a JMS Server. You can use an existing one if you do not want to create a new one.

   See Create JMS servers.

3. Create a JMS module that contains a JMS queue. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name.

   If you want the buffered Web Service to use the default Web Services queue, set the JNDI name of the JMS queue to `weblogic.wsee.DefaultQueue`. Otherwise, if you use a different JNDI name, be sure to use the `@BufferQueue` annotation in the JWS file to specify this JNDI name to the reliable Web Service. See “Programming Guidelines for the Buffered JWS File” on page 6-53.

   If you are using the buffered Web Service feature in a cluster, you must still create a local queue rather than a distributed queue. In addition, you must explicitly target this queue to each server in the cluster.
Creating Buffered Web Services

See Create JMS modules and Create queues.

Programming Guidelines for the Buffered JWS File

The following example shows a simple JWS file that implements a buffered Web Service; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```java
package examples.webservices.buffered;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.MessageBuffer;
import weblogic.jws.BufferQueue;

@WebService(name="BufferedPortType",
            serviceName="BufferedService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="buffered",
                serviceUri="BufferedService",
                portName="BufferedPort")
// Annotation to specify a specific JMS queue rather than the default
@BufferQueue(name="my.jms.queue")
/**
 * Simple buffered Web Service.
 */
public class BufferedImpl {
    @WebMethod()
    @MessageBuffer(retryCount=10, retryDelay="10 seconds")
    @Oneway()
    public void sayHelloNoReturn(String message) {
        System.out.println("sayHelloNoReturn: " + message);
    }
}
```
Follow these guidelines when programming the JWS file that implements a buffered Web Service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the JWS annotations used for buffered Web Services:
  ```java
  import javax.jws.Oneway;
  import weblogic.jws.MessageBuffer;
  import weblogic.jws.BufferQueue;
  ```
  See the following bullets for guidelines on which JWS annotations are required.

- Optionally use the class-level `@BufferQueue` JWS annotation to specify the JNDI name of the JMS queue used internally by WebLogic Server when it processes a buffered invoke; for example:
  ```java
  @BufferQueue(name="my.jms.queue")
  ```
  If you do not specify this JWS annotation, then WebLogic Server uses the default Web Services JMS queue (`weblogic.wsee.DefaultQueue`).
  You must create both the default JMS queue and any queues specified with this annotation before you can successfully invoke a buffered operation. See “Configuring the Host WebLogic Server Instance for the Buffered Web Service” on page 6-52 for details.

- Use the `@MessageBuffer` JWS annotation to specify the operations of the Web Service that are buffered. The annotation has two optional attributes:
  - retryCount: The number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web Service implementation (default 3).
  - retryDelay: The amount of time that the server should wait in between retries (default 5 minutes).
  For example:
  ```java
  @MessageBuffer(retryCount=10, retryDelay="10 seconds")
  ```
  You can use this annotation at the class-level to specify that all operations are buffered, or at the method-level to choose which operations are buffered.

- If you plan on invoking the buffered Web Service operation synchronously (or in other words, not using the asynchronous request-response feature), then the implementing method is required to be annotated with the `@Oneway` annotation to specify that the method is one-way. This means that the method cannot return a value, but rather, must explicitly return `void`. For example:
  ```java
  @Oneway()
  public void sayHelloNoReturn(String message) {
```
Conversely, if the method is *not* annotated with the `@Oneway` annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 and “Using the Asynchronous Features Together” on page 6-57.

**Programming the JWS File That Invokes the Buffered Web Service**

You can invoke a buffered Web Service from both a stand-alone Java application (if not using asynchronous request-response) and from another Web Service. Unlike other WebLogic Web Services asynchronous features, however, you do not use the `@ServiceClient` JWS annotation in the client Web Service, but rather, you invoke the service as you would any other. For details, see “Invoking a Web Service from Another Web Service” on page 9-12.

The following sample JWS file shows how to invoke the `sayHelloNoReturn` operation of the `BufferedService` Web Service:

```java
package examples.webservices.buffered;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
import examples.webservices.buffered.BufferedPortType;
import examples.webservices.buffered.BufferedService_Impl;
import examples.webservices.buffered.BufferedService;

@WebService(name="BufferedClientPortType",
            serviceName="BufferedClientService",
            targetNamespace="http://examples.org")
@WLHttpTransport(contextPath="bufferedClient",
                  serviceUri="BufferedClientService",
                  portName="BufferedClientPort")

public class BufferedClientImpl {
    @WebMethod()
    public String callBufferedService(String input, String serviceUrl)
        throws RemoteException {
```

```java
Programming Web Services for WebLogic Server  

Conversely, if the method is *not* annotated with the `@Oneway` annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 and “Using the Asynchronous Features Together” on page 6-57.

**Programming the JWS File That Invokes the Buffered Web Service**

You can invoke a buffered Web Service from both a stand-alone Java application (if not using asynchronous request-response) and from another Web Service. Unlike other WebLogic Web Services asynchronous features, however, you do not use the `@ServiceClient` JWS annotation in the client Web Service, but rather, you invoke the service as you would any other. For details, see “Invoking a Web Service from Another Web Service” on page 9-12.

The following sample JWS file shows how to invoke the `sayHelloNoReturn` operation of the `BufferedService` Web Service:

```java
package examples.webservices.buffered;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
import examples.webservices.buffered.BufferedPortType;
import examples.webservices.buffered.BufferedService_Impl;
import examples.webservices.buffered.BufferedService;

@WebService(name="BufferedClientPortType",
            serviceName="BufferedClientService",
            targetNamespace="http://examples.org")
@WLHttpTransport(contextPath="bufferedClient",
                  serviceUri="BufferedClientService",
                  portName="BufferedClientPort")

public class BufferedClientImpl {
    @WebMethod()
    public String callBufferedService(String input, String serviceUrl)
        throws RemoteException {
```
try {
    BufferedService service = new BufferedService_Impl(serviceUrl + "?WSDL");
    BufferedPortType port = service.getBufferedPort();

    // Invoke the sayHelloNoReturn() operation of BufferedService
    port.sayHelloNoReturn(input);
    return "Invoke went okay!";
} catch (ServiceException se) {
    System.out.println("ServiceExcpetion thrown");
    throw new RuntimeException(se);
}

Updating the build.xml File for a Client of the Buffered Web Service

To update a build.xml file to generate the JWS file that invokes a buffered Web Service operation, add taskdefs and a build-clientService targets that look something like the following example. See the description after the example for details.

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

<target name="build-clientService">

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

        <jws file="examples/webservices/buffered/BufferedClientImpl.java">
            <clientgen
                wsdl="http://${wls.hostname}:${wls.port}/buffered/BufferedService?WSDL"
                packageName="examples.webservices.buffered"/>

        </jws>

    </jwsc>

</target>
```
Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks.

Update the `jwsc` Ant task that compiles the client Web Service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `BufferedService` Web Service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web Service can immediately access the stubs. You do this because the `BufferedClientImpl` JWS file imports and uses one of the generated classes.

### Using the Asynchronous Features Together

The preceding sections describe how to use the WebLogic Web Service asynchronous features (Web Service reliable messaging, conversations, asynchronous request-response, and buffering) on their own. Typically, however, Web Services use the features together; see “Example of a JWS File That Implements a Reliable Conversational Web Service” on page 6-58 and “Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service” on page 6-59 for examples.

When used together, some restrictions described in the individual feature sections do not apply, and sometimes additional restrictions apply.

- **Asynchronous request-response with Web Service reliable messaging or buffering**—The asynchronous response from the reliable Web Service is also reliable. This means that you must also configure a JMS server, module, and queue on the source WebLogic Server instance, in a similar way you configured the destination WebLogic Server instance, to handle the response.

  When you create the JMS queue on the source WebLogic Server instance, you are required to specify a JNDI name of `weblogic.wsee.DefaultQueue`; you can name the queue anything you want. You must also ensure that you specify that this JMS queue is `local`, typically by setting the local JNDI name.

- **Asynchronous request-response with Web Service reliable messaging or buffering**—The reliable or buffered operation cannot be one-way; in other words, you cannot annotate the implementing method with the `@Oneway` annotation.

- **Asynchronous request-response with Web Service reliable messaging**—If you set a property in one of the asynchronous contexts (AsyncPreCallContext or AsyncPostCallContext), then the property must implement `java.io.Serializable`.

- **Asynchronous request-response with buffering**—You must use the `@ServiceClient` JWS annotation in the client Web Service that invokes the buffered Web Service operation.
Conversations with Web Service reliable messaging — If you set the property `WLStub.CONVERSATIONAL_METHOD_BLOCK_TIMEOUT` on the stub of the client Web Service, the property is ignored because the client does not block.

Conversations with Web Service reliable messaging — At least one method of the reliable conversational Web Service must not be marked with the `@Oneway` annotation.

Conversations with asynchronous request-response — Asynchronous responses between a client conversational Web Service and any other Web Service also participate in the conversation. For example, assume `WebServiceA` is conversational, and it invokes `WebServiceB` using asynchronous request-response. Because `WebServiceA` is conversational the asynchronous responses from `WebServiceB` also participate in the same conversation.

Example of a JWS File That Implements a Reliable Conversational Web Service

The following sample JWS file implements a Web Service that is both reliable and conversational:

```java
package examples.webservices.async_mega;
import java.io.Serializable;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Conversation;
import weblogic.jws.Policy;
import javax.jws.WebService;
import javax.jws.WebMethod;
@WebService(name="AsyncMegaPortType",
serviceName="AsyncMegaService",
targetNamespace="http://examples.org/")
@Policy(uri="AsyncReliableConversationPolicy.xml",
attachToWsdl=true)
@WLHttpTransport(contextPath="asyncMega",
serviceUri="AsyncMegaService",
portName="AsyncMegaServicePort")
/**
 * Web Service that is both reliable and conversational.
 */
public class AsyncMegaServiceImpl implements Serializable {
```
Using the Asynchronous Features Together

```java
@WebMethod
@Conversation (Conversation.Phase.START)
public String start() { 
    return "Starting conversation";
}

@WebMethod
@Conversation (Conversation.Phase.CONTINUE)
public String middle(String message) { 
    return "Middle of conversation; the message is: " + message;
}

@WebMethod
@Conversation (Conversation.Phase.FINISH)
public String finish(String message) {
    return "End of conversation; the message is: " + message;
}
```

### Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service

The following JWS file shows how to implement a client Web Service that reliably invokes the various conversational methods of the Web Service described in “Example of a JWS File That Implements a Reliable Conversational Web Service” on page 6-58; the client JWS file uses the asynchronous request-response feature as well.

```java
package examples.webservices.async_mega;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;
import javax.jws.WebService;
import javax.jws.WebMethod;
import weblogic.wsee.async.AsyncPreCallContext;
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPostCallContext;
import examples.webservices.async_mega.AsyncMegaPortType;
import examples.webservices.async_mega.AsyncMegaService;
import examples.webservices.async_mega.AsyncMegaService_Impl;
import java.rmi.RemoteException;
```
public class AsyncMegaClientImpl {

    @ServiceClient(
        serviceName="AsyncMegaService",
        portName="AsyncMegaServicePort")
    private AsyncMegaPortType port;

    @WebMethod
    public void runAsyncReliableConversation(String message) {
        AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
        apc.setProperty("message", message);
        try {
            port.startAsync(apc);
            System.out.println("start method executed.");
            port.middleAsync(apc, message);
            System.out.println("middle method executed.");
            port.finishAsync(apc, message);
            System.out.println("finish method executed.");
        }
        catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @AsyncResponse(target="port", operation="start")
    public void onStartAsyncResponse(AsyncPostCallContext apc, String message) {
        System.out.println("--------------");
        System.out.println("Got message " + message);
        System.out.println("--------------");
    }
}
Using Reliable Messaging or Asynchronous Request Response With a Proxy Server

Client applications that invoke reliable Web Services or use the asynchronous request-response feature might not invoke the operation directly, but rather, use a proxy server. Reasons for using a proxy include the presence of a firewall or the deployment of the invoked Web Service to a cluster.
In this case, the WebLogic Server instance that hosts the invoked Web Service must be configured with the address and port of the proxy server. If your Web Service is deployed to a cluster, you must configure every server in the cluster.

For each server instance:

1. Create a network channel for the protocol you use to invoke the Web Service. You must name the network channel `weblogic-wsee-proxy-channel-XXX`, where `XXX` refers to the protocol. For example, to create a network channel for HTTPS, call it `weblogic-wsee-proxy-channel-https`.

   See Configure Custom Network Channels for general information about creating a network channel.

2. Configure the network channel, updating the **External Listen Address** and **External Listen Port** fields with the address and port of the proxy server, respectively.
The following sections provide information about the following advanced JWS programming topics:

- “Using JMS Transport as the Connection Protocol” on page 7-1
- “Creating and Using SOAP Message Handlers” on page 7-12

**Using JMS Transport as the Connection Protocol**

Typically, client applications use HTTP/S as the connection protocol when invoking a WebLogic Web Service. You can, however, configure a WebLogic Web Service so that client applications use JMS as the transport instead. You configure transports using either JWS annotations or child elements of the `jwsc` Ant task, as described in later sections.

When a WebLogic Web Service is configured to use JMS as the connection transport, the endpoint address specified for the corresponding port in the generated WSDL of the Web Service uses `jms://` in its URL rather than `http://`. An example of a JMS endpoint address is as follows:

```
jms://myHost:7001/transports/JMSTransport?URI=JMSTransportQueue
```

The `URI=JMSTransportQueue` section of the URL specifies the JMS queue that has been configured for the JMS transport feature. Although you cannot invoke the Web Service using HTTP, you can view its WSDL using HTTP, which is how the `clientgen` is still able to generate JAX-RPC stubs for the Web Service.
For each transport that you specify, WebLogic Server generates an additional port in the WSDL. For this reason, if you want to give client applications a choice of transports they can use when they invoke the Web Service (JMS, HTTP, or HTTPS), you should explicitly add the transports using the appropriate JWS annotations or child elements of `jwsc`.

**Caution:** Using JMS transport is an added-value WebLogic feature; non-WebLogic client applications, such as a .NET client, may not be able to invoke the Web Service using the JMS port.

### Using JMS Transport Starting From Java: Main Steps

To use JMS transport when starting from Java, you must perform at least one of the following tasks:

- Add the `@WLJmsTransport` annotation to your JWS file.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task. This setting overrides the transports defined in the JWS file.

**Note:** Because you might not know at the time that you are coding the JWS file which transport best suits your needs, it is often better to specify the transport at build-time using the `<WLJmsTransport>` child element.

The following procedure describes the complete set of steps required so that your Web Service can be invoked using the JMS transport when starting from Java.

**Note:** It is assumed that you have already created a basic JWS file that implements a Web Service and that you want to configure the Web Service to be invoked using JMS. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes targets for running the `jwsc` Ant task and deploying the service. For more information, see Chapter 4, “Iterative Development of WebLogic Web Services,” and Chapter 5, “Programming the JWS File.”

1. **Configure the WebLogic Server domain for the required JMS components.**
   
   You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see “Configuring Your Domain For Web Services Features” on page 4-2.

   If, however, you prefer to configure the resources yourself, follow these steps:

   a. Invoke the Administration Console in your browser, as described in “Invoking the Administration Console” on page 11-4.
b. Using the Administration Console, create and configure the following JMS components, if they do not already exist:

- JMS Server. See Create JMS servers.
- JMS Module, targeted to the preceding JMS server. See Create JMS system modules.
- JMS Queue, contained within the preceding JMS module. You can either specify the JNDI name of the JMS queue that WebLogic Web Services listen to by default (weblogic.wsee.DefaultQueue) or specify a different name. If you specify a different JNDI name, you later pass this name to the Web Service itself. When you configure the queue, be sure you specify that this JMS queue is local, typically by setting the local JNDI name. See Create queues in a system module.

Except for the JNDI name of the JMS queue, you can name the other components anything you want.

2. Add the @WLJmsTransport annotation to your JWS file.

   This step is optional. If you do not add the @WLJmsTransport annotation to your JWS file, then you must add a <WLJmsTransport> child element to the jwsc Ant task, as described in Step 3. See “Using the @WLJmsTransport JWS Annotation” on page 7-6.

3. Add a <WLJmsTransport> child element to the jwsc Ant task.

   Use the <WLJmsTransport> child element to override the transports defined in the JWS file. This step is required if you did not add the @WLJmsTransport annotation to your JWS file in Step 2. Otherwise, this step is optional.

   See “Using the <WLJmsTransport> Child Element of the jwsc Ant Task” on page 7-7 for details.

4. Rebuild your Web Service by re-running the target in the build.xml Ant file that calls the jwsc task.

   For example, if the target that calls the jwsc Ant task is called build-service, then you would run:

   prompt> ant build-service

5. Redeploy your Web Service to WebLogic Server.

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

See “Invoking a WebLogic Web Service Using JMS Transport” on page 7-9 for information about updating your client application to invoke the Web Service using JMS transport.
Using JMS Transport Starting From WSDL: Main Steps

To use JMS transport when starting from WSDL, you must perform at least one of the following tasks:

- Update the WSDL to use JMS transport before running the `wsdlc` Ant task.
- Update the stubbed-out JWS implementation file generated by the `wsdlc` Ant task to add the `@WLJmsTransport` annotation.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task used to build the JWS implementation file. This setting overrides the transports defined in the JWS file.

**Note:** Because you might not know at the time that you are coding the JWS file which transport best suits your needs, it is often better to specify the transport at build-time using the `<WLJmsTransport>` child element.

The following procedure describes the complete set of steps required so that your Web Service can be invoked using the JMS transport when starting from WSDL.

**Note:** It is assumed in this procedure that you have an existing WSDL file.

1. Configure the WebLogic Server domain for the required JMS components.

   You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see “Configuring Your Domain For Web Services Features” on page 4-2.

   If, however, you prefer to configure the resources yourself, follow these steps:

   a. Invoke the Administration Console in your browser, as described in “Invoking the Administration Console” on page 11-4.

   b. Using the Administration Console, create and configure the following JMS components, if they do not already exist:

      - JMS Server. See Create JMS servers.
      - JMS Module, targeted to the preceding JMS server. See Create JMS system modules.
      - JMS Queue, contained within the preceding JMS module. You can either specify the JNDI name of the JMS queue that WebLogic Web Services listen to by default (`weblogic.wsee.DefaultQueue`) or specify a different name. If you specify a different JNDI name, you later pass this name to the Web Service itself. When you
configure the queue, be sure you specify that this JMS queue is local, typically by setting the local JNDI name. See Create queues in a system module.

Except for the JNDI name of the JMS queue, you can name the other components anything you want.

2. Update the WSDL to use JMS transport.

This step is optional. If you do not update the WSDL to use JMS transport, then you must do at least one of the following:

- Edit the stubbed out JWS file to add the @WLJmsTransport annotation to your JWS file, as described in Step 4.
- Add a <WLJmsTransport> child element to the jwsc Ant task, as described in Step 5.

See “Updating the WSDL to Use JMS Transport” on page 7-8.

3. Run the wsdlc Ant task against the WSDL file.

For example, if the target that calls the wsdlc Ant task is called generate-from-wsdl, then you would run:

prompt> ant generate-from-wsdl

See “Running the wsdlc WebLogic Web Services Ant Task” on page 4-11.

4. Update the stubbed-out JWS file.

The wsdlc Ant task generates a stubbed-out JWS file. You need to add your business code to the Web Service so it behaves as you want. See “Updating the Stubbed-Out JWS Implementation Class File Generated By wsdlc” on page 4-13.

If you updated the WSDL to use the JMS transport in Step 2, the JWS file includes the @WLJmsTransport annotation that defines the JMS transport. If the @WLJmsTransport annotation is not included in the JWS file, you must do at least one of the following:

- Edit the JWS file to add the @WLJmsTransport annotation to your JWS file, as described in “Using the @WLJmsTransport JWS Annotation” on page 7-6.
- Add a <WLJmsTransport> child element to the jwsc Ant task, as described in Step 5.

5. Add a <WLJmsTransport> child element to the jwsc Ant task.

Use the <WLJmsTransport> child element to override the transports defined in the JWS file. This step is required if the JWS file does not include the @WLJmsTransport annotation, as noted in Step 4. Otherwise, this step is optional. See “Using the <WLJmsTransport> Child Element of the jwsc Ant Task” on page 7-7.
6. Run the \texttt{jwsc} Ant task against the JWS file to build the Web Service. Specify the artifacts generated by the \texttt{wsdlc} Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web Service. See “Running the \texttt{jwsc} WebLogic Web Services Ant Task” on page 4-7.


See “Invoking a WebLogic Web Service Using JMS Transport” on page 7-9 for information about updating your client application to invoke the Web Service using JMS transport.

\textbf{Using the @WLJmsTransport JWS Annotation}

If you know at the time that you program the JWS file that you want client applications to use JMS transport (instead of HTTP/S) to invoke the Web Service, you can use the \texttt{@WLJmsTransport} to specify the details of the invoke. Later, at build-time, you can override the one in the JWS file and add additional JMS transport specifications, by specifying the \texttt{<WLJmsTransport>} child element of the \texttt{jwsc} Ant task, as described in “Using the \texttt{<WLJmsTransport>} Child Element of the \texttt{jwsc} Ant Task” on page 7-7.

Follow these guidelines when using the \texttt{@WLJmsTransport} annotation:

\begin{itemize}
\item You can include only one \texttt{@WLJmsTransport} annotation in a JWS file.
\item Use the \texttt{queue} attribute to specify the JNDI name of the JMS queue you configured earlier in the section. If you want to use the default Web Services queue (\texttt{weblogic.wsee.DefaultQueue}) then you do not have to specify the \texttt{queue} attribute.
\end{itemize}

The following example shows a simple JWS file that uses the \texttt{@WLJmsTransport} annotation, with the relevant code in bold:

```java
package examples.webservices.jmstransport;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import weblogic.jws.WLJmsTransport;
@WebService(name="JMSTransportPortType",
            serviceName="JMSTransportService",
            targetNamespace="http://example.org")
```
Using JMS Transport as the Connection Protocol

@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
// "transports/JMSTransport"

@WLJmsTransport(contextPath="transports", serviceUri="JMSTransport",
    queue="JMSTransportQueue", portName="JMSTransportServicePort")

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

public class JMSTransportImpl {

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

Using the <WLJmsTransport> Child Element of the jwsc Ant Task

You can also specify the JMS transport at build-time by using the <WLJmsTransport> child
element of the <jws> element of the jwsc Ant task. Reasons for specifying the transport at
build-time include:

- You need to override the attribute values specified in the JWS file.
- The JWS file specifies a different transport, and at build-time you decide that JMS should
  be the transport.
- The JWS file does not include a @WLXXXTransport annotation; thus by default the HTTP
  transport is used, but at build-time you decide you want to clients to use the JMS transport
to invoke the Web Service.
If you specify a transport to the jwsc Ant task, it takes precedence over any transport annotation in the JWS file.

The following example shows how to specify a transport to the jwsc Ant task:

```xml
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="${ear-dir}"
     file="examples/webservices/jmstransport/JMSTransportImpl.java">
    <WLJmsTransport
       contextPath="transports"
       serviceUri="JMSTransport"
       portName="JMSTransportServicePort"
       queue="JMSTransportQueue"/>
  </jwsc>
</target>
```

The preceding example shows how to specify the same values for the URL and JMS queue as were specified in the JWS file shown in “Using the @WLJmsTransport JWS Annotation” on page 7-6.

For more information about using the jwsc Ant task, see “jwsc” on page A-17.

**Updating the WSDL to Use JMS Transport**

To update the WSDL to use JMS transport, you need to add `<wsdl:binding>` and `<wsdl:service>` definitions that define JMS transport information. You can add the definitions in one of the following ways:

- Edit the existing HTTP `<wsdl:binding>` and `<wsdl:service>` definitions.
- To specify multiple transport options in the WSDL, copy the existing HTTP `<wsdl:binding>` and `<wsdl:service>` definitions and edit them to use JMS transport.
In either case, you must modify the `<wsdl:binding>` and `<wsdl:service>` definitions to use JMS transport as follows:

- Set the `transport` attribute of the `<soapwsdl:binding>` child element of the `<wsdl:binding>` element to `http://www.openuri.org/2002/04/soap/jms`. For example:

  ```xml
  <binding name="JmsTransportServiceSoapBindingjms" type="tns:JmsTransportPortType">
  </binding>
  ```

- Specify a JMS-style endpoint URL for the `location` attribute of the `<soapwsdl:address>` child element of the `<wsdl:service>`. For example:

  ```xml
  <s0:service name="JmsTransportService">
    <s0:port binding="s1:JmsTransportServiceSoapBindingjms" name="JmsTransportServicePort">
      <s2:address location="jms://localhost:7001/transports/JmsTransport?URI=JMSTransportQueue"/>
    </s0:port>
  </s0:service>
  ```

### Invoking a WebLogic Web Service Using JMS Transport

You write a client application to invoke a Web Service using JMS transport in the same way as you write one using the HTTP transport; the only difference is that you must ensure that the JMS queue (specified by the `@WLJmsTransport` annotation or `<WLJmsTransport>` child element of the `jwsc` Ant task) and other JMS objects have already been created. See “Using JMS Transport Starting From Java: Main Steps” on page 7-2 or “Using JMS Transport Starting From WSDL: Main Steps” on page 7-4 for more information.

Although you cannot `invoke` a JMS-transport-configured Web Service using HTTP, you can view its WSDL using HTTP, which is how the `clientgen` Ant task is still able to create the JAX-RPC stubs for the Web Service. For example, the URL for the WSDL of the deployed Web Service shown in this section would be:

http://host:port/transports/JMSTransport?WSDL

However, because the endpoint address in the WSDL of the deployed Web Service uses `jms://` instead of `http://`, and the address includes the qualifier `?URI=JMS_QUEUE`, the `clientgen` Ant task automatically creates the stubs needed to use the JMS transport when invoking the Web Service, and your client application need not do anything different than normal. An example of a JMS endpoint address is as follows:
jms://host:port/transports/JMSTransport?URI=JMSTransportQueue

**WARNING:** If you have specified that the Web Service you invoke using JMS transport also runs within the context of a transaction (in other words, the JWS file includes the `@weblogic.jws.Transactional` annotation), you must use asynchronous request-response when invoking the service. If you do not, a deadlock will occur and the invocation will fail.

For general information about invoking a Web Service, see Chapter 9, “Invoking Web Services.”

**Overriding the Default Service Address URL**

When you write a client application that uses the `clientgen`-generated JAX-RPC stubs to invoke a Web Service, the default service address URL of the Web Service is the one specified in the `<address>` element of the WSDL file argument of the `Service` constructor.

Sometimes, however, you might need to override this address, in particular when invoking a WebLogic Web Service that is deployed to a cluster and you want to specify the cluster address or a list of addresses of the managed servers in the cluster. You might also want to use the t3 protocol to invoke the Web Service. To override this service endpoint URL when using JMS transport, use the `weblogic.wsee.jaxrpc.WLStub.JMS_TRANSPORT_JNDI_URL` stub property as shown in the following example:

```java
package examples.webservices.jmstransport.client;
import weblogic.wsee.jaxrpc.WLStub;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
/**
 * This is a simple standalone client application that invokes the
 * the `<code>`sayHello</code>` operation of the JMSTransport Web service.
 * *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 * */
public class Main {
    public static void main(String[] args)
            throws ServiceException, RemoteException{
```
Using JMS Transport as the Connection Protocol

JMSTransportService service = new JMSTransportService_Impl(args[0] + "?WSDL");
JMSTransportPortType port = service.getJMSTransportServicePort();

Stub stub = (Stub) port;
stub._setProperty(WLStub.JMS_TRANSPORT_JNDI_URL, "t3://shackell01.amer.bea.com:7001");
try {
    String result = null;
    result = port.sayHello("Hi there! ");
    System.out.println("Got JMS result: " + result);
} catch (RemoteException e) {
    throw e;
}

See WLStub reference documentation for additional stub properties.

Using JMS BytesMessage Rather Than the Default TextMessage

When you use JMS transport, the Web Services runtime uses, by default, the javax.jms.TextMessage object to send the message. This is usually adequate for most client applications, but sometimes you might need to send binary data rather than ordinary text; in this case you must request that the Web Services runtime use javax.jms.BytesMessage instead.

To do this, use the WLStub.JMS_TRANSPORT_MESSAGE_TYPE stub property in your client application and set it to the value WLStub.JMS_BYTESMESSAGE, as shown in the following example:

    stub._setProperty(WLStub.JMS_TRANSPORT_MESSAGE_TYPE,
                     WLStub.JMS_BYTESMESSAGE);

The Web Services runtime sends back the response using the same message data type as the request.

See “Overriding the Default Service Address URL” on page 7-10 for a full example of a client application in which you can set this property. See WLStub reference documentation for additional stub properties.
Disabling HTTP Access to the WSDL File

As described in “Invoking a WebLogic Web Service Using JMS Transport” on page 7-9, the WSDL of the deployed Web Service is, by default, still accessible using HTTP. If you want to disable access to the WSDL file, in particular if your Web Service can be accessed outside of a firewall, then you can do one of the following:

- Use the `weblogic.jws.WSDL` annotation in your JWS file to programmatically disable access. For details, see “weblogic.jws.WSDL” on page B-62.

- Use the Administration Console to disable access to the WSDL file after the Web Service has been deployed. In this case, the configuration information will be stored in the deployment plan rather than through the annotation.

To use the Administration Console to perform this task, go to the Configuration -> General page of the deployed Web Service and uncheck the View Dynamic WSDL Enabled checkbox. After saving the configuration to the deployment plan, you must redeploy (update) the Web Service, or Enterprise Application which contains it, for the change to take effect.

Creating and Using SOAP Message Handlers

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

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

A simple example of using handlers is to access information in the header part of the SOAP message. You can use the SOAP header to store Web Service specific information and then use handlers to manipulate it.

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

The following table lists the standard JWS annotations that you can use in your JWS file to specify that a Web Service has a handler chain configured; later sections discuss how to use the
Creating and Using SOAP Message Handlers

annotations in more detail. For additional information, see the Web Services MetaData for the Java Platform (JSR-181) specification at http://www.jcp.org/en/jsr/detail?id=181.

Table 7-1  JWS Annotations Used To Configure SOAP Message Handler Chains

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

The following table describes the main classes and interfaces of the javax.xml.rpc.handler API, some of which you use when creating the handler itself. These APIs are discussed in detail
in a later section. For additional information about these APIs, see the JAX-RPC 1.1 specification at http://java.sun.com/xml/jaxrpc/index.jsp.

Table 7-2  JAX-RPC Handler Interfaces and Classes

<table>
<thead>
<tr>
<th>javax.xml.rpc.handler Classes and Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler</td>
<td>Main interface that is implemented when creating a handler. Contains methods to handle the SOAP request, response, and faults.</td>
</tr>
<tr>
<td>GenericHandler</td>
<td>Abstract class that implements the Handler interface. User should extend this class when creating a handler, rather than implement Handler directly. The GenericHandler class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the lifecycle methods init and destroy and also different handle methods. A handler developer should only override methods that it needs to specialize as part of the derived handler implementation class.</td>
</tr>
<tr>
<td>HandlerChain</td>
<td>Interface that represents a list of handlers. An implementation class for the HandlerChain interface abstracts the policy and mechanism for the invocation of the registered handlers.</td>
</tr>
<tr>
<td>HandlerRegistry</td>
<td>Interface that provides support for the programmatic configuration of handlers in a HandlerRegistry.</td>
</tr>
<tr>
<td>HandlerInfo</td>
<td>Class that contains information about the handler in a handler chain. A HandlerInfo instance is passed in the Handler.init method to initialize a Handler instance.</td>
</tr>
<tr>
<td>MessageContext</td>
<td>Abstracts the message context processed by the handler. The MessageContext properties allow the handlers in a handler chain to share processing state.</td>
</tr>
<tr>
<td>soap.SOAPMessageContext</td>
<td>Sub-interface of the MessageContext interface used to get at or update the SOAP message.</td>
</tr>
<tr>
<td>javax.xml.soap.SOAPMessage</td>
<td>Object that contains the actual request or response SOAP message, including its header, body, and attachment.</td>
</tr>
</tbody>
</table>
Adding SOAP Message Handlers to a Web Service: Main Steps

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

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

1. Design the handlers and handler chains.
   See “Designing the SOAP Message Handlers and Handler Chains” on page 7-15.

2. For each handler in the handler chain, create a Java class that extends the javax.xml.rpc.handler.GenericHandler abstract class.
   See “Creating the GenericHandler Class” on page 7-18.

3. Update your JWS file, adding annotations to configure the SOAP message handlers.
   See “Configuring Handlers in the JWS File” on page 7-26.

4. If you are using the @HandlerChain standard annotation in your JWS file, create the handler chain configuration file.
   See “Creating the Handler Chain Configuration File” on page 7-31.

5. Compile all handler classes in the handler chain and rebuild your Web Service.
   See “Compiling and Rebuilding the Web Service” on page 7-32.

For information about creating client-side SOAP message handlers and handler chains, see “Creating and Using Client-Side SOAP Message Handlers” on page 9-22.

Designing the SOAP Message Handlers and Handler Chains

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

- The number of handlers needed to perform all the work
- The sequence of execution
Each handler in a handler chain has one method for handling the request SOAP message and another method for handling the response SOAP message. An ordered group of handlers is referred to as a handler chain. You specify that a Web Service has a handler chain attached to it with one of two JWS annotations: @HandlerChain or @SOAPMessageHandler. When to use which is discussed in a later section.

When invoking a Web Service, WebLogic Server executes handlers as follows:

1. The handleRequest() methods of the handlers in the handler chain are all executed in the order specified by the JWS annotation. Any of these handleRequest() methods might change the SOAP message request.

2. When the handleRequest() method of the last handler in the handler chain executes, WebLogic Server invokes the back-end component that implements the Web Service, passing it the final SOAP message request.

3. When the back-end component has finished executing, the handleResponse() methods of the handlers in the handler chain are executed in the reverse order specified in by the JWS annotation. Any of these handleResponse() methods might change the SOAP message response.

4. When the handleResponse() method of the first handler in the handler chain executes, WebLogic Server returns the final SOAP message response to the client application that invoked the Web Service.

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

```xml
    xmlns:soap1="http://HandlerInfo.org/Server1"
    xmlns:soap2="http://HandlerInfo.org/Server2"
    xmlns="http://java.sun.com/xml/ns/j2ee">
    <jwshc:handler-chain>
        <jwshc:handler-chain-name>SimpleChain</jwshc:handler-chain-name>
        <jwshc:handler>
            <handler-name>handlerOne</handler-name>
            <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler1</handler-class>
        </jwshc:handler>
    </jwshc:handler-chain>
</jwshc:handler-config>
```
Creating and Using SOAP Message Handlers

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

**Figure 7-1  Order of Execution of Handler Methods**

Each SOAP message handler has a separate method to process the request and response SOAP message because the same type of processing typically must happen for the inbound and outbound message. For example, you might design an Encryption handler whose `handleRequest()` method decrypts secure data in the SOAP request and `handleResponse()` method encrypts the SOAP response.

You can, however, design a handler that process only the SOAP request and does no equivalent processing of the response.

You can also choose not to invoke the next handler in the handler chain and send an immediate response to the client application at any point.
Creating the GenericHandler Class

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

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

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

- init()
  See “Implementing the Handler.init() Method” on page 7-20.
- destroy()
  See “Implementing the Handler.destroy() Method” on page 7-21.
- getHeaders()
  See “Implementing the Handler.getHeaders() Method” on page 7-21.
- handleRequest()
  See “Implementing the Handler.handleRequest() Method” on page 7-21.
- handleResponse()
  See “Implementing the Handler.handleResponse() Method” on page 7-23.
- handleFault()
  See “Implementing the Handler.handleFault() Method” on page 7-24.

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the javax.xml.soap.SOAPMessage abstract class, which is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification. For details, see “Directly Manipulating the SOAP Request and Response Message Using SAAJ” on page 7-25.

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

```java
package examples.webservices.soap_handlers.global_handler;
```
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;
import weblogic.logging.NonCatalogLogger;

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

class ServerHandler1 extends GenericHandler {

    private NonCatalogLogger log;
    private HandlerInfo handlerInfo;

    /**
     * Initializes the instance of the handler. Creates a nonCatalogLogger to
     * log messages to.
     */
    public void init(HandlerInfo hi) {
        log = new NonCatalogLogger("WebService-LogHandler");
        handlerInfo = hi;
    }

    /**
     * Specifies that the SOAP request message be logged to a log file before the
     * message is sent to the Java class that implements the Web Service.
     */
    public boolean handleRequest(MessageContext context) {
        SOAPMessageContext messageContext = (SOAPMessageContext) context;
        System.out.println("** Request: " + messageContext.getMessage().toString());
        log.info(messageContext.getMessage().toString());
        return true;
    }
}
Advanced JWS Programming: JMS Transport and SOAP Message Handlers

/**
 * Specifies that the SOAP response message be logged to a log file before the 
 * message is sent back to the client application that invoked the Web 
 * service.
 */
public boolean handleResponse(MessageContext context) {
    SOAPMessageContext messageContext = (SOAPMessageContext) context;
    System.out.println("** Response: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;
}

/**
 * Specifies that a message be logged to the log file if a SOAP fault is 
 * thrown by the Handler instance.
 */
public boolean handleFault(MessageContext context) {
    SOAPMessageContext messageContext = (SOAPMessageContext) context;
    System.out.println("** Fault: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;
}

public QName[] getHeaders() {
    return handlerInfo.getHeaders();
}
}

Implementing the Handler.init() Method

The Handler.init() method is called to create an instance of a Handler object and to enable 
the instance to initialize itself. Its signature is:

    public void init(HandlerInfo config) throws JAXRPCException {}

The HandlerInfo object contains information about the SOAP message handler, in particular 
the initialization parameters. Use the HandlerInfo.getHandlerConfig() method to get the 
parameters; the method returns a java.util.Map object that contains name-value pairs.
Implement the `init()` method if you need to process the initialization parameters or if you have other initialization tasks to perform.

Sample uses of initialization parameters are to turn debugging on or off, specify the name of a log file to which to write messages or errors, and so on.

**Implementing the `Handler.destroy()` Method**

The `Handler.destroy()` method is called to destroy an instance of a `Handler` object. Its signature is:

```
public void destroy() throws JAXRPCException {}
```

Implement the `destroy()` method to release any resources acquired throughout the handler’s lifecycle.

**Implementing the `Handler.getHeaders()` Method**

The `Handler.getHeaders()` method gets the header blocks that can be processed by this `Handler` instance. Its signature is:

```
public QName[] getHeaders() {}
```

**Implementing the `Handler.handleRequest()` Method**

The `Handler.handleRequest()` method is called to intercept a SOAP message request before it is processed by the back-end component. Its signature is:

```
public boolean handleRequest(MessageContext mc)
  throws JAXRPCException, SOAPFaultException {}
```

Implement this method to perform such tasks as decrypting data in the SOAP message before it is processed by the back-end component, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message request. The SOAP message request itself is stored in a `javax.xml.soap.SOAPMessage` object. For detailed information on this object, see “Directly Manipulating the SOAP Request and Response Message Using SAAJ” on page 7-25.

The `SOAPMessageContext` class defines two methods for processing the SOAP request:
SOAPMessageContext.getMessage() returns a javax.xml.soap.SOAPMessage object that contains the SOAP message request.

SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage) updates the SOAP message request after you have made changes to it.

After you code all the processing of the SOAP request, code one of the following scenarios:

- **Invoke the next handler on the handler request chain by returning true.**
  The next handler on the request chain is specified as either the next <handler> subelement of the <handler-chain> element in the configuration file specified by the @HandlerChain annotation, or the next @SOAPMessageHandler in the array specified by the @SOAPMessageHandlers annotation. If there are no more handlers in the chain, the method either invokes the back-end component, passing it the final SOAP message request, or invokes the handleResponse() method of the last handler, depending on how you have configured your Web Service.

- **Block processing of the handler request chain by returning false.**
  Blocking the handler request chain processing implies that the back-end component does not get executed for this invoke of the Web Service. You might want to do this if you have cached the results of certain invokes of the Web Service, and the current invoke is on the list.

  Although the handler request chain does not continue processing, WebLogic Server does invoke the handler response chain, starting at the current handler. For example, assume that a handler chain consists of two handlers: handlerA and handlerB, where the handleRequest() method of handlerA is invoked before that of handlerB. If processing is blocked in handlerA (and thus the handleRequest() method of handlerB is not invoked), the handler response chain starts at handlerA and the handleRequest() method of handlerB is not invoked either.

- **Throw the javax.xml.rpc.soap.SOAPFaultException to indicate a SOAP fault.**
  If the handleRequest() method throws a SOAPFaultException, WebLogic Server catches the exception, terminates further processing of the handler request chain, and invokes the handleFault() method of this handler.

- **Throw a JAXRPCException for any handler-specific runtime errors.**
  If the handleRequest() method throws a JAXRPCException, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server log file, and invokes the handleFault() method of this handler.
Implementing the Handler.handleResponse() Method

The Handler.handleResponse() method is called to intercept a SOAP message response after it has been processed by the back-end component, but before it is sent back to the client application that invoked the Web Service. Its signature is:

```java
public boolean handleResponse(MessageContext mc) throws JAXRPCException {}
```

Implement this method to perform such tasks as encrypting data in the SOAP message before it is sent back to the client application, to further process returned values, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get or update the contents of the SOAP message response. The SOAP message response itself is stored in a `javax.xml.soap.SOAPMessage` object. See “Directly Manipulating the SOAP Request and Response Message Using SAAJ” on page 7-25.

The `SOAPMessageContext` class defines two methods for processing the SOAP response:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message response.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message response after you have made changes to it.

After you code all the processing of the SOAP response, code one of the following scenarios:

- **Invoke the next handler on the handler response chain by returning `true`**.

  The next response on the handler chain is specified as either the preceding `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the preceding `@SOAPMessageHandler` in the array specified by the `@SOAPMessageHandlers` annotation. (Remember that responses on the handler chain execute in the reverse order that they are specified in the JWS file. See “Designing the SOAP Message Handlers and Handler Chains” on page 7-15 for more information.)

  If there are no more handlers in the chain, the method sends the final SOAP message response to the client application that invoked the Web Service.

- **Block processing of the handler response chain by returning `false`**.
Blocking the handler response chain processing implies that the remaining handlers on the response chain do not get executed for this invoke of the Web Service and the current SOAP message is sent back to the client application.

- Throw a JAXRPCException for any handler specific runtime errors.

If the handleRequest() method throws a JAXRPCException, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, and invokes the handleFault() method of this handler.

### Implementing the Handler.handleFault() Method

The Handler.handleFault() method processes the SOAP faults based on the SOAP message processing model. Its signature is:

```java
public boolean handleFault(MessageContext mc) throws JAXRPCException {}
```

Implement this method to handle processing of any SOAP faults generated by the handleResponse() and handleRequest() methods, as well as faults generated by the back-end component.

The MessageContext object abstracts the message context processed by the SOAP message handler. The MessageContext properties allow the handlers in a handler chain to share processing state.

Use the SOAPMessageContext sub-interface of MessageContext to get at or update the contents of the SOAP message. The SOAP message itself is stored in a javax.xml.soap.SOAPMessage object. See “Directly Manipulating the SOAP Request and Response Message Using SAAJ” on page 7-25.

The SOAPMessageContext class defines the following two methods for processing the SOAP message:

- SOAPMessageContext.getMessage(): returns a javax.xml.soap.SOAPMessage object that contains the SOAP message.
- SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage): updates the SOAP message after you have made changes to it.

After you code all the processing of the SOAP fault, do one of the following:

- Invoke the handleFault() method on the next handler in the handler chain by returning true.
- Block processing of the handler fault chain by returning false.
Creating and Using SOAP Message Handlers

Directly Manipulating the SOAP Request and Response Message Using SAAJ

The `javax.xml.soap.SOAPMessage` abstract class is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification. You use the class to manipulate request and response SOAP messages when creating SOAP message handlers. This section describes the basic structure of a `SOAPMessage` object and some of the methods you can use to view and update a SOAP message.

A `SOAPMessage` object consists of a `SOAPPart` object (which contains the actual SOAP XML document) and zero or more attachments.

Refer to the SAAJ Javadocs for the full description of the `SOAPMessage` class. For more information on SAAJ, go to http://java.sun.com/xml/saaj/index.html.

The SOAPPart Object

The `SOAPPart` object contains the XML SOAP document inside of a `SOAPEnvelope` object. You use this object to get the actual SOAP headers and body.

The following sample Java code shows how to retrieve the SOAP message from a `MessageContext` object, provided by the `Handler` class, and get at its parts:

```java
SOAPMessage soapMessage = messageContext.getMessage();
SOAPPart soapPart = soapMessage.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPBody soapBody = soapEnvelope.getBody();
SOAPHeader soapHeader = soapEnvelope.getHeader();
```

The AttachmentPart Object

The `javax.xml.soap.AttachmentPart` object contains the optional attachments to the SOAP message. Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file.

**Caution:** If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see “Manipulating Image Attachments in a SOAP Message Handler” on page 7-26 for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments:

- `countAttachments()`: returns the number of attachments in this SOAP message.
- `getAttachments()`: retrieves all the attachments (as `AttachmentPart` objects) into an `Iterator` object.
Advanced JWS Programming: JMS Transport and SOAP Message Handlers

- `createAttachmentPart()`: create an `AttachmentPart` object from another type of Object.
- `addAttachmentPart()`: adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.

### Manipulating Image Attachments in a SOAP Message Handler

It is assumed in this section that you are creating a SOAP message handler that accesses a `java.awt.Image` attachment and that the Image has been sent from a client application that uses the client JAX-RPC stubs generated by the `clientgen` Ant task.

In the client code generated by the `clientgen` Ant task, a `java.awt.Image` attachment is sent to the invoked WebLogic Web Service with a MIME type of `text/xml` rather than `image/gif`, and the image is serialized into a stream of integers that represents the image. In particular, the client code serializes the image using the following format:

- `int width`
- `int height`
- `int[] pixels`

This means that, in your SOAP message handler that manipulates the received Image attachment, you must deserialize this stream of data to then re-create the original image.

### Configuring Handlers in the JWS File

There are two standard annotations you can use in your JWS file to configure a handler chain for a Web Service: `@javax.jws.HandlerChain` and `@javax.jws.soap.SOAPMessageHandlers`.

#### `@javax.jws.HandlerChain`

When you use the `@javax.jws.HandlerChain` annotation (also called `@HandlerChain` in this chapter for simplicity) you use the `file` attribute to specify an external file that contains the configuration of the handler chain you want to associate with the Web Service. The configuration includes the list of handlers in the chain, the order in which they execute, the initialization parameters, and so on.

Use the `@HandlerChain` annotation, rather than the `@SOAPMessageHandlers` annotation, in your JWS file if one or more of the following conditions apply:

- You want multiple Web Services to share the same configuration.
- Your handler chain includes handlers for multiple transports.
You want to be able to change the handler chain configuration for a Web Service without recompiling the JWS file that implements it.

The following JWS file shows an example of using the @HandlerChain annotation; the relevant Java code is shown in bold:

```java
class HandlerChainImpl {
    public String sayHello(String input) {
        weblogic.utils.Debug.say("in backend component. input:" + input);
    }
}
```
return "'" + input + "' to you too!";
}

Before you use the @HandlerChain annotation, you must import it into your JWS file, as shown in the preceding example.

Use the file attribute of the @HandlerChain annotation to specify the name of the external file that contains configuration information for the handler chain. The value of this attribute is a URL, which may be relative or absolute. Relative URLs are relative to the location of the JWS file at the time you run the jwsc Ant task to compile the file.

Use the name attribute to specify the name of the handler chain in the configuration file that you want to associate with the Web Service. The value of this attribute corresponds to the name attribute of the <handler-chain> element in the configuration file.

**WARNING:** It is an error to specify more than one @HandlerChain annotation in a single JWS file. It is also an error to combine the @HandlerChain annotation with the @SOAPMessageHandlers annotation.

For details about creating the external configuration file, see “Creating the Handler Chain Configuration File” on page 7-31.

For additional detailed information about the standard JWS annotations discussed in this section, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

@javax.jws.soap.SOAPMessageHandlers

When you use the @javax.jws.soap.SOAPMessageHandlers (also called @SOAPMessageHandlers in this section for simplicity) annotation, you specify, within the JWS file itself, an array of SOAP message handlers (specified with the @SOAPMessageHandler annotation) that execute before and after the operations of a Web Service. The @SOAPMessageHandler annotation includes attributes to specify the class name of the handler, the initialization parameters, list of SOAP headers processed by the handler, and so on. Because you specify the list of handlers within the JWS file itself, the configuration of the handler chain is embedded within the Web Service.

Use the @SOAPMessageHandlers annotation if one or more of the following conditions apply:

- You prefer to embed the configuration of the handler chain inside the Web Service itself, rather than specify the configuration in an external file.
● Your handler chain includes only SOAP handlers and none for any other transport.

● You prefer to recompile the JWS file each time you change the handler chain configuration.

The following JWS file shows a simple example of using the `@SOAPMessageHandlers` annotation; the relevant Java code is shown in bold:

```java
package examples.webservices.soap_handlers.simple;
import java.io.Serializable;
import javax.jws.soap.SOAPMessageHandlers;
import javax.jws.soap.SOAPMessageHandler;
import javax.jws.soap.SOAPBinding;
import javax.jws.WebService;
import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
@WebService(name="SimpleChainPortType",
   serviceName="SimpleChainService")
// Standard JWS annotation that specifies a list of SOAP message handlers
// that execute before and after an invocation of all operations in the
// Web Service.
@SOAPMessageHandlers ( {
   @SOAPMessageHandler ( className="examples.webservices.soap_handlers.simple.ServerHandler1"),
   @SOAPMessageHandler ( className="examples.webservices.soap_handlers.simple.ServerHandler2")
} )
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
   use=SOAPBinding.Use.LITERAL,
   parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
@WLHttpTransport(contextPath="SimpleChain", serviceUri="SimpleChain",
   portName="SimpleChainServicePort")
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
* Web Service with a single operation: sayHello. The Web Service also
* has a handler chain associated with it, as specified by the
* @SOAPMessageHandler/s annotations.
* <p>
* @author Copyright (c) 2005 by BEA Systems, Inc. All Rights Reserved.
*/

public class SimpleChainImpl {
    // by default all public methods are exposed as operations
    public String sayHello(String input) {
        weblogic.utils.Debug.say( "in backend component. input:" + input );
        return "'" + input + "' to you too!";
    }
}

Before you use the @SOAPMessageHandlers and @SOAPMessageHandler annotations, you
must import them into your JWS file, as shown in the preceding example. Note that these
annotations are in the javax.jws.soap package.

The order in which you list the handlers (using the @SOAPMessageHandler annotation) in the
@SOAPMessageHandlers array specifies the order in which the handlers execute: in forward
order before the operation, and in reverse order after the operation. The preceding example
configures two handlers in the handler chain, whose class names are
examples.webservices.soap_handlers.simple.ServerHandler1 and
to examples.webservices.soap_handlers.simple.ServerHandler2.

Use the initParams attribute of @SOAPMessageHandler to specify an array of initialization
parameters expected by a particular handler. Use the @InitParam standard JWS annotation to
specify the name/value pairs, as shown in the following example:

@SOAPMessageHandler(
    className = "examples.webservices.soap_handlers.simple.ServerHandler1",
    initParams = { @InitParam(name="logCategory", value="MyService")}
)

The @SOAPMessageHandler annotation also includes the roles attribute for listing the SOAP
roles implemented by the handler, and the headers attribute for listing the SOAP headers
processed by the handler.

**WARNING:** It is an error to combine the @SOAPMessageHandlers annotation with the
@HandlerChain annotation.
Creating and Using SOAP Message Handlers

For additional detailed information about the standard JWS annotations discussed in this section, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

Creating the Handler Chain Configuration File

If you decide to use the @HandlerChain annotation in your JWS file to associate a handler chain with a Web Service, you must create an external configuration file that specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on.

Because this file is external to the JWS file, you can configure multiple Web Services to use this single configuration file to standardize the handler configuration file for all Web Services in your enterprise. Additionally, you can change the configuration of the handler chains without needing to recompile all your Web Services. Finally, if you include handlers in your handler chain that use a non-SOAP transport, then you are required to use the @HandlerChain annotation rather than the @SOAPMessageHandler annotation.

The configuration file uses XML to list one or more handler chains, as shown in the following simple example:

```xml
xmlns:soap1="http://HandlerInfo.org/Server1"
xmlns:soap2="http://HandlerInfo.org/Server2"
xmlns="http://java.sun.com/xml/ns/j2ee">
    <jwshc:handler-chain>
        <jwshc:handler-chain-name>SimpleChain</jwshc:handler-chain-name>
        <jwshc:handler>
            <handler-name>handler1</handler-name>
            <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler1</handler-class>
        </jwshc:handler>
        <jwshc:handler>
            <handler-name>handler2</handler-name>
            <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler2</handler-class>
        </jwshc:handler>
    </jwshc:handler-chain>
</jwshc:handler-config>
```

In the example, the handler chain called SimpleChain contains two handlers: handler1 and handler2, implemented with the class names specified with the <handler-class> element.
The two handlers execute in forward order before the relevant Web Service operation executes, and in reverse order after the operation executes.

Use the `<init-param>`, `<soap-role>`, and `<soap-header>` child elements of the `<handler>` element to specify the handler initialization parameters, SOAP roles implemented by the handler, and SOAP headers processed by the handler, respectively.

For the XML Schema that defines the external configuration file, additional information about creating it, and additional examples, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

### Compiling and Rebuilding the Web Service

It is assumed in this section that you have a working `build.xml` Ant file that compiles and builds your Web Service, and you want to update the build file to include handler chain. See Chapter 4, “Iterative Development of WebLogic Web Services,” for information on creating this `build.xml` file.

Follow these guidelines to update your development environment to include message handler compilation and building:

- After you have updated the JWS file with either the `@HandlerChain` or `@SOAPMessageHandlers` annotation, you must rerun the `jwsc` Ant task to recompile the JWS file and generate a new Web Service. This is true anytime you make a change to an annotation in the JWS file.

  If you used the `@HandlerChain` annotation in your JWS file, reran the `jwsc` Ant task to regenerate the Web Service, and subsequently changed only the external configuration file, you do not need to rerun `jwsc` for the second change to take affect.

- The `jwsc` Ant task compiles SOAP message handler Java files into handler classes (and then packages them into the generated application) if all the following conditions are true:

  - The handler classes are referenced in the `@HandlerChain` or `@SOAPMessageHandler(s)` annotations of the JWS file.
  - The Java files are located in the directory specified by the `sourcepath` attribute.
  - The classes are not currently in your CLASSPATH.

If you want to compile the handler classes yourself, rather than let `jwsc` compile them automatically, ensure that the compiled classes are in your CLASSPATH before you run the `jwsc` Ant task.
Creating and Using SOAP Message Handlers

- You deploy and invoke a Web Service that has a handler chain associated with it in the same way you deploy and invoke one that has no handler chain. The only difference is that when you invoke any operation of the Web Service, the WebLogic Web Services runtime executes the handlers in the handler chain both before and after the operation invoke.
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 8-6 for a list of supported XML and Java data types.

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

The following table lists the supported XML Schema data types (target namespace `http://www.w3.org/2001/XMLSchema`) and their corresponding Java data types.

For a list of the supported user-defined XML data types, see “Java-to-XML Mapping for Built-In Data Types” on page 8-4.

<table>
<thead>
<tr>
<th>XML Schema Data Type</th>
<th>Equivalent Java Data Type (lower case indicates a primitive data type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>byte</td>
<td>byte</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
</tbody>
</table>

---

8-2 Programming Web Services for WebLogic Server
<table>
<thead>
<tr>
<th>XML Schema Data Type</th>
<th>Equivalent Java Data Type (lower case indicates a primitive data type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>integer</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>string</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>dateTime</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>hexBinary</td>
<td>byte[]</td>
</tr>
<tr>
<td>duration</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>time</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>date</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>gYear</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>gDay</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>gMonth</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>anyURI</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>NOTATION</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>token</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>normalizedString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>language</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>Name</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>
Data Types and Data Binding

### Table 8-1  Mapping XML Schema Data Types to Java Data Types

<table>
<thead>
<tr>
<th>XML Schema Data Type</th>
<th>Equivalent Java Data Type (lower case indicates a primitive data type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMToken</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NCName</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NMTokenS</td>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>ID</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>IDREF</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>ENTITY</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>IDREFS</td>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>positiveInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>long</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>int</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>short</td>
</tr>
<tr>
<td>QName</td>
<td>javax.xml.namespace.QName</td>
</tr>
</tbody>
</table>

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

For a list of the supported user-defined Java data types, see “Supported Java User-Defined Data Types” on page 8-8.
### Table 8-2  Mapping Java Data Types to XML Schema Data Types

<table>
<thead>
<tr>
<th>Java Data Type (lower case indicates a primitive data type)</th>
<th>Equivalent XML Schema Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>byte</td>
<td>byte</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>string (with facet of length=1)</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>int</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>short</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>long</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>float</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>double</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>byte</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>string (with facet of length=1)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>string</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>integer</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>decimal</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>dateTime</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>dateTime</td>
</tr>
</tbody>
</table>
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 8-2, then you must create the user-defined data type artifacts manually.

Supported XML User-Defined Data Types

The following table lists the XML Schema data types supported by the jwsc and 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 8-2  Mapping Java Data Types to XML Schema Data Types

<table>
<thead>
<tr>
<th>Java Data Type (lower case indicates a primitive data type)</th>
<th>Equivalent XML Schema Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[]</td>
<td>base64Binary</td>
</tr>
<tr>
<td>javax.xml.namespace.QName</td>
<td>QName</td>
</tr>
<tr>
<td>java.net.URI</td>
<td>anyURI</td>
</tr>
</tbody>
</table>

Table 8-3  Supported User-Defined XML Schema Data Types

<table>
<thead>
<tr>
<th>XML Schema Data Type</th>
<th>Equivalent Java Data Type or Mapping Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="">xsd:complexType</a> with elements of both simple and complex types.</td>
<td>JavaBean</td>
</tr>
<tr>
<td><a href="">xsd:complexType</a> with simple content.</td>
<td>JavaBean</td>
</tr>
<tr>
<td><a href="">xsd:attribute</a> in <a href="">xsd:complexType</a></td>
<td>Property of a JavaBean</td>
</tr>
<tr>
<td>Derivation of new simple types by restriction of an existing simple type.</td>
<td>Equivalent Java data type of simple type.</td>
</tr>
</tbody>
</table>
### Table 8-3  Supported User-Defined XML Schema Data Types

<table>
<thead>
<tr>
<th>XML Schema Data Type</th>
<th>Equivalent Java Data Type or Mapping Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facets used with restriction element.</td>
<td>Facets not enforced during serialization and deserialization.</td>
</tr>
<tr>
<td><a href="">xsd:list</a></td>
<td>Array of the list data type.</td>
</tr>
<tr>
<td>Array derived from soapenc:Array by restriction using the wsdl:arrayType attribute.</td>
<td>Array of the Java equivalent of the arrayType data type.</td>
</tr>
<tr>
<td>Array derived from soapenc:Array by restriction.</td>
<td>Array of Java equivalent.</td>
</tr>
<tr>
<td>Derivation of a complex type from a simple type.</td>
<td>JavaBean with a property called _value whose type is mapped from the simple type according to the rules in this section.</td>
</tr>
<tr>
<td><a href="">xsd:anyType</a></td>
<td>java.lang.Object</td>
</tr>
<tr>
<td><a href="">xsd:any</a></td>
<td>javax.xml.soap.SOAPElement or org.apache.xmlbeans.XmlObject</td>
</tr>
<tr>
<td><a href="">xsd:any[]</a></td>
<td>javax.xml.soap.SOAPElement[] or org.apache.xmlbeans.XmlObject</td>
</tr>
<tr>
<td><a href="">xsd:union</a></td>
<td>Common parent type of union members.</td>
</tr>
<tr>
<td><a href="">xsi:nil</a> and <a href="">xsd:nillable</a> attribute</td>
<td>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).</td>
</tr>
<tr>
<td>Derivation of complex types</td>
<td>Mapped using Java inheritance.</td>
</tr>
<tr>
<td>Abstract types</td>
<td>Abstract Java data type.</td>
</tr>
</tbody>
</table>
## 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 8-4 Supported User-Defined Java Data Types

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>Equivalent XML Schema Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaBean whose properties are any supported data type.</td>
<td><code>&lt;xsd:complexType&gt;</code> whose content model is a <code>&lt;xsd:sequence&gt;</code> of elements corresponding to JavaBean properties.</td>
</tr>
<tr>
<td>Array and multidimensional array of any supported data type (when used as a JavaBean property)</td>
<td>An element in a <code>&lt;xsd:complexType&gt;</code> with the <code>maxOccurs</code> attribute set to unbounded.</td>
</tr>
<tr>
<td><code>java.lang.Object</code></td>
<td><code>&lt;xsd:anyType&gt;</code></td>
</tr>
<tr>
<td><strong>Note:</strong> The data type of the runtime object must be a known type.</td>
<td></td>
</tr>
<tr>
<td>Apache XMLBeans (in package <code>org.apache.xmlbeans</code>)</td>
<td>See <a href="#">Apache XMLBeans</a>.</td>
</tr>
<tr>
<td><strong>Note:</strong> The 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.</td>
<td></td>
</tr>
<tr>
<td><code>java.util.Collection</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.List</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.ArrayList</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.LinkedList</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.Vector</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.Stack</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.Set</code></td>
<td>Literal Array</td>
</tr>
<tr>
<td><code>java.util.TreeSet</code></td>
<td>Literal Array</td>
</tr>
</tbody>
</table>
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:

```java
public ArrayList<String> echoGeneric(ArrayList<String> in) {
    return in;
}
```
Data Types and Data Binding
CHAPTER 9

Invoking Web Services

The following sections describe how to invoke WebLogic Web Services:

- “Overview of Web Services Invocation” on page 9-1
- “Invoking a Web Service from a Stand-alone Client: Main Steps” on page 9-4
- “Invoking a Web Service from Another Web Service” on page 9-12
- “Using a Proxy Server When Invoking a Web Service” on page 9-18
- “Client Considerations When Redeploying a Web Service” on page 9-21
- “Creating and Using Client-Side SOAP Message Handlers” on page 9-22
- “Using a Client-Side Security WS-Policy File” on page 9-27
- “WebLogic Web Services Stub Properties” on page 9-32

Note: The following sections do not include information about invoking message-secured Web Services; for that topic, see the message security section, in particular “Updating a Client Application to Invoke a Message-Secured Web Service” on page 10-41.

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 J2EE 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 J2EE 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.
Overview of Web Services Invocation

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.

Table 9-1  JAX-RPC Interfaces and Classes

<table>
<thead>
<tr>
<th>javax.xml.rpc Interface or Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Main client interface.</td>
</tr>
<tr>
<td>ServiceFactory</td>
<td>Factory class for creating Service instances.</td>
</tr>
<tr>
<td>Stub</td>
<td>Base class of the client proxy used to invoke the operations of a Web Service.</td>
</tr>
<tr>
<td>Call</td>
<td>Used to dynamically invoke a Web Service.</td>
</tr>
<tr>
<td>JAXRPCException</td>
<td>Exception thrown if an error occurs while invoking a Web Service.</td>
</tr>
</tbody>
</table>
• A client-side copy of the WSDL file.

For additional information about the clientgen Ant task, such as all the available attributes, see Appendix A, “Ant Task Reference.”

**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 9.2, 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 9-11.

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 9-5.

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 9-6.

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 9-8.

5. Compile and run your Java client application.

   See “Compiling and Running the Client Application” on page 9-9.

### 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:

```xml
<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`” on page A-5.

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 9-11 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.
Invoking a Web Service from a Stand-alone Client: Main Steps

- The `ServiceName.java` source file contains the `getPortName()` 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 4-17 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:

```xml
<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):

```xml
<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.

```java
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:

  ```java
  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:

  ```java
  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

```xml
<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"
```
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 prototyping, 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
```
Invoking a Web Service from a Stand-alone Client: Main Steps

```java
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 9-11 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 9-5 and “Compiling and Running the Client Application” on page 9-9 for explanations of the sections in bold.

```xml
<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" />
```

Programming Web Services for WebLogic Server  9-11
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 9-4. 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.
Invoking a Web Service from Another Web Service

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 J2EE 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 9-4. 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 9-14 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 9-16 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 9-8 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">

    </clientgen>
</jws>
</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>
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.

```java
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")
```
Invoking a Web Service from Another Web Service

```java
@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:
  ```java
  import examples.webservices.complex.BasicStruct;
  ```

- Import the JAX-RPC stubs of the ComplexService Web Service; the stubs are generated by the `<clientgen>` child element of `<jws>`:
  ```java
  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`:
  ```java
  throws ServiceException, RemoteException
  ```

- Create the JAX-RPC Service and Port instances for the ComplexService:
  ```java
  ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
  ComplexPortType port = service.getComplexServicePort();
  ```
Invoking Web Services

- Invoke the echoComplexType operation of ComplexService using the port you just instantiated:

  BasicStruct result = port.echoComplexType(input);

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:

```java
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;
  
  Using a Proxy Server When Invoking a Web Service
• Import the WebLogic HttpTransportInfo API:
  
  ```java
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

• Create a Proxy object that represents the proxy server:

  ```java
  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:

  ```java
  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:

  ```java
  ((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:

  ```java
  ((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",user);
  ((Stub)port)._setProperty("weblogic.webservice.client.proxypassword",pass);
  ```

  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:

  ```java
  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`
Client Considerations When Redeploying a Web Service

- 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:

```xml
<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.

## Creating and Using Client-Side SOAP Message Handlers

The section “Creating and Using SOAP Message Handlers” on page 7-12 describes how to create server-side SOAP message handlers that execute as part of the Web Service running on WebLogic Server. You can also create client-side handlers that execute as part of the client application that invokes a Web Service operation. In the case of a client-side handler, the handler executes twice:

- Directly before the client application sends the SOAP request to the Web Service
- Directly after the client application receives the SOAP response from the Web Service

You can configure client-side SOAP message handlers for both stand-alone clients and clients that run inside of WebLogic Server.

You create the actual Java client-side handler in the same way you create a server-side handler: write a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class. In many cases you can use the exact same handler class on both the Web Service running on WebLogic Server and the client applications that invoke the Web Service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client.
Similar to the server-side SOAP handler programming, you use an XML file to specify to the clientgen Ant task that you want to invoke client-side SOAP message handlers. However, the XML Schema of this XML file is slightly different, as described in the following procedure.

**Using Client-Side SOAP Message Handlers: Main Steps**

The following procedure describes the high-level steps to add client-side SOAP message handlers to the client application that invokes a Web Service operation.

It is assumed that you have already created the client application that invokes a deployed Web Service, and that you want to update the client application by adding client-side SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working build.xml file that includes a target for running the clientgen Ant task. For more information, see “Invoking a Web Service from a Stand-alone Client: Main Steps” on page 9-4.

1. Design the client-side SOAP handlers and the handler chain which specifies the order in which they execute. This step is almost exactly the same as that of designing the server-side SOAP message handlers, except the perspective is from the client application, rather than a Web Service.
   
   See “Designing the SOAP Message Handlers and Handler Chains” on page 7-15.

2. For each handler in the handler chain, create a Java class that extends the javax.xml.rpc.handler.GenericHandler abstract class. This step is very similar to the corresponding server-side step, except that the handler executes in a chain in the client rather than the server.
   
   See “Creating the GenericHandler Class” on page 7-18 for details about programming a handler class. See “Example of a Client-Side Handler Class” on page 9-24 for an example.

3. Create the client-side SOAP handler configuration file. This XML file describes the handlers in the handler chain, the order in which they execute, and any initialization parameters that should be sent.

   See “Creating the Client-Side SOAP Handler Configuration File” on page 9-25.

4. Update the build.xml file that builds your client application, specifying to the clientgen Ant task the name of the SOAP handler configuration file. Also ensure that the build.xml file compiles the handler files into Java classes and makes them available to your client application.

   See “Specifying the Client-Side SOAP Handler Configuration File to clientgen” on page 9-27.
5. Rebuild your client application by running the relevant task:

   prompt> ant build-client

When you next run the client application, the SOAP messaging handlers listed in the configuration file automatically execute before the SOAP request message is sent and after the response is received.

**Note:** You do not have to update your actual client application to invoke the client-side SOAP message handlers; as long as you specify to the `clientgen` Ant task the handler configuration file, the generated JAX-RPC stubs automatically take care of executing the handlers in the correct sequence.

**Example of a Client-Side Handler Class**

The following example shows a simple SOAP message handler class that you can configure for a client application that invokes a Web Service.

```java
package examples.webservices.client_handler.client;

import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;

public class ClientHandler1 extends GenericHandler {
    private QName[] headers;

    public void init(HandlerInfo hi) {
        System.out.println("in " + this.getClass() + " init()");
    }

    public boolean handleRequest(MessageContext context) {
        System.out.println("in " + this.getClass() + " handleRequest()");
        return true;
    }

    public boolean handleResponse(MessageContext context) {
        System.out.println("in " + this.getClass() + " handleResponse()");
        return true;
    }

    public boolean handleFault(MessageContext context) {
        System.out.println("in " + this.getClass() + " handleFault()");
        return true;
    }
}
```
Creating and Using Client-Side SOAP Message Handlers

public QName[] getHeaders() {
    return headers;
}

Creating the Client-Side SOAP Handler Configuration File

The client-side SOAP handler configuration file specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on. See “XML Schema for the Client-Side Handler Configuration File” on page 9-26 for a full description of this file.

The configuration file uses XML to describe a single handler chain that contains one or more handlers, as shown in the following simple example:

```xml
<weblogic-wsee-clientHandlerChain
    xmlns="http://www.bea.com/ns/weblogic/90"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee">
    <handler>
        <j2ee:handler-name>clienthandler1</j2ee:handler-name>
        <j2ee:handler-class>examples.webservices.client_handler.client.ClientHandler1</j2ee:handler-class>
        <j2ee:init-param>
            <j2ee:param-name>ClientParam1</j2ee:param-name>
            <j2ee:param-value>value1</j2ee:param-value>
        </j2ee:init-param>
    </handler>
    <handler>
        <j2ee:handler-name>clienthandler2</j2ee:handler-name>
        <j2ee:handler-class>examples.webservices.client_handler.client.ClientHandler2</j2ee:handler-class>
        <j2ee:init-param>
            <j2ee:param-name>ClientParam1</j2ee:param-name>
            <j2ee:param-value>value1</j2ee:param-value>
        </j2ee:init-param>
    </handler>
</weblogic-wsee-clientHandlerChain>
```

In the example, the handler chain contains two handlers: clienthandler1 and clienthandler2, implemented with the class names specified with the `<j2ee:handler-class>` element. The two handlers execute in forward order directly before the client application sends the SOAP request to the Web Service, and then in reverse order directly after the client application receives the SOAP response from the Web Service.

The example also shows how to use the `<j2ee:init-param>` element to specify one or more initialization parameters to a handler.
Use the `<soap-role>`, `<soap-header>`, and `<port-name>` child elements of the `<handler>` element to specify the SOAP roles implemented by the handler, the SOAP headers processed by the handler, and the port-name element in the WSDL with which the handler is associated with, respectively.

**XML Schema for the Client-Side Handler Configuration File**

The following XML Schema file defines the structure of the client-side SOAP handler configuration file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema
    targetNamespace="http://www.bea.com/ns/weblogic/90"
    xmlns:wls="http://www.bea.com/ns/weblogic/90"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified" />
<include schemaLocation="weblogic-j2ee.xsd"/>
<element name="weblogic-wsee-clientHandlerChain"
    type="wls:weblogic-wsee-clientHandlerChainType">
    <xsd:annotation>
        <xsd:documentation>
            Defines the name of the handler. The name must be unique within the chain.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:key name="wsee-clienthandler-name-key">
        <xsd:selector xpath="j2ee:handler"/>
        <xsd:field xpath="j2ee:handler-name"/>
    </xsd:key>
</element>
<complexType name="weblogic-wsee-clientHandlerChainType">
    <sequence>
        <xsd:element name="handler"
            type="j2ee:service-ref_handlerType"
            minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
</complexType>
</schema>
```
A single configuration file specifies a single client-side handler chain. The root of the configuration file is `<weblogic-wsee-clientHandlerChain>`, and the file contains zero or more `<handler>` child elements, each of which describes a handler in the chain.

The structure of the `<handler>` element is described by the J2EE `service-ref_handlerType` complex type, specified in the J2EE 1.4 Web Service client XML Schema.

**Specifying the Client-Side SOAP Handler Configuration File to clientgen**

Use the `handlerChainFile` attribute of the `clientgen` Ant task to specify the client-side SOAP handler configuration file, as shown in the following excerpt from a `build.xml` file:

```
<clientgen
    destDir="${clientclass-dir}"
    handlerChainFile="ClientHandlerChain.xml"
    packageName="examples.webservices.client_handler.client"/>
```

The JAX-RPC stubs generated by `clientgen` automatically ensure that the handlers described by the configuration file execute in the correct order before and after the client application invokes the Web Service operation.

**Using a Client-Side Security WS-Policy File**

The section “Using WS-Policy Files for Message-Level Security Configuration” on page 10-4 describes how a WebLogic Web Service can be associated with one or more WS-Policy files that describe the message-level security of the Web Service. These WS-Policy files are XML files that describe how a SOAP message should be digitally signed or encrypted and what sort of user authentication is required from a client that invokes the Web Service. Typically, the WS-Policy file associated with a Web Service is attached to its WSDL, which the Web Services client runtime reads to determine whether and how to digitally sign and encrypt the SOAP message request from an operation invoke from the client application.

Sometimes, however, a Web Service might not attach the WS-Policy file to its deployed WSDL or the Web Service might be configured to not expose its WSDL at all. In these cases, the Web Services client runtime cannot determine from the service itself the security that must be enabled for the SOAP message request. Rather, it must load a client-side copy of the WS-Policy file. This section describes how to update a client application to load a local copy of a WS-Policy file.
The client-side WS-Policy file is typically exactly the same as the one associated with a deployed Web Service. If the two files are different, and there is a conflict in the security assertions contained in the files, then the invoke of the Web Service operation returns an error.

You can specify that the client-side WS-Policy file be associated with the SOAP message request, response, or both. Additionally, you can specify that the WS-Policy file be associated with the entire Web Service, or just one of its operations.

**Associating a WS-Policy File with a Client Application: Main Steps**

The following procedure describes the high-level steps to associate a WS-Policy file with the client application that invokes a Web Service operation.

It is assumed that you have created the client application that invokes a deployed Web Service, and that you want to update it by associating a client-side WS-Policy file. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `clientgen` Ant task. See “Invoking a Web Service from a Stand-alone Client: Main Steps” on page 9-4.

1. Create the client-side WS-Policy files and save them in a location accessible by the client application. Typically, the WS-Policy files are the same as those configured for the Web Service you are invoking, but because the server-side files are not exposed to the client runtime, the client application must load its own local copies.

   See “Creating and Using a Custom WS-Policy File” on page 10-23 for information about creating WS-Policy files.

   **WARNING:** You can specify only concrete client-side WS-Policy files to a client application; you cannot use abstract WS-Policy files or the three pre-packaged security WS-Policy files.

2. Update the `build.xml` file that builds your client application by specifying to the `clientgen` Ant task that it should generate additional `getXXXPort()` methods in the JAX-RPC stub, where `XXX` refers to the name of the Web Service. These methods are later used by the client application to load the client-side WS-Policy files.

   See “Updating clientgen to Generate Methods That Load WS-Policy Files” on page 9-29.

3. Update your Java client application to load the client-side WS-Policy files using the additional `getXXXPort()` methods that the `clientgen` Ant task generates.

   See “Updating a Client Application To Load WS-Policy Files” on page 9-29.
4. Rebuild your client application by running the relevant task. For example:

prompt> ant build-client

When you next run the client application, it will load local copies of the WS-Policy files that the Web Service client runtime uses to enable security for the SOAP request message.

### Updating clientgen to Generate Methods That Load WS-Policy Files

Set the `generatePolicyMethods` attribute of the `clientgen` Ant task to `true` to specify that the Ant task should generate additional `getXXX()` methods in the implementation of the JAX-RPC `Service` interface for loading client-side copies of WS-Policy files when you get a port, as shown in the following example:

```xml
<clientgen
    destDir="${clientclass-dir}"
    generatePolicyMethods="true"
    packageName="examples.webservices.client_policy.client"/>
```

See “Updating a Client Application To Load WS-Policy Files” on page 9-29 for a description of the additional methods that are generated and how to use them in a client application.

### Updating a Client Application To Load WS-Policy Files

When you set `generatePolicyMethods="true"` for `clientgen`, the Ant task generates additional methods in the implementation of the JAX-RPC `Service` interface that you can use to load WS-Policy files, where `XXX` refers to the name of the Web Service. You can use either an `Array` or `Set` of WS-Policy files to associate multiple files to a Web Service. If you want to associate just a single WS-Policy file, create a single-member `Array` or `Set`.

- `getXXXPort(String operationName, java.util.Set<java.io.InputStream> inbound, java.util.Set<java.io.InputStream> outbound)`

  Loads two different sets of client-side WS-Policy files from InputStreams and associates the first set to the SOAP request and the second set to the SOAP response. Applies to a specific operation, as specified by the first parameter.

- `getXXXPort(String operationName, java.io.InputStream[] inbound, java.io.InputStream[] outbound)`
Invoking Web Services

Loads two different arrays of client-side WS-Policy files from InputStreams and associates the first array to the SOAP request and the second array to the SOAP response. Applies to a specific operation, as specified by the first parameter.

- `getXXXPort(java.util.Set<java.io.InputStream> inbound, java.util.Set<java.io.InputStream> outbound)`

Loads two different sets of client-side WS-Policy files from InputStreams and associates the first set to the SOAP request and the second set to the SOAP response. Applies to all operations of the Web Service.

- `getXXXPort(java.io.InputStream[] inbound, java.io.InputStream[] outbound)`

Loads two different arrays of client-side WS-Policy files from InputStreams and associates the first array to the SOAP request and the second array to the SOAP response. Applies to all operations of the Web Service.

Use these methods, rather than the normal `getXXXPort()` method with no parameters, for getting a Web Service port and specifying at the same time that invokes of all, or the specified, operation using that port have an associated WS-Policy file or files.

**Note:** The following methods from a previous release of WebLogic Server have been deprecated; if you want to associate a single client-side WS-Policy file, specify a single-member Array or Set and use the corresponding method described above.

- `getXXXPort(java.io.InputStream policyInputStream)`

  Loads a single client-side WS-Policy file from an InputStream and applies it to both the SOAP request (inbound) and response (outbound) messages.

- `getXXXPort(java.io.InputStream policyInputStream, boolean inbound, boolean outbound)`

  Loads a single client-side WS-Policy file from an InputStream and applies it to either the SOAP request or response messages, depending on the Boolean value of the second and third parameters.

The following simple client application shows an example of using these policy methods; the code in bold is described after the example.

```java
package examples.webservices.client_policy.client;

import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
```
import java.io.FileInputStream;
import java.io.IOException;

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the ClientPolicyService Web service.
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public class Main {
    public static void main(String[] args)
        throws ServiceException, RemoteException, IOException {
        FileInputStream[] inbound_policy_array = new FileInputStream[2];
        inbound_policy_array[0] = new FileInputStream(args[1]);
        inbound_policy_array[1] = new FileInputStream(args[2]);

        FileInputStream[] outbound_policy_array = new FileInputStream[2];
        outbound_policy_array[0] = new FileInputStream(args[1]);
        outbound_policy_array[1] = new FileInputStream(args[2]);

        ClientPolicyService service = new ClientPolicyService_Impl(args[0] + "?WSDL");

        // standard way to get the Web Service port
        ClientPolicyPortType normal_port = service.getClientPolicyPort();

        // specify an array of WS-Policy file for the request and response
        // of a particular operation
        ClientPolicyPortType array_of_policy_port =
            service.getClientPolicyPort("sayHello", inbound_policy_array,
            outbound_policy_array);

        try {
            String result = null;
            result = normal_port.sayHello("Hi there!");
            result = array_of_policy_port.sayHello("Hi there!");
            System.out.println("Got result: " + result);
        } catch (RemoteException e) {
            throw e;
        }
    }
}

The second and third argument to the client application are the two WS-Policy files from which
the application makes an array of FileInputStreams (inbound_policy_array and
outbound_policy_array). The normal_port uses the standard parameterless method for
going a port; the array_of_policy_port, however, uses one of the policy methods to specify
that an invoke of the sayHello operation using the port has multiple WS-Policy files (specified
with an Array of FileInputStream) associated with both the inbound and outbound SOAP
request and response:

```java
ClientPolicyPortType array_of_policy_port =
    service.getClientPolicyPort("sayHello", inbound_policy_array,
    outbound_policy_array);
```

## 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:

```java
((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>
<thead>
<tr>
<th>Stub Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weblogic.wsee.transport.connection.timeout</td>
<td>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.</td>
</tr>
<tr>
<td>weblogic.wsee.transport.read.timeout</td>
<td>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.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Stub Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weblogic.wsee.security.bst.serverVerifyCert</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

```xml
<soapenv:Envelope>
  ...
  <maxResults xsi:type="xs:int">10</maxResults>
  ...
</soapenv:Envelope>
```

By default (or if you set this property to False), the parameter element would look like the following example:

```xml
<soapenv:Envelope>
  ...
  <maxResults>10</maxResults>
  ...
</soapenv:Envelope>
```

Valid values for this property are True and False; default value is False.
CHAPTER 10

Configuring Security

The following sections describe how to configure security for your Web Service:

- “Overview of Web Services Security” on page 10-1
- “What Type of Security Should You Configure?” on page 10-2
- “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3
- “Updating a Client Application to Invoke a Message-Secured Web Service” on page 10-41
- “Configuring Transport-Level Security” on page 10-45
- “Configuring Access Control Security: Main Steps” on page 10-48

Overview of Web Services Security

To secure your WebLogic Web Service, you configure one or more of three different types of security:

- Message-level security, in which data in a SOAP message is digitally signed or encrypted.
  See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3.

- Transport-level security, in which SSL is used to secure the connection between a client application and the Web Service.
Configuring Security

- Access control security, which specifies which roles are allowed to access Web Services.

What Type of Security Should You Configure?

Access control security answers the question “who can do what?” First you specify the security roles that are allowed to access a Web Service; a security role is a privilege granted to users or groups based on specific conditions. Then, when a client application attempts to invoke a Web Service operation, the client authenticates itself to WebLogic Server, and if the client has the authorization, it is allowed to continue with the invocation. Access control security secures only WebLogic Server resources. That is, if you configure only access control security, the connection between the client application and WebLogic Server is not secure and the SOAP message is in plain text.

With transport-level security, you secure the connection between the client application and WebLogic Server with Secure Sockets Layer (SSL). SSL provides secure connections by allowing two applications connecting over a network to authenticate the other's identity and by encrypting the data exchanged between the applications. Authentication allows a server, and optionally a client, to verify the identity of the application on the other end of a network connection. Encryption makes data transmitted over the network intelligible only to the intended recipient.

Transport-level security, however, secures only the connection itself. This means that if there is an intermediary between the client and WebLogic Server, such as a router or message queue, the intermediary gets the SOAP message in plain text. When the intermediary sends the message to a second receiver, the second receiver does not know who the original sender was. Additionally, the encryption used by SSL is “all or nothing”: either the entire SOAP message is encrypted or it is not encrypted at all. There is no way to specify that only selected parts of the SOAP message be encrypted.

Message-level security includes all the security benefits of SSL, but with additional flexibility and features. Message-level security is end-to-end, which means that a SOAP message is secure even when the transmission involves one or more intermediaries. The SOAP message itself is digitally signed and encrypted, rather than just the connection. And finally, you can specify that only parts of the message be signed or encrypted.
Configuring Message-Level Security (Digital Signatures and Encryption)

Message-level security specifies whether the SOAP messages between a client application and the Web Service it is invoking should be digitally signed or encrypted or both. It also can specify a shared security context between the Web Service and client in the event that they exchange multiple SOAP messages.

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

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

These specifications provide security token propagation, message integrity, and message confidentiality. These mechanisms can be used independently (such as passing a username token for user authentication) or together (such as digitally signing and encrypting a SOAP message and specifying that a user must use X.509 certificates for authentication).

WebLogic Web Services also implement the Web Services Trust Language (WS-Trust) and Web Services Secure Conversation Language (WS-SecureConversation) specifications which together provide secure communication between Web Services and their clients (either other Web Services or standalone Java client applications.) In particular, the WS-SecureConversation specification defines mechanisms for establishing and sharing security contexts, and deriving keys from security contexts, to enable a secure conversation. Together, the security context and derived keys potentially increase the overall performance and security of the subsequent exchanges. This functionality is missing from the WS-Security specifications, which focus primarily on the message authentication model and consequently are subject to various security attacks.

You configure message-level security for a Web Service by attaching one or more WS-Policy files that contain security policy statements, as specified by the WS-Policy (dated September 2004) specification. See “Using WS-Policy Files for Message-Level Security Configuration” on page 10-4 for detailed information about how the Web Services runtime environment uses these files.
See “Configuring Simple Message-Level Security: Main Steps” on page 10-14 for the basic steps you must perform to configure simple message-level security. This section discusses configuration of the Web Services runtime environment, as well as configuration of message-level security for a particular Web Service and how to code a client application to invoke the service.

You can also configure message-level security for a Web Service at runtime, after a Web Service has been deployed. See “Associating WS-Policy Files at Runtime Using the Administration Console” on page 10-32 for details.

**Note:** You cannot digitally sign or encrypt a SOAP attachment.

## Main Use Cases of Message-Level Security

The BEA implementation of the *Web Services Security: SOAP Message Security* specification supports the following use cases:

- Use X.509 certificates to sign and encrypt a SOAP message, starting from the client application that invokes the message-secured Web Service, to the WebLogic Server instance that is hosting the Web Service and back to the client application.

- Specify the SOAP message targets that are signed or encrypted: the body, specific SOAP headers, or specific elements.

- Include a username, SAML, or X.509 token in the SOAP message for authentication.

- Specify that a Web Service and its client (either another Web Service or a standalone application) establish and share a security context when exchanging multiple messages.

- Derive keys for *each* key usage in a secure context, once the context has been established and is being shared between a Web Service and its client. This means that a particular SOAP message uses two derived keys, one for signing and another for encrypting, and each SOAP message uses a different pair of derived keys from other SOAP messages. Because each SOAP message uses its own pair of derived keys, the conversation between the client and Web Service is extremely secure.

## Using WS-Policy Files for Message-Level Security Configuration

You specify the details of message-level security for a WebLogic Web Service with one or more WS-Policy files. The *WS-Policy specification* provides a general purpose model and XML syntax to describe and communicate the policies of a Web Service.
Configuring Message-Level Security (Digital Signatures and Encryption)

Note: WebLogic Server includes pre-packaged WS-Policy files that BEA recommends for most use cases. See “WebLogic Server WS-Policy Files” on page 10-5. However, if you use SAML tokens for authentication, or you want to specify that particular parts of a SOAP message rather than the entire body be encrypted or digitally signed, you must create your own WS-Policy files. See “Creating and Using a Custom WS-Policy File” on page 10-23.

The WS-Policy files used for message-level security are XML files that describe whether and how the SOAP messages resulting from an invoke of an operation should be digitally signed or encrypted. They can also specify that a client application authenticate itself using a username, SAML, or X.509 token.

Note: The policy assertions used in the WS-Policy file to configure message-level security for a WebLogic Web Service are based on the assertions described in the December 18, 2002 version of the Web Services Security Policy Language (WS-SecurityPolicy) specification. This means that although the exact syntax and usage of the assertions in WebLogic Server are different, they are similar in meaning to those described in the specification. The assertions are not based on the latest update of the specification (13 July 2005.)

You use the @Policy and @Policies JWS annotations in your JWS file to associate WS-Policy files with your Web Service. You can associate any number of WS-Policy files with a Web Service, although it is up to you to ensure that the assertions do not contradict each other. You can specify a WS-Policy file at both the class- and method-level of your JWS file.

WebLogic Server WS-Policy Files

WebLogic Server includes a set of WS-Policy files that you can specify in your JWS file if you do not want to create your own WS-Policy files; BEA recommends that unless you have specific security needs, you use these pre-packaged files as often as possible. These WS-Policy files are all abstract; see “Abstract and Concrete WS-Policy Files” on page 10-13 for details.

The pre-packaged WS-Policy files are:

- **Auth.xml**—Specifies that the client must authenticate itself. Can be used on its own, or together with Sign.xml and Encrypt.xml.

- **Sign.xml**—Specifies that the SOAP messages are digitally signed. Can be used on its own, or together with Auth.xml and Encrypt.xml.

- **Encrypt.xml**—Specifies that the SOAP messages are encrypted. Can be used on its own, or together with Auth.xml and Sign.xml.
z Wssc-dk.xml—Specifies that the client and service share a security context when multiple messages are exchanged and that derived keys are used for encryption and digital signatures, as described by the WS-SecureConversation specification.

Note: This pre-packaged WS-Policy file is meant to be used on its own and not together with Auth.xml, Sign.xml, Encrypt.xml, or Wssc-sct.xml. Also, BEA recommends that you use this WS-Policy file, rather than Wssc-sct.xml, if you want the client and service to share a security context, due to its higher level of security.

z Wssc-sct.xml—Specifies that the client and service share a security context when multiple messages are exchanged, as described by the WS-SecureConversation specification.

Note: This pre-packaged WS-Policy file is meant to be used on its own and not together with Auth.xml, Sign.xml, Encrypt.xml, or Wssc-dk.xml. Also, BEA provides this WS-Policy file to support the various use cases of the WS-SecureConversation specification; however, BEA recommends that you use the Wssc-dk.xml WS-Policy file, rather than Wssc-sct.xml, if you want the client and service to share a security context, due to its higher level of security.

Auth.xml

The WebLogic Server Auth.xml file, shown below, specifies that the client application invoking the Web Service must authenticate itself with one of the tokens (username or X.509) that support authentication.

Because the pre-packaged WS-Policy files are abstract, there is no specific username or X.509 token assertions in the Auth.xml file at development-time. Depending on how you have configured security for WebLogic Server, either a username token, an X.509 token, or both will appear in the actual runtime-version of the Auth.xml WS-Policy file associated with your Web Service. Additionally, if the runtime-version of the WS-Policy file includes an X.509 token and it is applied to a client invoke, then the entire body of the SOAP message is signed.

If you want to specify that only X.509, and never username tokens, be used for identity, or want to specify that, when using X.509 for identity, only certain parts of the SOAP message be signed, then you must create a custom WS-Policy file. For details, see “Using Only X.509 Certificate Tokens for Identity” on page 10-36.

<?xml version="1.0"?>
<wsp:Policy
xmlns:wssp="http://www.bea.com/wls90/security/policy"
>


Configuring Message-Level Security (Digital Signatures and Encryption)

<wssp:Identity/>
</wssp:Policy>

**Sign.xml**

The WebLogic Server `Sign.xml` file specifies that the body and WebLogic-specific system headers of the SOAP message be digitally signed. It also specifies that the SOAP message include a Timestamp, which is digitally signed, and that the token used for signing is also digitally signed.

The token used for signing is included in the SOAP message.

The following headers are signed when using the `Sign.xml` WS-Policy file:

- SequenceAcknowledgement
- AckRequested
- Sequence
- Action
- FaultTo
- From
- MessageID
- relatesTo
- ReplyTo
- To
- SetCookie
- Timestamp

The WebLogic Server `Sign.xml` file is shown below:

```xml
<?xml version="1.0"?>
<wsp:Policy
 xmlns:wssp="http://www.bea.com/wls90/security/policy"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
 utility-1.0.xsd"
 xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
>
<wssp:Integrity>
  <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
</wssp:Integrity>
</wsp:Policy>
```
Configuring Security

<wssp:CanonicalizationAlgorithm
    URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>

<wssp:Target>
    <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <wssp:MessageParts
        Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
        wls:SystemHeaders()
    </wssp:MessageParts>
</wssp:Target>

<wssp:Target>
    <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <wssp:MessageParts
        Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
        wls:SecurityHeader(wsu:Timestamp)
    </wssp:MessageParts>
</wssp:Target>

<wssp:Target>
    <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <wssp:MessageParts
        Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
        wsp:Body()
    </wssp:MessageParts>
</wssp:Target>
</wssp:Integrity>

<wssp:MessageAge/>
</wsp:Policy>

Encrypt.xml

The WebLogic Server Encrypt.xml file specifies that the entire body of the SOAP message be encrypted. By default, the encryption token is not included in the SOAP message.

<?xml version="1.0"?>
<wsp:Policy
    xmlns:wssp="http://www.bea.com/wls90/security/policy"/>

<wssp:Confidentiality>
    <wssp:KeyWrappingAlgorithm URI="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    <wssp:Target>
        <wssp:EncryptionAlgorithm
            URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    </wssp:Target>
Configuring Message-Level Security (Digital Signatures and Encryption)

Wssc-dk.xml

Specifies that the client and Web Service share a security context, as described by the
WS-SecureConversation specification, and that a derived key token is used. This ensures the
highest form of security.

This WS-Policy file provides the following configuration:

- A derived key token is used to sign all system SOAP headers, the timestamp security
  SOAP header, and the SOAP body.
- A derived key token is used to encrypt the body of the SOAP message. This token is
different from the one used for signing.
- Each SOAP message uses its own pair of derived keys.
- For both digital signatures and encryption, they key length is 16 (as opposed to the default
  32)
- The lifetime of the security context is 12 hours.

If you need to change the default security context and derived key behavior, you will have to
create a custom WS-Policy file, described in later sections.

**WARNING:** If you specify this pre-packaged WS-Policy file, you should not also specify any
other pre-packaged WS-Policy file.
Configuring Security

<wssp:Integrity SupportTrust10="true">
  <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#hmac-shal"/>
  <wssp:CanonicalizationAlgorithm URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>
  <wssp:Target>
    <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <wssp:MessageParts Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
      wls:SystemHeaders()
    </wssp:MessageParts>
  </wssp:Target>
</wssp:Integrity>

<wssp:Confidentiality SupportTrust10="true">
  <wssp:Target>
    <wssp:EncryptionAlgorithm URI="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
    <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
      wsp:Body()
    </wssp:MessageParts>
  </wssp:Target>
</wssp:Confidentiality>
Configuring Message-Level Security (Digital Signatures and Encryption)

<wssp:KeyInfo>
  <wssp:SecurityToken IncludeInMessage="true">
    TokenTypeInfo='http://schemas.xmlsoap.org/ws/2005/02/sc/dk'
    DerivedFromTokenType='http://schemas.xmlsoap.org/ws/2005/02/sc/sct'>
      <wssp:Claims>
        <wssp:Label>WS-SecureConversation</wssp:Label>
        <wssp:Length>16</wssp:Length>
      </wssp:Claims>
    </wssp:SecurityToken>
  </wssp:KeyInfo>
</wssp:Confidentiality>
<wssp:MessageAge/>
</wssp:Policy>

Wssc-sct.xml
Specifications that the client and Web Service share a security context, as described by the WS-SecureConversation specification. In this case, security context tokens are used to encrypt and sign the SOAP messages, which differs from Wssc-dk.xml in which derived key tokens are used. The Wssc-sct.xml WS-Policy file is provided to support all the use cases of the specification; for utmost security, however, BEA recommends you always use Wssc-dk.xml when specifying shared security contexts due to its higher level of security.

This WS-Policy file provides the following configuration:

- A security context token is used to sign all system SOAP headers, the timestamp security SOAP header, and the SOAP body.
- A security context token is used to encrypt the body of the SOAP message.
- The lifetime of the security context is 12 hours.

If you need to change the default security context and derived key behavior, you will have to create a custom WS-Policy file, described in later sections.

WARNING: If you specify this pre-packaged WS-Policy file, you should not also specify any other pre-packaged WS-Policy file.

<?xml version="1.0"?>
<wsp:Policy
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
Configuring Security

```xml
<wsse:Security
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
>
  <wssp:Integrity SupportTrust10="true">
    <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
    <wssp:CanonicalizationAlgorithm URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
        wls:SystemHeaders()
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
        wls:SecurityHeader(wsu:Timestamp)
      </wssp:MessageParts>
    </wssp:Target>
    <wssp:Target>
      <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
        wsp:Body()
      </wssp:MessageParts>
    </wssp:Target>
  </wssp:Integrity>
  <wssp:Confidentiality SupportTrust10="true">
    <wssp:EncryptionAlgorithm URI="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
    <wssp:MessageParts Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
```

10-12 Programming Web Services for WebLogic Server
Abstract and Concrete WS-Policy Files

The WebLogic Web Services runtime environment recognizes two slightly different types of WS-Policy files: abstract and concrete. The pre-packaged WS-Policy files described in “WebLogic Server WS-Policy Files” on page 10-5 are all abstract.

Abstract WS-Policy files do not explicitly specify the security tokens that are used for authentication, encryption, and digital signatures, but rather, the Web Services runtime environment determines the security tokens when the Web Service is deployed. Specifically, this means the <Identity> and <Integrity> elements (or assertions) of the WS-Policy files do not contain a <SupportedTokens><SecurityToken> child element, and the <Confidentiality> element WS-Policy file does not contain a <KeyInfo><SecurityToken> child element.

If your Web Service is associated with only the pre-packaged WS-Policy files, then client authentication requires username tokens. Web Services support only one type of token for encryption and digital signatures (X.509), which means that in the case of the <Integrity> and <Confidentiality> elements, concrete and abstract WS-Policy files end up being essentially the same.

If your Web Service is associated with an abstract WS-Policy file and it is published as an attachment to the WSDL (which is the default behavior), the static WSDL file packaged in the Web Service archive file (JAR or WAR) will be slightly different than the dynamic WSDL of the deployed Web Service. This is because the static WSDL, being abstract, does not include specific <SecurityToken> elements, but the dynamic WSDL does include these elements because the Web Services runtime has automatically filled them in when it deployed the service. For this reason, in the code that creates the JAX-RPC stub in your client application, ensure that you specify the dynamic WSDL or you will get a runtime error when you try to invoke an operation:

HelloService service = new HelloService(Dynamic_WSDL);
You can specify either the static or dynamic WSDL to the clientgen Ant task in this case. See “Browsing to the WSDL of the Web Service” on page 4-17 for information on viewing the dynamic WSDL of a deployed Web Service.

Concrete WS-Policy files explicitly specify the details of the security tokens at the time the Web Service is programmed. Programmers create concrete WS-Policy files when they know, at the time they are programming the service, the details of the type of authentication (such as using x509 or SAML tokens); whether multiple private key and certificate pairs from the keystore are going to be used for encryption and digital signatures; and so on.

Configuring Simple Message-Level Security: Main Steps

The following procedure describes how to configure simple message-level security for the Web Services security runtime, a particular WebLogic Web Service, and a client application that invokes an operation of the Web Service. In this document, simple message-level security is defined as follows:

- The message-secured Web Service uses the pre-packaged WS-Policy files to specify its security requirements, rather than a user-created WS-Policy file. See “Using WS-Policy Files for Message-Level Security Configuration” on page 10-4 for a description of these files.

- The Web Service makes its associated WS-Policy files publicly available by attaching them to its deployed WSDL, which is also publicly visible.

- The Web Services runtime uses the out-of-the-box private key and X.509 certificate pairs, store in the default keystores, for its encryption and digital signatures, rather than its own key pairs. These out-of-the-box pairs are also used by the core WebLogic Server security subsystem for SSL and are provided for demonstration and testing purposes. For this reason BEA highly recommends you use your own keystore and key pair in production. To use key pairs other than out-of-the-box pairs, see “Using Key Pairs Other Than the Out-Of-The-Box SSL Pair” on page 10-20.

**WARNING:** If you plan to deploy the Web Service to a cluster in which different WebLogic Server instances are running on different computers, you must use a keystore and key pair other than the out-of-the-box ones, even for testing purposes. The reason is that the key pairs in the default WebLogic Server keystore, DemoIdentity.jks, are not guaranteed to be the same across WebLogic Servers running on different machines. If you were to use the default keystore, the WSDL of the deployed Web Service would specify the public key from one of these keystores, but the invoke of the service might actually be handled by a server running on a different computer, and in this
Configuring Simple Message-Level Security: Main Steps

Case the server’s private key would not match the published public key and the invoke would fail. This problem only occurs if you use the default keystore and key pairs in a cluster, and is easily resolved by using your own keystore and key pairs.

- The client invoking the Web Service uses a username token to authenticate itself, rather than an X.509 token.
- The client invoking the Web Service is a stand-alone Java application, rather than a module running in WebLogic Server.

Later sections describe some of the preceding scenarios in more detail, as well as additional Web Services security uses cases that build on the simple message-level security use case.

It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web Service and you want to update it so that the SOAP messages are digitally signed and encrypted. It is also assumed that you use Ant build scripts to iteratively develop your Web Service and that you have a working build.xml file that you can update with new information. Finally, it is assumed that you have a client application that invokes the non-secured Web Service. If these assumptions are not true, see:

- Chapter 5, “Programming the JWS File”
- Chapter 4, “Iterative Development of WebLogic Web Services”
- Chapter 9, “Invoking Web Services”

To configure simple message-level security for a WebLogic Web Service:

1. Update your JWS file, adding WebLogic-specific @Policy and @Policies JWS annotations to specify the pre-packaged WS-Policy files that are attached to either the entire Web Service or to particular operations.

   See “Updating the JWS File with @Policy and @Policies Annotations” on page 10-17, which describes how to specify any WS-Policy file. For this basic procedure, follow only the instructions for specifying the pre-packaged WS-Policy files: Auth.xml, Sign.xml, Encrypt.xml, Wssc-dk.xml, and Wssc-sct.xml.

2. Recompile and redeploy your Web Service as part of the normal iterative development process.

   See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.
3. Create a keystore used by the client application. BEA recommends that you create one client keystore per application user.

You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

See *Obtaining Private Keys and Digital Signatures* at 
`{DOCROOT}/secmanage/identity_trust.html#get_keys_certs_trustedcas`.

4. Create a private key and digital certificate pair, and load it into the client keystore. The same pair will be used to both digitally sign the client’s SOAP request and encrypt the SOAP responses from WebLogic Server.

Make sure that the certificate’s key usage allows both encryption and digital signatures. Also see “Ensuring That WebLogic Server Can Validate the Client’s Certificate” on page 10-17 for information about how WebLogic Server ensures that the client’s certificate is valid.

**WARNING:** BEA requires a key length of 1024 bits or larger.

You can use Sun Microsystem’s `keytool` utility to perform this step.

See *Obtaining Private Keys and Digital Signatures*.

5. Using the Administration Console, create users for authentication in your security realm.

See *Users, Groups, and Security Roles*.

6. Update your client application by adding the Java code to invoke the message-secured Web Service.

See “Updating a Client Application to Invoke a Message-Secured Web Service” on page 10-41.

7. Recompile your client application.

See “Compiling and Running the Client Application” on page 9-9 for general information.

See the following sections for information about additional Web Service security uses cases that build on the basic message-level security use case:

- “Using Key Pairs Other Than the Out-Of-The-Box SSL Pair” on page 10-20
- “Setting the SOAP Message Expiration” on page 10-22
- “Creating and Using a Custom WS-Policy File” on page 10-23
**Configuring Simple Message-Level Security: Main Steps**

- “Configuring and Using Security Contexts and Derived Keys (WS-SecureConversation)” on page 10-27
- “Associating WS-Policy Files at Runtime Using the Administration Console” on page 10-32
- “Using Security Assertion Markup Language (SAML) Tokens For Identity” on page 10-32
- “Using Only X.509 Certificate Tokens for Identity” on page 10-36
- “Using a Password Digest In the SOAP Message Rather Than Plaintext” on page 10-38
- “Invoking a Message-Secured Web Service From a Client Running in a WebLogic Server Instance” on page 10-44
- “Associating a Web Service with a Security Configuration Other Than the Default” on page 10-40

See “Using System Properties to Debug Message-Level Security” on page 10-40 for information on debugging problems with your message-secured Web Service.

**Ensuring That WebLogic Server Can Validate the Client’s Certificate**

You must ensure that WebLogic Server is able to validate the X.509 certificate that the client uses to digitally sign its SOAP request, and that WebLogic Server in turn uses to encrypt its SOAP responses to the client. Do one of the following:

- Ensure that the client application obtains a digital certificate that WebLogic Server automatically trusts, because it has been issued by a trusted certificate authority.
- Create a certificate registry which lists all the individual certificates trusted by WebLogic Server, and then ensure that the client uses one of these registered certificates.

For more information, see SSL Certificate Validation.

**Updating the JWS File with @Policy and @Policies Annotations**

Use the @Policy and @Policies annotations in your JWS file to specify that the Web Service has one or more WS-Policy files attached to it. You can use these annotations at either the class or method level.
The `@Policies` annotation simply groups two or more `@Policy` annotations together. Use the `@Policies` annotation if you want to attach two or more WS-Policy files to the class or method. If you want to attach just one WS-Policy file, you can use `@Policy` on its own.

The `@Policy` annotation specifies a single WS-Policy file, where it is located, whether the policy applies to the request or response SOAP message (or both), and whether to attach the WS-Policy file to the public WSDL of the service.

**WARNING:** As is true for all JWS annotations, the `@Policy` annotation cannot be overridden at runtime, which means that the WS-Policy file you specify at buildtime using the annotation will always be associated with the Web Service. This means, for example, that although you can view the associated WS-Policy file at runtime using the Administration Console, you cannot delete (unassociate) it. You can, however, associate additional WS-Policy files, as described in “Associating WS-Policy Files at Runtime Using the Administration Console” on page 10-32.

Use the `uri` attribute to specify the location of the WS-Policy file, as described below:

- To specify one of the five pre-packaged WS-Policy files that are installed with WebLogic Server, use the `policy: prefix and the name of one of the WS-Policy files (either Auth.xml, Encrypt.xml, Sign.xml, Wssc-dk.xml, or Wssc-sct.xml), as shown in the following example:

  ```java
  @Policy(uri="policy:Encrypt.xml")
  ```

  If you use the pre-packaged WS-Policy files, you do not have to create one yourself or package it in an accessible location. For this reason, BEA recommends that you use the pre-packaged WS-Policy files whenever you can.

  See “Using WS-Policy Files for Message-Level Security Configuration” on page 10-4 for information on the various types of message-level security provided by the pre-packaged WS-Policy files.

- To specify a user-created WS-Policy file, specify the path (relative to the location of the JWS file) along with its name, as shown in the following example:

  ```java
  @Policy(uri="../policies/MyPolicy.xml")
  ```

  In the example, the `MyPolicy.xml` file is located in the `policies` sibling directory of the one that contains the JWS file.

- You can also specify that a WS-Policy file that is located in a shared J2EE library; this method is useful if you want to share the file amongst multiple Web Services packaged in different J2EE archives.
In this case, it is assumed that the WS-Policy file is in the META-INF/policies or WEB-INF/policies directory of the shared J2EE library. Be sure, when you package the library, that you put the WS-Policy file in this directory.

To specify a WS-Policy file in a shared J2EE library, use the policy prefix and then the name of the WS-policy file, as shown in the following example:

```java
@Policy(uri="policy:MySharedPolicy.xml")
```

See Creating Shared J2EE Libraries and Optional Packages for information on creating shared libraries and setting up your environment so the Web Service can find the shared WS-Policy files.

You can also set the following attributes of the `@Policy` annotation:

- **direction**—Specifies whether the policy file should be applied to the request (inbound) SOAP message, the response (outbound) SOAP message, or both. The default value if you do not specify this attribute is both. The `direction` attribute accepts the following values:
  - Policy.Direction.both
  - Policy.Direction.inbound
  - Policy.Direction.outbound

- **attachToWsdl**—Specifies whether the policy file should be attached to the WSDL file that describes the public contract of the Web Service. The default value of this attribute is false. Abstract WS-Policy files cannot be attached at build time, but rather, they are attached at deploy time when the missing information is filled in by WebLogic Server.

The following example shows how to use the `@Policy` and `@Policies` JWS annotations, with the relevant sections shown in bold:

```java
package examples.webservices.security_jws;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Policies;
import weblogic.jws.Policy;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;

/**
 * @WebService(name="SecureHelloWorldPortType",
          serviceName="SecureHelloWorldService",
          targetNamespace="http://www.bea.com")
 */
Configuring Security

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

@WLHttpTransport(contextPath="SecureHelloWorldService",
    serviceUri="SecureHelloWorldService",
    portName="SecureHelloWorldServicePort")

@Policies(
    @Policy(uri="policy:Auth.xml", direction=Policy.Direction.inbound),
    @Policy(uri="policy:Sign.xml"),
    @Policy(uri="policy:Encrypt.xml"))

public class SecureHelloWorldImpl {

    @WebMethod()
    public String sayHello(String s) {
        return "Hello " + s;
    }
}
```

In the example, three WS-Policy files are attached to the Web Service at the class level, which means that all three WS-Policy files are applied to all public operations of the Web Service. The specified WS-Policy files are those pre-packaged with WebLogic Server, which means that the developers do not need to create their own files or package them in the corresponding archive.

The `Auth.xml` file is applied to only the request (inbound) SOAP message, as specified by the `direction` attribute. This means that only the client application needs to provide a username token; when WebLogic Server responds to the invoke, it does not provide a username token. The `Sign.xml` WS-Policy file specifies that the body and WebLogic system headers of both the request and response SOAP message be digitally signed. The `Encrypt.xml` policy file specifies that the body of both the request and response SOAP messages be encrypted.

**Using Key Pairs Other Than the Out-Of-The-Box SSL Pair**

In the simple message-level configuration procedure, documented in “Configuring Simple Message-Level Security: Main Steps” on page 10-14, it is assumed that the Web Services runtime uses the private key and X.509 certificate pair that is provided out-of-the-box with WebLogic Server; this same key pair is also used by the core security subsystem for SSL and is provided mostly for demonstration and testing purposes. In production environments, the Web Services runtime typically uses its own two private key and digital certificate pairs, one for signing and one for encrypting SOAP messages.

The following procedure describes the additional steps you must take to enable this use case.
1. Obtain two private key and digital certificate pairs to be used by the Web Services runtime. One of the pairs is used for digitally signing the SOAP message and the other for encrypting it. Although not required, BEA recommends that you obtain two pairs that will be used only by WebLogic Web Services. You must also ensure that both of the certificate’s key usage matches what you are configuring them to do. For example, if you are specifying that a certificate be used for encryption, be sure that the certificate’s key usage is specified as for encryption or is undefined. Otherwise, the Web Services security runtime will reject the certificate.

**WARNING:** BEA requires that the key length be 1024 bits or larger.

You can use the Cert Gen utility or Sun Microsystems's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

See [Obtaining Private Keys and Digital Signatures](#).

2. Create, if one does not currently exist, a custom identity keystore for WebLogic Server and load the private key and digital certificate pairs you obtained in the preceding step into the identity keystore.

   If you have already configured WebLogic Server for SSL, then you have already created a identity keystore which you can also use in this step.

   You can use WebLogic’s `ImportPrivateKey` utility and Sun Microsystem’s `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   See [Creating a Keystore and Loading Private Keys and Trusted Certificate Authorities Into the Keystore](#).

3. Using the Administration Console, configure WebLogic Server to locate the keystore you created in the preceding step. If you are using a keystore that has already been configured for WebLogic Server, you do not need to perform this step.

   See [Configuring Keystores for Production](#).

4. Using the Administration Console, create the default Web Service security configuration, which must be named `default_wss`. The default Web Service security configuration is used by all Web Services in the domain unless they have been explicitly programmed to use a different configuration.

   See [Create a Web Service security configuration](#).

5. Update the default Web Services security configuration you created in the preceding step to use one of the private key and digital certificate pairs for digitally signing SOAP messages.
See Specify the Key Pair Used to Sign SOAP Messages. In the procedure, when you create the properties used to identify the keystore and key pair, enter the exact value for the name of each property (such as IntegrityKeyStore, IntegrityKeyStorePassword, and so on), but enter the value that identifies your own previously-created keystore and key pair in the Value fields.

6. Similarly, update the default Web Services security configuration you created in a preceding step to use the second private key and digital certificate pair for encrypting SOAP messages.

See Create keystore used by SOAP message encryption. In the procedure, when you create the properties used to identify the keystore and key pair, enter the exact value for the name of each property (such as ConfidentialityKeyStore, ConfidentialityKeyStorePassword, and so on), but enter the value that identifies your own previously-created keystore and key pair in the Value fields.

**Setting the SOAP Message Expiration**

The `<MessageAge>` element in the WS-Policy file specifies whether SOAP messages resulting from an invoke of the Web Service associated with the WS-Policy file have an expiration. WebLogic Server rejects SOAP requests that have expired, based on their expiration time and the creation timestamp, which is included in the message. You can further configure expiration of messages by using the Administration Console to create and update the Web Services security configuration that is associated with the service.

The following bullets describe how the WebLogic Web Services runtime determines the expiration of a SOAP message for a particular Web Service:

- If the WS-Policy file associated with the Web Service does not include a `<MessageAge>` assertion, then the SOAP messages never expire.

- If the WS-Policy file includes a `<MessageAge>` assertion, but with no attributes, and the Web Service is not associated with a Web Service security configuration, then the expiration time is 60 seconds. If the Web Service is associated with a Web Service security configuration, then the expiration is the value of the `Validity Period` timestamp field of the associated Web Service security configuration (typically default_wss).

  The pre-packaged Sign.xml WS-Policy file falls into this category.

- If the WS-Policy file includes a `<MessageAge>` assertion with the `Age` attribute, then the expiration time is the value of the `Age` attribute. This value always overrides the value of the `Validity Period` field of any associated Web Service security configuration.

It is assumed in the following procedure that you have followed the steps in “Configuring Simple Message-Level Security: Main Steps” on page 10-14 and now want to set the message expiration.
To set the SOAP message expiration:

1. Ensure that the WS-Policy file associated with the Web Service includes a `<MessageAge>` assertion. The pre-packaged `Sign.xml` file includes one without any attributes.
   
   If you add a `<MessageAge>` assertion to a custom WS-Policy file, and specify the `Age` attribute, then you are done; the expiration is the value of this attribute and cannot be overridden with the Administration Console. If you do not specify an attribute because you want the default of 60 seconds, then you are also done. If, however, you want to change this default value, go to the next step.

2. Using the Administration Console, create (if you have not already done so) the default Web Service security configuration, which must be named `default_wss`. The default Web Service security configuration is used by all Web Services in the domain unless they have been explicitly programmed to use a different configuration.
   
   See Create a Web Service security configuration.

3. Update the default Web Services security configuration you created in the preceding step to specify a different expiration:
   
   a. In the left pane of the Administration Console, select `domain > Web Service Security`.
   
   b. Select `default_wss` in the Web Service Security Configuration table.
      
      a. Select `Web Service Security > Timestamp`.
      
      b. Update the **Validity Period** field with the new expiration time, in seconds.
      
      c. Optionally update the other fields. Click the Help link in the top right corner for detailed information about these fields.
      
      d. Click **Save**.

**Creating and Using a Custom WS-Policy File**

Although WebLogic Server includes five pre-packaged WS-Policy files that typically satisfy the security needs of most programmers, you can also create and use your own WS-Policy file if you need additional configuration. For example, you must create your own WS-Policy file if you want to:

- Use SAML tokens for authentication
• Specify that particular parts of the body of a SOAP message be encrypted or digitally signed, rather than the entire body, which is what the Encrypt.xml and Sign.xml pre-packaged WS-Policy files do.

• Configure security context and derived keys (part of the WS-SecureConversation specification) differently from the way they are configured in the pre-packaged files Wssc-dk.xml and Wssc-sct.xml.

See “Using WS-Policy Files for Message-Level Security Configuration” on page 10-4 for general information about WS-Policy files and how they are used for message-level security configuration.

When you create a custom WS-Policy file, you can separate out the three main security categories (authentication, encryption, and signing) into three separate WS-Policy files, as do the pre-packaged files, or create a single WS-Policy file that contains all three categories. You can also create a custom WS-Policy file that changes just one category (such as authentication) and use the pre-packaged files for the other categories (Sign.xml and Encrypt.xml). In other words, you can mix and match the number and content of the WS-Policy files that you associate with a Web Service. In this case, however, you must always ensure yourself that the multiple files do not contradict each other.

The root element of your WS-Policy file must be <Policy> and include the following namespace declarations:

```xml
<wsp:Policy
 xmlns:wssp="http://www.bea.com/wls90/security/policy"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
/>
```

**Defining Child Elements in a Custom WS-Policy File**

Define the following child elements of the <Policy> root element in your WS-Policy file (see Appendix D, “Security Policy Assertion Reference,” for complete reference information about the elements):

• <Identity>—Specifies the tokens that are supported for authentication. The <SupportedTokens> element groups one or more <SecurityTokens> elements for each type of supported tokens for identity: username, X.509, or SAML. Use the <Claims> element to specify the type of confirmation for SAML tokens (sender-vouches or holder-of-key) and to specify use of password digests when using username tokens.
● **<Confidentiality>**—Specifies what parts of the SOAP message must be encrypted. Optional child elements include: **<KeyWrappingAlgorithm>** to specify the algorithm used to wrap symmetric keys, **<Target>** to specify the blocks of the SOAP message that are encrypted, and **<KeyInfo>** to specify the tokens used for encryption (only X.509 tokens are supported.)

● **<Integrity>**—Specifies what parts of the SOAP message must be digitally signed. Optional child elements include: **<SignatureAlgorithm>** to specify the algorithm used to sign the message, **<CanonicalizationAlgorithm>** to specify the algorithm used for canonicalization, **<Target>** to specify the blocks of the SOAP message that are digitally signed, and **<SupportedTokens>** to specify the types of tokens that can be used for signing (only X.509 tokens are supported.)

● **<MessageAge>**—Specifies the maximum age, in seconds, of a SOAP message.

See “Example of a Custom WS-Policy File” on page 10-25 for an example of a custom WS-Policy file used to specify SAML tokens for identity. Because the **<Integrity>** and **<Confidentiality>** elements do not include **<KeyInfo>** and **<SupportedTokens>** child elements, respectively, these sections of the file are abstract. The **<Identity>** element does include the SAML token, so the identity section is concrete.

You can also use the abstract pre-packaged WS-Policy files as templates to create your own custom files. See:

- “Auth.xml” on page 10-6
- “Sign.xml” on page 10-7
- “Encrypt.xml” on page 10-8
- “Wssc-dk.xml” on page 10-9
- “Wssc-sct.xml” on page 10-11

**Example of a Custom WS-Policy File**

```xml
<?xml version="1.0"?>
<wsp:Policy
 xmlns:wssp="http://www.bea.com/wls90/security/policy"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
>
```
<wssp:Identity>
  <wssp:SupportedTokens>
    <wssp:SecurityToken
      TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID">"/>

    <wssp:Claims>
      <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
    </wssp:Claims>
  </wssp:SecurityToken>
  <wssp:SupportedTokens>
  </wssp:Identity>

<wssp:Integrity>
  <wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
  <wssp:CanonicalizationAlgorithm
    URI="http://www.w3.org/2001/10/xml-exc-c14n#"/>

  <wssp:Target>
    <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <wssp:MessageParts
      Dialect="http://schemas.xmlsoap.org/2002/12/wss#part">
      wsp:Body()
    </wssp:MessageParts>
  </wssp:Target>

  <wssp:Target>
    <wssp:DigestAlgorithm URI="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <wssp:MessageParts
      Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
      wls:SecurityHeader(Assertion)
    </wssp:MessageParts>
  </wssp:Target>
</wssp:Integrity>

<wssp:Confidentiality>
  <wssp:KeyWrappingAlgorithm URI="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>

  <wssp:Target>
    <wssp:EncryptionAlgorithm
      URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    <wssp:MessageParts
      Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
      wls:SecurityHeader(Assertion)
    </wssp:MessageParts>
  </wssp:Target>
</wssp:Confidentiality>
Configuring Simple Message-Level Security: Main Steps

```xml
<wssp:Target>
  <wssp:EncryptionAlgorithm
    URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
  <wssp:MessageParts
    Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
    wsp:Body()</wssp:MessageParts>
</wssp:Target>
<wssp:KeyInfo />
<wssp:Confidentiality>
</wssp:MessageAge/>
</wssp:Policy>
```

Configuring and Using Security Contexts and Derived Keys (WS-SecureConversation)

BEA provides two pre-packaged WS-Policy files (Wssc-dk.xml and Wssc-sct.xml) to configure security contexts and derived keys, as described by the WS-SecureConversation specification. It is highly recommended that you always use the pre-packaged files if you want to configure security contexts, because these two files provide most of the required functionality and typical default values. See “Wssc-dk.xml” on page 10-9 and “Wssc-sct.xml” on page 10-11 for the specific configuration provided by these files.

If, however, the values defined in these WS-Policy files are not adequate for your requirements, then you must create your own custom WS-Policy file. The following sections describe how to customize the configuration of secure contexts and derived keys and how a client application negotiates the security contexts with a Web Service.

**WARNING:** If you are deploying a Web Service that uses shared security contexts to a cluster, then you are required to also configure cross-cluster session state replication. For details, see Failover and Replication in a Cluster.

Using Custom WS-Policy Files To Configure Security Contexts and Derived Keys

If the pre-packaged WS-Policy files to configure security contexts are not adequate for your requirements, then you must create your own custom WS-Policy file and associate it with your Web Service using the @Policy annotation in your JWS file, or at runtime using the Administration Console. See “Creating and Using a Custom WS-Policy File” on page 10-23 for general information about creating custom WS-Policy files; the remainder of this section...
describes security context-specific information. BEA recommends you use the pre-packaged Wssc-dk.xml WS-Policy file as a guide and then customize it for your needs.

Follow these guidelines to create a custom security context WS-Policy file:

- **Use the** `SupportTrust20="true"` **attribute of the** `<wssp:Integrity>` **and** `<wssp:Confidentiality>` **assertions to specify that you want to configure shared security contexts when signing and encrypting:**

```xml
<wssp:Integrity SupportTrust10="true">
and
<wssp:Confidentiality SupportTrust10="true">
```

- **If you want to configure use of derived key token, set the** `TokenType` **attribute of the** `<wssp:SecurityToken>` **assertion (for both digitally signing and encryption) to** `http://schemas.xmlsoap.org/ws/2005/02/sc/dk` **and specify the** `DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct"` **attribute, as shown in the following example:**

```xml
<wssp:Confidentiality SupportTrust10="true">
  <wssp:Target>
    ...
  </wssp:Target>
  <wssp:KeyInfo>
    <wssp:SecurityToken IncludeInMessage="true"
      TokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
      DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
      ...
    </wssp:SecurityToken>
    </wssp:KeyInfo>
  </wssp:Confidentiality>
```

- **If you want to configure use of simple security context token, without using derived keys, set the** `TokenType` **attribute of the** `<wssp:SecurityToken>` **assertion (for both digitally signing and encryption) to** `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`, **as shown in the following example:**

```xml
<wssp:Confidentiality SupportTrust10="true">
  <wssp:Target>
    ...
  </wssp:Target>
  <wssp:KeyInfo>
    <wssp:SecurityToken>
      ...
    </wssp:SecurityToken>
  </wssp:KeyInfo>
</wssp:Confidentiality>
```
IncludeInMessage="true"
TokenTypet="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
</wssp:SecurityToken>
</wssp:KeyInfo>
</wssp:Confidentiality>

• The URI specified for the <wssp:SignatureAlgorithm> assertion of
<wssp:Integrity> must be http://www.w3.org/2000/09/xmldsig#hmac-sha1:
<wssp:SignatureAlgorithm SupportTrust10="true">
<wssp:SignatureAlgorithm URI="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
...

• You cannot specify the <wssp:KeyWrappingAlgorithm> assertion of
<wssp:Confidentiality>.

• Use child elements of the <wssp:Claims> assertion, which itself is a child element of
<wssp:SecurityToken> to further configure the security context token or derived key
<token>. In particular, use the <wssp:TokenLifeTime> assertion to change the lifetime, in
seconds, of the security context from the default 12 hours, <wssp:Length> assertion to
change the length of the key from the default 32, and <wssp:Label> assertion to set the
label, as shown in the following example:
<wssp:Confidentiality SupportTrust10="true">
<wssp:Target>
...
</wssp:Target>
<wssp:KeyInfo>
<wssp:SecurityToken IncludeInMessage="true"
TokenTypet="http://schemas.xmlsoap.org/ws/2005/02/sc/dk">
DerivedFromTokenType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct">
<wssp:Claims>
<wssp:TokenLifeTime>600</wssp:TokenLifeTime>
<wssp:Label>WS-SecureConversation</wssp:Label>
<wssp:Length>16</wssp:Length>
</wssp:Claims>
</wssp:SecurityToken>
</wssp:KeyInfo>
</wssp:Confidentiality>
Updating a Client Application to Negotiate Security Contexts

A client application that negotiates security contexts when invoking a Web Service is similar to a standard client application that invokes a message-secured Web Service, as described in “Updating a Client Application to Invoke a Message-Secured Web Service” on page 10-41. The only real difference is that you can use the weblogic.wsee.security.wssc.utils.WSSCClientUtil API to explicitly cancel the secure context token.

Note: WebLogic Server provides the WSSCClientUtil API for your convenience only; the Web Services runtime automatically cancels the secure context token when the configured timeout is reached. Use the API only if you want to have more control over when the token is cancelled.

The following client application shows a simple example of invoking a Web Service that is associated with the pre-packaged Wssc-dk.xml WS-Policy file that enables secure conversations; the sections in bold which are relevant to security contexts are discussed after the example:

```java
package examples.webservices.wssc.client;
import weblogic.security.SSL.TrustManager;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.bst.StubPropertyBSTCredProv;
import weblogic.wsee.security.wssc.utils.WSSCClientUtil;
import weblogic.wsee.security.util.CertUtils;
import javax.xml.rpc.Stub;
import java.util.List;
import java.util.ArrayList;
import java.security.cert.X509Certificate;

/**
 * Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */
public class WSSecureConvClient {
    public static void main(String[] args) throws Throwable {
        String clientKeyStore = args[0];
        String clientKeyStorePass = args[1];
        String clientKeyAlias = args[2];
        String clientKeyPass = args[3];
```
String serverCert = args[4];
String wsdl = args[5];

WSSecureConvService service = new WSSecureConvService_Impl(wsdl);
WSSecureConvPortType port = service.getWSSecureConvServicePort();

//create credential provider and set it to the Stub
List credProviders = new ArrayList();

//use x509 to secure wssc handshake
credProviders.add(new ClientBSTCredentialProvider(clientKeyStore,
clientKeyStorePass, clientKeyAlias, clientKeyPass));

Stub stub = (Stub)port;

stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
credProviders);
stub._setProperty(StubPropertyBSTCredProv.SERVER_ENCRYPT_CERT,
CertUtils.getCertificate(serverCert));

stub._setProperty(WSSecurityContext.TRUST_MANAGER,
new TrustManager(){
    public boolean certificateCallback(X509Certificate[] chain, int
    validateErr){
        //need to validate if the server cert can be trusted
        return true;
    }
});

System.out.println (port.sayHelloWithWSSC("Hello World, once"));
System.out.println (port.sayHelloWithWSSC("Hello World, twice"));
System.out.println (port.sayHelloWithWSSC("Hello World, thrice"));

//cancel SecureContextToken after done with invocation
WSSCClientUtil.terminateWssc(stub);
System.out.println("WSSC terminated!");

The points to notice in the preceding example are:

- Import the WebLogic API used to explicitly terminate the secure context token:
  import weblogic.wsee.security.wssc.utils.WSSCClientUtil;

- Set a property on the JAX-RPC stub which specifies that the client application must
  encrypt its request to WebLogic Server to cancel the secure context token with WebLogic
  Server’s public key:
stub._setProperty(StubPropertyBSTCredProv.SERVER_ENCRYPT_CERT, CertUtils.getCertificate(serverCert));

- Use the `terminateWssc()` method of the `WSSClientUtil` class to terminate the secure context token:
  
  ```java
  WSSClientUtil.terminateWssc(stub);
  ```

## Associating WS-Policy Files at Runtime Using the Administration Console

The simple message-level configuration procedure, documented in “Configuring Simple Message-Level Security: Main Steps” on page 10-14, describes how to use the `@Policy` and `@Policies` JWS annotations in the JWS file that implements your Web Service to specify one or more WS-Policy files that are associated with your service. This of course implies that you must already know, at the time you program your Web Service, which WS-Policy files you want to associate with your Web Service and its operations. This might not always be possible, which is why you can also associate WS-Policy files at runtime, after the Web Service has been deployed, using the Administration Console.

You can use no `@Policy` or `@Policies` JWS annotations at all in your JWS file and associate WS-Policy files only at runtime using the Administration Console, or you can specify some WS-Policy files using the annotations and then associate additional ones at runtime. However, once you associate a WS-Policy file using the JWS annotations, you cannot change this association at runtime using the Administration Console.

At runtime, the Administration Console allows you to associate as many WS-Policy files as you want with a Web Service and its operations, even if the policy assertions in the files contradict each other or contradict the assertions in WS-Policy files associated with the JWS annotations. It is up to you to ensure that multiple associated WS-Policy files work together. If any contradictions do exist, WebLogic Server returns a runtime error when a client application invokes the Web Service operation.

See Associate a WS-Policy file with a Web Service for detailed instructions on using the Administration Console to associate a WS-Policy file at runtime.

## Using Security Assertion Markup Language (SAML) Tokens For Identity

In the simple Web Services configuration procedure, described in “Configuring Simple Message-Level Security: Main Steps” on page 10-14, it is assumed that users use username
tokens to authenticate themselves. Because WebLogic Server implements the Web Services Security: SAML Token Profile of the Web Services Security specification, users can also use SAML tokens in the SOAP messages to authenticate themselves when invoking a Web Service operation, as described in this section.

Use of SAML tokens works server-to-server. This means that the client application is running inside of a WebLogic Server instance and then invokes a Web Service running in another WebLogic Server instance using SAML for identity. Because the client application is itself a Web Service, the Web Services security runtime takes care of all the SAML processing.

When you configure a Web Service to require SAML tokens for identity, you can specify one of the following confirmation methods:

- `sender-vouches`
- `holder-of-key`

See SAML Token Profile Support in WebLogic Web Services, as well as the Web Services Security: SAML Token Profile specification itself, for details about these confirmation methods.

**Note:** It is assumed in this section that you understand the basics of SAML and how it relates to core security in WebLogic Server. For general information, see Security Assertion Markup Language (SAML).

It is also assumed in the following procedure that you have followed the steps in “Configuring Simple Message-Level Security: Main Steps” on page 10-14 and now want to enable the additional use case of using SAML tokens, rather than username tokens, for identity.

To use SAML tokens for identity, follow these steps:

1. Using the Administration Console, configure a SAML identity assertion and credential mapping provider. This step configures the core WebLogic Server security subsystem. For details, see:
   - Configuring a SAML Identity Assertion Provider
   - Configuring a SAML Credential Mapping Provider

2. Create a custom WS-Policy file that specifies that SAML should be used for identity. The exact syntax depends on the type of confirmation method you want to configure (`sender-vouches` or `holder-of-key`).

   **To specify the sender-vouches confirmation method:**

   - Create a `<SecurityToken>` child element of the `<Identity><SupportedTokens>` elements and set the `TokenType` attribute to a value that indicates SAML token usage.
Configuring Security

- Add a `<Claims><Confirmationmethod>` child element of `<SecurityToken>` and specify `sender-vouches`.

For example:

```xml
<?xml version="1.0"?>
<wsp:Policy
    xmlns:wssp="http://www.bea.com/wls90/security/policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part">

    <wssp:Identity>
        <wssp:SupportedTokens>
            <wssp:SecurityToken
                TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID">
                <wssp:Claims>
                    <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
                </wssp:Claims>
            </wssp:SecurityToken>
        </wssp:SupportedTokens>
    </wssp:Identity>
</wsp:Policy>
```

To specify the holder-of-key confirmation method:

- Create a `<SecurityToken>` child element of the `<Integrity><SupportedTokens>` elements and set the `TokenType` attribute to a value that indicates SAML token usage.

The reason you put the SAML token in the `<Integrity>` assertion for the holder-of-key confirmation method is that the Web Service runtime must prove the integrity of the message, which is not required by `sender-vouches`.

- Add a `<Claims><Confirmationmethod>` child element of `<SecurityToken>` and specify `holder-of-key`.

For example:

```xml
<?xml version="1.0"?>
<wsp:Policy
    xmlns:wssp="http://www.bea.com/wls90/security/policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part">

    <wssp:Identity>
        <wssp:SupportedTokens>
            <wssp:SecurityToken
                TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID">
                <wssp:Claims>
                    <wssp:ConfirmationMethod>holder-of-key</wssp:ConfirmationMethod>
                </wssp:Claims>
            </wssp:SecurityToken>
        </wssp:SupportedTokens>
    </wssp:Identity>
</wsp:Policy>
```
By default, the WebLogic Web Services runtime always validates the X.509 certificate specified in the <KeyInfo> assertion of any associated WS-Policy file. To disable this validation when using SAML holder-of-key assertions, you must configure the Web Service security configuration associated with the Web service by setting a property on the SAML token handler. See Disable X.509 certificate validation when using SAML holder_of_key assertions for information on how to do this using the Administration Console.

3. Update the appropriate `@Policy` annotations in the JWS file that implements the Web Service to point to the custom WS-Policy file you created in the preceding step. For example, if you want invokes of *all* the operations of a Web Service to SAML for identity, specify the `@Policy` annotation at the class-level.

You can mix and match the WS-Policy files that you associate with a Web Service, as long as they do not contradict each other. For example, you can create a simple `MyAuth.xml` file that contains only the `<Identity>` security assertion to specify use of SAML for identity and then associate it with the Web Service together with the pre-packaged `Encrypt.xml` and `Sign.xml` files. It is, however, up to you to ensure that multiple associated WS-Policy files do not contradict each other; if they do, you will either receive a runtime error or the Web Service might not behave as you expect.

4. Recompile and redeploy your Web Service as part of the normal iterative development process.

See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.

5. Create a client application that runs in a WebLogic Server instance to invoke the main Web Service using SAML as identity. See “Invoking a Message-Secured Web Service From a Client Running in a WebLogic Server Instance” on page 10-44 for details.

**Using Only X.509 Certificate Tokens for Identity**

In the simple Web Services configuration procedure, described in “Configuring Simple Message-Level Security: Main Steps” on page 10-14, it is assumed that users use either username tokens, X.509 tokens, or both, to authenticate themselves. This is because it is assumed that users associate the `Auth.xml` pre-packaged WS-Policy file to their Web Service; because `Auth.xml` is abstract, the exact tokens that appear in the deployed WS-Policy file depend on how you have configured WebLogic Server, and thus it is undefined at the time you program the Web Service which tokens will actually be used for identity.

If, however, you want to specify that users authenticate themselves using only X.509 certificates when invoking a Web Service, and thus explicitly exclude username tokens from the WS-Policy file, then you cannot use `Auth.xml` and must instead create a custom WS-Policy file, as described in the following procedure. You must also use this procedure if you want to specify the parts of the SOAP message that are signed; by default, `Auth.xml`, when using X.509 for identity, also signs the entire SOAP body. Use the `<Integrity>` assertion to specify parts that should be signed.
Note: It is assumed in the following procedure that you have followed the steps in “Configuring Simple Message-Level Security: Main Steps” on page 10-14 and now want to enable the additional use case of using only X.509 certificates for identity.

1. Using the Administration Console, create (if you have not already done so) the default Web Service security configuration, which must be named default_wss. The default Web Service security configuration is used by all Web Services in the domain unless they have been explicitly programmed to use a different configuration.

See Create a Web Service security configuration.

2. Update the default Web Services security configuration you created in the preceding step to specify that X.509 certificates should be used for identity. See Use X.509 certificates to establish identity.

3. Create a custom WS-Policy file that specifies that X.509 certificates should be used for identity.

In particular, you must set the TokenType attribute of the <SecurityToken> child element of the <Identity><SupportedTokens> elements to http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3, as shown in the following simple example:

```xml
<?xml version="1.0"?>
           xmlns:wssp="http://www.bea.com/wls90/security/policy">
  <wssp:Identity>
    <wssp:SupportedTokens>
      <wssp:SecurityToken
          TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"/>
    </wssp:SupportedTokens>
  </wssp:Identity>
</wsp:Policy>
```

See “Creating and Using a Custom WS-Policy File” on page 10-23 for additional information about creating your own WS-Policy file. This section also includes information about adding <Integrity>.

4. Update the appropriate @Policy annotations in your JWS file to point to the custom WS-Policy file you created in the preceding step. For example, if you want invokes of all
the operations of a Web Service to use X.509 for identity, specify the @Policy annotation at the class-level.

You can mix and match the WS-Policy files that you associate with a Web Service, as long as they do not contradict each other. For example, you can create a simple MyAuth.xml file that contains only the <Identity> security assertion to specify use of X.509 certificates for identity and then associate it with the Web Service together with the pre-packaged Encrypt.xml and Sign.xml files. It is, however, up to you to ensure that multiple associated WS-Policy files do not contradict each other. If they do, you will receive a runtime error.

5. Recompile and redeploy your Web Service as part of the normal iterative development process.

See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.

6. You can use the same client application, described in “Updating a Client Application to Invoke a Message-Secured Web Service” on page 10-41, when using X.509 for identity. The only optional update is to remove the creation of the username token, which is not needed for this use case. The code to remove is:

```java
//client side UsernameToken credential provider
cp = new ClientUNTCredentialProvider(username, password);
credProviders.add(cp);
```

Using a Password Digest In the SOAP Message Rather Than Plaintext

By default, the WebLogic Web Services security runtime uses cleartext passwords, rather than the password digest, in the SOAP messages resulting from an invoke of a message-secured Web Service. The following procedure shows how to change this default behavior so that the SOAP messages use the password digest.

It is assumed in the following procedure that you have followed the steps in “Configuring Simple Message-Level Security: Main Steps” on page 10-14 and now want to specify that all SOAP messages use password digest rather than cleartext.

1. Using the Administration Console, create (if you have not already done so) the default Web Service security configuration, which must be named default_wss. The default Web Service security configuration is used by all Web Services in the domain unless they have been explicitly programmed to use a different configuration.
WARNING: If you have created Web Services security configuration in addition to the default one (default_wss), each configuration should specify the same password digest use. Inconsistent password digest use in different Web Service security configurations will result in a runtime error.

See Create a Web Service security configuration.

2. Update the default Web Services security configuration you created in the preceding step to specify that password digests should be used in SOAP messages. See Use a password digest in SOAP messages.

3. Update the default WebLogic Authentication provider of the core WebLogic Server security to store cleartext passwords rather than the digest. See Configure Authentication and Identity Assertion providers.

4. If you are not using the pre-packaged Auth.xml file and have instead created a custom WS-Policy file and have explicitly specified a username token with the <Identity><SupportedTokens><SecurityToken> elements, then you must add a <Claims><UsePassword> child element as shown below:

   <wssp:Identity>
     <wssp:SupportedTokens>
       <wssp:SecurityToken TokenType="#UsernameToken">
         <wssp:Claims>
           <wssp:UsePassword
           Type="http://www.docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest" />
         </wssp:Claims>
       </wssp:SecurityToken>
     </wssp:SupportedTokens>
   </wssp:Identity>

If you are using the pre-packaged Auth.xml file to configure authentication, you do not need to perform this step.

See “Creating and Using a Custom WS-Policy File” on page 10-23 for additional information about creating your own WS-Policy file.

5. If you created a custom WS-Policy file, update the appropriate @Policy annotations in your JWS file to point to it. See “Updating the JWS File with @Policy and @Policies Annotations” on page 10-17.
6. Recompile and redeploy your Web Service as part of the normal iterative development process.

   See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.

**Associating a Web Service with a Security Configuration Other Than the Default**

Many use cases previously discussed require you to use the Administration Console to create the default Web Service security configuration called default_wss. After you create this configuration, it is applied to all Web Services that either do not use the @weblogic.jws.security.WssConfiguration JWS annotation or specify the annotation with no attribute.

There are some cases, however, in which you might want to associate a Web Service with a security configuration other than the default; such use cases include specifying different timestamp values for different services.

To associate a Web Service with a security configuration other than the default:

1. **Create a Web Service security configuration** with a name that is not default_wss.

2. Update your JWS file, adding the @WssConfiguration annotation to specify the name of this security configuration. See “weblogic.jws.security.WssConfiguration” on page B-73 for additional information and an example.

3. Recompile and redeploy your Web Service as part of the normal iterative development process.

   See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.

**WARNING:** All Web Services security configurations are required to specify the same password digest use. Inconsistent password digest use in different Web Service security configurations will result in a runtime error.

**Using System Properties to Debug Message-Level Security**

The following table lists the system properties you can set to debug problems with your message-secured Web Service.
Updating a Client Application to Invoke a Message-Secured Web Service

When you update your Java code to invoke a message-secured Web Service, you must load a private key and digital certificate pair from the client’s keystore and pass this information, along with a username and password for user authentication if so required by the WS-Policy, to the secure WebLogic Web Service being invoked.

If the WS-Policy file of the Web Service specifies that the SOAP request must be encrypted, then the Web Services client runtime automatically gets the server’s certificate from the WS-Policy file that is attached to the WSDL of the service, and uses it for the encryption. If, however, the WS-Policy file is not attached to the WSDL, or the entire WSDL itself is not available, then the client application must use a client-side copy of the WS-Policy file; for details, see “Using a Client-Side Security WS-Policy File” on page 9-27.

The following example shows a Java client application that invokes the message-secured WebLogic Web Service described by the JWS file in “Updating the JWS File With the Security-Related Annotations” on page 10-50. The client application takes five arguments:

- Client username for client authentication
- Client password for client authentication
- Client private key file
- Client digital certificate

<table>
<thead>
<tr>
<th>System Property</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weblogic.xml.crypto.dsig.verbose</td>
<td>Boolean</td>
<td>Prints information about digital signature processing.</td>
</tr>
<tr>
<td>weblogic.xml.crypto.encrypt.verbose</td>
<td>Boolean</td>
<td>Prints information about encryption processing.</td>
</tr>
<tr>
<td>weblogic.xml.crypto.keyinfo.verbose</td>
<td>Boolean</td>
<td>Prints information about key resolution processing.</td>
</tr>
<tr>
<td>weblogic.xml.crypto.wss.verbose</td>
<td>Boolean</td>
<td>Prints information about Web Service security token and token reference processing.</td>
</tr>
</tbody>
</table>
Configuring Security

- WSDL of the deployed Web Service
  The security-specific code in the sample client application is shown in bold (and described after the example):

```java
package examples.webservices.security_jws.client;

import weblogic.security.SSL.TrustManager;
import weblogic.xml.crypto.wss.provider.CredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import javax.xml.rpc.Stub;
import java.util.List;
import java.util.ArrayList;
import java.security.cert.X509Certificate;

/**
 * Copyright (c) 2005 by BEA Systems. All Rights Reserved.
 */
public class SecureHelloWorldClient {
    public static void main(String[] args) throws Throwable {
        //username or password for the UsernameToken
        String username = args[0];
        String password = args[1];

        //client private key file
        String keyFile = args[2];

        //client certificate
        String clientCertFile = args[3];

        String wsdl = args[4];
        SecureHelloWorldService service = new SecureHelloWorldService_Impl(wsdl + "?WSDL");
        SecureHelloWorldPortType port = service.getSecureHelloWorldServicePort();

        //create credential provider and set it to the Stub
        List credProviders = new ArrayList();

        //client side BinarySecurityToken credential provider -- x509
        CredentialProvider cp = new ClientBSTCredentialProvider(clientCertFile, keyFile);
        credProviders.add(cp);
    }
}
```
//client side UsernameToken credential provider
cp = new ClientUNTCredentialProvider(username, password);
credProviders.add(cp);

Stub stub = (Stub)port;
stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
credProviders);

stub._setProperty(WSSecurityContext.TRUST_MANAGER,
new TrustManager(){
    public boolean certificateCallback(X509Certificate[] chain, int
    validateErr){
        return true;
    }
});

String response = port.sayHello("World");
System.out.println("response = " + response);
}

The main points to note about the preceding code are:

- Import the WebLogic security TrustManager API:
  
  import weblogic.security.SSL.TrustManager;

- Import the following WebLogic Web Services security APIs to create the needed client-side credential providers, as specified by the WS-Policy files that are associated with the Web Service:
  
  import weblogic.xml.crypto.wss.provider.CredentialProvider;
  import weblogic.xml.crypto.wss.WSSecurityContext;
  import weblogic.wsee.security.bst.ClientBSTCredentialProvider;
  import weblogic.wsee.security.unt.ClientUNTCredentialProvider;

- Use the ClientBSTCredentialProvider WebLogic API to create a binary security token credential provider from the client’s certificate and private key:
  
  CredentialProvider cp =
  new ClientBSTCredentialProvider(clientCertFile, keyFile);

- Use the ClientUNTCredentialProvider WebLogic API to create a username token from the client’s username and password, which are also known by WebLogic Server:
  
  cp = new ClientUNTCredentialProvider(username, password);

- Use the WSSecurityContext.CREDENTIAL_PROVIDER_LIST property to pass a List object that contains the binary security and username tokens to the JAX-RPC Stub:
stub._setProperty(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders)

- Use the weblogic.security.SSL.TrustManager WebLogic security API to verify that the certificate used to encrypt the SOAP request is valid. The Web Services client runtime gets this certificate from the deployed WSDL of the Web Service, which in production situations is not automatically trusted, so the client application must ensure that it is okay before it uses it to encrypt the SOAP request:

  stub._setProperty(WSSecurityContext.TRUST_MANAGER,
  new TrustManager(){
    public boolean certificateCallback(X509Certificate[] chain, int validateErr){
      return true;
    }
  });

Invoking a Message-Secured Web Service From a Client Running in a WebLogic Server Instance

In the simple Web Services configuration procedure, described in “Configuring Simple Message-Level Security: Main Steps” on page 10-14, it is assumed that a stand-alone client application invokes the message-secured Web Service. Sometimes, however, the client is itself running in a WebLogic Server instance, as part of an EJB, servlet, or another Web Service. In this case, you can use the core WebLogic Server security framework to configure the credential providers and trust manager so that your EJB, servlet, or JWS code contains only the simple invoke of the secured operation and no other security-related API usage. The following procedure describes the high level steps you must perform to make use of the core WebLogic Server security framework in this use case.

1. In your EJB, servlet, or JWS code, invoke the Web Service operation as if it were not configured for message-level security. Specifically, do not create a CredentialProvider object that contains username or X.509 tokens, and do not use the TrustManager core security API to validate the certificate from the WebLogic Server hosting the secure Web Service. The reason you should not use these APIs in your client code is that the Web Services runtime will perform this work for you.

2. Using the Administration Console, configure the required credential mapping providers of the core security of the WebLogic Server instance that hosts your client application. The list of required credential mapper providers depends on the WS-Policy file that is attached to the Web Service you are invoking. Typically, you must configure the credential mapper providers for both username/password and X.509 certificates. See Configuring a WebLogic Credential Mapping Provider.
Configuring Transport-Level Security

**Note:** WebLogic Server includes a credential mapping provider for username/passwords and X.509. However, only username/password is configured by default.

3. Using the Administration Console, create the actual credential mappings in the credential mapping providers you configured in the preceding step. You must map the user principal, associated with the client running in the server, to the credentials that are valid for the Web Service you are invoking. See Configuring a WebLogic Credential Mapping Provider.

4. Using the Administration Console, configure the core WebLogic Server security framework to trust the X.509 certificate of the invoked Web Service. See Configuring the Credential Lookup and Validation Framework.

You are not required to configure the core WebLogic Server security framework, as described in this procedure, if your client application does not want to use the out-of-the-box credential provider and trust manager. Rather, you can override all of this configuration by using the same APIs in your EJB, servlet, and JWS code as in the stand-alone Java code described in “Updating a Client Application to Invoke a Message-Secured Web Service” on page 10-41. However, using the core security framework standardizes the WebLogic Server configuration and simplifies the Java code of the client application that invokes the Web Service.

## Configuring Transport-Level Security

Transport-level security refers to securing the connection between a client application and a Web Service with Secure Sockets Layer (SSL).

See Secure Sockets Layer (SSL) for general information about SSL and the implementations included in WebLogic Server.

To configure transport-level Web Services security:

1. Configure SSL for the core WebLogic Server security subsystem.
   - You can configure one-way SSL where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.
   - To configure two-way or one-way SSL for the core WebLogic Server security subsystem, see Configuring SSL.

2. In the JWS file that implements your Web Service, add the @weblogic.jws.security.UserDataConstraint annotation to require that the Web Service be invoked using the HTTPS transport.
   - For details, see “weblogic.jws.security.UserDataConstraint” on page B-71.
3. Recompile and redeploy your Web Service as part of the normal iterative development process.

See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.

4. Update the build.xml file that invokes the clientgen Ant task to use a static WSDL to generate the JAX-RPC stubs of the Web Service, rather than the dynamic deployed WSDL of the service.

The reason clientgen cannot generate the stubs from the dynamic WSDL in this case is that when you specify the @UserDataConstraint annotation, all client applications are required to specify a truststore, including clientgen. However, there is currently no way for clientgen to specify a truststore, thus the Ant task must generate its client components from a static WSDL that describes the Web Service in the same way as the dynamic WSDL.

5. When you run the client application that invokes the Web Service, specify certain properties to indicate the SSL implementation that your application should use. In particular:

- To specify the Certicom SSL implementation, use the following properties
  -Djava.protocol.handler.pkgs=weblogic.net
  -Dweblogic.security.SSL.trustedCAKeyStore=trustStore

  where trustStore specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server’s certificate). To disable host name verification, also specify the following property:
  -Dweblogic.security.SSL.ignoreHostnameVerification=true

- To specify Sun’s SSL implementation, use the following properties:
  -Djavax.net.ssl.trustStore=trustStore

  where trustStore specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server’s certificate). To disable host name verification, also specify the following property:
  -Dweblogic.wsee.client.ssl.stricthostchecking=false

See “Configuring Two-Way SSL for a Client Application” on page 10-47 for details about two-way SSL.
Configuring Two-Way SSL for a Client Application

If you configured two-way SSL for WebLogic Server, the client application must present a certificate to WebLogic Server, in addition to WebLogic Server presenting a certificate to the client application as required by one-way SSL. You must also follow these requirements:

- Create a client-side keystore that contains the client’s private key and X.509 certificate pair.
  The SSL package of J2SE requires that the password of the client’s private key must be the same as the password of the client’s keystore. For this reason, the client keystore can include only one private key and X.509 certificate pair.
- Configure the core WebLogic Server’s security subsystem, mapping the client’s X.509 certificate in the client keystore to a user. See Configuring a User Name Mapper.
- Create a truststore which contains the certificates that the client trusts; the client application uses this truststore to validate the certificate it receives from WebLogic Server. Because of the J2SE password requirement described in the preceding bullet item, this truststore must be different from the keystore that contains the key pair that the client presents to the server.

You can use the Cert Gen utility or Sun Microsystem's keytool utility to perform this step. For development purposes, the keytool utility is the easiest way to get started.

See Obtaining Private Keys and Digital Signatures at {DOCROOT}/secmanage/identity_trust.html#get_keys_certs_trustedcas.

- When you run the client application that invokes the Web Service, specify the following properties:
  - -Djavax.net.ssl.trustStore=trustStore
  - -Djavax.net.ssl.trustStorePassword=trustStorePassword

  where trustStore specifies the name of the client-side truststore that contains the list of trusted certificates (one of which should be the server’s certificate) and trustStorePassword specifies the truststore’s password.

  The preceding properties are in addition to the standard properties you must set to specify the client-side keystore:
  - -Djavax.net.ssl.keyStore=keyStore
  - -Djavax.net.ssl.keyStorePassword=keyStorePassword
Additional Web Services SSL Examples

The dev2dev CodeShare is a community of developers that share ideas, code and best practices related to BEA technologies. The site includes code examples for a variety of BEA technologies, including using SSL with Web Services.

To view and download the SSL Web Services code examples on the dev2dev site, go to the main Projects page and click on Web Services in the By Technology column.

Configuring Access Control Security: Main Steps

Access control security refers to configuring the Web Service to control the users who are allowed to access it, and then coding your client application to authenticate itself, using HTTP/S or username tokens, to the Web Service when the client invokes one of its operations.

You specify access control security for your Web Service by using one or more of the following annotations in your JWS file:

- weblogic.jws.security.RolesAllowed
- weblogic.jws.security.SecurityRole
- weblogic.jws.security.RolesReferenced
- weblogic.jws.security.SecurityRoleRef
- weblogic.jws.security.RunAs

Note: The @weblogic.security.jws.SecurityRoles and @weblogic.security.jws.SecurityIdentity JWS annotations are deprecated as of WebLogic Server 9.1.

The following procedure describes the high-level steps to use these annotations to enable access control security; later sections in the chapter describe the steps in more detail.

Note: It is assumed in the following procedure that you have already created a JWS file that implements a WebLogic Web Service and you want to update it with access control security. It is also assumed that you use Ant build scripts to iteratively develop your Web Service and that you have a working build.xml file that you can update with new information. Finally, it is assumed that you have a client application that invokes the non-secured Web Service. If these assumptions are not true, see:

- Chapter 5, “Programming the JWS File”
- Chapter 4, “Iterative Development of WebLogic Web Services”
- Chapter 9, “Invoking Web Services”
1. Update your JWS file, adding the \texttt{@weblogic.jws.security.RolesAllowed}, \texttt{@weblogic.jws.security.SecurityRole}, \texttt{@weblogic.jws.security.RolesReferenced}, or \texttt{@weblogic.jws.security.SecurityRoleRef} annotations as needed at the appropriate level (class or operation).

   See “Updating the JWS File With the Security-Related Annotations” on page 10-50.

2. Optionally specify that WebLogic Server internally run the Web Service using a specific role, rather than the role assigned to the user who actually invokes the Web Service, by adding the \texttt{@weblogic.jws.security.RunAs} JWS annotation.

   See “Updating the JWS File With the \texttt{@RunAs} Annotation” on page 10-52.

3. Optionally specify that your Web Service can be, or is required to be, invoked using HTTPS by adding the \texttt{@weblogic.jws.security.UserDataConstraint} JWS annotation.

   See “Configuring Transport-Level Security” on page 10-45 for details. This section also discusses how to update your client application to use SSL.

4. Recompile and redeploy your Web Service as part of the normal iterative development process.

   See “Iterative Development of WebLogic Web Services Starting From Java: Main Steps” on page 4-3.

5. Using the Administration Console, create valid WebLogic Server users, if they do not already exist. If the mapping of users to roles is external, also use the Administration Console to create the roles specified by the \texttt{@SecurityRole} annotation and map the users to the roles.

   \textbf{Note:} The mapping of users to roles is defined externally if you do \textit{not} specify the \texttt{mapToPrincipals} attribute of the \texttt{@SecurityRole} annotation in your JWS file to list all users who can invoke the Web Service.

   See Users, Groups, and Security Roles at \{DOCROOT\}/secwlres/secroles.html.

6. Update your client application to use the \texttt{HttpTransportInfo} WebLogic API to specify the appropriate user and password when creating the JAX-RPC Service object.

   See “Setting the Username and Password When Creating the JAX-RPC Service Object” on page 10-53.

7. Update the \texttt{clientgen} Ant task in your \texttt{build.xml} file to specify the username and password of a valid WebLogic user (in the case where your Web Service uses the \texttt{@RolesAllowed} annotation) and the trust store that contains the list of trusted certificates, including WebLogic Server’s (in the case you specify \texttt{@UserDataConstraint}).
You do this by adding the standard Ant `<sysproperty>` nested element to the `clientgen` Ant task, and set the `key` attribute to the required Java property, as shown in the following example:

```xml
<clientgen
    wsdl="http://example.com/myapp/myservice.wsdl"
    destDir="/output/clientclasses"
    packageName="myapp.myservice.client"
    serviceName="StockQuoteService"
    <sysproperty
        key="javax.net.ssl.trustStore"
        value="/keystores/DemoTrust.jks"/>
    <sysproperty
        key="weblogic.wsee.client.ssl.stricthostchecking"
        value="false"/>
    <sysproperty
        key="javax.xml.rpc.security.auth.username"
        value="juliet"/>
    <sysproperty
        key="javax.xml.rpc.security.auth.password"
        value="secret"/>
</clientgen>
```

8. Regenerate client-side components and recompile client Java code as usual.

### Updating the JWS File With the Security-Related Annotations

Use the WebLogic-specific `@weblogic.jws.security.RolesAllowed` annotation in your JWS file to specify an array of `@weblogic.jws.security.SecurityRoles` annotations that list the roles that are allowed to invoke the Web Service. You can specify these two annotations at either the class- or method-level. When set at the class-level, the roles apply to all public operations. You can add additional roles to a particular operation by specifying the annotation at the method level.

The `@SecurityRole` annotation has the following two attributes:

- **role**—Name of the role that is allowed to invoke the Web Service.

- **mapToPrincipals**—List of users that map to the role. If you specify one or more users with this attribute, you do not have to externally create the mapping between users and roles, typically using the Administration Console. However, the mapping specified with this attribute applies only within the context of the Web Service.

The `@RolesAllowed` annotation does not have any attributes.

You can also use the `@weblogic.jws.security.RolesReferenced` annotation to specify an array of `@weblogic.jws.security.SecurityRoleRef` annotations that list references to existing roles. For example, if the role `manager` is already allowed to invoke the Web Service,
you can specify that the `mgr` role be linked to the `manager` role and any user mapped to `mgr` is also able to invoke the Web Service. You can specify these two annotations only at the class-level.

The `@SecurityRoleRef` annotation has the following two attributes:

- **role**—Name of the role reference.
- **link**—Name of the already-specified role that is allowed to invoke the Web Service. The value of this attribute corresponds to the value of the `role` attribute of a `@SecurityRole` annotation specified in the same JWS file.

The `@RolesReferenced` annotation does not have any attributes.

The following example shows how to use the annotations described in this section in a JWS file, with the relevant sections shown in bold:

```java
package examples.webservices.security_roles;
import javax.jws.WebMethod;
import javax.jws.WebService;
// WebLogic JWS annotations
import weblogic.jws.WLHttpTransport;
import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.RolesReferenced;
import weblogic.jws.security.SecurityRole;
import weblogic.jws.security.SecurityRoleRef;

@WebService(name="SecurityRolesPortType",
    serviceName="SecurityRolesService",
    targetNamespace="http://example.org")
@WLHttpTransport(contextPath="security",
    serviceUri="SecurityRolesService",
    portName="SecurityRolesPort")
@RolesAllowed ( {  
    @SecurityRole (role="manager",
        mapTo Principals={ "juliet","amanda" } ),
    @SecurityRole (role="vp")
} )
```
Configuring Security

```java
@RolesReferenced {
    @SecurityRoleRef (role="mgr", link="manager")
}

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 */
public class SecurityRolesImpl {
    @WebMethod()
    public String sayHello(String message) {
        System.out.println("sayHello:" + message);
        return "Here is the message: \'" + message + "\'";
    }
}
```

The example shows how to specify that only the manager, vp, and mgr roles are allowed to invoke the Web Service. The mgr role is actually a reference to the manager role. The users juliet and amanda are mapped to the manager role within the context of the Web Service. Because no users are mapped to the vp role, it is assumed that the mapping occurs externally, typically using the Administration Console to update the WebLogic Server security realm.

See Appendix B, “JWS Annotation Reference,” for reference information on these annotations.

### Updating the JWS File With the @RunAs Annotation

Use the WebLogic-specific `@weblogic.jws.security.RunAs` annotation in your JWS file to specify that the Web Service is always run as a particular role. This means that regardless of the user, and the role to which the user is mapped, initially invokes the Web Service, the service is internally executed as the specified role.

You can set the `@RunAs` annotation only at the class-level. The annotation has the following attributes:

- **role**—Role which the Web Service should run as.
- **mapToPrincipal**—Principal user that maps to the role.
The following example shows how to use the @RunAs annotation in a JWS file, with the relevant sections shown in bold:

```java
package examples.webservices.security_roles;
import javax.jws.WebMethod;
import javax.jws.WebService;
// WebLogic JWS annotations
import weblogic.jws.WLHttpTransport;
import weblogic.jws.security.RunAs;

@WebService(name="SecurityRunAsPortType",
serviceName="SecurityRunAsService",
targetNamespace="http://example.org")
@WLHttpTransport(contextPath="security_runas",
serviceUri="SecurityRunAsService",
portName="SecurityRunAsPort")
@RunAs (role="manager", mapToPrincipal="juliet")
/**
 * This JWS file forms the basis of simple WebLogic
 * Web Service with a single operation: sayHello
 * *
 */
public class SecurityRunAsImpl {
    @WebMethod()
    public String sayHello(String message) {
        System.out.println("sayHello:" + message);
        return "Here is the message: '" + message + '"';
    }
}
```

### Setting the Username and Password When Creating the JAX-RPC Service Object

When you use the @RolesAllowed JWS annotation to secure a Web Service, only the specified roles are allowed to invoke the Web Service operations. This means that you must specify the
username and password of a user that maps to the role when creating the JAX-RPC Service object in your client application that invokes the protected Web Service.

WebLogic Server provides the HttpTransportInfo class for setting the username and password and passing it to the Service constructor. The following example is based on the standard way to invoke a Web Service from a standalone Java client (as described in Chapter 9, “Invoking Web Services”) but also shows how to use the HttpTransportInfo class to set the username and password. The sections in bold are discussed after the example.

```
package examples.webservices.sec_wsdl.client;

import weblogic.wsee.connection.transport.http.HttpTransportInfo;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

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

    public static void main(String[] args)
            throws ServiceException, RemoteException{
            HttpTransportInfo info = new HttpTransportInfo();
            info.setUsername("juliet".getBytes());
            info.setPassword("secret".getBytes());

            SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL",
                    info);
            SecWsdlPortType port = service.getSecWsdlPort();

            try {
                String result = null;
                result = port.sayHello("Hi there!");
                System.out.println("Got result: " + result);
            } catch (RemoteException e) {
                throw e;
            }
        }
    }
```

WebLogic Server provides the HttpTransportInfo class for setting the username and password and passing it to the Service constructor. The following example is based on the standard way to invoke a Web Service from a standalone Java client (as described in Chapter 9, “Invoking Web Services”) but also shows how to use the HttpTransportInfo class to set the username and password. The sections in bold are discussed after the example.

```
package examples.webservices.sec_wsdl.client;

import weblogic.wsee.connection.transport.http.HttpTransportInfo;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

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

    public static void main(String[] args)
            throws ServiceException, RemoteException{
            HttpTransportInfo info = new HttpTransportInfo();
            info.setUsername("juliet".getBytes());
            info.setPassword("secret".getBytes());

            SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL",
                    info);
            SecWsdlPortType port = service.getSecWsdlPort();

            try {
                String result = null;
                result = port.sayHello("Hi there!");
                System.out.println("Got result: " + result);
            } catch (RemoteException e) {
                throw e;
            }
        }
    }
```
Configuring Access Control Security: Main Steps

The main points to note in the preceding example are as follows:

- **Import the `HttpTransportInfo` class into your client application:**
  
  ```java
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

- **Use the `setXXX()` methods of the `HttpTransportInfo` class to set the username and password:**
  
  ```java
  HttpTransportInfo info = new HttpTransportInfo();
  info.setUsername("juliet").getBytes());
  info.setPassword("secret").getBytes());
  ```

  In the example, it is assumed that the user `juliet` with password `secret` is a valid WebLogic Server user and has been mapped to the role specified in the `@RolesAllowed` JWS annotation of the Web Service.

  If you are accessing a Web Service using a proxy, the Java code would be similar to:

  ```java
  HttpTransportInfo info = new HttpTransportInfo();
  Proxy p = new Proxy(Proxy.Type.HTTP, new InetSocketAddress(proxyHost,
  Integer.parseInt(proxyPort)));
  info.setProxy(p); 
  info.setProxyUsername(user.getBytes()); 
  info.setProxyPassword(pass.getBytes());
  ```

- **Pass the `info` object that contains the username and password to the `Service` constructor as the second argument, in addition to the standard WSDL first argument:**

  ```java
  SecWsdlService service = new SecWsdlService_Impl(args[0] + "?WSDL", info);
  ```

Configuring Security
Administering Web Services

The following sections describe how to administer WebLogic Web Services:

- “Overview of WebLogic Web Services Administration Tasks” on page 11-1
- “Administration Tools” on page 11-2
- “Using the Administration Console” on page 11-3
- “Using the WebLogic Scripting Tool” on page 11-8
- “Using WebLogic Ant Tasks” on page 11-8
- “Using the Java Management Extensions (JMX)” on page 11-8
- “Using the J2EE Deployment API” on page 11-9
- “Using Work Managers to Prioritize Web Services Work and Reduce Stuck Execute Threads” on page 11-10

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 J2EE 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 J2EE modules and applications that run on WebLogic Server, including Web Services; use the tool that best fits your needs:

- Using the Administration Console
- Using the WebLogic Scripting Tool
- Using WebLogic Ant Tasks
Using the 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.
Web Services are typically deployed to WebLogic Server as part of an Enterprise Application. The Enterprise Application can be either packaged 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, \texttt{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 \textbf{Web Services} category. Click on the name of the Web Service to view or update its configuration.

The following figure shows how the \texttt{HandlerChainService} Web Service, packaged inside the \texttt{GlobalHandler} Enterprise application, is displayed in the \textbf{Deployments} table of the Administration Console.

\textbf{Figure 11-2} \hspace{1em} 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 11-3  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 J2EE modules and applications (including Web Services) to a particular WebLogic Server instance. Using WLST, system administrators and operators can initiate, manage, and persist WebLogic Server configuration changes.

Typically, the types of WLST commands you use to administer Web Services fall under the Deployment category.

For more information on using WLST, see WebLogic Scripting Tool at {DOCROOT}/config_scripting/index.html.

Using WebLogic Ant Tasks

WebLogic Server includes a variety of Ant tasks that you can use to centralize many of the configuration and administrative tasks into a single Ant build script. These Ant tasks can:

- 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 J2EE 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:
Using the J2EE Deployment API

In J2EE 1.4, the J2EE Application Deployment specification (JSR-88) defines a standard API that you can use to configure an application for deployment to a target application server environment.

The specification describes the J2EE 1.4 Deployment architecture, which in turn defines the contracts that enable tools or application programmers to configure and deploy applications on any J2EE platform product. The contracts define a uniform model between tools and J2EE platform products for application deployment configuration and deployment. The Deployment architecture makes it easier to deploy applications: Deployers do not have to learn all the features of many different J2EE deployment tools in order to deploy an application on many different J2EE platform products.
See Deploying Applications to WebLogic Server for more information.

**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.

For more information about Work Managers and how to configure them for your Web Service, see Using Work Managers to Optimize Scheduled Work.
The following sections provide information about publishing and finding Web Services through the UDDI registry:

- “Overview of UDDI” on page 12-1
- “WebLogic Server UDDI Features” on page 12-4
- “UDDI 2.0 Server” on page 12-5
- “UDDI Directory Explorer” on page 12-20
- “UDDI Client API” on page 12-20
- “Pluggable tModel” on page 12-21

Overview of UDDI

UDDI stands for Universal Description, Discovery, and Integration. The UDDI Project is an industry initiative aims to enable businesses to quickly, easily, and dynamically find and carry out transactions with one another.

A populated UDDI registry contains cataloged information about businesses; the services that they offer; and communication standards and interfaces they use to conduct transactions.

Built on the Simple Object Access Protocol (SOAP) data communication standard, UDDI creates a global, platform-independent, open architecture space that will benefit businesses.

The UDDI registry can be broadly divided into two categories:
UDDI and Web Services

The owners of Web Services publish them to the UDDI registry. Once published, the UDDI registry maintains pointers to the Web Service description and to the service.

The UDDI allows clients to search this registry, find the intended service, and retrieve its details. These details include the service invocation point as well as other information to help identify the service and its functionality.

Web Service capabilities are exposed through a programming interface, and usually explained through Web Services Description Language (WSDL). In a typical publish-and-inquire scenario, the provider publishes its business; registers a service under it; and defines a binding template with technical information on its Web Service. The binding template also holds reference to one or several tModels, which represent abstract interfaces implemented by the Web Service. The tModels might have been uniquely published by the provider, with information on the interfaces and URL references to the WSDL document.

A typical client inquiry may have one of two objectives:

- To find an implementation of a known interface. In other words, the client has a tModel ID and seeks binding templates referencing that tModel.
- To find the updated value of the invocation point (that is, access point) of a known binding template ID.

UDDI and Business Registry

As a Business Registry solution, UDDI enables companies to advertise the business products and services they provide, as well as how they conduct business transactions on the Web. This use of UDDI complements business-to-business (B2B) electronic commerce.

The minimum required information to publish a business is a single business name. Once completed, a full description of a business entity may contain a wealth of information, all of which helps to advertise the business entity and its products and services in a precise and accessible manner.

A Business Registry can contain:
• **Business Identification**—Multiple names and descriptions of the business, comprehensive contact information, and standard business identifiers such as a tax identifier.

• **Categories**—Standard categorization information (for example a D-U-N-S business category number).

• **Service Description**—Multiple names and descriptions of a service. As a container for service information, companies can advertise numerous services, while clearly displaying the ownership of services. The bindingTemplate information describes how to access the service.

• **Standards Compliance**—In some cases it is important to specify compliance with standards. These standards might display detailed technical requirements on how to use the service.

• **Custom Categories**—It is possible to publish proprietary specifications (tModels) that identify or categorize businesses or services.

**UDDI Data Structure**

The data structure within UDDI consists of four constructions: a businessEntity structure, a businessService structure, a bindingTemplate structure and a tModel structure.

The following table outlines the difference between these constructions when used for Web Service or Business Registry applications.
WebLogic Server UDDI Features

WebLogic Server provides the following UDDI features:

- UDDI 2.0 Server
- UDDI Directory Explorer

Table 12-1  UDDI Data Structure

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Web Service</th>
<th>Business Registry</th>
</tr>
</thead>
</table>
| businessEntity | Represents a Web Service provider:  
• Company name  
• Contact detail  
• Other business information | Represents a company, a division or a department within a company:  
• Company name(s)  
• Contact details  
• Identifiers and Categories |
| businessService | A logical group of one or several Web Services.  
API(s) with a single name stored as a child element, contained by the business entity named above. | A group of services may reside in a single businessEntity.  
• Multiple names and descriptions  
• Categories  
• Indicators of compliance with standards |
| bindingTemplate | A single Web Service.  
Technical information needed by client applications to bind and interact with the target Web Service.  
Contains access point (that is, the URI to invoke a Web Service). | Further instances of standards conformity.  
Access points for the service in form of URLs, phone numbers, email addresses, fax numbers or other similar address types. |
| tModel | Represents a technical specification; typically a specifications pointer, or metadata about a specification document, including a name and a URL pointing to the actual specifications. In the context of Web Services, the actual specifications document is presented in the form of a WSDL file. | Represents a standard or technical specification, either well established or registered by a user for specific use. |
The UDDI 2.0 Server is part of WebLogic Server and is started automatically when WebLogic Server is started. The UDDI Server implements the UDDI 2.0 server specification at http://www.uddi.org/specification.html.

**Configuring the UDDI 2.0 Server**

To configure the UDDI 2.0 Server:

1. Stop WebLogic Server.

2. Update the uddi.properties file, located in the WL_HOME/server/lib directory, where WL_HOME refers to the main WebLogic Server installation directory.

   **WARNING:** If your WebLogic Server domain was created by a user different from the user that installed WebLogic Server, the WebLogic Server administrator must change the permissions on the uddi.properties file to give access to all users.


   Never edit the uddi.properties file while WebLogic Server is running. Should you modify this file in a way that prevents the successful startup of the UDDI Server, refer to the WL_HOME/server/lib/uddi.properties.booted file for the last known good configuration.

   To restore your configuration to its default, remove the uddi.properties file from the WL_HOME/server/lib directory. BEA strongly recommends that you move this file to a backup location, because a new uddi.properties file will be created and with its successful startup, the uddi.properties.booted file will also be overwritten. After removing the properties file, start the server. Minimal default properties will be loaded and written to a newly created uddi.properties file.

   The following section describes the UDDI Server properties that you can include in the uddi.properties file. The list of properties has been divided according to component, usage, and functionality. At any given time, you do not need all these properties to be present.
Configuring an External LDAP Server

The UDDI 2.0 Server is automatically configured with an embedded LDAP server. You can, however, also configure an external LDAP Server by following the procedure in this section.

Note: Currently, WebLogic Server supports only the SunOne Directory Server for use with the UDDI 2.0 Server.

To configure the SunOne Directory Server to be used with UDDI, follow these steps:

1. Create a file called 51acumen.ldif in the
   LDAP_DIR/Sun/MPS/slapd-LDAP_INSTANCE_NAME/config/schema directory, where
   LDAP_DIR refers to the root installation directory of your SunOne Directory Server and
   LDAP_INSTANCE_NAME refers to the instance name.

2. Update the 51acumen.ldif file with the content described in “51acumen.ldif File Contents” on page 12-6.

3. Restart the SunOne Directory Server.

4. Update the uddi.properties file of the WebLogic UDDI 2.0 Server, adding the following properties:
   
   datasource.ldap.manager.password
   datasource.ldap.manager.uid
   datasource.ldap.server.root
   datasource.ldap.server.url
   
   The value of the properties depends on the configuration of your SunOne Directory Server. The following example shows a possible configuration that uses default values:
   
   datasource.ldap.manager.password=password
   datasource.ldap.manager.uid=cn=Directory Manager
   datasource.ldap.server.root=dc=beasys,dc=com
   datasource.ldap.server.url=ldap://host:port
   
   See Table 12-11 for information about these properties.

5. Restart WebLogic Server.

51acumen.ldif File Contents

Use the following content to create the 51acumen.ldif file:

```
dn: cn=schema
#
# attribute_types:
#
```
attributeTypes: ( 11827.0001.1.0 NAME 'uddi-Business-Key' DESC 'Business Key' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.1 NAME 'uddi-Authorized-Name' DESC 'Authorized Name for publisher of data' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.2 NAME 'uddi-Operator' DESC 'Name of UDDI Registry Operator' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.3 NAME 'uddi-Name' DESC 'Business Entity Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{258} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.4 NAME 'uddi-Description' DESC 'Description of Business Entity' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.7 NAME 'uddi-Use-Type' DESC 'Name of convention that the referenced document follows' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.8 NAME 'uddi-URL' DESC 'URL' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.9 NAME 'uddi-Person-Name' DESC 'Name of Contact Person' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.10 NAME 'uddi-Phone' DESC 'Telephone Number' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{50} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.11 NAME 'uddi-Email' DESC 'Email address' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.12 NAME 'uddi-Sort-Code' DESC 'Code to sort addresses' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{10} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.13 NAME 'uddi-tModel-Key' DESC 'Key to reference a tModel entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.14 NAME 'uddi-Address-Line' DESC 'Actual address lines in free form text' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{80} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.15 NAME 'uddi-Service-Key' DESC 'Service Key' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.16 NAME 'uddi-Service-Name' DESC 'Service Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.17 NAME 'uddi-Binding-Key' DESC 'Binding Key' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.18 NAME 'uddi-Access-Point' DESC 'A text field to convey the entry point address for calling a web service' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{50} X-ORIGIN 'acumen defined' )
Publishing and Finding Web Services Using UDDI

1.3.6.1.4.1.1466.115.121.1.15(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.19 NAME 'uddi-Hosting-Redirector' DESC 'Provides a Binding Key attribute to redirect reference to a different binding template' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(41) SINGLE-VALUE X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.20 NAME 'uddi-Instance-Parms' DESC 'Parameters to use a specific facet of a bindingTemplate description' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.21 NAME 'uddi-Overview-URL' DESC 'URL reference to a long form of an overview document' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.22 NAME 'uddi-From-Key' DESC 'Unique key reference to first businessEntity assertion is made for' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(41) SINGLE-VALUE X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.23 NAME 'uddi-To-Key' DESC 'Unique key reference to second businessEntity assertion is made for' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(41) SINGLE-VALUE X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.24 NAME 'uddi-Key-Name' DESC 'An attribute of the KeyedReference structure' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.25 NAME 'uddi-Key-Value' DESC 'An attribute of the KeyedReference structure' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.26 NAME 'uddi-Auth-Info' DESC 'Authorization information' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(4096) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.27 NAME 'uddi-Key-Type' DESC 'The key for all UDDI entries' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(16) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.28 NAME 'uddi-Upload-Register' DESC 'The upload register' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.29 NAME 'uddi-URL-Type' DESC 'The type for the URL' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15(16) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.30 NAME 'uddi-Ref-Keyed-Reference' DESC 'reference to a keyedReference entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.31 NAME 'uddi-Ref-Category-Bag' DESC 'reference to a categoryBag entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.32 NAME 'uddi-Ref-Identifier-Bag' DESC 'reference to a identifierBag entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12(255) X-ORIGIN 'acumen defined'

attributeTypes: ( 11827.0001.1.33 NAME 'uddi-Ref-TModel' DESC 'reference to a TModel entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12(255) SINGLE-VALUE X-ORIGIN 'acumen defined'

# id names for each entry
attributeTypes: ( 11827.0001.1.34 NAME 'uddi-Contact-ID' DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.35 NAME 'uddi-Discovery-URL-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.36 NAME 'uddi-Address-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.37 NAME 'uddi-Overview-Doc-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.38 NAME 'uddi-Instance-Details-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.39 NAME 'uddi-tModel-Instance-Info-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.40 NAME 'uddi-Publisher-Assertions-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.41 NAME 'uddi-Keyed-Reference-ID' DESC 'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.42 NAME 'uddi-Ref-Attribute' DESC 'a reference to another entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12{255} X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.43 NAME 'uddi-Entity-Name' DESC 'Business entity Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{258} X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.44 NAME 'uddi-tModel-Name' DESC 'tModel Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.45 NAME 'uddi-tMII-TModel-Key' DESC 'tModel key referenced in tModelInstanceInfo' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.46 NAME 'uddi-Keyed-Reference-TModel-Key' DESC 'tModel key referenced in KeyedReference' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.47 NAME 'uddi-Address-tModel-Key' DESC 'tModel key referenced in Address' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.48 NAME 'uddi-isHidden' DESC 'a flag to indicate whether an entry is hidden' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.49 NAME 'uddi-Time-Stamp' DESC 'modification time stamp' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined'
attributeTypes: ( 11827.0001.1.50 NAME 'uddi-next-id' DESC 'generic counter' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN 'acumen defined'
Publishing and Finding Web Services Using UDDI

attributeTypes: ( 11827.0001.1.51 NAME 'uddi-tModel-origin' DESC 'tModel origin' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.52 NAME 'uddi-tModel-type' DESC 'tModel type' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.53 NAME 'uddi-tModel-checked' DESC 'tModel field to check or not' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.54 NAME 'uddi-user-quota-entity' DESC 'quota for business entity' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.55 NAME 'uddi-user-quota-service' DESC 'quota for business services per entity' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.56 NAME 'uddi-user-quota-binding' DESC 'quota for binding templates per service' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.57 NAME 'uddi-user-quota-tmodel' DESC 'quota for tmodels' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.58 NAME 'uddi-user-quota-assertion' DESC 'quota for publisher assertions' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.59 NAME 'uddi-user-quota-messagesize' DESC 'quota for maximum message size' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.60 NAME 'uddi-user-language' DESC 'user language' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.61 NAME 'uddi-Name-Soundex' DESC 'name in soundex format' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{258} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.62 NAME 'uddi-var' DESC 'generic variable' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'acumen defined' )

# objectclasses:
# objectClasses: ( 11827.0001.2.0 NAME 'uddi-Business-Entity' DESC 'Business Entity object' SUP top STRUCTURAL MUST (uddi-Business-Key $ uddi-Entity-Name $ uddi-Ref-Identifier-Bag $ uddi-Ref-Category-Bag ) MAY ( uddi-Ref-Category-Bag $ uddi-Ref-Category-Bag ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.1 NAME 'uddi-Business-Service' DESC 'Business Service object' SUP top STRUCTURAL MUST ( uddi-Service-Key $ uddi-Service-Name $ uddi-Service-Category-Bag ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.2 NAME 'uddi-Binding-Template' DESC 'Binding Template object' SUP TOP STRUCTURAL MUST ( uddi-Binding-Key $ uddi-isHidden ) MAY ( uddi-Description $ uddi-Access-Point $ uddi-Hosting-Redirector ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.3 NAME 'uddi-tModel' DESC 'tModel object' SUP top STRUCTURAL MUST ( uddi-tModel-Key $ uddi-tModel-Name $ uddi-namespace $ uddi-Authorized-Name ) MAY ( uddi-Name-Soundex $ uddi-Operator $ uddi-Description $ uddi-Ref-Identifier-Bag $ uddi-Ref-Category-Bag $ uddi-tModel-origin $ uddi-tModel-checked $ uddi-tModel-type ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.4 NAME 'uddi-Publisher-Assertion' DESC 'Publisher Assertion object' SUP TOP STRUCTURAL MUST ( uddi-Publisher-Assertions-ID $ uddi-From-Key $ uddi-To-Key $ uddi-Ref-Keyed-Reference ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.5 NAME 'uddi-Discovery-URL' DESC 'Discovery URL' SUP TOP STRUCTURAL MUST ( uddi-Discovery-URL-ID $ uddi-Use-Type $ uddi-URL ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.6 NAME 'uddi-Contact' DESC 'Contact Information' SUP TOP STRUCTURAL MUST ( uddi-Contact-ID $ uddi-Person-Name ) MAY ( uddi-Use-Type $ uddi-Description $ uddi-Phone $ uddi-Email $ uddi-tModel-Key ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.7 NAME 'uddi-Address' DESC 'Address information for a contact entry' SUP TOP STRUCTURAL MUST ( uddi-Address-ID ) MAY ( uddi-Use-Type $ uddi-Sort-Code $ uddi-Address-tModel-Key $ uddi-Address-Line ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.8 NAME 'uddi-Keyed-Reference' DESC 'KeyedReference' SUP TOP STRUCTURAL MUST ( uddi-Keyed-Reference-ID $ uddi-Key-Value ) MAY ( uddi-Key-Name $ uddi-Keyed-Reference-TModel-Key ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.9 NAME 'uddi-tModel-Instance-Info' DESC 'tModelInstanceInfo' SUP TOP STRUCTURAL MUST ( uddi-tMII-TModel-Key ) MAY ( uddi-Description ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.10 NAME 'uddi-Instance-Details' DESC 'instanceDetails' SUP TOP STRUCTURAL MUST ( uddi-Instance-Details-ID ) MAY ( uddi-Description $ uddi-Instance-Parms ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.11 NAME 'uddi-Overview-Doc' DESC 'overviewDoc' SUP TOP STRUCTURAL MUST ( uddi-Overview-Doc-ID ) MAY ( uddi-Description $ uddi-Overview-URL ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.12 NAME 'uddi-Ref-Object' DESC 'an object class contains a reference to another entry' SUP TOP STRUCTURAL MUST ( uddi-Ref-Attribute ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.13 NAME 'uddi-Ref-Auxiliary-Object' DESC 'an auxiliary type object used in another structural class to hold a reference to a third entry' SUP TOP AUXILIARY MUST ( uddi-Ref-Attribute ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.14 NAME 'uddi-ou-container' DESC 'an organizational unit with uddi attributes' SUP organizationalunit STRUCTURAL MAY ( uddi-next-id $ uddi-var ) X-ORIGIN 'acumen defined' )

objectClasses: ( 11827.0001.2.15 NAME 'uddi-User' DESC 'a
Description of Properties in the uddi.properties File

The following tables describe properties of the `uddi.properties` file, categorized by the type of UDDI feature they describe:

- Basic UDDI Configuration
- UDDI User Defaults
- General Server Configuration
- Logger Configuration
- Connection Pools
- LDAP Datastore Configuration
- Replicated LDAP Datastore Configuration
- File Datastore Configuration
- General Security Configuration
- LDAP Security Configuration
- File Security Configuration

Table 12-2  Basic UDDI Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auddl.discoveryurl</td>
<td>DiscoveryURL prefix that is set for each saved business entity. Typically this is the full URL to the uddilistener servlet, so that the full DiscoveryURL results in the display of the stored BusinessEntity data.</td>
</tr>
<tr>
<td>auddl.inquiry.secure</td>
<td>Permissible values are true and false. When set to true, inquiry calls to UDDI Server are limited to secure https connections only. Any UDDI inquiry calls through a regular http URL are rejected.</td>
</tr>
</tbody>
</table>
Table 12-2  Basic UDDI Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuddi.publish.secure</td>
<td>Permissible values are true and false. When set to true, publish calls to UDDI Server are limited to secure https connections only. Any UDDI publish calls through a regular http URL are rejected.</td>
</tr>
<tr>
<td>uuddi.search.maxrows</td>
<td>Maximum number of returned rows for search operations. When the search results in a higher number of rows then the limit set by this property, the result is truncated.</td>
</tr>
<tr>
<td>uuddi.search.timeout</td>
<td>Timeout value for search operations. The value is indicated in milliseconds.</td>
</tr>
<tr>
<td>uuddi.siteoperator</td>
<td>Name of the UDDI registry site operator. The specified value will be used as the operator attribute, saved in all future BusinessEntity registrations. This attribute will later be returned in responses, and indicates which UDDI registry has generated the response.</td>
</tr>
<tr>
<td>security.cred.life</td>
<td>Credential life, specified in seconds, for authentication. Upon authentication of a user, an AuthToken is assigned which will be valid for the duration specified by this property.</td>
</tr>
<tr>
<td>pluggableTModel.file.list</td>
<td>UDDI Server is pre-populated with a set of Standard TModels. You can further customize the UDDI server by providing your own taxonomies, in the form of TModels. Taxonomies must be defined in XML files, following the provided XML schema. The value of this property a comma-separated list of URIs to such XML files. Values that refer to these TModels are checked and validated against the specified taxonomy.</td>
</tr>
<tr>
<td>UDDI Property Key</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>auddi.default.lang</td>
<td>User's initial language, assigned to user profile by default at the time of creation. User profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>auddi.default.quota.assertion</td>
<td>User's initial assertion quota, assigned to user profile by default at the time of creation. The assertion quota is the maximum number of publisher assertions that the user is allowed to publish. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>auddi.default.quota.binding</td>
<td>User's initial binding quota, assigned to user profile by default at the time of creation. The binding quota is the maximum number of binding templates that the user is allowed to publish, per each business service. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>auddi.default.quota.entity</td>
<td>User's initial business entity quota, assigned to user profile by default at the time of creation. The entity quota is the maximum number of business entities that the user is allowed to publish. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>auddi.default.quota.messageSize</td>
<td>User's initial message size limit, assigned to his user profile by default at the time of creation. The message size limit is the maximum size of a SOAP call that the user may send to UDDI Server. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>auddi.default.quota.service</td>
<td>User's initial service quota, assigned to user profile by default at the time of creation. The service quota is the maximum number of business services that the user is allowed to publish, per each business entity. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>auddi.default.quota.tmodel</td>
<td>User's initial TModel quota, assigned to user profile by default at the time of creation. The TModel quota is the maximum number of TModels that the user is allowed to publish. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.</td>
</tr>
<tr>
<td>UDDI Property Keys</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>auddl.datasource.type</td>
<td>Location of physical storage of UDDI data. This value defaults to WLS, which indicates that the internal LDAP directory of WebLogic Server is to be used for data storage. Other permissible values include LDAP, ReplicaLDAP, and File.</td>
</tr>
<tr>
<td>auddl.security.type</td>
<td>UDDI Server's security module (authentication). This value defaults to WLS, which indicates that the default security realm of WebLogic Server is to be used for UDDI authentication. As such, a WebLogic Server user would be an UDDI Server user and any WebLogic Server administrator would also be an UDDI Server administrator, in addition to members of the UDDI Server administrator group, as defined in UDDI Server settings. Other permissible values include LDAP and File.</td>
</tr>
<tr>
<td>auddl.license.dir</td>
<td>Location of the UDDI Server license file. In the absence of this property, the WL_HOME/server/lib directory is assumed to be the default license directory, where WL_HOME is the main WebLogic Server installation directory. Some WebLogic users are exempt from requiring a UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (for example, UDDI Server Browser).</td>
</tr>
<tr>
<td>auddl.license.file</td>
<td>Name of the license file. In the absence of this property, uddilicense.xml is presumed to be the default license filename. Some WebLogic users are exempt from requiring an UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (e.g., UDDI Server Browser).</td>
</tr>
<tr>
<td>UDDI Property Key</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>logger.file.maxsize</td>
<td>Maximum size of logger output files (if output is sent to file), in Kilobytes. Once an output file reaches maximum size, it is closed and a new log file is created.</td>
</tr>
<tr>
<td>logger.indent.enabled</td>
<td>Permissible values are <code>true</code> and <code>false</code>. When set to <code>true</code>, log messages beginning with &quot;+&quot; and &quot;.&quot;, typically TRACE level logs, cause an increase or decrease of indentation in the output.</td>
</tr>
<tr>
<td>logger.indent.size</td>
<td>Size of each indentation (how many spaces for each indent), specified as an integer.</td>
</tr>
<tr>
<td>logger.log.dir</td>
<td>Absolute or relative path to a directory where log files are stored.</td>
</tr>
<tr>
<td>logger.log.file.stem</td>
<td>String that is prefixed to all log file names.</td>
</tr>
<tr>
<td>logger.log.type</td>
<td>Determines whether log messages are sent to the screen, to a file or to both destinations. Permissible values, respectively, are: <code>LOG_TYPE_SCREEN</code>, <code>LOG_TYPE_FILE</code>, and <code>LOG_TYPE_SCREEN_FILE</code>.</td>
</tr>
<tr>
<td>logger.output.style</td>
<td>Determines whether logged output will simply contain the message, or thread and timestamp information will be included. Permissible values are <code>OUTPUT_LONG</code> and <code>OUTPUT_SHORT</code>.</td>
</tr>
<tr>
<td>logger.quiet</td>
<td>Determines whether the logger itself displays information messages. Permissible values are <code>true</code> and <code>false</code>.</td>
</tr>
<tr>
<td>logger.verbosity</td>
<td>Logger's verbosity level. Permissible values (case sensitive) are <code>TRACE</code>, <code>DEBUG</code>, <code>INFO</code>, <code>WARNING</code> and <code>ERROR</code>, where each severity level includes the following ones accumulatively.</td>
</tr>
</tbody>
</table>
Note: In a replicated LDAP environment, there are \(m\) LDAP masters and \(n\) LDAP replicas, respectively numbered from 0 to \((m-1)\) and from 0 to \((n-1)\). The fifth part of the property keys below, quoted as \(i\), refers to this number and differs for each LDAP server instance defined.

### Table 12-6  Connection Pools

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datasource.ldap.pool.increment</td>
<td>Number of new connections to create and add to the pool when all connections in the pool are busy</td>
</tr>
<tr>
<td>datasource.ldap.pool.initialsize</td>
<td>Number of connections to be stored at the time of creation and initialization of the pool.</td>
</tr>
<tr>
<td>datasource.ldap.pool.maxsize</td>
<td>Maximum number of connections that the pool may hold.</td>
</tr>
<tr>
<td>datasource.ldap.pool.systemmaxsize</td>
<td>Maximum number of connections created, even after the pool has reached its capacity. Once the pool reaches its maximum size, and all connections are busy, connections are temporarily created and returned to the client, but not stored in the pool. However, once the system max size is reached, all requests for new connections are blocked until a previously busy connection becomes available.</td>
</tr>
</tbody>
</table>

### Table 12-7  LDAP Datastore Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datasource.ldap.manager.uid</td>
<td>Back-end LDAP server administrator or privileged user ID, (for example, cn=Directory Manager) who can save data in LDAP.</td>
</tr>
<tr>
<td>datasource.ldap.manager.password</td>
<td>Password for the datasource.ldap.manager.uid, establishes connections with the LDAP directory used for data storage.</td>
</tr>
<tr>
<td>datasource.ldap.server.url</td>
<td>&quot;ldap://&quot; URL to the LDAP directory used for data storage.</td>
</tr>
<tr>
<td>datasource.ldap.server.root</td>
<td>Root entry of the LDAP directory used for data storage (e.g., dc=acumenat, dc=com).</td>
</tr>
</tbody>
</table>
### Table 12-8  Replicated LDAP Datastore Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datasource.ldap.server.master.i.manager.uid</td>
<td>Administrator or privileged user ID for this &quot;master&quot; LDAP server node, (for example, cn=Directory Manager) who can save data in LDAP.</td>
</tr>
<tr>
<td>datasource.ldap.server.master.i.manager.password</td>
<td>Password for the datasource.ldap.server.master.i.manager.uid, establishes connections with the relevant &quot;master&quot; LDAP directory to write data.</td>
</tr>
<tr>
<td>datasource.ldap.server.master.i.url</td>
<td>&quot;ldap://&quot; URL to the corresponding LDAP directory node.</td>
</tr>
<tr>
<td>datasource.ldap.server.master.i.root</td>
<td>Root entry of the corresponding LDAP directory node (for example, dc=acumenat, dc=com).</td>
</tr>
<tr>
<td>datasource.ldap.server.replica.i.manager.uid</td>
<td>User ID for this &quot;replica&quot; LDAP server node (for example, cn=Directory Manager); this person can read the UDDI data from LDAP.</td>
</tr>
<tr>
<td>datasource.ldap.server.replica.i.manager.password</td>
<td>Password for datasource.ldap.server.replica.i.manager.uid, establishes connections with the relevant &quot;replica&quot; LDAP directory to read data.</td>
</tr>
<tr>
<td>datasource.ldap.server.replica.i.url</td>
<td>&quot;ldap://&quot; URL to the corresponding LDAP directory node.</td>
</tr>
<tr>
<td>datasource.ldap.server.replica.i.root</td>
<td>Root entry of the corresponding LDAP directory node (for example, dc=acumenat, dc=com).</td>
</tr>
</tbody>
</table>

### Table 12-9  File Datastore Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datasource.file.directory</td>
<td>Directory where UDDI data is stored in the file system.</td>
</tr>
</tbody>
</table>
### Table 12-10 General Security Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.custom.group/operators</td>
<td>Security group name, where the members of this group are treated as UDDI administrators.</td>
</tr>
</tbody>
</table>

### Table 12-11 LDAP Security Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.custom.ldap.manager.uid</td>
<td>Security LDAP server administrator or privileged user ID (for example, cn=Directory Manager); this person can save data in LDAP.</td>
</tr>
<tr>
<td>security.custom.ldap.manager.password</td>
<td>The value of this property is the password for the above user ID, and is used to establish connections with the LDAP directory used for security.</td>
</tr>
<tr>
<td>security.custom.ldap.url</td>
<td>The value of this property is an &quot;ldap://&quot; URL to the LDAP directory used for security.</td>
</tr>
<tr>
<td>security.custom.ldap.root</td>
<td>Root entry of the LDAP directory used for security (for example, dc=acumenat, dc=com).</td>
</tr>
<tr>
<td>security.custom.ldap.userroot</td>
<td>User’s root entry on the security LDAP server. For example, ou=People.</td>
</tr>
<tr>
<td>security.custom.ldap.group.root</td>
<td>Operator entry on the security LDAP server. For example, &quot;cn=UDDI Administrators, ou=Groups&quot;. This entry contains IDs of all UDDI administrators.</td>
</tr>
</tbody>
</table>

### Table 12-12 File Security Configuration

<table>
<thead>
<tr>
<th>UDDI Property Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.custom.file.userdir</td>
<td>Directory where UDDI security information (users and groups) is stored in the file system.</td>
</tr>
</tbody>
</table>
UDDI Directory Explorer

The UDDI Directory Explorer allows authorized users to publish Web Services in private WebLogic Server UDDI registries and to modify information for previously published Web Services. The Directory Explorer provides access to details about the Web Services and associated WSDL files (if available.)

The UDDI Directory Explorer also enables you to search both public and private UDDI registries for Web Services and information about the companies and departments that provide these Web Services.

To invoke the UDDI Directory Explorer in your browser, enter:
http://host:port/uddiexplorer

where

- host is the computer on which WebLogic Server is running.
- port is the port number where WebLogic Server listens for connection requests. The default port number is 7001.

You can perform the following tasks with the UDDI Directory Explorer:

- Search public registries
- Search private registries
- Publish to a private registry
- Modify private registry details
- Setup UDDI directory explorer

For more information about using the UDDI Directory Explorer, click the Explorer Help link on the main page.

UDDI Client API

WebLogic Server includes an implementation of the client-side UDDI API that you can use in your Java client applications to programmatically search for and publish Web Services.

The two main classes of the UDDI client API are Inquiry and Publish. Use the Inquiry class to search for Web Services in a known UDDI registry and the Publish class to add your Web Service to a known registry.
WebLogic Server provides an implementation of the following client UDDI API packages:

- `weblogic.uddi.client.service`
- `weblogic.uddi.client.structures.datatypes`
- `weblogic.uddi.client.structures.exception`
- `weblogic.uddi.client.structures.request`
- `weblogic.uddi.client.structures.response`

For detailed information on using these packages, see the UDDI API Javadocs at `{DOCROOT}/javadocs/index.html`.

**Pluggable tModel**

A taxonomy is basically a tModel used as reference by a categoryBag or identifierBag. A major distinction is that in contrast to a simple tModel, references to a taxonomy are typically checked and validated. WebLogic Server’s UDDI Server takes advantage of this concept and extends this capability by introducing custom taxonomies, called "pluggable tModels". Pluggable tModels allow users (UDDI administrators) to add their own checked taxonomies to the UDDI registry, or overwrite standard taxonomies.

To add a pluggable tModel:

1. Create an XML file conforming to the specified format described in “XML Schema for Pluggable tModels” on page 12-23, for each tModelKey/categorization.

2. Add the comma-delimited, fully qualified file names to the `pluggableTModel.file.list` property in the `uddi.properties` file used to configure UDDI Server. For example:

   `pluggableTModel.file.list=c:/temp/cat1.xml,c:/temp/cat2.xml`

   See “Configuring the UDDI 2.0 Server” on page 12-5 for details about the `uddi.properties` file.


The following sections include a table detailing the XML elements and their permissible values, the XML schema against which pluggable tModels are validated, and a sample XML.

**XML Elements and Permissible Values**

The following table describes the elements of the XML file that describes your pluggable tModels.
<table>
<thead>
<tr>
<th><strong>Element/Attribute</strong></th>
<th><strong>Required</strong></th>
<th><strong>Role</strong></th>
<th><strong>Values</strong></th>
<th><strong>Comments</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxonomy</td>
<td>Required</td>
<td>Root Element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>checked</td>
<td>Required</td>
<td>Whether this categorization is checked or not.</td>
<td>true / false</td>
<td>If false, keyValue will not be validated.</td>
</tr>
<tr>
<td>type</td>
<td>Required</td>
<td>The type of the tModel.</td>
<td>categorization / identifier / valid values as defined in uddi-org-types</td>
<td>See uddi-org-types tModel for valid values.</td>
</tr>
<tr>
<td>applicability</td>
<td>Optional</td>
<td>Constraints on where the tModel may be used.</td>
<td></td>
<td>No constraint is assumed if this element is not provided</td>
</tr>
<tr>
<td>scope</td>
<td>Required if the applicability element is included.</td>
<td>businessEntity / businessService / bindingTemplate / tModel</td>
<td>tModel may be used in tModelInstanceI nfo if scope “bindingTemplat e” is specified.</td>
<td></td>
</tr>
<tr>
<td>tModel</td>
<td>Required</td>
<td>The actual tModel, according to the UDDI data structure.</td>
<td>Valid tModelKey must be provided.</td>
<td></td>
</tr>
<tr>
<td>categories</td>
<td>Required if checked is set to true.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>category</td>
<td>Required if element categories is included</td>
<td>Holds actual keyName and keyValue pairs.</td>
<td>keyName / keyValue pairs</td>
<td>category may be nested for grouping or tree structure.</td>
</tr>
</tbody>
</table>
**XML Schema for Pluggable tModels**

The XML Schema against which pluggable tModels are validated is as follows:

```
<simpleType name="type">
  <restriction base="string"/>
</simpleType>

<simpleType name="checked">
  <restriction base="NMTOKEN">
    <enumeration value="true"/>
    <enumeration value="false"/>
  </restriction>
</simpleType>

<element name="scope" type="string"/>

<complexType name="applicability">
  <sequence>
    <element ref="uddi:scope" minOccurs="1" maxOccurs="4"/>
  </sequence>
</complexType>

<element name="category" type="uddi:category"/>

<complexType name="category">
  <sequence>
    <element ref="uddi:category" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="keyName" use="required" type="string"/>
  <attribute name="keyValue" use="required" type="string"/>
</complexType>
```

<table>
<thead>
<tr>
<th>Element/Attribute</th>
<th>Required</th>
<th>Role</th>
<th>Values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyName</td>
<td>Required</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>keyValue</td>
<td>Required</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12-13 Description of the XML Elements to Configure Pluggable tModels
Sample XML for a Pluggable tModel

The following shows a sample XML for a pluggable tModel:

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <Taxonomy checked="true" type="categorization" xmlns="urn:uddi-org:api_v2">
      <applicability>
        <scope>businessEntity</scope>
        <scope>businessService</scope>
        <scope>bindingTemplate</scope>
      </applicability>
      <tModel tModelKey="uuid:C0B9FE13-179F-41DF-8A5B-5004DB444tt2">
        <name> sample pluggable tModel </name>
        <description>used for test purpose only </description>
        <overviewDoc>
          <overviewURL>http://www.abc.com</overviewURL>
        </overviewDoc>
      </tModel>
    </Taxonomy>
    <categories>
      <category keyName="name1" keyValue="1">
      </category>
    </categories>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
<category keyName="name1" keyValue="12">
  <category keyName="name11" keyValue="111">
    <category keyName="name111" keyValue="1111"/>
    <category keyName="name1112" keyValue="1112"/>
  </category>
</category>

<category keyName="name11" keyValue="112">
  <category keyName="name111" keyValue="1112"/>
  <category keyName="name112" keyValue="1121"/>
  <category keyName="name1122" keyValue="1122"/>
</category>

<category keyName="name2" keyValue="2">
  <category keyName="name21" keyValue="22">
    <category keyName="name211" keyValue="211">
      <category keyName="name2111" keyValue="2111"/>
      <category keyName="name2112" keyValue="2112"/>
    </category>
    <category keyName="name212" keyValue="212">
      <category keyName="name2121" keyValue="2121"/>
      <category keyName="name2122" keyValue="2122"/>
    </category>
  </category>
</category>
</categories>
</Taxonomy>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Upgrading WebLogic Web Services From Previous Releases to 9.2

The following sections describe how to upgrade a pre-9.2 WebLogic Server Web Service to run in the 9.2 Web Service runtime environment:

- “Upgrading a 9.0 or 9.1 WebLogic Web Service to 9.2” on page 13-1
- “Upgrading an 8.1 WebLogic Web Service to 9.2” on page 13-2

Upgrading a 9.0 or 9.1 WebLogic Web Service to 9.2

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 9.2:

- 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 9.2 without making any changes or recompiling it.
Upgrading WebLogic Web Services From Previous Releases to 9.2

Upgrading an 8.1 WebLogic Web Service to 9.2

This section describes how to upgrade an 8.1 WebLogic Web Service to use the new Version 9.2 Web Services runtime environment. The 9.2 runtime is based on the Implementing Enterprise Web Services 1.1 specification (JSR-921, which is the 1.1 maintenance release of JSR-109). The 9.2 programming model uses standard JDK 1.5 metadata annotations, as specified by the Web Services Metadata for the Java Platform specification (JSR-181).

Note: 8.1 WebLogic Web Services will continue to run, without any changes, on Version 9.2 of WebLogic Server because the 8.1 Web Services runtime is still supported in 9.2, although it is deprecated and will be removed from the product in future releases. For this reason, BEA highly recommends that you follow the instructions in this chapter to upgrade your 8.1 Web Service to 9.2.

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 9.2 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 9.2 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 J2EE artifacts that run on WebLogic Server. The procedures in this section direct you to update existing Ant build.xml files.

- You have access to the Java class or EJB source code for your 8.1 Web Service.

This section does not discuss the following topics:

- Upgrading a JMS-implemented 8.1 Web Service.
- Upgrading Web Services from versions previous to 8.1.
• Upgrading a client application that invokes an 8.1 Web Service to one that invokes a 9.2 Web Service. For details on how to write a client application that invokes a 9.2 Web Service, see Chapter 9, “Invoking Web Services.”

Upgrading an 8.1 Java Class-Implemented WebLogic Web Service to 9.2: Main Steps

To upgrade an 8.1 Java class-implemented Web Service to use the 9.2 WebLogic Web Services runtime:

1. Open a command window and set your WebLogic Server 9.2 environment by executing the setDomainEnv.cmd (Windows) or setDomainEnv.sh (UNIX) script, located in the bin subdirectory of your 9.2 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 9.2 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 9.2 JWS File” on page 13-5 for specific examples.

   a. If needed, change the package name and class name of the Java file to reflect the new 9.2 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:
Upgrading WebLogic Web Services From Previous Releases to 9.2

- 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.


d. You might need to add additional annotations to your JWS file, depending on the 8.1 Web Service features you want to carry forward to 9.2. 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 13-19 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 9.2 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 9.2 Ant Build File for Java Class-Implemented Web Services” on page 13-7 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 9.2 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 9.2. 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 13-19 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 9.2 Web Service.

See “Deploying and Undeploying WebLogic Web Services” on page 4-15 and “Browsing to the WSDL of the Web Service” on page 4-17 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 9.2 Web Service is:

http://host:port/upgradePOJO/HelloWorld?WSDL

Example of an 8.1 Java File and the Corresponding 9.2 JWS File
Assume that the following sample Java class implemented a 8.1 Web Service:

```java
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.
 */
```
Upgrading WebLogic Web Services From Previous Releases to 9.2

```java
/*
 * public String sayHello(int num, String s) {
 *     System.out.println("sayHello operation has been invoked with arguments " +
 *             s + " and " + num);
 *     String returnValue = "This message brought to you by the letter "+s+" and
 *             the number "+num;
 *     return returnValue;
 * }
 */

An equivalent JWS file for a 9.2 Java class-implemented Web Service is shown below, with the differences shown in bold. Note that some of the JWS annotation values are taken from attributes of the 8.1 servicegen Ant task shown in “Example of an 8.1 and Updated 9.2 Ant Build File for Java Class-Implemented Web Services” on page 13-7:

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 annotation
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 9.2 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
Upgrading WebLogic Web Services From Previous Releases to 9.2

```
destEar="${build}/${ear_file}"
warName="${war_file}">
<service
   javaClassComponents="examples.javaclass.HelloWorld81"
   targetNamespace="${namespace}"
   serviceName="HelloWorld"
   serviceURI="/HelloWorld"
   generateTypes="True"
   expandMethods="True">
</service>
</servicegen>
</target>
</project>
```

An equivalent build.xml file that calls the jwsc Ant task to build a 9.2 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 9.2 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" />
```
Upgrading an 8.1 WebLogic Web Service to 9.2

The following procedure describes how to upgrade an 8.1 EJB-implemented Web Service to use the 9.2 WebLogic Web Services runtime.

The 9.2 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 9.2 equivalent to the 8.1 EJB-implemented Web Service.

1. Open a command window and set your 9.2 WebLogic Server environment by executing the
   setDomainEnv.cmd (Windows) or setDomainEnv.sh (UNIX) script, located in the bin
   subdirectory of your 9.2 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 9.2 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 9.2 JWS File” on page 13-12 for
   specific examples.
   
   a. If needed, change the package name and class name of the Java file to reflect the new 9.2
      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.


   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 9.2. 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 13-19 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 9.2 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.”
Upgrading WebLogic Web Services From Previous Releases to 9.2

See “Example of an 8.1 and Updated 9.2 Ant Build File for an 8.1 EJB-Implemented Web Service” on page 13-17 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 9.2 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 9.2. 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 13-19 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 9.2 Web Service.

See “Deploying and Undeploying WebLogic Web Services” on page 4-15 and “Browsing to the WSDL of the Web Service” on page 4-17 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 9.2 Web Service is:

http://host:port/upgradeEJB/HelloWorldService?WSDL

Example of 8.1 EJB Files and the Corresponding 9.2 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 9.2 JWS file is shown in “Equivalent 9.2 JWS File” on page 13-15. The differences between the 8.1 and 9.2 classes are shown in bold. Note that some of the JWS annotation values are taken from attributes of the 8.1 servicegen Ant task shown in “Example of an 8.1 and Updated 9.2 Ant Build File for an 8.1 EJB-Implemented Web Service” on page 13-17.
8.1 SessionBean Class

```java
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");
    }
}
```

Upgrading an 8.1 WebLogic Web Service to 9.2
/**
 * 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.
 */
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 9.2 JWS File
The differences between the 8.1 and 9.2 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 9.2 Ant Build File for an 8.1 EJB-Implemented Web Service” on page 13-17
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;
    }
}

Upgrading WebLogic Web Services From Previous Releases to 9.2

Programming Web Services for WebLogic Server

13-16
Example of an 8.1 and Updated 9.2 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 9.2 Web Service.

```xml
<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
```
Upgrading WebLogic Web Services From Previous Releases to 9.2

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 \texttt{jwsc} Ant task to build a 9.2 Web Service is shown below, with the relevant tasks discussed in this section in bold:

\begin{verbatim}
<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="${example-output}/upgradeEJB" />
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy" />
  <target name="all" depends="clean,build-service,deploy" />
\end{verbatim}
<target name="clean" depends="undeploy">
  <delete dir="${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 9.2 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 Appendix B, “JWS Annotation Reference,” and “jwsc” on page A-17 for more information about the 9.2 JWS annotations and jwsc Ant task.

### Table 13-1  Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

<table>
<thead>
<tr>
<th>servicegen or Child Element of servicegen Attribute</th>
<th>Equivalent JWS Annotation or jwsc Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>contextURI</td>
<td>contextPath attribute of the WebLogic-specific @WLHttpTransport annotation.</td>
</tr>
<tr>
<td>destEAR</td>
<td>destdir attribute of the jwsc Ant task.</td>
</tr>
<tr>
<td>keepGenerated</td>
<td>keepGenerated attribute of the jwsc Ant task.</td>
</tr>
<tr>
<td>mergeWithExistingWS</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>overwrite</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>warName</td>
<td>name attribute of the &lt;jws&gt; child element of the jwsc Ant task.</td>
</tr>
<tr>
<td>ejbJar (attribute of the service child element)</td>
<td>No direct equivalent, because the jwsc Ant task generates Web Service artifacts from a JWS file, rather than a compiled EJB or Java class.</td>
</tr>
<tr>
<td></td>
<td>Indirect equivalent is the file attribute of the &lt;jws&gt; child element of the jwsc Ant task that specifies the name of the JWS file.</td>
</tr>
<tr>
<td>excludeEJBs (attribute of the service child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>expandMethods (attribute of the service child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>generateTypes (attribute of the service child element)</td>
<td>No equivalent.</td>
</tr>
</tbody>
</table>
### Table 13-1  Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

<table>
<thead>
<tr>
<th>servicegen or Child Element of servicegen Attribute</th>
<th>Equivalent JWS Annotation or jwsc Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignoreAuthHeader (attribute of the service child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>includeEJBs (attribute of the service child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>javaClassComponents (attribute of the service child element)</td>
<td>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 <code>&lt;jws&gt;</code> child element of the jwsc Ant task that specifies the name of the JWS file.</td>
</tr>
<tr>
<td>JMSOperationName (attribute of the service child element)</td>
<td>No equivalent because JMS-implemented Web Services are not supported in the 9.2 release.</td>
</tr>
<tr>
<td>JMSDestinationType (attribute of the service child element)</td>
<td>No equivalent because JMS-implemented Web Services are not supported in the 9.2 release.</td>
</tr>
<tr>
<td>JMSMessageTypeName (attribute of the service child element)</td>
<td>No equivalent because JMS-implemented Web Services are not supported in the 9.2 release.</td>
</tr>
</tbody>
</table>
Table 13-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

<table>
<thead>
<tr>
<th>servicegen or Child Element of servicegen Attribute</th>
<th>Equivalent JWS Annotation or jwsc Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol (attribute of the service child element)</td>
<td>One of the following WebLogic-specific annotations:</td>
</tr>
<tr>
<td></td>
<td>• @WLHttpTransport</td>
</tr>
<tr>
<td></td>
<td>• @WLJmsTransport</td>
</tr>
<tr>
<td>serviceName (attribute of the service child element)</td>
<td>serviceName attribute of the standard @WebService annotation.</td>
</tr>
<tr>
<td>serviceURI (attribute of the service child element)</td>
<td>serviceUri attribute of the WebLogic-specific @WLHttpTransport or @WLJmsTransport annotations.</td>
</tr>
<tr>
<td>style (attribute of service child element)</td>
<td>style attribute of the standard @SOAPBinding annotation.</td>
</tr>
<tr>
<td>typeMappingFile (attribute of the service child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>targetNamespace (attribute of the service child element)</td>
<td>targetNamespace attribute of the standard @WebService annotation.</td>
</tr>
<tr>
<td>userSOAP12 (attribute of the service child element)</td>
<td>value attribute of the WebLogic-specific @weblogic.jws.Binding JWS annotation</td>
</tr>
<tr>
<td>clientJarName (attribute of client child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>packageName (attribute of client child element)</td>
<td>No direct equivalent. Use the packageName attribute of the clientgen Ant task to generate client-side Java code and artifacts.</td>
</tr>
<tr>
<td>saveWSDL (attribute of the client child element)</td>
<td>No equivalent.</td>
</tr>
</tbody>
</table>
### Table 13-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

<table>
<thead>
<tr>
<th>servicegen or Child Element of servicegen Attribute</th>
<th>Equivalent JWS Annotation or jwsc Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>userServerTypes (attribute of the client child element)</td>
<td>No equivalent.</td>
</tr>
<tr>
<td>handlers (attribute of the handlerChain child element)</td>
<td>Standard @HandlerChain or @SOAPMessageHandlers annotation.</td>
</tr>
<tr>
<td>name (attribute of the handlerChain child element)</td>
<td>Standard @HandlerChain or @SOAPMessageHandlers annotation.</td>
</tr>
<tr>
<td>duplicateElimination (attribute of the reliability child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains Web Service reliable messaging policy assertions. See “Using Web Service Reliable Messaging” on page 6-1.</td>
</tr>
<tr>
<td>persistDuration (attribute of the reliability child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains Web Service reliable messaging policy assertions. See “Using Web Service Reliable Messaging” on page 6-1.</td>
</tr>
<tr>
<td>enablePasswordAuth (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3.</td>
</tr>
<tr>
<td>encryptKeyName (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3.</td>
</tr>
</tbody>
</table>
Table 13-1 Mapping of servicegen Attributes to JWS Annotations or jwsc Attributes

<table>
<thead>
<tr>
<th>servicegen or Child Element of servicegen Attribute</th>
<th>Equivalent JWS Annotation or jwsc Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>encryptKeyPass (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3</td>
</tr>
<tr>
<td>password (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3</td>
</tr>
<tr>
<td>signKeyName (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3</td>
</tr>
<tr>
<td>signKeyPass (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3</td>
</tr>
<tr>
<td>username (attribute of the security child element)</td>
<td>No direct equivalent. Use WebLogic-specific @Policy attribute to specify a WS-Policy file that contains message-level security policy assertions. See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3</td>
</tr>
</tbody>
</table>
Ant Task Reference

The following sections provide reference information about the WebLogic Web Services Ant tasks:

- “Overview of WebLogic Web Services Ant Tasks” on page A-1
- “clientgen” on page A-5
- “jwsc” on page A-17
- “wsdlc” on page A-53

For detailed information on how to integrate and use these Ant tasks in your development environment to program a Web Service and a client application that invokes the Web Service, see:

- “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
- Chapter 9, “Invoking Web Services”

Overview of WebLogic Web Services Ant Tasks

Ant is a Java-based build tool, similar to the make command but much more powerful. Ant uses XML-based configuration files (called build.xml by default) to execute tasks written in Java.
BEA provides a number of Ant tasks that help you generate important Web Service-related artifacts.

The Apache Web site provides other useful Ant tasks for packaging EAR, WAR, and EJB JAR files. For more information, see http://jakarta.apache.org/ant/manual/.

Note: The Apache Jakarta Web site publishes online documentation for only the most current version of Ant, which might be different from the version of Ant that is bundled with WebLogic Server. To determine the version of Ant that is bundled with WebLogic Server, run the following command after setting your WebLogic environment:

```
prompt> ant -version
```

To view the documentation for a specific version of Ant, download the Ant zip file from http://archive.apache.org/dist/ant/binaries/ and extract the documentation.

**List of Web Services Ant Tasks**

The following table provides an overview of the Web Service Ant tasks provided by BEA.

<table>
<thead>
<tr>
<th>Ant Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientgen</td>
<td>Generates the JAX-RPC Service stubs and other client-side files used to invoke a Web Service.</td>
</tr>
<tr>
<td>jwsc</td>
<td>Compiles a JWS-annotated file into a Web Service. JWS refers to Java Web Service.</td>
</tr>
<tr>
<td>wsdlc</td>
<td>Generates a partial Web Service implementation based on a WSDL file.</td>
</tr>
</tbody>
</table>

**Using the Web Services Ant Tasks**

To use the Ant tasks:

1. Set your environment.

   On Windows NT, execute the `setDomainEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is

   \`BEA_HOME\user_projects\domains\domainName\`, where \`BEA_HOME\` is the top-level installation directory of the BEA products and \`domainName\` is the name of your domain.
On UNIX, execute the `setDomainEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is `BEA_HOME/user_projects/domains/domainName`, where `BEA_HOME` is the top-level installation directory of the BEA products and `domainName` is the name of your domain.

2. Create a file called `build.xml` that will contain a call to the Web Services Ant tasks. The following example shows a simple `build.xml` file with a single target called `clean`:

```xml
<project name="my-webservice">
  <target name="clean">
    <delete>
      <fileset dir="tmp" />
    </delete>
  </target>
</project>
```

This `clean` target deletes all files in the `temp` subdirectory.

Later sections provide examples of specifying the Ant task in the `build.xml` file.

3. For each WebLogic Web Service Ant task you want to execute, add an appropriate task definition and target to the `build.xml` file using the `<taskdef>` and `<target>` elements. The following example shows how to add the `jwsc` Ant task to the build file; the attributes of the task have been removed for clarity:

```xml
<taskdef name="jwsc" class="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
  <jwsc attributes go here...>
    ...
  </jwsc>
</target>
```

You can, of course, name the WebLogic Web Services Ant tasks anything you want by changing the value of the `name` attribute of the relevant `<taskdef>` element. For consistency, however, this document uses the names `jwsc`, `clientgen`, and `wsdlc` throughout.

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

```
prompt> ant build-service
```
Setting the Classpath for the WebLogic Ant Tasks

Each WebLogic Ant task accepts a classpath attribute or element so that you can add new directories or JAR files to your current CLASSPATH environment variable.

The following example shows how to use the classpath attribute of the jwsc Ant task to add a new directory to the CLASSPATH variable:

```xml
<jwsc srcdir="MyJWSFile.java"
     classpath="${java.class.path};my_fab_directory"
     ...
</jwsc>
```

The following example shows how to add to the CLASSPATH by using the <classpath> element:

```xml
<jwsc ...
     <classpath>
         <pathelement path="${java.class.path}" />
         <pathelement path="my_fab_directory" />
     </classpath>
     ...
</jwsc>
```

The following example shows how you can build your CLASSPATH variable outside of the WebLogic Web Service Ant task declarations, then specify the variable from within the task using the <classpath> element:

```xml
<path id="myClassID">
    <pathelement path="${java.class.path}"/>
    <pathelement path="${additional.path1}"/>
    <pathelement path="${additional.path2}"/>
</path>

<jwsc ...
     <classpath refid="myClassID" />
     ...
</jwsc>
```
Note: The Java Ant utility included in WebLogic Server uses the ant (UNIX) or ant.bat (Windows) configuration files in the $WL_HOME\server\bin directory to set various Ant-specific variables, where $WL_HOME$ is the top-level directory of your WebLogic Server installation. If you need to update these Ant variables, make the relevant changes to the appropriate file for your operating system.

Differences in Operating System Case Sensitivity When Manipulating WSDL and XML Schema Files

Many WebLogic Web Service Ant tasks have attributes that you can use to specify a file, such as a WSDL or an XML Schema file.

The Ant tasks process these files in a case-sensitive way. This means that if, for example, the XML Schema file specifies two user-defined types whose names differ only in their capitalization (for example, MyReturnType and MYRETURNTYPE), the clientgen Ant task correctly generates two separate sets of Java source files for the Java representation of the user-defined data type: MyReturnType.java and MYRETURNTYPE.java.

However, compiling these source files into their respective class files might cause a problem if you are running the Ant task on Microsoft Windows, because Windows is a case insensitive operating system. This means that Windows considers the files MyReturnType.java and MYRETURNTYPE.java to have the same name. So when you compile the files on Windows, the second class file overwrites the first, and you end up with only one class file. The Ant tasks, however, expect that two classes were compiled, thus resulting in an error similar to the following:

c:\src\com\bea\order\MyReturnType.java:14:
class MYRETURNTYPE is public, should be declared in a file named MYRETURNTYPE.java
^  

To work around this problem rewrite the XML Schema so that this type of naming conflict does not occur, or if that is not possible, run the Ant task on a case sensitive operating system, such as Unix.

clientgen

The clientgen Ant task generates, from an existing WSDL file, the client component files that client applications use to invoke both WebLogic and non-WebLogic Web Services. These files include:
The Java source code for the JAX-RPC Stub and Service interface implementations for the particular Web Service you want to invoke.

- The Java source code for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

Two types of client applications use the generated artifacts of `clientgen` to invoke Web Services:

- Stand-alone Java clients that do not use the J2EE client container.
- J2EE clients, such as EJBs, JSPs, and Web Services, that use the J2EE client container.

You typically use the `destDir` attribute of `clientgen` to specify the directory into which all the artifacts should be generated, and then compile the generate Java files yourself using the `javac` Ant task. However, `clientgen` also provides a `destFile` attribute if you want the Ant task to compile the Java files for you and package them, along with the other generated artifacts, into the specified JAR file. You must specify one of either `destFile` or `destDir`, although you cannot specify both.

**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 9.2, 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.

**Taskdef Classname**

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

```xml
<taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
...
<target name="build_client">
    <clientgen
        wsdl="http://example.com/myapp/myservice.wsdl"
        destDir="/output/clientclasses"
        packageName="myapp.myservice.client"
        serviceName="StockQuoteService" />
    <javac ... />
</target>
```

When the sample `build_client` target is executed, `clientgen` uses the WSDL file specified by the `wsdl` attribute to generate all the client-side artifacts needed to invoke the Web Service specified by the `serviceName` attribute. The `clientgen` Ant task generates all the artifacts into the `/output/clientclasses` directory. All generated Java code is in the `myapp.myservice.client` package. After `clientgen` has finished, the `javac` Ant task then compiles the Java code, both `clientgen`-generated as well as your own client application that uses the generated artifacts and contains your business code.

If you want the `clientgen` Ant task to compile and package the generated artifacts for you, specify the `destFile` attribute rather than `destDir`:

```xml
<clientgen
    wsdl="http://example.com/myapp/myservice.wsdl"
    destFile="/output/jarfiles/myclient.jar"
    packageName="myapp.myservice.client"
    serviceName="StockQuoteService" />
```

In the preceding example, you do not need to also specify the `javac` Ant task after `clientgen` in the `build.xml` file, because the Java code has already been compiled.

You typically execute the `clientgen` Ant task on a WSDL file that is deployed on the Web and accessed using HTTP. Sometimes, however, you might want to execute `clientgen` on a static WSDL file that is packaged in an archive file, such as the WAR or JAR file generated by the `jwsc` Ant task. In this case you must use the following syntax for the `wsdl` attribute:

```xml
wsdl="jar:file:archive_file!WSDL_file"
```
where \textit{archive\_file} refers to the full (or relative to the current directory) name of the archive file and \textit{WSDL\_file} refers to the full pathname of the WSDL file, relative to the root directory of the archive file. For example:

\begin{verbatim}
<clientgen
  wsdll="jar:file:output/myEAR/examples/webservices/simple/SimpleImpl.war!/WEB-INF/SimpleService.wsdl"
  destDir="/output/clientclasses"
  packageName="myapp.myservice.client"/>
\end{verbatim}

The preceding example shows how to execute \texttt{clientgen} on a static WSDL file called \texttt{SimpleService.wsdl}, which is packaged in the \texttt{WEB-INF} directory of a WAR file called \texttt{SimpleImpl.war}, which is located in the \texttt{output/myEAR/examples/webservices/simple} sub-directory of the directory that contains the build.xml file.

You can use the standard Ant \texttt{<sysproperty>} nested element to set Java properties, such as the username and password of a valid WebLogic Server user (if you have enabled access control on the Web Service) or the name of a client-side trust store that contains trusted certificates, as shown in the following example:

\begin{verbatim}
<clientgen
  wsdll="http://example.com/myapp/mySecuredService.wsdl"
  destDir="/output/clientclasses"
  packageName="myapp.mysecuredservice.client"
  serviceName="SecureStockQuoteService"
  <sysproperty key="javax.net.ssl.trustStore"
    value="/keystores/DemoTrust.jks"/>
  <sysproperty key="weblogic.wsee.client.ssl.stricthostchecking"
    value="false"/>
  <sysproperty key="javax.xml.rpc.security.auth.username"
    value="juliet"/>
  <sysproperty key="javax.xml.rpc.security.auth.password"
    value="secret"/>
</clientgen>
\end{verbatim}

\textbf{Child Element}

The \texttt{clientgen} Ant task has one WebLogic-specific child element: \texttt{<xsdConfig>}. 

\textbf{A-8} Programming Web Services for WebLogic Server
Use the `<xsdConfig>` child element to specify one or more XMLBeans configuration files, which by convention end in `.xsdconfig`. Use this element if your Web Service uses Apache XMLBeans data types as parameters or return values.

The `<xsdConfig>` element is similar to the standard Ant `<Fileset>` element and has all the same attributes. See the Apache Ant documentation on the `Fileset` element for the full list of attributes you can specify.

See “Standard Ant Attributes and Elements That Apply To clientgen” on page A-16 for the list of elements associated with the standard Ant `javac` task that you can also set for the `clientgen` Ant task.

**Attributes**

The table in the following section describes the attributes of the `clientgen` Ant task. See “Standard Ant Attributes and Elements That Apply To clientgen” on page A-16 for the list of attributes associated with the standard Ant `javac` task that you can also set for the `clientgen` Ant task.
# WebLogic-Specific clientgen Attributes

## Table A-2  Attributes of the clientgen Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoDetectWrapped</td>
<td>Specifies whether the <code>clientgen</code> Ant task should try to determine whether the parameters and return type of document-literal Web Services are of type wrapped or bare. When the <code>clientgen</code> Ant task parses a WSDL file to create the JAX-RPC stubs, it attempts to determine whether a document-literal Web Service uses wrapped or bare parameters and return types based on the names of the XML Schema elements, the name of the operations and parameters, and so on. Depending on how the names of these components match up, the <code>clientgen</code> Ant task makes a best guess as to whether the parameters are wrapped or bare. In some cases, however, you might want the Ant task to always assume that the parameters are of type bare; in this case, set the <code>autoDetectWrapped</code> attribute to False. Valid values for this attribute are True or False. The default value is True.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
<tr>
<td>destDir</td>
<td>Directory into which the <code>clientgen</code> Ant task generates the client source code, WSDL, and client deployment descriptor files. You can set this attribute to any directory you want. However, if you are generating the client component files to invoke a Web Service from an EJB, JSP, or other Web Service, you typically set this attribute to the directory of the J2EE component which holds shared classes, such as META-INF for EJBs, WEB-INF/classes for Web Applications, or APP-INF/classes for Enterprise Applications. If you are invoking the Web Service from a stand-alone client, then you can generate the client component files into the same source code directory hierarchy as your client application code.</td>
<td>String</td>
<td>You must specify either the <code>destFile</code> or <code>destDir</code> attribute, but not both.</td>
</tr>
</tbody>
</table>
### Table A-2 Attributes of the clientgen Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>destFile</td>
<td>Name of a JAR file or exploded directory into which the <code>clientgen</code> task packages the client source code, compiled classes, WSDL, and client deployment descriptor files. If you specify this attribute, the <code>clientgen</code> Ant task also compiles all Java code into classes. To create or update a JAR file, use a <code>.jar</code> suffix when specifying the JAR file, such as <code>myclientjar.jar</code>. If the attribute value does not have a <code>.jar</code> suffix, then the <code>clientgen</code> task assumes you are referring to a directory name. If you specify a JAR file or directory that does not exist, the <code>clientgen</code> task creates a new JAR file or directory.</td>
<td>String</td>
<td>You must specify either the <code>destFile</code> or <code>destDir</code> attribute, but not both.</td>
</tr>
<tr>
<td>failonerror</td>
<td>Specifies whether the <code>clientgen</code> Ant task continues executing in the event of an error. Valid values for this attribute are <code>True</code> or <code>False</code>. The default value is <code>True</code>, which means <code>clientgen</code> continues executing even after it encounters an error. <strong>Note:</strong> When this attribute is set to <code>False</code>, it aborts the Ant task only when there is an issue with Java compilation. It does not abort the Ant task when other problems occur.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
</tbody>
</table>
**generateAsyncMethods**

Specifies whether the `clientgen` Ant task should include methods in the generated JAX-RPC stubs that client applications can use to invoke a Web Service operation asynchronously.

For example, if you specify `True` (which is also the default value), and one of the Web Service operations in the WSDL is called `getQuote`, then the `clientgen` Ant task also generates a method called `getQuoteAsync` in the JAX-RPC stubs which client applications invoke instead of the original `getQuote` method. This asynchronous flavor of the operation also has an additional parameter, of data type `weblogic.wsee.async.AsyncPreCallContext`, that client applications can use to set asynchronous properties, contextual variables, and so on.

See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 for full description and procedures about this feature.

**Note:** If the Web Service operation is marked as one-way, the `clientgen` Ant task never generates the asynchronous flavor of the JAX-RPC stub, even if you explicitly set the `generateAsyncMethods` attribute to `True`.

Valid values for this attribute are `True` or `False`. The default value is `True`, which means the asynchronous methods are generated by default.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>generateAsyncMethods</td>
<td>Specifies whether the <code>clientgen</code> Ant task should include methods in the generated JAX-RPC stubs that client applications can use to invoke a Web Service operation asynchronously.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
</tbody>
</table>
generatePolicyMethods specifies whether the clientgen Ant task should include WS-Policy-loading methods in the generated JAX-RPC stubs. These methods can be used by client applications to load a local WS-Policy file.

If you specify True, four flavors of a method called getXXXSoapPort() are added as extensions to the JAX-RPC Service interface in the generated client stubs, where XXX refers to the name of the Web Service. Client applications can use these methods to load and apply local WS-Policy files, rather than apply any WS-Policy files deployed with the Web Service itself. Client applications can specify whether the local WS-Policy file applies to inbound, outbound, or both SOAP messages and whether to load the local WS-Policy from an InputStream or a URI.

Valid values for this attribute are True or False. The default value is False, which means the additional methods are not generated.


<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>generatePolicyMethods</td>
<td>Specifies whether the clientgen Ant task should include WS-Policy-loading methods in the generated JAX-RPC stubs. These methods can be used by client applications to load a local WS-Policy file. If you specify True, four flavors of a method called getXXXSoapPort() are added as extensions to the JAX-RPC Service interface in the generated client stubs, where XXX refers to the name of the Web Service. Client applications can use these methods to load and apply local WS-Policy files, rather than apply any WS-Policy files deployed with the Web Service itself. Client applications can specify whether the local WS-Policy file applies to inbound, outbound, or both SOAP messages and whether to load the local WS-Policy from an InputStream or a URI. Valid values for this attribute are True or False. The default value is False, which means the additional methods are not generated. See “Using a Client-Side Security WS-Policy File” on page 9-27 for more information.</td>
<td>Boolean</td>
<td>No</td>
</tr>
<tr>
<td>handlerChainFile</td>
<td>Specifies the name of the XML file that describes the client-side SOAP message handlers that execute when a client application invokes a Web Service. Each handler specified in the file executes twice: • directly before the client application sends the SOAP request to the Web Service • directly after the client application receives the SOAP response from the Web Service If you do not specify this clientgen attribute, then no client-side handlers execute, even if they are in your CLASSPATH. See “Creating and Using Client-Side SOAP Message Handlers” on page 9-22 for details and examples about creating client-side SOAP message handlers.</td>
<td>String</td>
<td>No</td>
</tr>
</tbody>
</table>
### Table A-2 Attributes of the clientgen Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>includeGlobalTypes</td>
<td>Specifies that the clientgen Ant task should generate Java representations of all XML Schema data types in the WSDL, rather than just the data types that are explicitly used in the Web Service operations. Valid values for this attribute are True or False. The default value is False, which means that clientgen generates Java representations for only the actively-used XML data types.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
<tr>
<td>jaxRPCWrappedArrayStyle</td>
<td>When the clientgen Ant task is generating the Java equivalent to XML Schema data types in the WSDL file, and the task encounters an XML complex type with a single enclosing sequence with a single element with the <code>maxOccurs</code> attribute equal to unbounded, the task generates, by default, a Java structure whose name is the lowest named enclosing complex type or element. To change this behavior so that the task generates a literal array instead, set the <code>jaxRPCWrappedArrayStyle</code> to False. Valid values for this attribute are True or False. The default value is True.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Specifies whether the client component files (source code, WSDL, and deployment descriptor files) generated by this Ant task should be overwritten if they already exist. If you specify True, new artifacts are always generated and any existing artifacts are overwritten. If you specify False, the Ant task overwrites only those artifacts that have changed, based on the timestamp of any existing artifacts. Valid values for this attribute is True or False. The default value is True.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
</tbody>
</table>
### Table A-2 Attributes of the clientgen Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>packageName</td>
<td>Package name into which the generated JAX-RPC client interfaces and stub files are packaged. If you do not specify this attribute, the clientgen Ant task generates Java files whose package name is based on the targetNamespace of the WSDL file. For example, if the targetNamespace is <a href="http://example.org">http://example.org</a>, then the package name might be org.example or something similar. If you want control over the package name, rather than let the Ant task generate one for you, then you should specify this attribute. If you do specify this attribute, BEA recommends you use all lower-case letters for the package name.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>serviceName</td>
<td>Name of the Web Service in the WSDL file for which the corresponding client component files should be generated. The Web Service name corresponds to the &lt;service&gt; element in the WSDL file. The generated JAX-RPC mapping file and client-side copy of the WSDL file will use this name. For example, if you set serviceName to CuteService, the JAX-RPC mapping file will be called cuteService_java_wsdl_mapping.xml and the client-side copy of the WSDL will be called CuteService_saved_wsdl.wsdl.</td>
<td>String</td>
<td>This attribute is required only if the WSDL file contains more than one &lt;service&gt; element. The Ant task returns an error if you do not specify this attribute and the WSDL file contains more than one &lt;service&gt; element.</td>
</tr>
<tr>
<td>wsdl</td>
<td>Full path name or URL of the WSDL that describes a Web Service (either WebLogic or non-WebLogic) for which the client component files should be generated. The generated stub factory classes in the client JAR file use the value of this attribute in the default constructor.</td>
<td>String</td>
<td>Yes.</td>
</tr>
</tbody>
</table>
Standard Ant Attributes and Elements That Apply To clientgen

In addition to the WebLogic-defined clientgen attributes, you can also define the following standard javac attributes; see the Ant documentation for additional information about each attribute:

- bootclasspath
- bootClasspathRef
- classpath
- classpathRef
- compiler
- debug
- debugLevel
- depend
- deprecation
- destdir
- encoding
- extdirs
- failonerror
- fork
- includeantruntime
- includejavaruntime
- listfiles
- memoryInitialSize
- memoryMaximumSize
- nowarn
- optimize
- proceed
- source
- sourcepath
- sourcepathRef
- tempdir
You can use the standard Ant `<sysproperty>` child element to specify properties required by the Web Service from which you are generating client-side artifacts. For example, if the Web Service is secured, you can use the `javax.xml.rpc.security.auth.username|password` properties to set the authenticated username and password. See the Ant documentation for the java Ant task for additional information about `<sysproperty>`.

You can also use the following standard Ant child elements with the `clientgen` Ant task:

- `<FileSet>`
- `<SourcePath>`
- `<Classpath>`
- `<Extdirs>`

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

- Java source files that implement a standard JSR-921 Web Service, such as the service endpoint interface (called `JWS_ClassNamePortType.java`, where `JWS_ClassName` refers to the JWS class).
- 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-webservices.xml`).
- The XML Schema representation of any Java user-defined types used as parameters or return values to the methods of the JWS files that are specified to be exposed as public operations.
- The WSDL file that publicly describes the Web Service.

After generating all the artifacts, the `jwsc` Ant task compiles the Java and JWS files, packages the compiled classes and generated artifacts into a deployable Web application WAR file, and finally creates an exploded Enterprise Application directory that contains the JAR file. You then deploy this Enterprise Application to WebLogic Server.

**Note:** Although not typical, you can code your JWS file to explicitly implement `javax.ejb.SessionBean`. See “Should You Implement a Stateless Session EJB?” on page 5-16 for details. In this case, `jwsc` packages the Web Service in an EJB JAR file.
and generates the required EJB-related artifacts, such as the ejb-jar.xml and weblogic.ejb-jar.xml deployment descriptor files. However, because this case is not typical, it is assumed in this section that jwsc packages your Web Service in a Web application WAR file, and EJB-specific information is called out only when necessary.

You specify the JWS file or files you want the jwsc Ant task to compile using the <jws> element. If the <jws> element is an immediate child of the jwsc Ant task, then jwsc generates a separate WAR file for each JWS file. If you want all the JWS files, along with their supporting artifacts, to be packaged in a single WAR file, then group all the <jws> elements under a single <module> element. A single WAR file reduces WebLogic server resources and allows the Web Services to share common objects, such as user-defined data types. Using this method you can also specify the same context path for the Web Services; if they are each packaged in their own WAR file then each service must also have a unique context path.

When you use the <module> element, you can use the <jwsfileset> child element to search for a list of JWS files in one or more directories, rather than list each one individually using <jws>.

The following sections discuss additional important information about jwsc:

- “Specifying the Transport Used to Invoke the Web Service” on page A-18
- “How to Determine the Final Context Root of a WebLogic Web Service” on page A-19
- “Generating Client Artifacts for an Invoked Web Service” on page A-21
- “Updating an Existing Enterprise Application or Web Application” on page A-21

See “Examples” on page A-21 for examples of all the topics discussed in these sections.

### Specifying the Transport Used to Invoke the Web Service

When you program your JWS file, you can use an annotation to specify the transport that clients use to invoke the Web Service, in particular @weblogic.jws.WLHttpTransport or @weblogic.jws.WLJMSTransport. You can specify only one of instance of a particular transport annotation in the JWS file. For example, although you cannot specify two different @WLHttpTransport annotations, you can specify one @WLHttpTransport and one @WLJmsTransport annotation. However, you might not know at the time that you are coding the JWS file which transport best suits your needs. For this reason, it is often better to specify the transport at build-time.

The <jws> element includes the following optional child-elements for specifying the transports (HTTP/S or JMS) that are used to invoke the Web Service:
• **WLHttpTransport**—Specifies the context path and service URI sections of the URL used to invoke the Web Service over the HTTP/S transport, as well as the name of the port in the generated WSDL.

• **WLJMSTransport**—Specifies the context path and service URI sections of the URL used to invoke the Web Service over the JMS transport, as well as the name of the port in the generated WSDL. You also specify the name of the JMS queue and connection factory that you have already configured for JMS transport.

The following guidelines describe the usage of the transport elements for the jwsc Ant task:

• The transports you specify to jwsc always override any corresponding transport annotations in the JWS file. In addition, all attributes of the transport annotation are ignored, even if you have not explicitly specified the corresponding attribute for the transport element, in which case the default value of the transport element attribute is used.

• You can specify both transport elements for a particular JWS file. However, you can specify only one instance of a particular transport element. For example, although you cannot specify two different `<WLHttpTransport>` elements for a given JWS file, you can specify one `<WLHttpTransport>` and one `<WLJmsTransport>` element.

• The value of the serviceURI attribute must be unique for `<WLJMSTransport>` and `<WLHttpTransport>` when you specify both transports in one JWS file.

• All transports associated with a particular JWS file must specify the same contextPath attribute value.

• If you specify more than one transport element for a particular JWS file, the value of the portName attribute for each element must be unique among all elements. This means that you must explicitly specify this attribute if you add more than one transport child element to `<jws>`, because the default value of the element will always be the same and thus cause an error when running the jwsc Ant task.

• If you do not specify any transport, as either one of the transport elements to the jwsc Ant task or a transport annotation in the JWS file, then the Web Service’s default URL corresponds to the default value of the WLHttpTransport element.

**How to Determine the Final Context Root of a WebLogic Web Service**

There are a variety of places where the context root (also called context path) of a WebLogic Web Service can be specified. This section describes how to determine which is the true context root of the service based on its configuration, even if it is has been set in multiple places.
In the context of this discussion, a Web Service context root is the string that comes after the `host:port` portion of the Web Service URL. For example, if the deployed WSDL of a WebLogic Web Service is as follows:

```
```

The context root for this Web Service is `financial`.

The following list describes the order of precedence, from most to least important, of all possible context root specifications:

1. The `contextPath` attribute of the `<module>` element and `<jws>` element (when used as a direct child of the `jwsc` Ant task.)
2. The `contextPath` attribute of the `<WLXXXTransport>` child elements of `<jws>`.
3. The `contextPath` attribute of the `@WLXXXTransport` JWS annotations.
4. The default value of the context root, which is the name of the JWS file without any extension.

Suppose, for example, that you specified the `@WLHttpTransport` annotation in your JWS file and set its `contextPath` attribute to `financial`. If you do not specify any additional `contextPath` attributes in the `jwsc` Ant task in your `build.xml` file, then the context root for this Web Service would be `financial`.

Assume that you then update the `build.xml` file and add a `<WLHttpTransport>` child element to the `<jws>` element that specifies the JWS file and set its `contextPath` attribute to `finance`. If, however, you then group the `<jws>` element (including its child `<WLHttpTransport>` element) under a `<module>` element, and set its `contextPath` attribute to `money`, then the context root of the Web Service would now be `money`.

If you do not specify any `contextPath` attribute in either the JWS file or the `jwsc` Ant task, then the context root of the Web Service is the default value: the name of the JWS file without its `*.java` extension. This means that if you have not specified either the `@WLXXXTransport` annotations or `<WLXXXTransport>` child elements of `<jws>`, but group two or more `<jws>` elements under a `<module>` element, then you must specify the `contextPath` attribute of `<module>` to specify the common context root used by all the Web Services in the module. This is because the default context roots for all the Web Services in the module are most likely going to be different (due to different names of the implementing JWS files), which is not allowed in a single WAR file.
Generating Client Artifacts for an Invoked Web Service

If one or more of the JWS files to be compiled itself includes an invoke of a different Web Service, then you can use the \texttt{<clientgen>} element of \texttt{jwsc} to generate and compile the required client component files, such as the JAX-RPC 	exttt{Stub} and 	exttt{Service} interface implementations for the particular Web Service you want to invoke. These files are packaged in the generated WAR file so as to make them available to the invoking Web Service.

Updating an Existing Enterprise Application or Web Application

Typically, \texttt{jwsc} generates a new Enterprise Application exploded directory at the location specified by the \texttt{destDir} attribute. However, if you specify an \textit{existing} Enterprise Application as the destination directory, \texttt{jwsc} updates any existing \texttt{application.xml} file with the new Web Services information.

Similarly, \texttt{jwsc} typically generates new Web application deployment descriptors (\texttt{web.xml} and \texttt{weblogic.xml}) that describe the generated Web application. If, however, you have an existing Web application to which you want to add Web Services, you can use the \texttt{<descriptor>} child element of the \texttt{<module>} element to specify existing \texttt{web.xml} and \texttt{weblogic.xml} files; in this case, \texttt{jwsc} copies these files to the \texttt{destDir} directory and adds new information to them. Use the standard Ant \texttt{<fileset>} element to copy the other existing Web application files to the \texttt{destDir} directory.

\textbf{WARNING:} The existing \texttt{web.xml} and \texttt{weblogic.xml} files pointed to by the \texttt{<descriptor>} element must be XML Schema-based, not DTD-based which will cause the \texttt{jwsc} Ant task to fail with a validation error.

\textbf{Taskdef Classname}

\begin{verbatim}
<taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
\end{verbatim}

\textbf{Examples}

The following examples show how to use the \texttt{jwsc} Ant task by including it in a \texttt{build-service} target of the \texttt{build.xml} Ant file that iteratively develops your Web Service. See Chapter 3, “Common Web Services Use Cases and Examples,” and Chapter 4, “Iterative Development of WebLogic Web Services,” for samples of complete \texttt{build.xml} files that contain many other targets that are useful when iteratively developing a WebLogic Web Service, such as \texttt{clean}, \texttt{deploy}, \texttt{client}, and \texttt{run}.
The following sample shows a very simple usage of `jwsc`:

```xml
<target name="build-service">
  <jwsc srcdir="src" destdir="output/TestEar">
    <jws file="examples/webservices/jwsc/TestServiceImpl.java"/>
  </jwsc>
</target>
```

In the preceding example, the JWS file called `TestServiceImpl.java` is located in the `src/examples/webservices/jwsc` sub-directory of the directory that contains the `build.xml` file. The `jwsc` Ant task generates the Web Service artifacts in the `output/TestEar` sub-directory. In addition to the Web Service JAR file, the `jwsc` Ant task also generates the `application.xml` file that describes the Enterprise Application in the `output/TestEar/META-INF` directory.

The following example shows a more complicated use of `jwsc`:

```xml
<path id="add.class.path">
  <pathelement path="${myclasses-dir}"/>
  <pathelement path="${java.class.path}"/>
</path>
...

<target name="build-service2">
  <jwsc srcdir="src" destdir="output/TestEar" verbose="on" debug="on" keepGenerated="yes" classpathref="add.class.path">
    <jws file="examples/webservices/jwsc/TestServiceImpl.java"/>
    <jws file="examples/webservices/jwsc/AnotherTestServiceImpl.java"/>
    <jws file="examples/webservices/jwsc/SecondTestServiceImpl.java"/>
  </jwsc>
</target>
```

The preceding example shows how to enable debugging and verbose output, and how to specify that `jwsc` not regenerate any existing temporary files in the output directory. The example shows
how to use `classpathref` attribute to add to the standard CLASSPATH by referencing a path called `add.class.path` that has been specified elsewhere in the `build.xml` file using the standard Ant `<path>` target.

The example also shows how to specify multiple JWS files, resulting in separate Web Services packaged in their own Web application WAR files, although all are still deployed as part of the same Enterprise Application. If you want all three Web Services packaged in a single WAR file, group the `<jws>` elements under a `<module>` element, as shown in the following example:

```xml
<target name="build-service3">
  <jwsc
    srcdir="src"
    destdir="output/TestEar">
    <module contextPath="test" name="myJar">
      <jws file="examples/webservices/jwsc/TestServiceImpl.java"/>
      <jws file="examples/webservices/jwsc/AnotherTestServiceImpl.java"/>
      <jws file="examples/webservices/jwsc/SecondTestServiceImpl.java"/>
    </module>
  </jwsc>
</target>
```

The preceding example shows how to package all three Web Services in a WAR file called `myJAR.war`, located at the top level of the Enterprise Application exploded directory. The `contextPath` attribute of `<module>` specifies that the context path of all three Web Services is `test`; this value overrides any context path specified in a transport annotation of the JWS files.

The following example shows how to specify that the Web Service can be invoked using all transports (HTTP/HTTPS/JMS):

```xml
<target name="build-service4">
  <jwsc
    srcdir="src"
    destdir="output/TestEar">
    <jws file="examples/webservices/jwsc/TestServiceImpl.java">
      <WLHttpTransport
        contextPath="TestService" serviceUri="TestService"
        portName="TestServicePortHTTP"/>
      <WLJmsTransport
        contextPath="TestService" serviceUri="JMSTestService"
        portName="TestServicePortJMS"/>  ```
The preceding example also shows how to use the `<clientgen>` element to generate and include the client-side artifacts (such as the JAX-RPC Stub and Service implementations) of the Web Service described by http://examples.org/complex/ComplexService?WSDL. This indicates that the TestServiceImpl.java JWS file, in addition to implementing a Web Service, must also acts as a client to the ComplexService Web Service and must include Java code to invoke operations of ComplexService.

The following example is very similar to the preceding one, except that it groups the `<jws>` elements under a `<module>` element:

```
<target name="build-service5">
  <jwsc srcdir="src" destdir="output/TestEar">
    <module contextPath="TestService" >
      <jws file="examples/webservices/jwsc/TestServiceImpl.java">
        <WLHttpTransport
          serviceUri="TestService"
          portName="TestServicePort1"/>
      </jws>
      <jws file="examples/webservices/jwsc/AnotherTestServiceImpl.java" />
      <jws file="examples/webservices/jwsc/SecondTestServiceImpl.java" />
    </module>
  </jwsc>
</target>
```
In the preceding example, the individual transport elements no longer define their own contextPath attributes; rather, the parent <module> element defines it instead. This improves maintenance and understanding of what jwsc actually does. Also note that the <clientgen> element is a child of <module>, and not <jws> as in the previous example.

The following example show how to use the <jwsfileset> element:

```xml
<target name="build-service6">
  <jwsc srcdir="src" destdir="output/TestEar" >
    <module contextPath="test" name="myJar" >
      <jwsfileset srcdir="src/examples/webservices/jwsc" >
        <include name="**/*.java" />
      </jwsfileset>
    </module>
  </jwsc>
</target>
```

In the example, jwsc searches for *.java files in the directory src/examples/webservices/jwsc, relative to the directory that contains build.xml, determines which Java files contain JWS annotations, and then processes each file as if it had been specified with a <jws> child element of <module>. The <include> element is a standard Ant element, described in the documentation for the standard <FilesSet> task.

The following example shows how to specify that jwsc Ant task not create new Web application deployment descriptors, but rather, add to existing ones:

```xml
<target name="build-service7">
  <jwsc srcdir="src" destdir="output/TestEar" >
    <module contextPath="test" name="myJar" explode="true" >
      <jws file="examples/webservices/jwsc/AnotherTestServiceImpl.java" />
      <FileSet dir="webapp" >
        <include name="**/*.java" />
      </FileSet>
      <descriptor file="webapp/WEB-INF/web.xml" />
      <descriptor file="webapp/WEB-INF/weblogic.xml" />
    </module>
  </jwsc>
</target>
```
In the preceding example, the `explode="true"` attribute of `<module>` specifies that the generated Web application should be in exploded directory format, rather than the default WAR archive file. The `<descriptor>` child elements specify jwsc should copy the existing `web.xml` and `weblogic.xml` files, located in the `webapp/WEB-INF` subdirectory of the directory that contains the `build.xml` file, to the new Web application exploded directory, and that new Web Service information from the specified JWS file should be added to the files, rather than jwsc creating new ones. The example also shows how to use the standard Ant `<FileSet>` task to copy additional files to the generated WAR file; if any of the copied files are Java files, the jwsc Ant task compiles the files and puts the compiled classes into the `classes` directory of the Web application.

Attributes and Child Elements of the jwsc Ant Task

The jwsc Ant task has a variety of attributes and two child elements: `<jws>` and `<module>`. The `<module>` element simply groups one or more JWS files (also specified with the `<jws>` element) into a single module (WAR file); if you do not specify `<module>`, then each JWS file is packaged into its own module, or WAR file.

The `<jws>` element (when used as either a child element of `<jwsc>` or `<module>`) has three optional child elements: `<WLHttpTransport>`, `<WLHttpsTransport>`, and `<WLJMSTransport>`. See “Specifying the Transport Used to Invoke the Web Service” on page A-18 for common information about using the transport elements.

The `<clientgen>` and `<descriptor>` elements are children only of the elements that generate modules: either the actual `<module>` element itself, or `<jws>` when used as a child of jwsc, rather than a child of `<module>`.

The `<jwsfileset>` element can be used only as a child of `<module>`.

The following graphic describes the hierarchy of the jwsc Ant task.
The table in the following section describes the attributes of the jwsc Ant task. See “Standard Ant Attributes and Child Elements That Apply to jwsc” on page A-31 for the list of attributes associated with the standard Ant javac task that you can also set for the jwsc Ant task.
## WebLogic-Specific jwsc Attributes

### Table A-3 Attributes of the jwsc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationXml</td>
<td>Specifies the full name and path of the application.xml deployment descriptor of the Enterprise Application. If you specify an existing file, the jwsc Ant task updates it to include the Web Services information. If the file does not exist, jwsc creates it. The jwsc Ant task also creates or updates the corresponding weblogic-application.xml file in the same directory. If you do not specify this attribute, jwsc creates or updates the file destDir/META-INF/application.xml, where destDir is the jwsc attribute.</td>
<td>No.</td>
</tr>
<tr>
<td>destdir</td>
<td>The full pathname of the directory that will contain the compiled JWS files, XML Schemas, WSDL, and generated deployment descriptor files, all packaged into a JAR or WAR file. The jwsc Ant task creates an exploded Enterprise Application at the specified directory, or updates one if you point to an existing application directory. The jwsc task generates the JAR or WAR file that implements the Web Service in this directory, as well as other needed files, such as the application.xml file in the META-INF directory; the jwsc Ant task updates an existing application.xml file if it finds one, or creates a new one if not. Use the applicationXML attribute to specify a different application.xml from the default.</td>
<td>Yes.</td>
</tr>
<tr>
<td>destEncoding</td>
<td>Specifies the character encoding of the output files, such as the deployment descriptors and XML files. Examples of character encodings are SHIFT-JIS and UTF-8. The default value of this attribute is UTF-8.</td>
<td>No.</td>
</tr>
</tbody>
</table>
dotNetStyle Specifies that the jwsc Ant task should generate a .NET-style Web Service. In particular, this means that, in the WSDL of the Web Service, the value of the `name` attribute of the `<part>` element that corresponds to the return parameter is `parameters` rather than `returnParameters`. This applies only to document-literal-wrapped Web Services. The valid values for this attribute are `true` and `false`. The default value is `true`, which means .NET-style Web Service are generated by default.

enableAsyncService Specifies whether the Web Service is using one or more of the asynchronous features of WebLogic Web Service: Web Service reliable messaging, asynchronous request-response, buffering, or conversations. In the case of Web Service reliable messaging, you must ensure that this attribute is enabled for both the reliable Web Service and the Web Service that is invoking the operations reliably. In the case of the other features (conversations, asynchronous request-response, and buffering), the attribute must be enabled only on the client Web Service. When this attribute is set to `true` (default value), WebLogic Server automatically deploys internal modules that handle the asynchronous Web Service features. Therefore, if you are not using any of these features in your Web Service, consider setting this attribute to `false` so that WebLogic Server does not waste resources by deploying unneeded internal modules.

 Valid values for this attribute are `true` and `false`. The default value is `true`.

Note: This attribute is deprecated as of Version 9.2 of WebLogic Server.

jaxRpcByteArrayStyle This attribute is used for mapping `byte[]` and `Byte[]` data types to `xsd:base64Binary` type for a RPC/encoded Web Service. It applies only for JAX-RPC based Web Services. When this attribute is set to `true`, generated mappings for `byte[]` and `Byte[]` data types are compliant with the JAX-RPC guidelines.

 Valid values for this attribute are `true` and `false`. The default value is `false`.

---

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dotNetStyle</td>
<td>Specifies that the jwsc Ant task should generate a .NET-style Web Service. In particular, this means that, in the WSDL of the Web Service, the value of the <code>name</code> attribute of the <code>&lt;part&gt;</code> element that corresponds to the return parameter is <code>parameters</code> rather than <code>returnParameters</code>. This applies only to document-literal-wrapped Web Services. The valid values for this attribute are <code>true</code> and <code>false</code>. The default value is <code>true</code>, which means .NET-style Web Service are generated by default.</td>
<td>No</td>
</tr>
<tr>
<td>enableAsyncService</td>
<td>Specifies whether the Web Service is using one or more of the asynchronous features of WebLogic Web Service: Web Service reliable messaging, asynchronous request-response, buffering, or conversations. In the case of Web Service reliable messaging, you must ensure that this attribute is enabled for both the reliable Web Service and the Web Service that is invoking the operations reliably. In the case of the other features (conversations, asynchronous request-response, and buffering), the attribute must be enabled only on the client Web Service. When this attribute is set to <code>true</code> (default value), WebLogic Server automatically deploys internal modules that handle the asynchronous Web Service features. Therefore, if you are not using any of these features in your Web Service, consider setting this attribute to <code>false</code> so that WebLogic Server does not waste resources by deploying unneeded internal modules. Valid values for this attribute are <code>true</code> and <code>false</code>. The default value is <code>true</code>. Note: This attribute is deprecated as of Version 9.2 of WebLogic Server.</td>
<td>No</td>
</tr>
<tr>
<td>jaxRpcByteArrayStyle</td>
<td>This attribute is used for mapping <code>byte[]</code> and <code>Byte[]</code> data types to <code>xsd:base64Binary</code> type for a RPC/encoded Web Service. It applies only for JAX-RPC based Web Services. When this attribute is set to <code>true</code>, generated mappings for <code>byte[]</code> and <code>Byte[]</code> data types are compliant with the JAX-RPC guidelines. Valid values for this attribute are <code>true</code> and <code>false</code>. The default value is <code>false</code>.</td>
<td>No</td>
</tr>
</tbody>
</table>
keepGenerated Specifies whether the Java source files and artifacts generated by this Ant task should be regenerated if they already exist.

If you specify no, new Java source files and artifacts are always generated and any existing artifacts are overwritten.

If you specify yes, the Ant task regenerates only those artifacts that have changed, based on the timestamp of any existing artifacts.

Valid values for this attribute are yes or no. The default value is no.

sourcepath The full pathname of top-level directory that contains the Java files referenced by the JWS file, such as JavaBeans used as parameters or user-defined exceptions. The Java files are in sub-directories of the sourcepath directory that correspond to their package names. The sourcepath pathname can be either absolute or relative to the directory which contains the Ant build.xml file.

For example, if sourcepath is /src and the JWS file references a JavaBean called MyType.java which is in the webservices.financial package, then this implies that the MyType.java Java file is stored in the /src/webservices/financial directory.

The default value of this attribute is the value of the srcdir attribute. This means that, by default, the JWS file and the objects it references are in the same package. If this is not the case, then you should specify the sourcepath accordingly.
Standard Ant Attributes and Child Elements That Apply to jwsc
In addition to the WebLogic-defined jwsc attributes, you can also define the following standard javac attributes; see the Ant documentation for additional information about each attribute:

- bootclasspath
- bootClasspathRef
- classpath
- classpathRef
- compiler
- debug
- debugLevel
- depend
- deprecation
- destdir
- encoding
- extdirs

Table A-3 Attributes of the jwsc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcdir</td>
<td>The full pathname of top-level directory that contains the JWS file you want to compile (specified with the file attribute of the &lt;jws&gt; child element). The JWS file is in sub-directories of the srcdir directory that corresponds to its package name. The srcdir pathname can be either absolute or relative to the directory which contains the Ant build.xml file. For example, if srcdir is /src and the JWS file called MyService.java is in the webservices.financial package, then this implies that the MyService.java JWS file is stored in the /src/webservices/financial directory.</td>
<td>Yes.</td>
</tr>
<tr>
<td>srcEncoding</td>
<td>Specifies the character encoding of the input files, such as the JWS file or configuration XML files. Examples of character encodings are SHIFT-JIS and UTF-8. The default value of this attribute is the character encoding set for the JVM.</td>
<td>No.</td>
</tr>
</tbody>
</table>
You can also use the following standard Ant child elements with the jwsc Ant task:

- `<SourcePath>`
- `<Classpath>`
- `<Extdirs>`

**jws**

The `<jws>` element specifies the name of a JWS file that implements your Web Service and for which the Ant task should generate Java code and supporting artifacts and then package into a deployable WAR file inside of an Enterprise Application.

You can specify the `<jws>` element in the following two different levels of the jwsc element hierarchy:

- An immediate child element of the jwsc Ant task. In this case, jwsc generates a separate WAR file for each JWS file. You typically use this method if you are specifying just one JWS file to the jwsc Ant task.

- A child element of the `<module>` element, which in turn is a child of jwsc. In this case, jwsc generates a single WAR file that includes all the generated code and artifacts for all
the JWS files grouped within the <module> element. This method is useful if you want all
JWS files to share supporting files, such as common Java data types.

You are required to specify either a <jws> or <module> child element of jwsc.

See Figure A-1 for a visual description of where this element fits in the jwsc element hierarchy.
See “Examples” on page A-21 for examples of using the element.

You can use the standard Ant <FileSet> child element with the <jws> element of jwsc.

The following table describes the attributes of the <jws> element. The description specifies
whether the attribute applies in the case that <jws> is a child of jwsc, is a child of <module> or
in both cases.
Table A-4  Attributes of the <jws> Element of the jwsc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiledWsd</td>
<td>Full pathname of the JAR file generated by the wsdlc Ant task based on an existing WSDL file. The JAR file contains the JWS interface file that implements a Web Service based on this WSDL, as well as data binding artifacts for converting parameter and return value data between its Java and XML representations; the XML Schema section of the WSDL defines the XML representation of the data. You use this attribute only in the “starting from WSDL” use case, in which you first use the wsdlc Ant task to generate the JAR file, along with the JWS file that implements the generated JWS interface. After you update the JWS implementation class with business logic, you run the jwsc Ant task to generate a deployable Web Service, using the file attribute to specify this updated JWS implementation file. You do not use the compiledWsd attribute for the “starting from Java” use case in which you write your JWS file from scratch and the WSDL file that describes the Web Service is generated by the WebLogic Web Services runtime. Applies to &lt;jws&gt; when used as a child of both jwsc and &lt;module&gt;.</td>
<td>Only required for the “starting from WSDL” use case.</td>
</tr>
</tbody>
</table>
| contextPath | Context root of the Web Service. For example, assume the deployed WSDL of a WebLogic Web Service is as follows:  


  The context root for this Web Service is financial. The value of this attribute overrides any other context path set for the JWS file. This includes the transport-related JWS annotations, as well as the transport-related child elements of <jws>. The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is HelloWorldImpl.java, then the default value of its contextPath is HelloWorldImpl. Applies only when <jws> is a direct child of jwsc.                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | No.                                                                      |
**Attributes of the `<jws>` Element of the jwsc Ant Task**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>explode</td>
<td>Specifies whether the generated WAR file that contains the deployable Web Service is in exploded directory format or not.</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>Valid values for this attribute are true or false. Default value is false, which means that jwsc generates an actual WAR archive file, and not an exploded directory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Applies only when <code>&lt;jws&gt;</code> is a direct child of jwsc.</td>
<td></td>
</tr>
<tr>
<td>file</td>
<td>The name of the JWS file that you want to compile. The jwsc Ant task looks for the file in the srcdir directory.</td>
<td>Yes.</td>
</tr>
<tr>
<td></td>
<td>Applies to <code>&lt;jws&gt;</code> when used as a child of both jwsc and <code>&lt;module&gt;</code>.</td>
<td></td>
</tr>
<tr>
<td>includeSchemas</td>
<td>The full pathname of the XML Schema file that describes an XMLBeans parameter or return value of the Web Service.</td>
<td>Required if you are using an XMLBeans data type as a parameter or return value.</td>
</tr>
<tr>
<td></td>
<td>To specify more than one XML Schema file, use either a comma or semi-colon as a delimiter: includeSchemas=&quot;po.xsd, customer.xsd&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This attribute is only supported in the case where the JWS file explicitly uses an XMLBeans data type as a parameter or return value of a Web Service operation. If you are not using the XMLBeans data type, the jwsc Ant task returns an error if you specify this attribute.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Additionally, you can use this attribute only for Web Services whose SOAP binding is document-literal-bare. Because the default SOAP binding of a WebLogic Web Service is document-literal-wrapped, the corresponding JWS file must include the following JWS annotation: @SOAPBinding( style=SOAPBinding.Style.DOCUMENT, use=SOAPBinding.Use.LITERAL, parameterStyle=SOAPBinding.ParameterStyle.BARE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For more information on XMLBeans, see <a href="http://dev2dev.bea.com/technologies/xmlbeans/index.jsp">http://dev2dev.bea.com/technologies/xmlbeans/index.jsp</a>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Applies to <code>&lt;jws&gt;</code> when used as a child of both jwsc and <code>&lt;module&gt;</code>.</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** 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.

---

Programming Web Services for WebLogic Server  
A-35
module

The `<module>` element groups one or more `<jws>` elements together so that their generated code and artifacts are packaged in a single Web application (WAR) file. The `<module>` element is a child of the main `<jwsc>` Ant task.

You can group only Web Services implemented with the same backend component (Java class or stateless session EJB) under a single `<module>` element; you cannot mix and match. By default, `<jwsc>` always implements your Web Service as a plain Java class; the only exception is if you have explicitly implemented `javax.ejb.SessionBean` in your JWS file. This means, for example, that if one of the JWS files specified by the `<jws>` child element of `<module>` implements
javax.ejb.SessionBean, then all its sibling <jws> files must also implement javax.ejb.SessionBean. If this is not possible, then you can not group all the JWS files under a single <module>.

The Web Services within a module must have the same contextPath, but must have unique serviceURIs. You can set the common contextPath by specifying it as an attribute to the <module> element, or ensuring that the @WLXXXTransport annotations and/or <WLXXXTrasnsport> elements for each Web Service have the same value for the contextPath attribute. The jwsc Ant task validates these values and returns an error if they are not unique.

You must specify at least one <jws> child element of <module>.

See Figure A-1 for a visual description of where this element fits in the jwsc element hierarchy. See “Examples” on page A-21 for examples of using the element.

The following table describes the attributes of the <module> element.
### Table A-5 Attributes of the `<module>` Element of the jwsc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
</table>
| contextPath | Context root of all the Web Services contained in this module. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: 


   The context root for this Web Service is financial. 

   The value of this attribute overrides any other context path set for any of the JWS files contained in this module. This includes the transport-related JWS annotations, as well as the transport-related child elements of `<jws>`. 

   The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is HelloWorldImpl.java, then the default value of its contextPath is HelloWorldImpl. | Only required to ensure that the context roots of multiple Web Services in a single WAR are the same. See “How to Determine the Final Context Root of a WebLogic Web Service” on page A-19 |
| explode | Specifies whether the generated WAR file that contains the deployable Web Service(s) is in exploded directory format or not. Valid values for this attribute are true or false. Default value is false, which means that jwsc generates an actual WAR archive file, and not an exploded directory. | No. |
Use the `WLHttpTransport` element 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.

The `<WLHttpTransport>` element is a child of the `<jws>` element.

You can specify one or zero `<WLHttpTransport>` elements for a given JWS file.

See “Specifying the Transport Used to Invoke the Web Service” on page A-18 for guidelines to follow when specifying this element.

See Figure A-1 for a visual description of where this element fits in the `jwsc` element hierarchy. See “Examples” on page A-21 for examples of using the element.
The following table describes the attributes of `<WLHttpTransport>`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>contextPath</td>
<td>Context root of the Web Service.</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>For example, assume the deployed WSDL of a WebLogic Web Service is as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The contextPath for this Web Service is financial.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value of this attribute is the name of the JWS file,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>without its extension. For example, if the name of the JWS file is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HelloWorldImpl.java, then the default value of its contextPath</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is HelloWorldImpl.</td>
<td></td>
</tr>
<tr>
<td>serviceUri</td>
<td>Web Service URI portion of the URL.</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>For example, assume the deployed WSDL of a WebLogic Web Service is as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The serviceUri for this Web Service is GetQuote.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value of this attribute is the name of the JWS file,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>without its extension. For example, if the name of the JWS file is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HelloWorldImpl.java, then the default value of its serviceUri</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is HelloWorldImpl.</td>
<td></td>
</tr>
<tr>
<td>portName</td>
<td>The name of the port in the generated WSDL. This attribute maps to the</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>the name attribute of the <code>&lt;port&gt;</code> element in the WSDL.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value of this attribute is based on the @javax.jws.WebService</td>
<td></td>
</tr>
<tr>
<td></td>
<td>annotation of the JWS file. In particular, the default portName is the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value of the name attribute of @WebService annotation, plus the actual text</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SoapPort. For example, if @WebService.name is set to MyService, then the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>default portName is MyServiceSoapPort.</td>
<td></td>
</tr>
</tbody>
</table>

**WLHttpsTransport**

**WARNING:** The `<WLHttpsTransport>` element is deprecated as of version 9.2 of WebLogic Server. You should use the `<WLHttpTransport>` element instead because it now
supports both the HTTP and HTTPS protocols. If you want client applications to access the Web Service using only the HTTPS protocol, then you must specify the @weblogic.jws.security.UserDataConstraint JWS annotation in your JWS file.

Use the WLHttpsTransport element to specify the context path and service URI sections of the URL used to invoke the Web Service over the secure HTTPS transport, as well as the name of the port in the generated WSDL.

The <WLHttpsTransport> element is a child of the <jws> element.

You can specify one or zero <WLHttpsTransport> elements for a given JWS file.

See “Specifying the Transport Used to Invoke the Web Service” on page A-18 for guidelines to follow when specifying this element.

See Figure A-1 for a visual description of where this element fits in the jwsc element hierarchy.

The following table describes the attributes of <WLHttpsTransport>.
Use the **WLJMSTransport** element to specify the context path and service URI sections of the URL used to invoke the Web Service over the JMS transport, as well as the name of the port in the generated WSDL. This attribute maps to the `name` attribute of the `<port>` element in the WSDL.

The default value of this attribute is based on the `@javax.jws.WebService` annotation of the JWS file. In particular, the default `portName` is the value of the `name` attribute of the `@WebService` annotation, plus the actual text `SoapPort`. For example, if `@WebService.name` is set to `MyService`, then the default `portName` is `MyServiceSoapPort`.

### Table A-7 Attributes of the `<WLHttpTransport>` Child Element of the `<jws>` Element

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contextPath</code></td>
<td>Context root of the Web Service. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: <code>https://hostname:7001/financial/GetQuote?WSDL</code> The contextPath for this Web Service is <code>financial</code>. The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is <code>HelloWorldImpl.java</code>, then the default value of its <code>contextPath</code> is <code>HelloWorldImpl</code>.</td>
<td>No.</td>
</tr>
<tr>
<td><code>serviceUri</code></td>
<td>Web Service URI portion of the URL. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: <code>https://hostname:7001/financial/GetQuote?WSDL</code> The serviceUri for this Web Service is <code>GetQuote</code>. The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is <code>HelloWorldImpl.java</code>, then the default value of its <code>serviceUri</code> is <code>HelloWorldImpl</code>.</td>
<td>No.</td>
</tr>
<tr>
<td><code>portName</code></td>
<td>The name of the port in the generated WSDL. This attribute maps to the <code>name</code> attribute of the <code>&lt;port&gt;</code> element in the WSDL. The default value of this attribute is based on the <code>@javax.jws.WebService</code> annotation of the JWS file. In particular, the default <code>portName</code> is the value of the <code>name</code> attribute of the <code>@WebService</code> annotation, plus the actual text <code>SoapPort</code>. For example, if <code>@WebService.name</code> is set to <code>MyService</code>, then the default <code>portName</code> is <code>MyServiceSoapPort</code>.</td>
<td>No.</td>
</tr>
</tbody>
</table>
the generated WSDL. You also specify the name of the JMS queue and connection factory that you have already configured for JMS transport.

The `<WLHJmsTransport>` element is a child of the `<jws>` element.

You can specify one or zero `<WLHJmsTransport>` elements for a given JWS file.

See “Specifying the Transport Used to Invoke the Web Service” on page A-18 for guidelines to follow when specifying this element.

See Figure A-1 for a visual description of where this element fits in the jwsc element hierarchy. See “Examples” on page A-21 for examples of using the element.

The following table describes the attributes of `<WLHJmsTransport>`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>contextPath</td>
<td>Context root of the Web Service.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>For example, assume the deployed WSDL of a WebLogic Web Service is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>as follows: <a href="http://hostname:7001/financial/GetQuote?WSDL">http://hostname:7001/financial/GetQuote?WSDL</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The contextPath for this Web Service is financial.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value of this attribute is the name of the JWS file, without its</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extension. For example, if the name of the JWS file is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HelloWorldImpl.java, then the default value of its contextPath is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HelloWorldImpl.</td>
<td></td>
</tr>
<tr>
<td>serviceUri</td>
<td>Web Service URI portion of the URL.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>For example, assume the deployed WSDL of a WebLogic Web Service is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>as follows: <a href="http://hostname:7001/financial/GetQuote?WSDL">http://hostname:7001/financial/GetQuote?WSDL</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The serviceUri for this Web Service is GetQuote.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value of this attribute is the name of the JWS file, without its</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extension. For example, if the name of the JWS file is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HelloWorldImpl.java, then the default value of its serviceUri is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HelloWorldImpl.</td>
<td></td>
</tr>
</tbody>
</table>
clientgen

Use the `<clientgen>` element if the JWS file itself invokes another Web Service and you want the `jwsc` Ant task to automatically generate and compile the required client-side artifacts and package them in the Web application WAR file together with the Web Service. The client-side 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.

Table A-8 Attributes of the `<WLJMSTransport>` Child Element of the `<jws>` Element

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>portName</td>
<td>The name of the port in the generated WSDL. This attribute maps to the <code>name</code> attribute of the <code>&lt;port&gt;</code> element in the WSDL. The default value of this attribute is based on the <code>@javax.jws.WebService</code> annotation of the JWS file. In particular, the default portName is the value of the <code>name</code> attribute of <code>@WebService</code> annotation, plus the actual text <code>SoapPort</code>. For example, if <code>@WebService.name</code> is set to <code>MyService</code>, then the default portName is <code>MyServiceSoapPort</code>.</td>
<td>No.</td>
</tr>
<tr>
<td>queue</td>
<td>The JNDI name of the JMS queue that you have configured for the JMS transport. See “Using JMS Transport as the Connection Protocol” on page 7-1 for details about using JMS transport. The default value of this attribute, if you do not specify it, is <code>weblogic.wsee.DefaultQueue</code>. You must still create this JMS queue in the WebLogic Server instance to which you deploy your Web Service.</td>
<td>No.</td>
</tr>
<tr>
<td>connectionFactory</td>
<td>The JNDI name of the JMS connection factory that you have configured for the JMS transport. The default value of this attribute is the default JMS connection factory for your WebLogic Server instance.</td>
<td>No.</td>
</tr>
</tbody>
</table>
See Figure A-1 for a visual description of where this element fits in the `jwsc` element hierarchy. See “Examples” on page A-21 for examples of using the element.

You can specify the standard Ant `<sysproperty>` child element to specify properties required by the Web Service from which you are generating client-side artifacts. For example, if the Web Service is secured, you can use the `javax.xml.rpc.security.auth.username|password` properties to set the authenticated username and password. See the Ant documentation for the java Ant task for additional information about `<sysproperty>`.

The following table describes the attributes of the `<clientgen>` element.
### Table A-9  Attributes of the `<clientgen>` Element

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoDetectWrapped</td>
<td>Specifies whether the <code>&lt;jwsc&gt;</code> Ant task should try to determine whether the parameters and return type of document-literal Web Services are of type <em>wrapped</em> or <em>bare</em>. When the <code>&lt;jwsc&gt;</code> Ant task parses a WSDL file to create the JAX-RPC stubs, it attempts to determine whether a document-literal Web Service uses wrapped or bare parameters and return types based on the names of the XML Schema elements, the name of the operations and parameters, and so on. Depending on how the names of these components match up, the <code>&lt;jwsc&gt;</code> Ant task makes a best guess as to whether the parameters are wrapped or bare. In some cases, however, you might want the Ant task to <em>always</em> assume that the parameters are of type bare; in this case, set the <code>autoDetectWrapped</code> attribute to <code>False</code>. Valid values for this attribute are <code>True</code> or <code>False</code>. The default value is <code>True</code>.</td>
<td>No.</td>
</tr>
<tr>
<td>handlerChainFile</td>
<td>Specifies the name of the XML file that describes the client-side SOAP message handlers that execute when the JWS file invokes a Web Service. Each handler specified in the file executes twice: • directly before the JWS sends the SOAP request to the invoked Web Service. • directly after the JWS receives the SOAP response from the invoked Web Service. If you do not specify this attribute, then no client-side handlers execute when the Web Service is invoked from the JWS file, even if they are in your CLASSPATH. See “Creating and Using Client-Side SOAP Message Handlers” on page 9-22 for details and examples about creating client-side SOAP message handlers.</td>
<td>No.</td>
</tr>
</tbody>
</table>
generateAsyncMethods Specifies whether the jwsc Ant task should include methods in the generated JAX-RPC stubs that the JWS file can use to invoke a Web Service operation asynchronously.

For example, if you specify True (which is also the default value), and one of the Web Service operations in the WSDL is called getQuote, then the jwsc Ant task also generates a method called getQuoteAsync in the JAX-RPC stubs which the JWS file can use instead of the original getQuote method. This asynchronous flavor of the operation also has an additional parameter, of data type weblogic.wsee.async.AsyncPreCallContext, that the JWS file can use to set asynchronous properties, contextual variables, and so on.

See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 for full description and procedures about this feature.

Note: If the operation of the Web Service being invoked in the JWS file is marked as one-way, the jwsc Ant task never generates the asynchronous flavor of the JAX-RPC stub, even if you explicitly set the generateAsyncMethods attribute to True.

Valid values for this attribute are True or False. The default value is True, which means the asynchronous methods are generated by default.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>generateAsyncMethods</td>
<td>Specifies whether the jwsc Ant task should include methods in the generated JAX-RPC stubs that the JWS file can use to invoke a Web Service operation asynchronously.</td>
<td>No.</td>
</tr>
</tbody>
</table>
Ant Task Reference

Table A-9  Attributes of the <clientgen> Element

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>generatePolicyMethods</td>
<td>Specifies whether the jwsc Ant task should include WS-Policy-loading methods in the generated JAX-RPC stubs. You can use these methods in your JWS file, when invoking the Web Service, to load a local WS-Policy file. If you specify True, four flavors of a method called getXXXSoapPort() are added as extensions to the JAX-RPC Service interface in the generated client stubs, where XXX refers to the name of the Web Service. You can program the JWS file to use these methods to load and apply local WS-Policy files, rather than apply any WS-Policy file deployed with the Web Service itself. You can specify in the JWS file whether the local WS-Policy file applies to inbound, outbound, or both SOAP messages and whether to load the local WS-Policy file from an InputStream or a URI. Valid values for this attribute are True or False. The default value is False, which means the additional methods are not generated. See “Using a Client-Side Security WS-Policy File” on page 9-27 for more information.</td>
<td>No.</td>
</tr>
<tr>
<td>includeGlobalTypes</td>
<td>Specifies that the jwsc Ant task should generate Java representations of all XML Schema data types in the WSDL, rather than just the data types that are explicitly used in the Web Service operations. Valid values for this attribute are True or False. The default value is False, which means that jwsc generates Java representations for only the actively-used XML data types.</td>
<td></td>
</tr>
</tbody>
</table>
When the `jwsc` Ant task is generating the Java equivalent to XML Schema data types in the WSDL file, and the task encounters an XML complex type with a single enclosing sequence with a single element with the `maxOccurs` attribute equal to `unbounded`, the task generates, by default, a Java structure whose name is the lowest named enclosing complex type or element. To change this behavior so that the task generates a literal array instead, set the `jaxRPCWrappedArrayStyle` to `false`.

Valid values for this attribute are `true` or `false`. The default value is `true`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaxRPCWrappedArrayStyle</td>
<td>When the <code>jwsc</code> Ant task is generating the Java equivalent to XML Schema data types in the WSDL file, and the task encounters an XML complex type with a single enclosing sequence with a single element with the <code>maxOccurs</code> attribute equal to <code>unbounded</code>, the task generates, by default, a Java structure whose name is the lowest named enclosing complex type or element. To change this behavior so that the task generates a literal array instead, set the <code>jaxRPCWrappedArrayStyle</code> to <code>false</code>. Valid values for this attribute are <code>true</code> or <code>false</code>. The default value is <code>true</code>.</td>
<td>No.</td>
</tr>
<tr>
<td>packageName</td>
<td>Package name into which the generated JAX-RPC client interfaces and stub files are packaged. BEA recommends you use all lower-case letters for the package name.</td>
<td>Yes.</td>
</tr>
<tr>
<td>serviceName</td>
<td>Name of the Web Service in the WSDL file for which the corresponding client-side artifacts should be generated. The Web Service name corresponds to the <code>&lt;service&gt;</code> element in the WSDL file. The generated JAX-RPC mapping file and client-side copy of the WSDL file will use this name. For example, if you set <code>serviceName</code> to <code>CuteService</code>, the JAX-RPC mapping file will be called <code>cuteService_java_wsd_mapping.xml</code> and the client-side copy of the WSDL will be called <code>CuteService_saved_wsd.wsd</code>. This attribute is required only if the WSDL file contains more than one <code>&lt;service&gt;</code> element. The Ant task returns an error if you do not specify this attribute and the WSDL file contains more than one <code>&lt;service&gt;</code> element.</td>
<td>Yes.</td>
</tr>
<tr>
<td>wsdl</td>
<td>Full path name or URL of the WSDL that describes a Web Service (either WebLogic or non-WebLogic) for which the client artifacts should be generated. The generated JAX-RPC stub factory classes use the value of this attribute in the default constructor.</td>
<td>Yes.</td>
</tr>
</tbody>
</table>
**descriptor**

Use the `<descriptor>` element to specify that, rather than create new Web application deployment descriptors when generating the WAR that will contain the implementation of the Web Service, the `<jwsc>` task should instead copy existing files and update them with the new information. This is useful when you have an existing Web application to which you want to add one or more Web Services. You typically use this element together with the standard `<FileSet>` Ant task to copy other existing Web application artifacts, such as HTML files and Java classes, to the `<jwsc>`-generated Web application.

You can use this element with only the following two deployment descriptor files:

- `web.xml`
- `weblogic.xml`

Use a separate `<descriptor>` element for each deployment descriptor file.

The `<descriptor>` element is a child of either `<module>` or `<jws>`, when the latter is a direct child of the main `<jwsc>` Ant task.

**WARNING:** The existing `web.xml` and `weblogic.xml` files pointed to by the `<descriptor>` element must be XML Schema-based, not DTD-based which will cause the `<jwsc>` Ant task to fail with a validation error.

See Figure A-1 for a visual description of where this element fits in the `<jwsc>` element hierarchy. See “Examples” on page A-21 for examples of using the element.

The following table describes the attributes of the `<descriptor>` element.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Full pathname (either absolute or relative to the directory that contains the build.xml file) of the existing deployment descriptor file. The deployment descriptor must be XML Schema-based, not DTD-based. The <code>&lt;jwsc&gt;</code> Ant task does not update this file directly, but rather, copies it to the newly-generated Web application.</td>
<td>Yes.</td>
</tr>
</tbody>
</table>
jwsfileset

Use the `<jwsfileset>` child element of `<module>` to specify one or more directories in which the jwsc Ant task searches for JWS files to compile. The list of JWS files that jwsc finds is then treated as if each file had been individually specified with the `<jws>` child element of `<module>`.

Use the standard nested elements of the `<FileSet>` Ant task to narrow the search. For example, use the `<include>` element to specify the pattern matching that `<jwsfileset>` should follow when determining the JWS files it should include in the list. See the Ant documentation for details about `<FileSet>` and its nested elements.

See Figure A-1 for a visual description of where this element fits in the jwsc element hierarchy. See “Examples” on page A-21 for examples of using the element.

The following table describes the attributes of the `<jwsfileset>` element.
Use the `<xsdConfig>` child element to specify one or more XMLBeans configuration files, which by convention end in `.xsdconfig`. Use this element if your Web Service uses Apache XMLBeans data types as parameters or return values.
The `<xsdConfig>` element is similar to the standard Ant `<Fileset>` element and has all the same attributes. See the Apache Ant documentation on the `<Fileset>` element for the full list of attributes.

**wsdlc**

The `wsdlc` Ant task generates, from an existing WSDL file, a set of artifacts that together provide a partial Java implementation of the Web Service described by the WSDL file.

By default, it is assumed that the WSDL file includes a single `<service>` element from which the `wsdlc` Ant task generates artifacts. You can, however, use the `srcServiceName` attribute to specify a specific Web Service, in the case that there is more than one `<service>` element in the WSDL file, or use the `srcPortName` attribute to specify a specific port of a Web Service in the case that there is more than one `<port>` child element for a given Web Service.

- A JWS interface file that implements the Web Service described by the WSDL file. The interface includes full method signatures that implement the Web Service operations, and JWS annotations (such as `@WebService` and `@SOAPBinding`) that implement other aspects of the Web Service.
  
  **WARNING:** The JWS interface is generated into a JAR file, neither of which you should ever update. It is discussed in this section only because later you need to specify this JAR file to the `jwsc` Ant task when you compile your JWS implementation file into a 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. The XML Schema of the data types is specified in the WSDL, and the Java representation is generated by the `wsdlc` Ant task.
  
  **WARNING:** These artifacts are generated into a JAR file, along with the JWS interface file, none of which you should ever update. It is discussed in this section only because later you need to specify this JAR file to the `jwsc` Ant task when you compile your JWS implementation file into a Web Service.

- A JWS file that contains a stubbed-out implementation of the generated JWS interface.

- Optional Javadocs for the generated JWS interface.

After running the `wsdlc` Ant task, (which typically you only do once) you update the generated JWS implementation file, in particular by adding Java code to the methods so that they function as you want. The generated JWS implementation file does not initially contain any business logic.
because the \texttt{wsdlc} Ant task obviously does not know how you want your Web Service to function, although it does know the \textit{shape} of the Web Service, based on the WSDL file.

When you code the JWS implementation file, you can also add additional JWS annotations, although you must abide by the following rules:

- The \textit{only} standard JSR-181 JWS annotations you can include in the JWS implementation file are \texttt{@WebService}, \texttt{@HandlerChain}, \texttt{@SOAPMessageHandler}, and \texttt{@SOAPMessageHandlers}. If you specify any other JWS-181 JWS annotations, the \texttt{jwsc} Ant task will return an error when you try to compile the JWS file into a Web Service.

- Additionally, you can specify \textit{only} the \texttt{serviceName} and \texttt{endpointInterface} attributes of the \texttt{@WebService} annotation. Use the \texttt{serviceName} attribute to specify a different \texttt{<service>} WSDL element from the one that the \texttt{wsdlc} Ant task used, in the rare case that the WSDL file contains more than one \texttt{<service>} element. Use the \texttt{endpointInterface} attribute to specify the JWS interface generated by the \texttt{wsdlc} Ant task.

- You can specify any WebLogic-specific JWS annotation that you want.

Finally, after you have coded the JWS file so that it works as you want, iteratively run the \texttt{jwsc} Ant task to generate a complete Java implementation of the Web Service. Use the \texttt{compiledWsdl} attribute of \texttt{jwsc} to specify the JAR file generated by the \texttt{wsdlc} Ant task which contains the JWS interface file and data binding artifacts. By specifying this attribute, the \texttt{jwsc} Ant task does not generate a new WSDL file but instead uses the one in the JAR file. Consequently, when you deploy the Web Service and view its WSDL, the deployed WSDL will look just like the one from which you initially started.

\textbf{Note:} The only potential difference between the original and deployed WSDL is the value of the \texttt{location} attribute of the \texttt{<address>} element of the port(s) of the Web Service. The deployed WSDL will specify the actual hostname and URI of the deployed Web Service, which is most likely different from that of the original WSDL. This difference is to be expected when deploying a real Web Service based on a static WSDL.

See “Creating a Web Service from a WSDL File” on page 3-14 for a complete example of using the \texttt{wsdlc} Ant task in conjunction with \texttt{jwsc}.

\textbf{Taskdef Classname}

\begin{verbatim}
<taskdef name="wsdlc"
    classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
\end{verbatim}
Example

The following excerpt from an Ant build.xml file shows how to use the wsdlc and jwsc Ant tasks together to build a WebLogic Web Service. The build file includes two different targets:

```
generate-from-wsdl that runs the wsdlc Ant task against an existing WSDL file, and
build-service that runs the jwsc Ant task to build a deployable Web Service from the artifacts generated by the wsdlc Ant task:
```

```
<target name="generate-from-wsdl">
  <wsdlc
    srcWsdl="wsdl_files/TemperatureService.wsdl"
    destJwsDir="output/compiledWsdl"
    destImplDir="output/impl"
    packageName="examples.webservices.wsdlc" />
</target>

<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 example, the wsdlc Ant task takes as input the TemperatureService.wsdl file and generates the JAR file that contains the JWS interface and data binding artifacts into the directory output/compiledWsdl. The name of the JAR file is TemperatureService_wsdl.jar. The Ant task also generates a JWS file that contains a stubbed-out implementation of the JWS interface into the output/impl/examples/webservices/wsdlc directory (a combination of the value of the destImplDir attribute and the directory hierarchy corresponding to the specified packageName). The name of the stubbed-out JWS implementation file is based on the name of the <portType> element in the WSDL file that corresponds to the first <service> element. For
example, if the portType name is TemperaturePortType, then the generated JWS implementation file is called TemperaturePortTypeImpl.java.

After running wsdlc, you code the stubbed-out JWS implementation file, adding your business logic. Typically, you move this JWS file from the wsdlc-output directory to a more permanent directory that contains your application source code; in the example, the fully coded TemperaturePortTypeImpl.java JWS file has been moved to the directory src/examples/webservices/wsdlc/. You then run the jwsc Ant task, specifying this JWS file as usual. The only additional attribute you must specify is compiledWsdl to point to the JAR file generated by the wsdlc Ant task, as shown in the preceding example. This indicates that you do not want the jwsc Ant task to generate a new WSDL file, because you want to use the original one that has been compiled into the JAR file.

**Child Element**

The wsdlc Ant task has one WebLogic-specific child element: `<xsdConfig>`.

Use the `<xsdConfig>` child element to specify one or more XMLBeans configuration files, which by convention end in `.xsdconfig`. Use this element if your Web Service uses Apache XMLBeans data types as parameters or return values.

The `<xsdConfig>` element is similar to the standard Ant `<Fileset>` element and has all the same attributes. See the Apache Ant documentation on the `<Fileset>` element for the full list of attributes you can specify.

See “Standard Ant javac Attributes That Apply To wsdlc” on page A-62 for the list of elements associated with the standard Ant javac task that you can also set for the wsdlc Ant task.

**Attributes**

The table in the following section describes the attributes of the wsdlc Ant task. See “Standard Ant javac Attributes That Apply To wsdlc” on page A-62 for the list of attributes associated with the standard Ant javac task that you can also set for the wsdlc Ant task.
## WebLogic-Specific wsdlc Attributes

### Table A-12  Attributes of the wsdlc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoDetectWrapped</td>
<td>Specifies whether the <code>wsdlc</code> Ant task should try to determine whether the parameters and return type of document-literal Web Services are of type wrapped or bare.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>When the <code>wsdlc</code> Ant task parses a WSDL file to create the partial JWS file that implements the Web Service, it attempts to determine whether a document-literal Web Service uses wrapped or bare parameters and return types based on the names of the XML Schema elements, the name of the operations and parameters, and so on. Depending on how the names of these components match up, the <code>wsdlc</code> Ant task makes a best guess as to whether the parameters are wrapped or bare. In some cases, however, you might want the Ant task to always assume that the parameters are of type bare; in this case, set the <code>autoDetectWrapped</code> attribute to False. Valid values for this attribute are <code>True</code> or <code>False</code>. The default value is <code>True</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>desImplDir</td>
<td>Directory into which the stubbed-out JWS implementation file is generated. The generated JWS file implements the generated JWS interface file (contained within the JAR file). You update this JWS implementation file, adding Java code to the methods so that they behave as you want, then later specify this updated JWS file to the <code>jwsc</code> Ant task to generate a deployable Web Service.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>
### Table A-12 Attributes of the wsdlc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>destJavadocDir</td>
<td>Directory into which Javadoc that describes the JWS interface is generated. Because you should never unjar or update the generated JAR file that contains the JWS interface file that implements the specified Web Service, you can get detailed information about the interface file from this generated Javadoc. You can then use this documentation, together with the generated stubbed-out JWS implementation file, to add business logic to the partially generated Web Service.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>destJwsDir</td>
<td>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 <code>WSDLFile_wsdl.jar</code>, where <code>WSDLFile</code> refers to the root name of the WSDL file. For example, if the name of the WSDL file you specify to the <code>file</code> attribute is <code>MyService.wsdl</code>, then the generated JAR file is <code>MyService_wsdl.jar</code>.</td>
<td>String</td>
<td>Yes.</td>
</tr>
<tr>
<td>explode</td>
<td>Specifies whether the generated JAR file that contains the generated JWS interface file and data binding artifacts is in exploded directory format or not. Valid values for this attribute are <code>true</code> or <code>false</code>. Default value is <code>false</code>, which means that wsdlc generates an actual JAR archive file, and not an exploded directory.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
<tr>
<td>jaxRPCWrappedArrayStyle</td>
<td>When the <code>wsdlc</code> Ant task is generating the Java equivalent to XML Schema data types in the WSDL file, and the task encounters an XML complex type with a single enclosing sequence with a single element with the <code>maxOccurs</code> attribute equal to <code>unbounded</code>, the task generates, by default, a Java structure whose name is the lowest named enclosing complex type or element. To change this behavior so that the task generates a literal array instead, set the <code>jaxRPCWrappedArrayStyle</code> to <code>false</code>. Valid values for this attribute are <code>True</code> or <code>False</code>. The default value is <code>True</code>.</td>
<td>Boolean</td>
<td>No.</td>
</tr>
</tbody>
</table>
### Table A-12  Attributes of the wsdlc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>packageName</td>
<td>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.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>
| srcBindingName   | Name of the WSDL binding from which the JWS interface file should be generated. The wsdlc Ant task runs against the first `<service>` element it finds in the WSDL file. Therefore, you only need to specify the srcBindingName attribute if there is more than one `<binding>` element associated with this first `<service>` element. If the namespace of the binding is the same as the namespace of the service, then you just need to specify the name of the binding for the value of this attribute. For example:  
  `srcBindingName="MyBinding"`  
  However, if the namespace of the binding is different from the namespace of the service, then you must also specify the namespace URI, using the following format:  
  `srcBindingName="{URI}BindingName"`  
  For example, if the namespace URI of the MyBinding binding is `www.examples.org`, then you specify the attribute value as follows:  
  `srcBindingName="{www.examples.org}MyBinding"`  
  **Note:** This attribute is deprecated as of Version 9.2 of WebLogic Server. Use `srcPortName` or `srcServiceName` instead. | String | Only if the WSDL file contains more than one `<binding>` element |

Programming Web Services for WebLogic Server  A-59
**srcPortName**
Name of the WSDL port from which the JWS interface file should be generated.
Set the value of this attribute to the value of the `name` attribute of the `<port>` element that corresponds to the Web Service port for which you want to generate a JWS interface file. The `<port>` element is a child element of the `<service>` element in the WSDL file.
If you specify this attribute, you cannot also specify `srcServiceName`. If you do not specify this attribute, `wsdlc` generates a JWS interface file from the service specified by `srcServiceName`. 

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcPortName</td>
<td>Name of the WSDL port from which the JWS interface file should be generated.</td>
<td>String</td>
<td>No</td>
</tr>
</tbody>
</table>
Programming Web Services for WebLogic Server

Table A-12  Attributes of the wsdlc Ant Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcServiceName</td>
<td>Name of the Web Service from which the JWS interface file should be generated.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Set the value of this attribute to the value of the name attribute of the &lt;service&gt; element that corresponds to the Web Service for which you want to generate a JWS interface file.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The wsdlc Ant task generates a single JWS endpoint interface and data binding JAR file for a given Web Service. This means that if the &lt;service&gt; element contains more than one &lt;port&gt; element, the following must be true:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The bindings for each port must be the same or equivalent to each other.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The transport for each port must be different. The wsdlc Ant task determines the transport for a port from the address listed in its &lt;address&gt; child element. Because WebLogic Web Services support only three transports (JMS, HTTP, and HTTPS), this means that there can be at most three &lt;port&gt; child elements for the &lt;service&gt; element specified by this attribute. The generated JWS implementation file will then include the corresponding @WLXXXTransport annotations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you specify this attribute, you cannot also specify srcPortName.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you do not specify either this or the srcPortName attribute, the WSDL file must include only one &lt;service&gt; element. The wsdlc Ant task generates the JWS interface file and data binding JAR file from this single Web Service.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srcWsdl</td>
<td>Name of the WSDL from which to generate the JAR file that contains the JWS interface and data binding artifacts.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>The name must include its pathname, either absolute or relative to the directory which contains the Ant build.xml file.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Standard Ant javac Attributes That Apply To wsdlc

In addition to the WebLogic-defined wsdlc attributes, you can also define the following standard javac attributes; see the Ant documentation for additional information about each attribute:

- bootclasspath
- bootClasspathRef
- classpath
- classpathRef
- compiler
- debug
- debugLevel
- depend
- deprecation
- destdir
- encoding
- extdirs
- failonerror
- fork
- includeantruntime
- includejavaruntime
- listfiles
- memoryInitialSize
- memoryMaximumSize
- nowarn
- optimize
- proceed
- source
- sourcepath
- sourcepathRef
- tempdir
You can also use the following standard Ant child elements with the \textit{wsdlc} Ant task:

- \texttt{<FileSet>}
- \texttt{<SourcePath>}
- \texttt{<Classpath>}
- \texttt{<Extdirs>}

\textbf{verbose}
JWS Annotation Reference

The following sections provide reference documentation about standard (JSR-181) and WebLogic-specific JWS annotations:

- “Overview of JWS Annotation Tags” on page B-1
- “Standard JSR-181 JWS Annotations Reference” on page B-4
- “WebLogic-Specific JWS Annotations Reference” on page B-16

Overview of JWS Annotation Tags

The WebLogic Web Services programming model uses the new JDK 5.0 metadata annotations feature (specified by JSR-175). In this programming model, you create an annotated Java file and then use Ant tasks to compile the file into the Java source code and generate all the associated artifacts.

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

You can target a JWS annotation at either the class-, method- or parameter-level in a JWS file. Some annotations can be targeted at more than one level, such as @SecurityRoles that can be
targeted at both the class- and method-level. The documentation in this section lists the level to which you can target each annotation.

The following example shows a simple JWS file that uses both standard JSR-181 and WebLogic-specific JWS annotations, shown in bold:

```java
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;

// 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)
 */
```
Overview of JWS Annotation Tags

* 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;
  }
}
Standard JSR-181 JWS Annotations Reference

The Web Services Metadata for the Java Platform (JSR-181) specification defines the standard annotations you can use in your JWS file to specify the shape and behavior of your Web Service. This section briefly describes each annotation, along with its attributes. See Chapter 5, “Programming the JWS File,” for examples. For more detailed information about the annotations, such as the Java annotation type definition and additional examples, see the specification.

This section documents the following standard JWS annotations:

- `javax.jws.WebService`
- `javax.jws.WebMethod`
- `javax.jws.Oneway`
- `javax.jws.WebParam`
- `javax.jws.WebResult`
- `javax.jws.HandlerChain`
- `javax.jws.soap.SOAPBinding`
- `javax.jws.soap.SOAPMessageHandler`
- `javax.jws.soap.InitParam`
- `javax.jws.soap.SOAPMessageHandlers`

### `javax.jws.WebService`

**Description**

**Target:** Class

Specifies that the JWS file implements a Web Service.
**Attributes**

Table B-1  Attributes of the javax.jws.WebService JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the Web Service. Maps to the <a href="">wsdl:portType</a> element in the WSDL file. Default value is the unqualified name of the Java class in the JWS file.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>targetNamespace</td>
<td>The XML namespace used for the WSDL and XML elements generated from this Web Service. The default value is specified by the JAX-RPC specification.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>serviceName</td>
<td>Service name of the Web Service. Maps to the <a href="">wsdl:service</a> element in the WSDL file. Default value is the unqualified name of the Java class in the JWS file, appended with the string Service.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>wsdlLocation</td>
<td>Relative or absolute URL of a pre-defined WSDL file. If you specify this attribute, the jwsc Ant task does not generate a WSDL file, and returns an error if the JWS file is inconsistent with the port types and bindings in the WSDL file. Note: The wsdlc Ant task uses this attribute when it generates the endpoint interface JWS file from a WSDL. Typically, users never use the attribute in their own JWS files.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>endpointInterface</td>
<td>Fully qualified name of an existing service endpoint interface file. If you specify this attribute, it is assumed that you have already created the endpoint interface file and it is in your CLASSPATH.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>

**Example**

```java
@WebService(name="JMSTransportPortType",
            serviceName="JMSTransportService",
            targetNamespace="http://example.org")
```
javax.jws.WebMethod

Description

Target: Method

Specifies that the method is exposed as a public operation of the Web Service. You must explicitly use this annotation to expose a method; if you do not specify this annotation, the method by default is not exposed.

Attributes

Table B-2  Attributes of the javax.jws.WebMethod JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>operationName</td>
<td>Name of the operation. Maps to the <a href="">wsdl:operation</a> element in the WSDL file. Default value is the name of the method.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>action</td>
<td>The action for this operation. For SOAP bindings, the value of this attribute determines the value of the SOAPAction header in the SOAP messages.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>

Example

```java
@WebMethod(operationName="echoComplexType")
public BasicStruct echoStruct(BasicStruct struct) {
    ...
}
```

javax.jws.Oneway

Description

Target: Method

Specifies that the method has only input parameters, but does not return a value. This annotation must be used only in conjunction with the @WebMethod annotation.
It is an error to use this annotation on a method that returns anything other than \texttt{void}, takes a \texttt{Holder} class as an input parameter, or throws checked exceptions.

This annotation does not have any attributes.

\textbf{Example}

\begin{verbatim}
@WebMethod()
@Oneway()
public void helloWorld(String input) {
    ...
}
\end{verbatim}

\textbf{javax.jws.WebParam}

\textbf{Description}

\textbf{Target:} Parameter

Customizes the mapping between operation input parameters of the Web Service and elements of the generated WSDL file. Also used to specify the behavior of the parameter.

\textbf{Attributes}

\begin{table}[h]
\centering
\caption{Attributes of the \texttt{javax.jws.WebParam} JWS Annotation}
\begin{tabular}{llll}
\hline
Name & Description & Data Type & Required? \\
\hline
name & Name of the parameter in the WSDL file. For RPC-style Web Services, the name maps to the \texttt{<wsdl:part>} element that represents the parameter. For document-style Web Services, the name is the local name of the XML element that represents the parameter. The default value is the name of the method’s parameter. & String & No. \\
\hline
\texttt{targetNamespace} & The XML namespace of the parameter. This value is used only for document-style Web Services, in which the parameter maps to an XML element. The default value is the \texttt{targetNamespace} of the Web Service. & String & No. \\
\hline
\end{tabular}
\end{table}
Table B-3  Attributes of the javax.jws.WebParam JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>The direction in which the parameter is flowing.</td>
<td>enum</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>Valid values are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WebParam.Mode.IN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WebParam.Mode.OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WebParam.Mode.INOUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default value is WebParam.Mode.IN.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you specify WebParam.Mode.OUT or WebParam.Mode.INOUT, then the data type of the parameter must be Holder, or extend Holder. For details, see the JAX-RPC specification.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WebParam.Mode.OUT and WebParam.Mode.INOUT modes are only supported for RPC-style Web Services or for parameters that map to headers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>header</td>
<td>Specifies whether the value of the parameter is found in the SOAP header. By default parameters are in the SOAP body.</td>
<td>boolean</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>Valid values are true and false. Default value is false.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

```java
@WebMethod()
public int echoInt(
    @WebParam(name="IntegerInput",
        targetNamespace="http://example.org/complex")
    int input)
{
    ...
}
```

descriptions

javajws.WebResult

Description

Target: Method
Customizes the mapping between the Web Service operation return value and the corresponding element of the generated WSDL file.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the parameter in the WSDL file.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>For RPC-style Web Services, the name maps to the <code>&lt;wsdl:part&gt;</code> element that represents the return value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>For document-style Web Services, the name is the local name of the XML element that represents the return value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value is the hard-coded name <code>result</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>targetNamespace</td>
<td>The XML namespace of the return value. This value is used only used for document-style Web Services, in which the return value maps to an XML element.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>The default value is the targetNamespace of the Web Service.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

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

`javax.jws.HandlerChain`

Description

Target: Class
Associates a Web Service with an external file that contains the configuration of a handler chain. The configuration includes the list of handlers in the chain, the order in which they execute, the initialization parameters, and so on.

Use the `@HandlerChain` annotation, rather than the `@SOAPMessageHandlers` annotation, in your JWS file if:

- You want multiple Web Services to share the same configuration.
- Your handler chain includes handlers for multiple transports.
- You want to be able to change the handler chain configuration for a Web Service without recompiling the JWS file that implements it.

It is an error to combine this annotation with the `@SOAPMessageHandlers` annotation.

For the XML Schema of the external configuration file, additional information about creating it, and additional examples, see the Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

### Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>URL, either relative or absolute, of the handler chain configuration file. Relative URLs are relative to the location of JWS file.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>name</td>
<td>Name of the handler chain (in the configuration file pointed to by the file attribute) that you want to associate with the Web Service.</td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Example

```java
package examples.webservices.handler;
...

@WebService (...)  
@HandlerChain(file="HandlerConfig.xml", name="SimpleChain")
```
public class HandlerChainImpl {
...
}

javax.jws.soap.SOAPBinding

Description
Target: Class
Specifies the mapping of the Web Service onto the SOAP message protocol.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>Specifies the message style of the request and response SOAP messages.</td>
<td>enum</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>Valid values are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SOAPBinding.Style.RPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SOAPBinding.Style.DOCUMENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default value is SOAPBinding.Style.DOCUMENT.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```java
package examples.webservices.bindings;
...

@WebService (...)

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

public class BindingsImpl {
    ...
```
javax.jws.soap.SOAPMessageHandler

Description

Target: None; this annotation can be used only inside of a @SOAPMessageHandler array.

Specifies a particular SOAP message handler in a @SOAPMessageHandler array. The annotation includes attributes to specify the class name of the handler, the initialization parameters, list of SOAP headers processed by the handler, and so on.

Attributes

Table B-7 Attributes of the javax.jws.soap.SOAPMessageHandler JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the SOAP message handler.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>The default value is the name of the class that implements the Handler interface (or extends the GenericHandler abstract class.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>className</td>
<td>Name of the handler class.</td>
<td>String</td>
<td>Yes.</td>
</tr>
<tr>
<td>initParams</td>
<td>Array of name/value pairs that is passed to the handler class during initialization.</td>
<td>Array of @InitParam</td>
<td>No.</td>
</tr>
<tr>
<td>roles</td>
<td>List of SOAP roles implemented by the handler.</td>
<td>Array of String</td>
<td>No.</td>
</tr>
<tr>
<td>headers</td>
<td>List of SOAP headers processed by the handler.</td>
<td>Array of String</td>
<td>No.</td>
</tr>
</tbody>
</table>

Example

```java
package examples.webservices.handlers;
...

@WebService (...)```
@SOAPMessageHandler ( {
    @SOAPMessageHandler ( 
        className="examples.webservices.soap_handlers.simple.ServerHandler1"),
    @SOAPMessageHandler ( 
        className="examples.webservices.soap_handlers.simple.ServerHandler2"
    )
})

public class HandlersImpl {
    ...
}

javax.jws.soap.InitParam

Description
Target: None; this annotation can be used only as a value to the initParams attribute of the @SOAPMessageHandler annotation.

Use this annotation in the initParams attribute of the @SOAPMessageHandler annotation to specify the array of parameters (name/value pairs) that are passed to a handler class during initialization.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the initialization parameter.</td>
<td>String</td>
<td>Yes.</td>
</tr>
<tr>
<td>value</td>
<td>Value of the initialization parameter.</td>
<td>String</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

javax.jws.soap.SOAPMessageHandlers

Description
Target: Class
Specifies an array of SOAP message handlers that execute before and after the operations of a Web Service. Use the @SOAPMessageHandler annotation to specify a particular handler. Because you specify the list of handlers within the JWS file itself, the configuration of the handler chain is embedded within the Web Service.

Use the @SOAPMessageHandlers annotation, rather than @HandlerChain, if:

- You prefer to embed the configuration of the handler chain inside the Web Service itself, rather than specify the configuration in an external file.
- Your handler chain includes only SOAP handlers and none for any other transport.
- You prefer to recompile the JWS file each time you change the handler chain configuration.

The @SOAPMessageHandlers annotation is an array of @SOAPMessageHandler types. The handlers run in the order in which they appear in the annotation, starting with the first handler in the array.

This annotation does not have any attributes.

Example

```java
package examples.webservices.handlers;
...

@WebService (...)  
@SOAPMessageHandlers ( {  
    @SOAPMessageHandler (  
        className="examples.webservices.soap_handlers.simple.ServerHandler1"),  
        @SOAPMessageHandler (  
            className="examples.webservices.soap_handlers.simple.ServerHandler2")
    )
} )

public class HandlersImpl {
...  
}
```
WebLogic-Specific JWS Annotations Reference

WebLogic Web Services define a set of JWS annotations that you can use to specify behavior and features in addition to the standard JSR-181 JWS annotations. In particular, the WebLogic-specific annotations are:

- `weblogic.jws.AsyncFailure` on page B-17
- `weblogic.jws.AsyncResponse` on page B-20
- `weblogic.jws.Binding` on page B-23
- `weblogic.jws.BufferQueue` on page B-25
- `weblogic.jws.Callback` on page B-26
- `weblogic.jws.CallbackMethod` on page B-28
- `weblogic.jws.CallbackService` on page B-29
- `weblogic.jws.Context` on page B-31
- `weblogic.jws.Conversation` on page B-33
- `weblogic.jws.Conversational` on page B-35
- `weblogic.jws.MessageBuffer` on page B-38
- `weblogic.jws.Policies` on page B-40
- `weblogic.jws.Policy` on page B-41
- `weblogic.jws.ReliabilityBuffer` on page B-43
- `weblogic.jws.ReliabilityErrorHandler` on page B-45
- `weblogic.jws.ServiceClient` on page B-47
- `weblogic.jws.StreamAttachments` on page B-51
- `weblogic.jws.Transactional` on page B-52
- `weblogic.jws.Types` on page B-54
- `weblogic.jws.WildcardBinding` on page B-56
- `weblogic.jws.WildcardBindings` on page B-57
weblogic.jws.AsyncFailure

Description

Target: Method

Specifies the method that handles a potential failure when the main JWS file invokes an operation of another Web Service asynchronously.

When you invoke, from within a JWS file, a Web Service operation asynchronously, the response (or exception, in the case of a failure) does not return immediately after the operation invocation, but rather, at some later point in time. Because the operation invocation did not wait for a response, a separate method in the JWS file must handle the response when it does finally return; similarly, another method must handle a potential failure. Use the @AsyncFailure annotation to specify the method in the JWS file that will handle the potential failure of an asynchronous operation invocation.
The `@AsyncFailure` annotation takes two parameters: the name of the JAX-RPC stub for the Web Service you are invoking and the name of the operation that you are invoking asynchronously. The JAX-RPC stub is the one that has been annotation with the `@ServiceClient` annotation.

The method that handles the asynchronous failure must follow these guidelines:

- **Return** void.
- **Be named** `onMethodNameAsyncFailure`, where `MethodName` is the name of the method you are invoking asynchronously (with initial letter always capitalized.)

  In the main JWS file, the call to the asynchronous method will look something like:

  ```java
  port.getQuoteAsync (apc, symbol);
  ```

  where `getQuote` is the non-asynchronous name of the method, `apc` is the asynchronous pre-call context, and `symbol` is the usual parameter to the `getQuote` operation.

- **Have two parameters**: the asynchronous post-call context (contained in the `weblogic.wsee.async.AsyncPostCallContext` object) and the `Throwable` exception, potentially thrown by the asynchronous operation call.

  Within the method itself you can get more information about the method failure from the context, and query the specific type of exception and act accordingly.

  Typically, you always use the `@AsyncFailure` annotation to explicitly specify the method that handles asynchronous operation failures. The only time you would not use this annotation is if you want a single method to handle failures for two or more stubs that invoke different Web Services. In this case, although the stubs connect to different Web Services, each Web Service must have a similarly named method, because the Web Services runtime relies on the name of the method (`onMethodNameAsyncFailure`) to determine how to handle the asynchronous failure, rather than the annotation. However, if you always want a one-to-one correspondence between a stub and the method that handles an asynchronous failure from one of the operations, then BEA recommends that you explicitly use `@AsyncFailure`.

  See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 for detailed information and examples of using this annotation.
Attributes

Table B-9 Attributes of the weblogic.jws.AsyncFailure JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>The name of the JAX-RPC stub of the Web Service for which you want to invoke an operation asynchronously. The stub is the one that has been annotated with the @ServiceClient field-level annotation.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>operation</td>
<td>The name of the operation that you want to invoke asynchronously. This is the actual name of the operation, as it appears in the WSDL file. When you invoke this operation in the main code of the JWS file, you add Async to its name. For example, if set operation=&quot;getQuote&quot;, then in the JWS file you invoke it asynchronously as follows: port.getQuoteAsync (apc, symbol);</td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example

The following sample snippet shows how to use the @AsyncFailure annotation in a JWS file that invokes the operation of another Web Service asynchronously; only the relevant Java code is included:

```java
package examples.webservices.async_req_res;
...

public class StockQuoteClientImpl {

    @ServiceClient(wsdlLocation="http://localhost:7001/async/StockQuote?WSDL", 
                   serviceName="StockQuoteService", portName="StockQuote")
    private StockQuotePortType port;

    @WebMethod
    public void getQuote (String symbol) {
        AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
        apc.setProperty("symbol", symbol);

        try {
            port.getQuoteAsync(apc, symbol);
        }
    }

```
System.out.println("in getQuote method of StockQuoteClient WS");
}

catch (RemoteException e) {
    e.printStackTrace();
}

...
Return void.

Be named onMethodNameAsyncResponse, where MethodName is the name of the method you are invoking asynchronously (with initial letter always capitalized.)

In the main JWS file, the call to the asynchronous method will look something like:

```java
port.getQuoteAsync (apc, symbol);
```

where `getQuote` is the non-asynchronous name of the method, `apc` is the asynchronous pre-call context, and `symbol` is the usual parameter to the `getQuote` operation.

Have two parameters: the asynchronous post-call context (contained in the `weblogic.wsee.async.AsyncPostCallContext` object) and the usual return value of the operation.

Within the asynchronous-response method itself you add the code to handle the response. You can also get more information about the method invocation from the context.

Typically, you always use the @AsyncResponse annotation to explicitly specify the method that handles asynchronous operation responses. The only time you would not use this annotation is if you want a single method to handle the response for two or more stubs that invoke different Web Services. In this case, although the stubs connect to different Web Services, each Web Service must have a similarly named method, because the Web Services runtime relies on the name of the method (onMethodNameAsyncResponse) to determine how to handle the asynchronous response, rather than the annotation. However, if you always want a one-to-one correspondence between a stub and the method that handles an asynchronous response from one of the operations, then BEA recommends that you explicitly use @AsyncResponse.

See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 for detailed information and examples of using this annotation.
### Attributes

**Table B-10**  Attributes of the `weblogic.jws.AsyncResponse` JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>The name of the JAX-RPC stub of the Web Service for which you want to invoke an operation asynchronously. The stub is the one that has been annotated with the @ServiceClient field-level annotation.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>operation</td>
<td>The name of the operation that you want to invoke asynchronously. This is the <em>actual</em> name of the operation, as it appears in the WSDL file. When you invoke this operation in the main code of the JWS file, you add Async to its name. For example, if set <code>operation=&quot;getQuote&quot;</code>, then in the JWS file you invoke it asynchronously as follows: <code>port.getQuoteAsync (apc, symbol);</code></td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Example

The following sample snippet shows how to use the @AsyncResponse annotation in a JWS file that invokes the operation of another Web Service asynchronously; only the relevant Java code is included:

```java
package examples.webservices.async_req_res;
...

public class StockQuoteClientImpl {
  @ServiceClient(wsdlLocation="http://localhost:7001/async/StockQuote?WSDL",
                 serviceName="StockQuoteService", portName="StockQuote")
  private StockQuotePortType port;

  @WebMethod
  public void getQuote (String symbol) {
    AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
    apc.setProperty("symbol", symbol);
    try {
      port.getQuoteAsync(apc, symbol);
    }
  }
```

---

B-22  Programming Web Services for WebLogic Server
System.out.println("in getQuote method of StockQuoteClient WS");
}
catch (RemoteException e) {
    e.printStackTrace();
}

...
Attributes

Table B-11  Attributes of the weblogic.jws.Binding JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
</table>
| value | Specifies the version of SOAP used in the request and response SOAP messages when the Web Service is invoked. Valid values for this attribute are:
  - Type.SOAP11
  - Type.SOAP12
  The default value is Type.SOAP11. | enum | No |

Example

The following example shows how to specify SOAP 1.2; only the relevant code is shown:

```java
package examples.webservices.soap12;

import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.Binding;

@WebService(name="SOAP12PortType",
            serviceName="SOAP12Service",
            targetNamespace="http://example.org")
@Binding(Binding.Type.SOAP12)
public class SOAP12Impl {
    @WebMethod()
    public String sayHello(String message) {
        ...
    }
}
```
weblogic.jws.BufferQueue

Description

**Target:** Class

Specifies the JNDI name of the JMS queue to which WebLogic Server:

- stores a buffered Web Service operation invocation.
- stores a reliable Web Service operation invocation.

When used with buffered Web Services, you use this annotation in conjunction with `@MessageBuffer`, which specifies the methods of a JWS that are buffered. When used with reliable Web Services, you use this annotation in conjunction with `@Policy`, which specifies the reliable messaging WS-Policy file associated with the Web Service.

If you have enabled buffering or reliable messaging for a Web Service, but do not specify the `@BufferQueue` annotation, WebLogic Server uses the default Web Services JMS queue (`weblogic.wsee.DefaultQueue`) to store buffered or reliable operation invocations. This JMS queue is also the default queue for the JMS transport features. It is assumed that you have already created this JMS queue if you intend on using it for any of these features.

See “Creating Buffered Web Services” on page 6-50 and “Using Web Service Reliable Messaging” on page 6-1 for detailed information and examples of creating buffered or reliable Web Services.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The JNDI name of the JMS queue to which the buffered or reliable operation invocation is queued.</td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example

The following example shows a code snippet from a JWS file in which the public operation is buffered and the JMS queue to which WebLogic Server queues the operation invocation is called `my.buffer.queue`; only the relevant Java code is shown:
package examples.webservices.buffered;

...

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

@BufferQueue(name="my.buffer.queue")

public class BufferedImpl {

...

@WebMethod()
@MessageBuffer(retryCount=10, retryDelay="10 seconds")
@Oneway()
public void sayHelloNoReturn(String message) {
    System.out.println("sayHelloNoReturn: " + message);
}
}

**weblogic.jws.Callback**

**Description**

**Target:** Field

Specifies that the annotated variable is a callback, which means that you can use the variable to send callback events back to the client Web Service that invoked an operation of the target Web Service.

You specify the @Callback annotation in the target Web Service so that it can call back to the client Web Service. The data type of the annotated variable is the callback interface.

The callback feature works between two WebLogic Web Services. When you program the feature, however, you create the following three Java files:

- **Callback interface:** Java interface file that defines the callback methods. You do not explicitly implement this file yourself; rather, the jwsc Ant task automatically generates an implementation of the interface. The implementation simply passes a message from the target Web Service back to the client Web Service. The generated Web Service is deployed to the same WebLogic Server that hosts the client Web Service.
JWS file that implements the **target Web Service**: The target Web Service includes one or more standard operations that invoke a method defined in the callback interface; this method in turn sends a message back to the client Web Service that originally invoked the operation of the target Web Service.

JWS file that implements the **client Web Service**: The client Web Service invokes an operation of the target Web Service. This Web Service includes one or more methods that specify what the client should do when it receives a callback message back from the target Web Service via a callback method.

See “Using Callbacks to Notify Clients of Events” on page 6-27 for additional overview and procedural information about programming callbacks.

The @Callback annotation does not have any attributes.

**Example**

The following example shows a very simple target Web Service in which a variable called `callback` is annotated with the `@Callback` annotation. The data type of the variable is `CallbackInterface`; this means a callback Web Service must exist with this name. After the variable is injected with the callback information, you can invoke the callback methods defined in `CallbackInterface`; in the example, the callback method is `callbackOperation()`.

The text in bold shows the relevant code:

```java
package examples.webservices.callback;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Callback;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(name="CallbackPortType",
    serviceName="TargetService",
    targetNamespace="http://examples.org/"
)
@WLHttpTransport(contextPath="callback",
    serviceUri="TargetService",
    portName="TargetServicePort")

public class TargetServiceImpl {
    @Callback
    CallbackInterface callback;
```
@WebMethod
public void targetOperation (String message) {
    callback.callbackOperation (message);
}
}

weblogic.jws.CallbackMethod

Description

Target: Method

Specifies the method in the client Web Service that handles the messages it receives from the callback Web Service. Use the attributes to link the callback message handler methods in the client Web Service with the callback method in the callback interface.

The callback feature works between two WebLogic Web Services. When you program the feature, however, you create the following three Java files:

- **Callback interface**: Java interface file that defines the callback methods. You do not explicitly implement this file yourself; rather, the jwsc Ant task automatically generates an implementation of the interface. The implementation simply passes a message from the target Web Service back to the client Web Service. The generated Web Service is deployed to the same WebLogic Server that hosts the client Web Service.

- **JWS file that implements the target Web Service**: The target Web Service includes one or more standard operations that invoke a method defined in the callback interface; this method in turn sends a message back to the client Web Service that originally invoked the operation of the target Web Service.

- **JWS file that implements the client Web Service**: The client Web Service invokes an operation of the target Web Service. This Web Service includes one or more methods that specify what the client should do when it receives a callback message back from the target Web Service via a callback method.

See “Using Callbacks to Notify Clients of Events” on page 6-27 for additional overview and procedural information about programming callbacks.
Attributes

Table B-13  Attributes of the weblogic.jws.CallbackMethod JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>Specifies the name of the callback method in the callback interface for which this method will handle callback messages.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>target</td>
<td>Specifies the name of the JAX-RPC stub for which you want to receive callbacks. The stub is the one that has been annotated with the @ServiceClient field-level annotation.</td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example

The following example shows a method of a client Web Service annotated with the @CallbackMethod annotation. The attributes show that a variable called port must have previously been injected with JAX-RPC stub information and that the annotated method will handle messages received from a callback operation called callbackOperation().

```java
@CallbackMethod(target="port", operation="callbackOperation")
@CallbackRolesAllowed(@SecurityRole(role="engineer", mapToPrincipals="shackell"))
public void callbackHandler(String msg) {
    System.out.println (msg);
}
```

weblogic.jws.CallbackService

Description

Target: Class

Specifies that the JWS file is actually a Java interface that describes a callback Web Service. This annotation is analogous to the @javax.jws.WebService, but specific to callbacks and with a reduced set of attributes.
The callback feature works between two WebLogic Web Services. When you program the feature, however, you create the following three Java files:

- **Callback interface**: Java interface file that defines the callback methods. You do not explicitly implement this file yourself; rather, the jwsc Ant task automatically generates an implementation of the interface. The implementation simply passes a message from the target Web Service back to the client Web Service. The generated Web Service is deployed to the same WebLogic Server that hosts the client Web Service.

- **JWS file that implements the target Web Service**: The target Web Service includes one or more standard operations that invoke a method defined in the callback interface; this method in turn sends a message back to the client Web Service that originally invoked the operation of the target Web Service.

- **JWS file that implements the client Web Service**: The client Web Service invokes an operation of the target Web Service. This Web Service includes one or more methods that specify what the client should do when it receives a callback message back from the target Web Service via a callback method.

Use the @CallbackInterface annotation to specify that the Java file is a callback interface file.

When you program the callback interface, you specify one or more callback methods; as with standard non-callback Web Services, you annotate these methods with the @javax.jws.WebMethod annotation to specify that they are Web Service operations. However, contrary to non-callback methods, you never write the actual implementation code for these callback methods; rather, when you compile the client Web Service with the jwsc Ant task, the task automatically creates an implementation of the interface and packages it into a Web Service. This generated implementation specifies that the callback methods all do the same thing: send a message from the target Web Service that invokes the callback method back to the client Web Service.

See “Using Callbacks to Notify Clients of Events” on page 6-27 for additional overview and procedural information about programming callbacks.
Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the callback Web Service. Maps to the <a href="">wsdl:portType</a> element in the WSDL file. Default value is the unqualified name of the Java class in the JWS file.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>serviceName</td>
<td>Service name of the callback Web Service. Maps to the <a href="">wsdl:service</a> element in the WSDL file. Default value is the unqualified name of the Java class in the JWS file, appended with the string Service.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>

Example

The following example shows a very simple callback interface. The resulting callback Web Service has one callback method, callbackOperation().

```java
package examples.webservices.callback;

import weblogic.jws.CallbackService;
import javax.jws.Oneway;
import javax.jws.WebMethod;

@CallbackService
public interface CallbackInterface {
    @WebMethod
    @Oneway
    public void callbackOperation (String msg);
}
```

weblogic.jws.Context

Description

Target: Field
Specifies that the annotated field provide access to the runtime context of the Web Service. 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. 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. The data type of the annotation field must be weblogic.wsee.jws.JwsContext, which is a WebLogic Web Service API that includes methods to query the context.

For additional information about using this annotation, see “Accessing Runtime Information about a Web Service Using the JwsContext” on page 5-11.

This annotation does not have any attributes.

**Example**

The following snippet of a JWS file shows how to use the @Context annotation; only parts of the file are shown, with relevant code in bold:

```java
import weblogic.jws.Context;
import weblogic.wsee.jws.JwsContext;

public class JwsContextImpl {
    @Context
    private JwsContext ctx;
    @WebMethod()
    public String getProtocol() {
        ...
    }
```
**weblogic.jws.Conversation**

**Description**

**Target:** Method

Specifies that a method annotated with the `@Conversation` annotation can be invoked as part of a conversation between two WebLogic Web Services or a stand-alone Java client and a conversational Web Service.

The conversational Web Service typically specifies three methods, each annotated with the `@Conversation` annotation that correspond to the start, continue, and finish phases of a conversation. Use the `@Conversational` annotation to specify, at the class level, that a Web Service is conversational and to configure properties of the conversation, such as the maximum idle time.

If the conversation is between two Web Services, the client service uses the `@ServiceClient` annotation to specify the wsdl, service name, and port of the invoked conversational service. In both the service and stand-alone client cases, the client then invokes the start, continue, and finish methods in the appropriate order to conduct a conversation. The only additional requirement to make a Web Service conversational is that it implement `java.io.Serializable`.

See “Creating Conversational Web Services” on page 6-37 for detailed information and examples of using this annotation.
Attributes

Table B-15 Attributes of the weblogic.jws.Conversation JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Specifies the phase of a conversation that the annotated method implements.</td>
<td>enum</td>
<td>No.</td>
</tr>
</tbody>
</table>

Possible values are:

- **Phase.START**
  Specifies that the method starts a new conversation. A call to this method creates a new conversation ID and context, and resets its idle and age timer.

- **Phase.CONTINUE**
  Specifies that the method is part of a conversation in progress. A call to this method resets the idle timer. This method must always be called after the start method and before the finish method.

- **Phase.FINISH**
  Specifies that the method explicitly finishes a conversation in progress.

Default value is **Phase.CONTINUE**

Example

The following sample snippet shows a JWS file that contains three methods, start, middle, and finish that are annotated with the @Conversation annotation to specify the start, continue, and finish phases, respectively, of a conversation.

```java
public class ConversationalServiceImpl implements Serializable {

    @WebMethod
    @Conversation (Conversation.Phase.START)
    public String start() {
        // Java code for starting a conversation goes here
    }

    @WebMethod
    @Conversation (Conversation.Phase.CONTINUE)
```
public String middle(String message) {
    // Java code for continuing a conversation goes here
}

@WebMethod
@Conversation (Conversation.Phase.FINISH)
public String finish(String message ) {
    // Java code for finishing a conversation goes here
}

weblogic.jws.Conversational

Description
Target: Class
Specifies that a JWS file implements a conversational Web Service.

You are not required to use this annotation to specify that a Web Service is conversational; by simply annotating a single method with the @Conversation annotation, all the methods of the JWS file are automatically tagged as conversational. Use the class-level @Conversational annotation only if you want to change some of the conversational behavior or if you want to clearly show at the class level that the JWS if conversational.

If you do use the @Conversational annotation in your JWS file, you can specify it without any attributes if their default values suit your needs. However, if you want to change values such as the maximum amount of time that a conversation can remain idle, the maximum age of a conversation, and so on, specify the appropriate attribute.

See “Creating Conversational Web Services” on page 6-37 for detailed information and examples of using this annotation.
## Attributes

### Table B-16 Attributes of the weblogic.jws.Conversational JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
</table>
| maxIdleTime | Specifies the amount of time that a conversation can remain idle before it is finished by WebLogic Server. Activity is defined by a client Web Service executing one of the phases of the conversation. Valid values 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, specify the annotation as follows:  
@Conversational(maxIdleTime="10 minutes")  
If you specify a zero-length value (such as 0 seconds, or 0 minutes and so on), then the conversation never times out due to inactivity. Default value is 0 seconds. | String    | No         |
| maxAge    | The amount of time that a conversation can remain active before it is finished by WebLogic Server. Valid values are a number and one of the following terms:  
- seconds  
- minutes  
- hours  
- days  
- years  
For example, to specify a maximum age of three days, specify the annotation as follows:  
@Conversational(maxAge="3 days")  
Default value is 1 day. | String    | No         |
Example

The following sample snippet shows how to specify that a JWS file implements a conversational Web Service. The maximum amount of time the conversation can be idle is ten minutes, and the maximum age of the conversation, regardless of activity, is one day. The continue and finish phases of the conversation can be executed by a user other than the one that started the conversation; if this happens, then the corresponding methods are run as the new user, not the original user.

```
package examples.webservices.conversation;

...

@Conversational(maxIdleTime="10 minutes",
               maxAge="1 day")
```
public class ConversationalServiceImpl implements Serializable {
  ...

weblogic.jws.MessageBuffer

Description

Target: Class, Method

Specifies which public methods of a JWS are buffered. If specified at the class-level, then all public methods are buffered; if you want only a subset of the methods to be buffered, specify the annotation at the appropriate method-level.

When a client Web Service invokes a buffered operation of a different WebLogic Web Service, WebLogic Server (hosting the invoked Web Service) puts the invoke message on a JMS queue and the actual invoke is dealt with later on when the WebLogic Server delivers the message from the top of the JMS queue to the Web Service implementation. The client does not need to wait for a response, but rather, continues on with its execution. For this reason, buffered operations (without any additional asynchronous features) can only return void and must be marked with the @Oneway annotation. If you want to buffer an operation that returns a value, you must use asynchronous request-response from the invoking client Web Service. See “Invoking a Web Service Using Asynchronous Request-Response” on page 6-19 for more information.

Buffering works only between two Web Services in which one invokes the buffered operations of the other.

Use the optional attributes of @MessageBuffer to specify the number of times the JMS queue attempts to invoke the buffered Web Service operation until it is invoked successfully, and the amount of time between attempts.

Use the optional class-level @BufferQueue annotation to specify the JMS queue to which the invoke messages are queued. If you do not specify this annotation, the messages are queued to the default Web Service queue, weblogic.wsee.DefaultQueue.

See “Creating Buffered Web Services” on page 6-50 for detailed information and examples for using this annotation.
Attributes

Table B-17  Attributes of the weblogic.jws.MessageBuffer JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>retryCount</td>
<td>Specifies the number of times that the JMS queue on the invoked WebLogic Server instance attempts to deliver the invoking message to the Web Service implementation until the operation is successfully invoked. Default value is 3.</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>retryDelay</td>
<td>Specifies the amount of time that elapses between message delivery retry attempts. The retry attempts are between the invoke message on the JMS queue and delivery of the message to the Web Service implementation. Valid values are a number and one of the following terms: • seconds • minutes • hours • days • years For example, to specify a retry delay of two days, specify: @MessageBuffer(retryDelay=&quot;2 days&quot;) Default value is 5 seconds.</td>
<td>String</td>
<td>No</td>
</tr>
</tbody>
</table>

Example

The following example shows a code snippet from a JWS file in which the public operation sayHelloNoReturn is buffered and the JMS queue to which WebLogic Server queues the operation invocation is called my.buffere.queue. The WebLogic Server instance that hosts the invoked Web Service tries a maximum of 10 times to deliver the invoke message from the JMS queue to the Web Service implementation, waiting 10 seconds between each retry. Only the relevant Java code is shown in the following snippet:

package examples.webservices.buffered;

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

@BufferQueue(name="my.buffer.queue")

public class BufferedImpl {
...
@WebMethod()
@MessageBuffer(retryCount=10, retryDelay="10 seconds")
@Oneway()

  public void sayHelloNoReturn(String message) {
    System.out.println("sayHelloNoReturn: " + message);
  }
}  

weblogic.jws.Policies

Description  
Target: Class, Method

Specifies an array of @weblogic.jws.Policy annotations.

Use this annotation if you want to attach more than one WS-Policy files to a class or method of a JWS file. If you want to attach just one WS-Policy file, you can use the @weblogic.jws.Policy on its own.

See “Using Web Service Reliable Messaging” on page 6-1 and “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3 for detailed information and examples of using this annotation.

This JWS annotation does not have any attributes.

Example

@Policies({
  @Policy(uri="policy:findFirstPolicy.xml"),
  @Policy(uri="policy:findSecondPolicy.xml")
})
weblogic.jws.Policy

Description

**Target:** Class, Method

Specifies that a WS-Policy file, which contains information about digital signatures, encryption, or Web Service reliable messaging, should be applied to the request or response SOAP messages.

This annotation can be used on its own to apply a single WS-Policy file to a class or method. If you want to apply more than one WS-Policy file to a class or method, use the `@weblogic.jws.Policies` annotation to group them together.

If this annotation is specified at the class level, the indicated WS-Policy file or files are applied to every public operation of the Web Service. If the annotation is specified at the method level, then only the corresponding operation will have the WS-Policy file applied.

By default, WS-Policy files are applied to both the request (inbound) and response (outbound) SOAP messages. You can change this default behavior with the `direction` attribute.

Also by default, the specified WS-Policy file is attached to the generated and published WSDL file of the Web Service so that consumers can view all the WS-Policy requirements of the Web Service. Use the `attachToWsdl` attribute to change this default behavior.

See “Using Web Service Reliable Messaging” on page 6-1 and “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3 for detailed information and examples of using this annotation.

**WARNING:** As is true for all JWS annotations, the `@Policy` annotation cannot be overridden at runtime, which means that the WS-Policy file you specify at buildtime using the annotation will always be associated with the Web Service. This means, for example, that although you can view the associated WS-Policy file at runtime using the Administration Console, you cannot delete (unassociate) it. You can, however, associate additional WS-Policy files using the console; see Associate a WS-Policy file with a Web Service for detailed instructions.
### Attributes

#### Table B-18 Attributes of the weblogic.jws.Policies JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>Specifies the location from which to retrieve the WS-Policy file.</td>
<td>String</td>
<td>Yes.</td>
</tr>
<tr>
<td></td>
<td>Use the <code>http:</code> prefix to specify the URL of a WS-Policy file on the Web.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use the <code>policy:</code> prefix to specify that the WS-Policy file is packaged in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>the Web Service archive file or in a shareable J2EE library of WebLogic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Server, as shown in the following example:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@Policy(uri=&quot;policy:MyPolicyFile.xml&quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you are going to publish the WS-Policy file in the Web Service archive,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>the WS-Policy XML file must be located in either the <code>META-INF/policies</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>or <code>WEB-INF/policies</code> directory of the EJB JAR file (for EJB implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Web Services) or WAR file (for Java class implemented Web Services),</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>respectively.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>For information on publishing the WS-Policy file in a library, see Creating</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared J2EE Libraries and Optional Packages.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>direction</td>
<td>Specifies when to apply the policy: on the inbound request SOAP message,</td>
<td>enum</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>the outbound response SOAP message, or both (default).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Valid values for this attribute are:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Policy.Direction.both</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Policy.Direction.inbound</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Policy.Direction.outbound</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value is Policy.Direction.both.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>attachToWsdl</td>
<td>Specifies whether the WS-Policy file should be attached to the WSDL that</td>
<td>boolean</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>describes the Web Service.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Valid values are <code>true</code> and <code>false</code>. Default value is <code>false</code>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```java
@Policy(uri="policy:myPolicy.xml",
   attachToWsdl=true,
   direction=Policy.Direction.outbound)
```

### weblogic.jws.ReliabilityBuffer

#### Description

**Target:** Method

Use this annotation to configure reliable messaging properties for an operation of a reliable Web Service, such as the number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web Service implementation, and the amount of time that the server should wait in between retries.

**Note:** It is assumed when you specify this annotation in a JWS file that you have already enabled reliable messaging for the Web Service by also including a `@Policy` annotation that specifies a WS-Policy file that has Web Service reliable messaging policy assertions.

If you specify the `@ReliabilityBuffer` annotation, but do not enable reliable messaging with an associated WS-Policy file, then WebLogic Server ignores this annotation.

See “Using Web Service Reliable Messaging” on page 6-1 for detailed information about enabling Web Services reliable messaging for your Web Service.
Attributes

Table B-19  Attributes of the weblogic.jws.ReliabilityBuffer JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>retryCount</td>
<td>Specifies the number of times that the JMS queue on the destination WebLogic Server instance attempts to deliver the message from a client that invokes the reliable operation to the Web Service implementation. Default value is 3.</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>retryDelay</td>
<td>Specifies the amount of time that elapses between message delivery retry attempts. The retry attempts are between the client’s request message on the JMS queue and delivery of the message to the Web Service implementation. Valid values are a number and one of the following terms: • seconds • minutes • hours • days • years For example, to specify a retry delay of two days, specify: @ReliabilityBuffer(retryDelay=&quot;2 days&quot;) Default value is 5 seconds.</td>
<td>String</td>
<td>No</td>
</tr>
</tbody>
</table>

Example

The following sample snippet shows how to use the @ReliabilityBuffer annotation at the method-level to change the default retry count and delay of a reliable operation; only relevant Java code is shown:

```java
package examples.webservices.reliable;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.Oneway;
```
... import weblogic.jws.ReliabilityBuffer;
import weblogic.jws.Policy;

@WebService(name="ReliableHelloWorldPortType",
            serviceName="ReliableHelloWorldService")

... @Policy(uri="ReliableHelloWorldPolicy.xml",
             direction=Policy.Direction.inbound,
             attachToWsdl=true)
public class ReliableHelloWorldImpl {

    @WebMethod()
    @Oneway()
    @ReliabilityBuffer(retryCount=10, retryDelay="10 seconds")
    public void helloWorld(String input) {
        System.out.println(" Hello World " + input);
    }
}

weblogic.jws.ReliabilityErrorHandler

Description

Target: Method

Specifies the method that handles the error that results when a client Web Service invokes a reliable Web Service, but the client does not receive an acknowledgement that the reliable Web Service actually received the message.

This annotation is relevant only when you implement the Web Service reliable messaging feature; you specify the annotation in the client-side Web Service that invokes a reliable Web Service.

The method you annotate with the @ReliabilityErrorHandler annotation takes a single parameter of data type weblogic.wsee.reliability.ReliabilityErrorHandler. You can use this context to get more information about the cause of the error, such as the operation that caused it, the target Web Service, the fault, and so on. The method must return void.
The single attribute of the @ReliabilityErrorHandler annotation specifies the variable into which you have previously injected the JAX-RPC stub information of the reliable Web Service that the client Web Service is invoking; you inject this information in a variable using the @weblogic.jws.ServiceClient annotation.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>Specifies the target stub name for which this method handles reliability failures.</td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Example**

The following code snippet from a client Web Service that invokes a reliable Web Service shows how to use the @ReliabilityErrorHandler annotation; not all code is shown, and the code relevant to this annotation is shown in bold:

```java
package examples.webservices.reliable;

... import weblogic.jws.ServiceClient; import weblogic.jws.ReliabilityErrorHandler; import examples.webservices.reliable.ReliableHelloWorldPortType; import weblogic.wsee.reliability.ReliabilityErrorContext; import weblogic.wsee.reliability.ReliableDeliveryException;

@WebService(name="ReliableClientPortType", ...

public class ReliableClientImpl
{
    @ServiceClient(

```
private ReliableHelloWorldPortType port;

@WebMethod
public void callHelloWorld(String input, String serviceUrl)
    throws RemoteException {
    ...
}

@ReliabilityErrorHandler(target="port")
public void onReliableMessageDeliveryError(ReliabilityErrorContext ctx) {
    ReliableDeliveryException fault = ctx.getFault();
    String message = null;
    if (fault != null) {
        message = ctx.getFault().getMessage();
    }
    String operation = ctx.getOperationName();
    System.out.println("Reliable operation " + operation + " may have not
    invoked. The error message is " + message);
}

In the example, the port variable has been injected with the JAX-RPC stub that corresponds to the ReliableHelloWorldService Web Service, and it is assumed that at some point in the client Web Service an operation of this stub is invoked. Because the onReliableMessageDeliveryError method is annotated with the @ReliabilityErrorHandler annotation and is linked with the port JAX-RPC stub, the method is invoked if there is a failure in an invoke of the reliable Web Service. The reliable error handling method uses the ReliabilityErrorContext object to get more details about the cause of the failure.

**weblogic.jws.ServiceClient**

**Description**

**Target:** Field
Specifies that the annotated variable in the JWS file is a JAX-RPC stub used to invoke another WebLogic Web Service when using the following features:

- Web Service reliable messaging
- asynchronous request-response
- conversations

You use the reliable messaging and asynchronous request-response features only between two Web Services; this means, for example, that you can invoke a reliable Web Service operation only from within another Web Service, not from a stand-alone client. In the case of reliable messaging, the feature works between any two application servers that implement the WS-ReliableMessaging 1.0 specification. In the case of asynchronous request-response, the feature works only between two WebLogic Server instances.

You use the @ServiceClient annotation in the client Web Service to specify which variable is a JAX-RPC port type for the Web Service described by the @ServiceClient attributes. The Enterprise Application that contains the client Web Service must also include the JAX-RPC stubs of the Web Service you are invoking; you generate the stubs with the clientgen Ant task.

See Chapter 6, “Advanced JWS Programming: Implementing Asynchronous Features,” for additional information and examples of using the @ServiceClient annotation.
# Attributes

Table B-21  Attributes of the weblogic.jws.ServiceClient JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>serviceName</td>
<td>Specifies the name of the Web Service that you are invoking. Corresponds to the name attribute of the <code>&lt;service&gt;</code> element in the WSDL of the invoked Web Service. If you used a JWS file to implement the invoked Web Service, this attribute corresponds to the serviceName attribute of the @WebService JWS annotation in the invoked Web Service.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>portName</td>
<td>Specifies the name of the port of the Web Service you are invoking. Corresponds to the name attribute of the <code>&lt;port&gt;</code> child element of the <code>&lt;service&gt;</code> element. If you used a JWS file to implement the invoked Web Service, this attribute corresponds to the portName attribute of the @WLHttpTransport JWS annotation in the invoked Web Service. If you do not specify this attribute, it is assumed that the <code>&lt;service&gt;</code> element in the WSDL contains only one <code>&lt;port&gt;</code> child element, which @ServiceClient uses. If there is more than one port, the client Web Service returns a runtime exception.</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>wsdlLocation</td>
<td>Specifies the WSDL file that describes the Web Service you are invoking. If you do not specify this attribute, the client Web Service uses the WSDL file from which the clientgen Ant task created the JAX-RPC Service implementation of the Web Service to be invoked.</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>endpointAddress</td>
<td>Specifies the endpoint address of the Web Service you are invoking. If you do not specify this attribute, the client Web Service uses the endpoint address specified in the WSDL file.</td>
<td>String</td>
<td>No</td>
</tr>
</tbody>
</table>
Example

The following JWS file excerpt shows how to use the `@ServiceClient` annotation in a client Web Service to annotate a field (`port`) with the JAX-RPC stubs of the Web Service being invoked (called `ReliableHelloWorldService` whose WSDL is at the URL `http://localhost:7001/ReliableHelloWorld/ReliableHelloWorld?WSDL`); only relevant parts of the example are shown:

```java
package examples.webservices.reliable;
import javax.jws.WebService;
...
import weblogic.jws.ServiceClient;
import examples.webservices.reliable.ReliableHelloWorldPortType;
@WebService(...
public class ReliableClientImpl {
    @ServiceClient(
    serviceName="ReliableHelloWorldService",
    portName="ReliableHelloWorldServicePort")
    private ReliableHelloWorldPortType port;
    @WebMethod
    public void callHelloWorld(String input, String serviceUrl)
    throws RemoteException {
        port.helloWorld(input);
        System.out.println(" Invoked the ReliableHelloWorld.helloWorld operation reliably.");
    }
}
```
weblogic.jws.StreamAttachments

Description
Target: Class
Specifies that the WebLogic Web Services runtime use streaming APIs when reading the parameters of all methods of the Web Service. This increases the performance of Web Service operation invocation, in particular when the parameters are large, such as images.
You cannot use this annotation if you are also using the following features in the same Web Service:

- Conversations
- Reliable Messaging
- JMS Transport
- A proxy server between the client application and the Web Service it invokes

The @StreamAttachments annotation does not have any attributes.

Example
The following simple JWS file shows how to specify the @StreamAttachments annotation; the single method, echoAttachment(), simply takes a DataHandler parameter and echoes it back to the client application that invoked the Web Service operation. The WebLogic Web Services runtime uses streaming when reading the DataHandler content.

```java
package examples.webservices.stream_attach;
import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.StreamAttachments;
import javax.activation.DataHandler;
import java.rmi.RemoteException;

@WebService(name="StreamAttachPortType",
    serviceName="StreamAttachService",
    targetNamespace="http://example.org")
```
**weblogic.jws.Transactional**

**Description**

**Target:** Class, Method

Specifies whether the annotated operation, or all the operations of the JWS file when the annotation is specified at the class-level, runs or run inside of a transaction. By default, the operations do not run inside of a transaction.
## Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Specifies whether the operation (when used at the method level) or all the operations of the Web Service (when specified at the class level) run inside of a transaction. Valid values are true and false. Default value is false.</td>
<td>boolean</td>
<td>No</td>
</tr>
<tr>
<td>timeout</td>
<td>Specifies a timeout value, in seconds, for the current transaction. The default value for this attribute is 30 seconds.</td>
<td>int</td>
<td>No</td>
</tr>
</tbody>
</table>

## Example

The following example shows how to use the @Transactional annotation to specify that an operation of a Web Service executes as part of a transaction:

```java
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.
 * @author Copyright (c) 2004 by BEA Systems. All rights reserved.
 */
```
public class TransactionPojoImpl {

@WebMethod()
@Transactional(value=true)

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

weblogic.jws.Types

Description

Target: Method, Parameter

Specifies a comma-separated list of fully qualified Java class names of the alternative data types for a return type or parameter. The alternative data types must extend the data type specified in the method signature; if this is not the case, the jwsc Ant task returns a validation error when you compile the JWS file into a Web Service.

For example, assume you have created the Address base data type, and then created USAAddress and CAAddress that extend this base type. If the method signature specifies that it takes an Address parameter, you can annotate the parameter with the @Types annotation to specify that that the public operation also takes USAAddress and CAAddress as a parameter, in addition to the base Address data type.

You can also use this annotation to restrict the data types that can be contained in parameters or return values of collection data types, such as java.util.Collection or java.util.List. By restricting the allowed contained data types, the generated WSDL is specific and unambiguous, and the Web Services runtime can do a better job of qualifying the parameters when a client application invokes a Web Service operation.

If you specify this annotation at the method-level, then it applies only to the return value. If you want the annotation to apply to parameters, you must specify it at the parameter-level for each relevant parameter.
Attributes

Table B-23 Attributes of the weblogic.jws.Types JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Comma-separated list of fully qualified class names for either the alternative data types that can also be used instead of the original data type, or the allowed data types contained in the collection-type parameter or return value.</td>
<td>String[]</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example

The following example shows a simple JWS file that uses the @Types annotation, with relevant Java code shown in bold:

```java
package examples.webservices.types;
import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Types;
import examples.webservices.types.BasicStruct;

@WebService(serviceName="TypesService",
    name="TypesPortType",
    targetNamespace="http://example.org")
@WLHttpTransport(contextPath="types",
    serviceUri="TypesService",
    portName="TypesServicePort")
public class TypesImpl {
    @WebMethod()
    @Types("examples.webservices.types.ExtendedStruct")
    public BasicStruct echoStruct(@Types("examples.webservices.types.ExtendedStruct") BasicStruct struct) {
        System.out.println("echoStruct called");
    }
}
```
In the example, the signature of the `echoStruct()` method shows that it takes a `BasicStruct` value as both a parameter and a return value. However, because both the method and the `struct` parameter are annotated with the `@Types` annotation, a client application invoking the `echoStruct` operation can also pass it a parameter of data type `ExtendedStruct`; in this case the operation also returns an `ExtendedStruct` value. It is assumed that `ExtendedStruct extends BasicStruct`.

**weblogic.jws.WildcardBinding**

**Description**

**Target:** Class

Specifies the XML Schema data type to which a wildcard class, such as `javax.xml.soap.SOAPElement` or `org.apache.xmlbeans.XmlObject`, binds. By default, these Java data types bind to the `<xsd:any>` XML Schema data type. By using this class-level annotation, you can specify that the wildcard classes bind to `<xsd:anyType>` instead.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>Specifies the fully qualified name of the wildcard class for which this binding applies. Typical values are <code>javax.xml.soap.SOAPElement</code> and <code>org.apache.xmlbeans.XmlObject</code>.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>binding</td>
<td>Specifies the XML Schema data type to which the wildcard class should bind. You can specify one of the following values:</td>
<td>enum</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>• <code>WildcardParticle.ANY</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>WildcardParticle.ANYTYPE</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
WebLogic-Specific JWS Annotations Reference

Example
The following example shows how to use the `@WildcardBinding` annotation to specify that the Apache XMLBeans data type `XMLObject` should bind to the `<xsd:any>` XML Schema data type for this Web Service:

```java
@WildcardBindings({
    @WildcardBinding(className="org.apache.xmlbeans.XmlObject",
        binding=WildcardParticle.ANY),
    @WildcardBinding(className="org.apache.xmlbeans.XmlObject[]",
        binding=WildcardParticle.ANY))

public class SimpleImpl {
...
```

weblogic.jws.WildcardBindings

Description
Target: Class

Specifies an array of `weblogic.jws.WildcardBinding` annotations.
This JWS annotation does not have any attributes.

See “weblogic.jws.WildcardBinding” on page B-56 for an example.

weblogic.jws.WLHttpTransport

Description
Target: Class

Specifies the context path and service URI sections of the URL used to invoke the Web Service over the HTTP transport, as well as the name of the port in the generated WSDL.
You can specify this annotation only once (maximum) in a JWS file.
Attributes

Table B-25 Attributes of the weblogic.jws.WLHttpTransport JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>contextPath</td>
<td>Context path of the Web Service. You use this value in the URL that invokes the Web Service. For example, assume you set the context path for a Web Service to financial; a possible URL for the WSDL of the deployed WebLogic Web Service is as follows: <a href="http://hostname:7001/financial/GetQuote?WSDL">http://hostname:7001/financial/GetQuote?WSDL</a> The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is HelloWorldImpl.java, then the default value of its contextPath is HelloWorldImpl.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>serviceUri</td>
<td>Web Service URI portion of the URL. You use this value in the URL that invokes the Web Service. For example, assume you set this attribute to GetQuote; a possible URL for the deployed WSDL of the service is as follows: <a href="http://hostname:7001/financial/GetQuote?WSDL">http://hostname:7001/financial/GetQuote?WSDL</a> The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is HelloWorldImpl.java, then the default value of its serviceUri is HelloWorldImpl.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td>portName</td>
<td>The name of the port in the generated WSDL. This attribute maps to the name attribute of the &lt;port&gt; element in the WSDL. The default value of this attribute is based on the @javax.jws.WebService annotation of the JWS file. In particular, the default portName is the value of the name attribute of @WebService annotation, plus the actual text SoapPort. For example, if @WebService.name is set to MyService, then the default portName is MyServiceSoapPort.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>
Example

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

weblogic.jws.WLHttpsTransport

Description

Target: Class

WARNING: The @weblogic.jws.WLHttpsTransport annotation is deprecated as of version 9.2 of WebLogic Server. You should use the @weblogic.jws.WLHttpTransport annotation instead because it now supports both the HTTP and HTTPS protocols. If you want client applications to access the Web Service using only the HTTPS protocol, then you must specify the @weblogic.jws.security.UserDataConstraint JWS annotation in your JWS file.

Specifies the context path and service URI sections of the URL used to invoke the Web Service over the HTTPS transport, as well as the name of the port in the generated WSDL.

You can specify this annotation only once (maximum) in a JWS file.
Attributes

Table B-26  Attributes of the weblogic.jws.WLHttpsTransport JWS Annotation Tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
</table>
| contextPath| Context path of the Web Service. You use this value in the URL that invokes the Web Service. For example, assume you set the context path for a Web Service to financial; a possible URL for the WSDL of the deployed WebLogic Web Service is as follows:  
The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is HelloWorldImpl.java, then the default value of its contextPath is HelloWorldImpl. | String    | No.       |
| serviceUri | Web Service URI portion of the URL. You use this value in the URL that invokes the Web Service. For example, assume you set this attribute to GetQuote; a possible URL for the deployed WSDL of the service is as follows:  
The default value of this attribute is the name of the JWS file, without its extension. For example, if the name of the JWS file is HelloWorldImpl.java, then the default value of its serviceUri is HelloWorldImpl. | String    | No.       |
| portName   | The name of the port in the generated WSDL. This attribute maps to the name attribute of the <port> element in the WSDL. The default value of this attribute is based on the @javax.jws.WebService annotation of the JWS file. In particular, the default portName is the value of the name attribute of @WebService annotation, plus the actual text SoapPort. For example, if @WebService.name is set to MyService, then the default portName is MyServiceSoapPort. | String    | No.       |
Example

```java
@WLHttpsTransport(portName="helloSecurePort",
                   contextPath="secure",
                   serviceUri="SimpleSecureBean")
```

**weblogic.jws.WLJmsTransport**

**Description**

**Target:** Class

Specifies the context path and service URI sections of the URL used to invoke the Web Service over the JMS transport, as well as the name of the port in the generated WSDL. You also use this annotation to specify the JMS queue to which WebLogic Server queues the SOAP request messages from invokes of the operations.

You can specify this annotation only once (maximum) in a JWS file.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contextPath</code></td>
<td>Context root of the Web Service. You use this value in the URL that invokes the Web Service.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td><code>serviceUri</code></td>
<td>Web Service URI portion of the URL used by client applications to invoke the Web Service.</td>
<td>String</td>
<td>No.</td>
</tr>
<tr>
<td><code>queue</code></td>
<td>The JNDI name of the JMS queue that you have configured for the JMS transport. See “Using JMS Transport as the Connection Protocol” on page 7-1 for details about using JMS transport. The default value of this attribute, if you do not specify it, is <code>weblogic.wsee.DefaultQueue</code>. You must still create this JMS queue in the WebLogic Server instance to which you deploy your Web Service.</td>
<td>String</td>
<td>No.</td>
</tr>
</tbody>
</table>
Example

The following example shows how to specify that the JWS file implements a Web Service that is invoked using the JMS transport. The JMS queue to which WebLogic Server queues SOAP message requests from invokes of the service operations is `JMSTransportQueue`; it is assumed that this JMS queue has already been configured for WebLogic Server.

```java
WLJmsTransport(contextPath="transports",
    serviceUri="JMSTransport",
    queue="JMSTransportQueue",
    portName="JMSTransportServicePort")
```

weblogic.jws.WSDL

Description

**Target:** Class

Specifies whether to expose the WSDL of a deployed WebLogic Web Service.

By default, the WSDL is exposed at the following URL:

```
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* and *serviceUri* refer to the value of the *contextPath* and *serviceUri* attributes, respectively, of the @WLHttpTransport JWS annotation of the JWS file that implements your Web Service.

For example, assume you used the following @WLHttpTransport annotation:

```java
@WLHttpTransport (portName="helloPort",
contextPath="hello",
serviceUri="SimpleImpl")
```

The URL to get view the WSDL of the Web Service, assuming the service is running on a host called *ariel* at the default port number, is:


## Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>exposed</td>
<td>Specifies whether to expose the WSDL of a deployed Web Service.</td>
<td>boolean</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>Valid values are <em>true</em> and <em>false</em>. Default value is <em>true</em>, which means</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>that by default the WSDL is exposed.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Example

The following use of the @WSDL annotation shows how to specify that the WSDL of a deployed Web Service not be exposed; only relevant Java code is shown:

```java
package examples.webservices;
import...

@WebService (name="Wsd1AnnotationPortType",
serviceName="Wsd1AnnotationService",
targetNamespace="http://example.org")

@WSDL (exposed=false)
```
public class WsdlAnnotationImpl {
...
}

weblogic.jws.security.CallbackRolesAllowed

Description
Target: Method, Field
Specifies an array of @SecurityRole JWS annotations that list the roles that are allowed to invoke the callback methods of the Web Service. A user that is mapped to an unspecified role, or is not mapped to any role at all, would not be allowed to invoke the callback methods.

If you use this annotation at the field level, then the specified roles are allowed to invoke all callback operations of the Web Service. If you use this annotation at the method-level, then the specified roles are allowed to invoke only that callback method. If specified at both levels, the method value overrides the field value if there is a conflict.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Array of @weblogic.jws.security.RolesAllowed that list the roles allowed to invoke the callback methods.</td>
<td>String[]</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example
The following example shows how to use the @CallbackRolesAllowed annotation at the method level to specify that the role engineer is allowed to invoke the callback method:

```java
@CallbackMethod(target="port", operation="callbackOperation")
@CallbackRolesAllowed(@SecurityRole(role="engineer",
mapToPrincipals="shackell"))
public void callbackHandler(String msg) {
    System.out.println (msg);
}
weblogic.jws.security.RolesAllowed

Description

**Target:** Class, Method

JWS annotation used to enable basic authentication for a Web Service. In particular, it specifies an array of @SecurityRole JWS annotations that describe the list of roles that are allowed to invoke the Web Service. A user that is mapped to an unspecified role, or is not mapped to any role at all, would not be allowed to invoke the Web Service.

If you use this annotation at the class-level, then the specified roles are allowed to invoke all operations of the Web Service. To specify roles for just a specific set of operations, specify the annotation at the operation-level.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Array of @weblogic.jws.security.RolesAllowed that list the roles allowed to invoke the Web Service methods.</td>
<td>String[]</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example

```java
package examples.webservices.security_roles;

...
import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.SecurityRole;

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

@RolesAllowed (@SecurityRole (role="manager",
```

```
mapToPrincipals={ "juliet","amanda" },
@SecurityRole (role="vp")
} }

In the example, only the roles manager and vp are allowed to invoke the Web Service. Within
the context of the Web Service, the users juliet and amanda are assigned the role manager. The
role vp, however, does not include a mapToPrincipals attribute, which implies that users have
been mapped to this role externally. It is assumed that you have already added the two users
(juliet and amanda) to the WebLogic Server security realm.

**weblogic.jws.security.RolesReferenced**

**Description**

**Target:** Class

JWS annotation used to specify the list of role names that reference actual roles that are allowed
to invoke the Web Service. In particular, it specifies an array of @SecurityRoleRef JWS
annotations, each of which describe a link between a referenced role name and an actual role
defined by a @SecurityRole annotation.

This JWS annotation does not have any attributes.

**Example**

```java
package examples.webservices.security_roles;
...
import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.SecurityRole;
import weblogic.jws.security.RolesReferenced;
import weblogic.jws.security.SecurityRoleRef;

@WebService(name="SecurityRolesPortType",
    serviceName="SecurityRolesService",
    targetNamespace="http://example.org")
@RolesAllowed ( { 
    @SecurityRole (role="manager",}
mapToPrincipals={ "juliet","amanda" },
@SecurityRole (role="vp")
} )

@RolesReferenced {
    @SecurityRoleRef (role="mgr", link="manager")
}

public class SecurityRolesImpl {
    ...

    In the example, the role mgr is linked to the role manager, which is allowed to invoke the Web Service. This means that any user who is assigned to the role of mgr is also allowed to invoke the Web Service.

**weblogic.jws.security.RunAs**

**Description**

**Target:** Class

Specifies the role and user identity which actually runs the Web Service in WebLogic Server.

For example, assume that the @RunAs annotation specifies the roleA role and userA principal. This means that even if the Web Service is invoked by userB (mapped to roleB), the relevant operation is actually executed internal as userA.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td>Specifies the role which the Web Service should be run as.</td>
<td>String</td>
<td>Yes.</td>
</tr>
<tr>
<td>mapToPrincipal</td>
<td>Specifies the principal user that maps to the role. It is assumed that you have already configured the specified principal (user) as a valid WebLogic Server user, typically using the Administration Console. See Create Users for details.</td>
<td>String</td>
<td>Yes.</td>
</tr>
</tbody>
</table>
Example

```java
package examples.webservices.security_roles;
import weblogic.jws.security.RunAs;
...
@WebService(name="SecurityRunAsPortType",
    serviceName="SecurityRunAsService",
    targetNamespace="http://example.org")
@RunAs (role="manager", mapToPrincipal="juliet")
public class SecurityRunAsImpl {
...

The example shows how to specify that the Web Service is always run as user juliet, mapped
to the role manager, regardless of who actually invoked the Web Service.
```

**weblogic.jws.security.SecurityRole**

**Description**

**Target:** Class, Method

Specifies the name of a role that is allowed to invoke the Web Service. This annotation is always
specified in the JWS file as a member of a @RolesAllowed array.

When a client application invokes the secured Web Service, it specifies a user and password as
part of its basic authentication. It is assumed that an administrator has already configured the user
as a valid WebLogic Server user using the Administration Console; for details see Create Users.

The user that is going to invoke the Web Service must also be mapped to the relevant role. You
can perform this task in one of the following two ways:

- Use the Administration Console to map the user to the role. In this case, you do not specify
the mapToPrincipals attribute of the @SecurityRole annotation. For details, see Add
Users to Roles.

- Map the user to a role only within the context of the Web Service by using the
mapToPrincipals attribute to specify one or more users.

To specify that multiple roles are allowed to invoke the Web Service, include multiple
@SecurityRole annotations within the @RolesAllowed annotation.
Attributes

Table B-32 Attributes of the weblogic.jws.security.SecurityRole JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td>The name of the role that is allowed to invoke the Web Service.</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>mapToPrincipals</td>
<td>An array of user names that map to the role.</td>
<td>String[]</td>
<td>No</td>
</tr>
</tbody>
</table>

Example

```java
package examples.webservices.security_roles;
...
import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.SecurityRole;

@WebService(name="SecurityRolesPortType",
            serviceName="SecurityRolesService",
            targetNamespace="http://example.org")
@RolesAllowed (  
    @SecurityRole (role="manager",
                   mapToPrincipals={ "juliet","amanda" }),
    @SecurityRole (role="vp")
) )
public class SecurityRolesImpl {
...
```

In the example, only the roles manager and vp are allowed to invoke the Web Service. Within the context of the Web Service, the users juliet and amanda are assigned the role manager. The role vp, however, does not include a mapToPrincipals attribute, which implies that users have been mapped to this role externally. It is assumed that you have already added the two users (juliet and amanda) to the WebLogic Server security realm.
### weblogic.jws.security.SecurityRoleRef

**Description**

**Target:** Class

Specifies a role name reference that links to an already-specified role that is allowed to invoke the Web Service.

Users that are mapped to the role reference can invoke the Web Service as long as the referenced role is specified in the `@RolesAllowed` annotation of the Web Service.

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td>Name of the role reference.</td>
<td>String</td>
<td>Yes.</td>
</tr>
<tr>
<td>link</td>
<td>Name of the already-specified role that is allowed to invoke the Web Service.</td>
<td>String</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

**Example**

```java
package examples.webservices.security_roles;
...
import weblogic.jws.security.RolesAllowed;
import weblogic.jws.security.SecurityRole;
import weblogic.jws.security.RolesReferenced;
import weblogic.jws.security.SecurityRoleRef;

@ServicePort(name="SecurityRolesPortType",
             serviceName="SecurityRolesService",
             targetNamespace="http://example.org")
@RolesAllowed ()
    @SecurityRole (role="manager",
```
mapToPrincipals={ "juliet","amanda" },
    @SecurityRole (role="vp")
} )

@RolesReferenced {
    @SecurityRoleRef (role="mgr", link="manager")
}

public class SecurityRolesImpl {
...

In the example, the role mgr is linked to the role manager, which is allowed to invoke the Web Service. This means that any user who is assigned to the role of mgr is also allowed to invoke the Web Service.

**weblogic.jws.security.UserDataConstraint**

**Description**

**Target:** Class

Specifies whether the client is required to use the HTTPS transport when invoking the Web Service.

WebLogic Server establishes a Secure Sockets Layer (SSL) connection between the client and Web Service if the transport attribute of this annotation is set to either Transport.INTEGRAL or Transport.CONFIDENTIAL in the JWS file that implements the Web Service.

If you specify this annotation in your JWS file, you must also specify the weblogic.jws.WLHttpTransport annotation (or the <WLHttpTransport> element of the jwsc Ant task) to ensure that an HTTPS binding is generated in the WSDL file by the jwsc Ant task.
Attributes

Table B-34  Attributes of the weblogic.jws.security.UserDataConstraint JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td>Specifies whether the client is required to use the HTTPS transport when invoking the Web Service. Valid values are:</td>
<td>enum</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>• Transport.NONE—Specifies that the Web Service does not require any transport guarantees.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Transport.INTEGRAL—Specifies that the Web Service requires that the data be sent between the client and Web Service in such a way that it cannot be changed in transit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Transport.CONFIDENTIAL—Specifies that the Web Service requires that data be transmitted so as to prevent other entities from observing the contents of the transmission.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default value is Transport.NONE.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

```java
package examples.webservices.security_https;

import weblogic.jws.security.UserDataConstraint;

...

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

@UserDataConstraint(
    transport=UserDataConstraint.Transport.CONFIDENTIAL)

public class SecurityHttpsImpl {
    ...
```
weblogic.jws.security.WssConfiguration

Description

Target: Class

Specifies the name of the Web Service security configuration you want the Web Service to use. If you do not specify this annotation in your JWS file, the Web Service is associated with the default security configuration (called default_wss) if it exists in your domain.

The @WssConfiguration annotation only makes sense if your Web Service is configured for message-level security (encryption and digital signatures). The security configuration, associated to the Web Service using this annotation, 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 digital signatures, and so on.

WebLogic Web Services are not required to be associated with a security configuration; if the default behavior of the Web Services security runtime is adequate then no additional configuration is needed. If, however, a Web Service requires different behavior from the default (such as using an X.509 certificate for identity, rather than the default username/password token), then the Web Service must be associated with a security configuration.

Before you can successfully invoke a Web Service that specifies a security configuration, you must use the Administration Console to create it. For details, see Create a Web Services security configuration. For general information about message-level security, see “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Specifies the name of the Web Service security configuration that is associated with this Web Service. The default configuration is called default_wss. You must create the security configuration (even the default one) using the Administration Console before you can successfully invoke the Web Service.</td>
<td>String</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Example

The following example shows how to specify that a Web Service is associated with the my_security_configuration security configuration; only the relevant Java code is shown:

```java
package examples.webservices.wss_configuration;
import javax.jws.WebService;
...
import weblogic.jws.security.WssConfiguration;
@WebService(...
...
@WebssConfiguration(value="my_security_configuration")
public class WssConfigurationImpl {
...
```

**weblogic.jws.soap.SOAPBinding**

**Description**

**Target:** Method

Specifies the mapping of a Web Service operation onto the SOAP message protocol.

This annotation is analogous to `@javax.jws.soap.SOAPBinding` except that it applies to a method rather than the class. With this annotation you can specify, for example, that one Web Service operation uses RPC-encoded SOAP bindings and another operation in the same Web Service uses document-literal-wrapped SOAP bindings.

**Note:** Because `@weblogic.jws.soap.SOAPBinding` and `@javax.jws.soap.SOAPBinding` have the same class name, be careful which annotation you are referring to when using it in your JWS file.
### Attributes

#### Table B-36  Attributes of the `weblogic.jws.soap.SOAPBinding` JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>Specifies the message style of the request and response SOAP messages of the invoked annotated operation. Valid values are: • <code>SOAPBinding.Style.RPC</code> • <code>SOAPBinding.Style.DOCUMENT</code>. Default value is <code>SOAPBinding.Style.DOCUMENT</code>.</td>
<td>enum</td>
<td>No.</td>
</tr>
<tr>
<td>use</td>
<td>Specifies the formatting style of the request and response SOAP messages of the invoked annotated operation. Valid values are: • <code>SOAPBinding.Use.LITERAL</code> • <code>SOAPBinding.Use.ENCODED</code> Default value is <code>SOAPBinding.Use.LITERAL</code>.</td>
<td>enum</td>
<td>No.</td>
</tr>
<tr>
<td>parameterStyle</td>
<td>Determines whether method parameters represent the entire message body, or whether the parameters are elements wrapped inside a top-level element named after the operation. Valid values are: • <code>SOAPBinding.ParameterStyle.BARE</code> • <code>SOAPBinding.ParameterStyle.WRAPPED</code> Default value is <code>SOAPBinding.ParameterStyle.WRAPPED</code> Note: This attribute applies only to Web Services of style document-literal. Or in other words, you can specify this attribute only if you have also set the <strong>style</strong> attribute to <code>SOAPBinding.Style.DOCUMENT</code> and the <strong>use</strong> attribute to <code>SOAPBinding.Use.LITERAL</code>.</td>
<td>enum</td>
<td>No.</td>
</tr>
</tbody>
</table>
Example
The following simple JWS file shows how to specify that, by default, the operations of the Web Service use document-literal-wrapped SOAP bindings; you specify this by using the
@javax.jws.soap.SOAPBinding annotation at the class-level. The example then shows how to specify different SOAP bindings for individual methods by using the
@weblogic.jws.soap.SOAPBinding annotation at the method-level. In particular, the sayHelloDocLitBare() method uses document-literal-bare SOAP bindings, and the sayHelloRPCEncoded() method uses RPC-encoded SOAP bindings.

```java
package examples.webservices.soap_binding_method;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import weblogic.jws.WLHttpTransport;
@WebService(name="SoapBindingMethodPortType",
    serviceName="SoapBindingMethodService",
    targetNamespace="http://example.org")
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
    use=SOAPBinding.Use.LITERAL,
    parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
@WLHttpTransport(contextPath="soap_binding_method",
    serviceUri="SoapBindingMethodService",
    portName="SoapBindingMethodServicePort")
/**
 * Simple JWS example that shows how to specify soap bindings for a method.
 */
public class SoapBindingMethodImpl {
    @WebMethod()
    @weblogic.jws.soap.SOAPBinding(
        style=SOAPBinding.Style.DOCUMENT,
        use=SOAPBinding.Use.LITERAL,
        parameterStyle=SOAPBinding.ParameterStyle.BARE)
    public String sayHelloDocLitBare(String message) {
        System.out.println("sayHelloDocLitBare" + message);
    }
```
return "Here is the message: '" + message + "'");
}

@WebMethod()
@weblogic.jws.soap.SOAPBinding(
    style=SOAPBinding.Style.RPC,
    use=SOAPBinding.Use.ENCODED)

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

weblogic.jws.security.SecurityRoles (deprecated)

Description

Target: Class, Method

Note: The @weblogic.security.jws.SecurityRoles JWS annotation is deprecated beginning in WebLogic Server 9.0.

Specifies the roles that are allowed to access the operations of the Web Service.

If you specify this annotation at the class level, then the specified roles apply to all public operations of the Web Service. You can also specify a list of roles at the method level if you want to associate different roles to different operations of the same Web Service.

Note: The @SecurityRoles annotation is supported only within the context of an EJB-implemented Web Service. For this reason, you can specify this annotation only inside of a JWS file that explicitly implements javax.ejb.SessionBean. See Securing Enterprise JavaBeans (EJBs) for conceptual information about what it means to secure access to an EJB. See “Should You Implement a Stateless Session EJB?” on page 5-16 for information about explicitly implementing an EJB in a JWS file.
Attributes

Table B-37 Attributes of the weblogic.jws.security.SecurityRoles JWS Annotation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolesAllowed</td>
<td>Specifies the list of roles that are allowed to access the Web Service.</td>
<td>Array of String</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>This annotation is the equivalent of the &lt;method-permission&gt; element in the ejb-jar.xml deployment descriptor of the stateless session EJB that implements the Web Service.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rolesReferenced</td>
<td>Specifies a list of roles referenced by the Web Service. The Web Service may access other resources using the credentials of the listed roles.</td>
<td>Array of String</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>This annotation is the equivalent of the &lt;security-role-ref&gt; element in the ejb-jar.xml deployment descriptor of the stateless session EJB that implements the Web Service.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

The following example shows how to specify, at the class-level, that the Web Service can be invoked only by the Admin role; only relevant parts of the example are shown:

```java
package examples.webservices.security_roles;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import weblogic.ejbgen.Session;
import javax.jws.WebService;
...
import weblogic.jws.security.SecurityRoles;
@Session(ejbName="SecurityRolesEJB")
@WebService(...
// Specifies the roles who can invoke the entire Web Service
```
public class SecurityRolesImpl implements SessionBean {
...

weblogic.jws.security.SecurityIdentity (deprecated)

Description
Target: Class

Note: The `@weblogic.security.jws.SecurityIdentity` JWS annotation is deprecated beginning in WebLogic Server 9.1.

Specifies the identity assumed by the Web Service when it is invoked.

Unless otherwise specified, a Web Service assumes the identity of the authenticated invoker. This annotation allows the developer to override this behavior so that the Web Service instead executes as a particular role. The role must map to a user or group in the WebLogic Server security realm.

Note: The `@SecurityIdentity` annotation only makes sense within the context of an EJB-implemented Web Service. For this reason, you can specify this annotation only inside of a JWS file that explicitly implements `javax.ejb.SessionBean`. See Securing Enterprise JavaBeans (EJBs) for conceptual information about what it means to secure access to an EJB. See “Should You Implement a Stateless Session EJB?” on page 5-16 for information about explicitly implementing an EJB in a JWS file.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Specifies the role which the Web Service assumes when it is invoked. The role must map to a user or group in the WebLogic Server security realm.</td>
<td>String</td>
<td>Yes.</td>
</tr>
</tbody>
</table>
Example
The following example shows how to specify that the Web Service, when invoked, runs as the Admin role:

```java
package examples.webservices.security_roles;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import weblogic.ejbgen.Session;
import javax.jws.WebService;
...
import weblogic.jws.security.SecurityIdentity;
@Session ejbName="SecurityRolesEJB"
@WebService(...
// Specifies that the Web Service runs as the Admin role
@SecurityIdentity( value="Admin")
public class SecurityRolesImpl implements SessionBean {
...
```
Web Service Reliable Messaging Policy Assertion Reference

The following sections provide reference information about Web Service reliable messaging policy assertions in a WS-Policy file:

- “Overview of a WS-Policy File That Contains Web Service Reliable Messaging Assertions” on page C-1
- “Graphical Representation” on page C-2
- “Example of a WS-Policy File With Web Service Reliable Messaging Assertions” on page C-3
- “Element Description” on page C-3

Overview of a WS-Policy File That Contains Web Service Reliable Messaging Assertions

You use WS-Policy files to configure the reliable messaging capabilities of a WebLogic Web Service running on a destination endpoint. Use the @Policy JWS annotations in the JWS file that implements the Web Service to specify the name of the WS-Policy file that is associated with a Web Service.

A WS-Policy file is an XML file that conforms to the WS-Policy specification. The root element of a WS-Policy file is always <wsp:Policy>. To configure Web Service reliable messaging, you first add a <wsrm:RMAssertion> child element; its main purpose is to group all the reliable messaging policy assertions together. Then you add as child elements to <wsrm:RMAssertion>
the assertions that enable the type of Web Service reliable messaging you want. All these assertions conform to the WS-PolicyAssertions specification.

**WARNING:** You must enter the assertions in the ordered listed in the graphic below. See “Graphical Representation” on page C-2.

WebLogic Server includes two WS-Policy files (DefaultReliability.xml and LongRunningReliability.xml) that contain typical reliable messaging assertions that you can use if you do not want to create your own WS-Policy file. For details about these two files, see “Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 6-2.

See “Using Web Service Reliable Messaging” on page 6-1 for task-oriented information about creating a reliable WebLogic Web Service.

**Graphical Representation**

The following graphic describes the element hierarchy of Web Service reliable messaging policy assertions in a WS-Policy file.

**Figure 13-1 Element Hierarchy of Web Service Reliable Messaging Policy Assertions**
Example of a WS-Policy File With Web Service Reliable Messaging Assertions

The following example shows a simple WS-Policy file used to configure reliable messaging for a WebLogic Web Service:

```xml
<?xml version="1.0"?>
<wsp:Policy
    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
    xmlns:beapolicy="http://www.bea.com/wsrm/policy"
    >
    <wsrm:RMAssertion >
        <wsrm:InactivityTimeout
            Milliseconds="600000" />
        <wsrm:BaseRetransmissionInterval
            Milliseconds="3000" />
        <wsrm:ExponentialBackoff />
        <wsrm:AcknowledgementInterval
            Milliseconds="200" />
        <beapolicy:Expires Expires="P1D" optional="true"/>
    </wsrm:RMAssertion>
</wsp:Policy>
```

Element Description

**beapolicy:Expires**

Specifies an amount of time after which the reliable Web Service expires and does not accept any new sequences. Client applications invoking this instance of the reliable Web Service will receive an error if they try to invoke an operation after the expiration duration.

The default value of this element, if not specified in the WS-Policy file, is for the Web Service to never expires.
beapolicy:QOS

Specifies the delivery assurance (or Quality Of Service) of the Web Service:

- **AtMostOnce**—Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
- **AtLeastOnce**—Every message is delivered at least once. It is possible that some messages be delivered more than once.
- **ExactlyOnce**—Every message is delivered exactly once, without duplication.
- **InOrder**—Messages are delivered in the order that they were sent. This delivery assurance can be combined with the preceding three assurances.

The default value of this element, if not specified in the WS-Policy file, is `ExactlyOnce` `InOrder`. 

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expires</td>
<td>The amount of time after which the reliable Web Service expires. The format of this attribute conforms to the XML Schema duration data type. For example, to specify that the reliable Web Service expires after 3 hours, specify <code>Expires=&quot;P3H&quot;</code>.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**wsrm:AcknowledgementInterval**

Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand alone acknowledgement.

A destination endpoint can send an acknowledgement on the return message immediately after it has received a message from a source endpoint, or it can send one separately in a stand alone acknowledgement. In the case that a return message is not available to send an acknowledgement, a destination endpoint may wait for up to the acknowledgement interval before sending a stand alone acknowledgement. If there are no unacknowledged messages, the destination endpoint may choose not to send an acknowledgement.

This assertion does not alter the formulation of messages or acknowledgements as transmitted. Its purpose is to communicate the timing of acknowledgements so that the source endpoint may tune appropriately.

This element is optional. If you do not specify this element, the default value is set by the store and forward (SAF) agent configured for the destination endpoint.

---

**Table C-2 Attributes of `<beapolicy:QOS>`**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
</table>
| QOS       | Specifies the delivery assurance. You can specify exactly one of the following values:  
  - AtMostOnce  
  - AtLeastOnce  
  - ExactlyOnce  
  You can also add the `InOrder` string to specify that the messages be delivered in order.  
  If you specify one of the `XXXOnce` values, but do not specify `InOrder`, then the messages are *not* guaranteed to be in order. This is different from the default value if the entire QOS element is not specified (exactly once in order).  
  Example: `<beapolicy:QOS QOS="AtMostOnce InOrder"/>` | Yes |
wsrm:BaseRetransmissionInterval

Specifies the interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message.

If the source endpoint does not receive an acknowledgement for a given message within the interval specified by this element, the source endpoint retransmits the message. The source endpoint can modify this retransmission interval at any point during the lifetime of the sequence of messages. This assertion does not alter the formulation of messages as transmitted, only the timing of their transmission.

This element can be used in conjunction with the `<wsrm:ExponentialBackoff>` element to specify that the retransmission interval will be adjusted using the algorithm specified by the `<wsrm:ExponentialBackoff>` element.

This element is optional. If you do not specify this element, the default value is set by the store and forward (SAF) agent configured for the source endpoint. If using the Administration Console to configure the SAF agent, this value is labeled Retry Delay Base.

### Table C-3 Attributes of `<wsrm:AcknowledgementInterval>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milliseconds</td>
<td>Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand alone acknowledgement.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table C-4 Attributes of `<wsrm:BaseRetransmissionInterval>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milliseconds</td>
<td>Number of milliseconds the source endpoint waits to retransmit message.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

wsrm:ExponentialBackoff

Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm.
This element is used in conjunction with the `<wsrm:BaseRetransmissionInterval>` element. If a destination endpoint does not acknowledge a sequence of messages for the amount of time specified by `<wsrm:BaseRetransmissionInterval>`, the exponential backoff algorithm will be used for timing of successive retransmissions by the source endpoint, should the message continue to go unacknowledged.

The exponential backoff algorithm specifies that successive retransmission intervals should increase exponentially, based on the base retransmission interval. For example, if the base retransmission interval is 2 seconds, and the exponential backoff element is set in the WS-Policy file, successive retransmission intervals if messages continue to be unacknowledged are 2, 4, 8, 16, 32, and so on.

This element is optional. If not set, the same retransmission interval is used in successive retries, rather than the interval increasing exponentially.

This element has no attributes.

**wsrm:InactivityTimeout**

Specifies (in milliseconds) a period of inactivity for a sequence of messages. A sequence of messages is defined as a set of messages, identified by a unique sequence number, for which a particular delivery assurance applies; typically a sequence originates from a single source endpoint. If, during the duration specified by this element, a destination endpoint has received no messages from the source endpoint, the destination endpoint may consider the sequence to have been terminated due to inactivity. The same applies to the source endpoint.

This element is optional. If it is not set in the WS-Policy file, then sequences never time-out due to inactivity.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milliseconds</td>
<td>The number of milliseconds that defines a period of inactivity.</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

**wsrm:RMAssertion**

Main Web Service reliable messaging assertion that groups all the other assertions under a single element.
The presence of this assertion in a WS-Policy file indicates that the corresponding Web Service must be invoked reliably.

Table C-6 Attributes of `<wsrm:RMAssertion>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Specifies whether the Web Service requires the operations to be invoked reliably. Valid values for this attribute are <code>true</code> and <code>false</code>. Default value is <code>false</code>.</td>
<td>No.</td>
</tr>
</tbody>
</table>
Security Policy Assertion Reference

The following sections provide reference information about the security assertions you can configure in a WS-Policy file:

- “Overview of a WS-Policy File That Contains Security Assertions” on page D-1
- “Graphical Representation” on page D-2
- “Example of a Policy File With Security Elements” on page D-5
- “Element Description” on page D-6
- “Using MessageParts To Specify Parts of the SOAP Messages that Must Be Encrypted or Signed” on page D-21

Overview of a WS-Policy File That Contains Security Assertions

You use WS-Policy files to configure the message-level security of a WebLogic Web Service. Use the @Policy and @Policies JWS annotations in the JWS file that implements the Web Service to specify the name of the WS-Policy file that is associated with a WebLogic Web Service.

A WS-Policy file is an XML file that conforms to the WS-Policy specification. The root element of a WS-Policy file is always <wsp:Policy>. To configure message-level security, you add policy assertions that specify the type of tokens supported for authentication and how the SOAP messages should be encrypted and digitally signed.
Note: These security policy assertions are based on the assertions described in the December 18, 2002 version of the Web Services Security Policy Language (WS-SecurityPolicy) specification. This means that although the exact syntax and usage of the assertions in WebLogic Server are different, they are similar in meaning to those described in the specification. The assertions are not based on the latest update of the specification (13 July 2005.)

WebLogic Server includes five WS-Policy files (Auth.xml, Sign.xml, Encrypt.xml, Wssc-dk.xml, and Wssc-sct.xml) that contain typical security assertions that you can use if you do not want to create your own WS-Policy file. For details about these files, see “Using WS-Policy Files for Message-Level Security Configuration” on page 10-4.

See “Configuring Message-Level Security (Digital Signatures and Encryption)” on page 10-3 for task-oriented information about creating a message-level secured WebLogic Web Service.

Graphical Representation

The following graphic describes the element hierarchy of the security assertions in a WS-Policy file.
Figure 13-2  Element Hierarchy of Security WS-Policy Assertions
Security Policy Assertion Reference

Policy
  Identity
    SupportedTokens ?
      SecurityToken +
      Claims ?
        UsePassword ?
        ConfirmationMethod ?
    Target +
      DigestAlgorithm
      Transform *
      MessageParts

Integrity
  SignatureAlgorithm
  CanonicalizationAlgorithm
  SupportedTokens ?
    SecurityToken +
    Target +
      DigestAlgorithm
      Transform *
      MessageParts

Confidentiality
  KeyWrappingAlgorithm
  Target +
    EncryptionAlgorithm
    Transform *
    MessageParts
    KeyInfo
      SecurityToken *
      SecurityTokenReference *
  MessageAge
Example of a Policy File With Security Elements

<?xml version="1.0"?>
<wsp:Policy
 xmlns:wssp="http://www.bea.com/wls90/security/policy"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-wssecurity-utility-1.0.xsd"
 xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part"
>
  <wssp:Identity>
    <wssp:SupportedTokens>
      <wssp:SecurityToken
        TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID">
        <wssp:Claims>
          <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
        </wssp:Claims>
      </wssp:SecurityToken>
    </wssp:SupportedTokens>
  </wssp:Identity>

  <wssp:Confidentiality>
    <wssp:KeyWrappingAlgorithm
      URI="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>

    <wssp:Target>
      <wssp:EncryptionAlgorithm
        URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <wssp:MessageParts
        Dialect="http://www.bea.com/wls90/security/policy/wsee#part"
        wls:SecurityHeader(Assertion)
      />
    </wssp:Target>

    <wssp:Target>
      <wssp:EncryptionAlgorithm
        URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <wssp:MessageParts
        Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
        wsp:Body()"/
      />
    </wssp:Target>

    <wssp:KeyInfo />
  </wssp:Confidentiality>
</wsp:Policy>
Element Description

CanonicalizationAlgorithm

Specifies the algorithm used to canonicalize the SOAP message elements that are digitally signed.

**Note:** The WebLogic Web Services security runtime does not support specifying an InclusiveNamespaces PrefixList that contains a list of namespace prefixes or a token indicating the presence of the default namespace to the canonicalization algorithm.

Table D-1 Attributes of `<CanonicalizationAlgorithm>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>The algorithm used to canonicalize the SOAP message being signed. You can specify only the following canonicalization algorithm: <a href="http://www.w3.org/2001/10/xml-exc-c14n#">http://www.w3.org/2001/10/xml-exc-c14n#</a></td>
<td>Yes.</td>
</tr>
</tbody>
</table>

**Claims**

Specifies additional metadata information that is associated with a particular type of security token. Depending on the type of security token, you can or must specify the following child elements:

- For username tokens, you can define a `<UsePassword>` child element to specify whether you want the SOAP messages to use password digests.

- For SAML tokens, you must define a `<ConfirmationMethod>` child element to specify the type of SAML confirmation (sender-vouches or holder-of-key).

By default, a security token for a secure conversation has a lifetime of 12 hours. To change this default value, define a `<TokenLifeTime>` child element to specify a new lifetime, in milliseconds, of the security token.

This element does not have any attributes.
Confidentiality

Specifies that part or all of the SOAP message must be encrypted, as well as the algorithms and keys that are used to encrypt the SOAP message.

For example, a Web Service may require that the entire body of the SOAP message must be encrypted using triple-DES.

Confidentiality

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SupportTrust10</td>
<td>.</td>
<td>No.</td>
</tr>
</tbody>
</table>

The valid values for this attribute are true and false. The default value is false.

ConfirmationMethod

Specifies the type of confirmation method that is used when using SAML tokens for identity. You must specify one of the following two values for this element: sender-vouches or holder-of-key. For example:

```xml
<wssp:Claims>
  <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
</wssp:Claims>
```

This element does not have any attributes.

The `<ConfirmationMethod>` element is required only if you are using SAML tokens.

The exact location of the `<ConfirmationMethod>` assertion in the WS-Policy file depends on the type configuration method you are configuring. In particular:

**sender-vouches:**

Specify the `<ConfirmationMethod>` assertion within an `<Identity>` assertion, as shown in the following example:

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

<wsp:Policy
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
```
holder-of-key:

Specify the `<ConfirmationMethod>` assertion within an `<Integrity>` assertion. The reason you put the SAML token in the `<Integrity>` assertion for this confirmation method is that the Web Service runtime must prove the integrity of the message, which is not required by `sender-vouches`.

For example:

```xml
<?xml version="1.0"?>
<wsp:Policy
 xmlns:wssp="http://www.bea.com/wls90/security/policy"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:wls="http://www.bea.com/wls90/security/policy/wsee#part">
  <wssp:SignatureAlgorithm
    URI="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
  <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
  <wssp:ConfirmationMethod>sender-vouches</wssp:ConfirmationMethod>
</wsp:Policy>
```
<wssp:CanonicalizationAlgorithm
    URI="http://www.w3.org/2001/10/xml-exc-c14n#" />
<wssp:Target>
    <wssp:DigestAlgorithm
        URI="http://www.w3.org/2000/09/xmldsig#sha1" />
    <wssp:MessageParts
        Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">wsp:Body()
    </wssp:MessageParts>
</wssp:Target>
<wssp:SupportedTokens>
    <wssp:SecurityToken
        IncludeInMessage="true"
        TokenType="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID">
        <wssp:Claims>
            <wssp:ConfirmationMethod>holder-of-key</wssp:ConfirmationMethod>
        </wssp:Claims>
    </wssp:SecurityToken>
</wssp:SupportedTokens>
</wssp:Integrity>
</wsp:Policy>

For more information about the two SAML confirmation methods (sender-vouches or holder-of-key), see SAML Token Profile Support in WebLogic Web Services.

**DigestAlgorithm**

Specifies the digest algorithm that is used when digitally signing the specified parts of a SOAP message. Use the `<MessageParts>` sibling element to specify the parts of the SOAP message you want to digitally sign.
Table D-3  Attributes of <DigestAlgorithm>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>The digest algorithm that is used when digitally signing the specified parts of a SOAP message. You can specify only the following digest algorithm: <a href="http://www.w3.org/2000/09/xmldsig#sha1">http://www.w3.org/2000/09/xmldsig#sha1</a></td>
<td>Yes</td>
</tr>
</tbody>
</table>

EncryptionAlgorithm

Specifies the encryption algorithm that is used when encrypting the specified parts of a SOAP message. Use the <MessageParts> sibling element to specify the parts of the SOAP message you want to digitally sign.

Table D-4  Attributes of <EncryptionAlgorithm>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>The encryption algorithm used to encrypt specified parts of the SOAP message. Valid values are: <a href="http://www.w3.org/2001/04/xmlenc#tripledes-cbc">http://www.w3.org/2001/04/xmlenc#tripledes-cbc</a> <a href="http://www.w3.org/2001/04/xmlenc#kw-tripledes">http://www.w3.org/2001/04/xmlenc#kw-tripledes</a> <a href="http://www.w3.org/2001/04/xmlenc#aes128-cbc">http://www.w3.org/2001/04/xmlenc#aes128-cbc</a></td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note: When interoperating with Web Services built with WebLogic Workshop 8.1, you must specify http://www.w3.org/2001/04/xmlenc#aes128-cbc as the encryption algorithm.

Identity

Specifies the type of security tokens (username, X.509, or SAML) that are supported for authentication.

This element has no attributes.
**Integrity**

Specifies that part or all of the SOAP message must be digitally signed, as well as the algorithms and keys that are used to sign the SOAP message.

For example, a Web Service may require that the entire body of the SOAP message must be digitally signed and only algorithms using SHA1 and an RSA key are accepted.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignToken</td>
<td>Specifies whether the security token, specified using the &lt;SecurityToken&gt; child element of &lt;Integrity&gt;, should also be digitally signed, in addition to the specified parts of the SOAP message. The valid values for this attribute are true and false. The default value is true.</td>
<td>No.</td>
</tr>
<tr>
<td>SupportTrust10</td>
<td></td>
<td>No.</td>
</tr>
<tr>
<td>X509AuthCond</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**KeyInfo**

Used to specify the security tokens that are used for encryption.

This element has no attributes.

**KeyWrappingAlgorithm**

Specifies the algorithm used to encrypt the message encryption key.
Table D-6  Attributes of <KeyWrappingAlgorithm>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
</table>
| URI       | The algorithm used to encrypt the SOAP message encryption key. Valid values are:  
- http://www.w3.org/2001/04/xmlenc#rsa-1_5  
  (to specify the RSA-v1.5 algorithm)  
- http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p  
  (to specify the RSA-OAEP algorithm) | Yes. |

Label

Specifies a label for the security context token. Used when configuring WS-SecureConversation security contexts.

This element has no attributes.

Length

Specifies the length of the key when using security context tokens and derived key tokens. This assertion only applies to WS-SecureConversation security contexts.

The default value is 32.

This element has no attributes.

MessageAge

Specifies the acceptable time period before SOAP messages are declared stale and discarded.

When you include this security assertion in your WS-Policy file, the Web Services runtime adds a <Timestamp> header to the request or response SOAP message, depending on the direction (inbound, outbound, or both) to which the WS-Policy file is associated. The <Timestamp> header indicates to the recipient of the SOAP message when the message expires.

For example, assume that your WS-Policy file includes the following <MessageAge> assertion:

```xml
```
The resulting generated SOAP message will have a `<Timestamp>` header similar to the following excerpt:

```xml
<wsu:Timestamp
    wsu:Id="Dy2PFsX32QacqNKEANpXbNMnMhm2BmG0A2Wdc2E0JpiaaTmbYNwT"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <wsu:Created>2005-11-09T17:46:55Z</wsu:Created>
</wsu:Timestamp>
```

In the example, the recipient of the SOAP message discards the message if received after 2005-11-09T17:51:55Z, or five minutes after the message was created.

The Web Services runtime, when generating the SOAP message, sets the `<Created>` header to the time when the SOAP message was created and the `<Expires>` header to the creation time plus the value of the `Age` attribute of the `<MessageAge>` assertion.

The following table describes the attributes of the `<MessageAge>` assertion.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Specifies the actual maximum age time-out for a SOAP message, in seconds.</td>
<td>No.</td>
</tr>
</tbody>
</table>
The following table lists the properties that describe the timestamp behavior of the WebLogic Web Services security runtime, along with their default values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Synchronized</td>
<td>Specifies whether the Web Service assumes synchronized clocks.</td>
<td>true</td>
</tr>
<tr>
<td>Clock Precision</td>
<td>If clocks are synchronized, describes the accuracy of the synchronization.</td>
<td>60000</td>
</tr>
<tr>
<td></td>
<td>Note: This property is deprecated as of release 9.2 of WebLogic Web Services. Use the Clock Skew property instead. If both properties are set, then Clock Skew takes precedence.</td>
<td></td>
</tr>
<tr>
<td>Clock Skew</td>
<td>Specifies the allowable difference, in milliseconds, between the sender and receiver of the message.</td>
<td>60000</td>
</tr>
<tr>
<td>Lax Precision</td>
<td>Allows you to relax the enforcement of the clock precision property.</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>Note: This property is deprecated as of release 9.2 of WebLogic Web Services. Use the Clock Skew property instead.</td>
<td></td>
</tr>
<tr>
<td>Max Processing Delay</td>
<td>Specifies the freshness policy for received messages.</td>
<td>-1</td>
</tr>
<tr>
<td>Validity Period</td>
<td>Represents the length of time the sender wants the outbound message to be valid.</td>
<td>60 seconds</td>
</tr>
</tbody>
</table>

You typically never need to change the values of the preceding timestamp properties. However, if you do need to, you must use the Administration Console to create the `default_wss` Web Service Security Configuration, if it does not already exist, and then update its timestamp configuration by clicking on the **Timestamp** tab. See [Create a Web Service security configuration](#) for task information and **Domains: Web Services Security: Timestamp** for additional reference information about these timestamp properties.
MessageParts

Specifies the parts of the SOAP message that should be signed or encrypted, depending on the grand-parent of the element. You can use either an XPath 1.0 expression or a set of pre-defined functions within this assertion to specify the parts of the SOAP message.

The `MessageParts` assertion is always a child of a `Target` assertion. The `Target` assertion can be a child of either an `Integrity` assertion (to specify how the SOAP message is digitally signed) or a `Confidentiality` assertion (to specify how the SOAP messages are encrypted.)

See “Using MessageParts To Specify Parts of the SOAP Messages that Must Be Encrypted or Signed” on page D-21 for detailed information about using this assertion, along with a variety of examples.

### Table D-8 Attributes of `<MessageParts>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialect</td>
<td>Identifies the dialect used to identity the parts of the SOAP message that should be signed or encrypted. If this attribute is not specified, then XPath 1.0 is assumed.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The value of this attribute must be one of the following:

- `http://www.w3.org/TR/1999/REC-xpath-19991116`: Specifies that an XPath 1.0 expression should be used against the SOAP message to specify the part to be signed or encrypted.
- `http://schemas.xmlsoap.org/2002/12/wsse#part`: Convenience dialect used to specify that the entire SOAP body should be signed or encrypted.
- `http://www.bea.com/wls90/security/policy/wsee#part`: Convenience dialect to specify that the WebLogic-specific headers should be signed or encrypted. You can also use this dialect to use QNames to specify the parts of the security header that should be signed or encrypted.

See “Using MessageParts To Specify Parts of the SOAP Messages that Must Be Encrypted or Signed” on page D-21 for examples of using these dialects.
Security Token

Specifies the security token that is supported for authentication, encryption or digital signatures, depending on the parent element.

For example, if this element is defined in the <Identity> parent element, then it specifies that a client application, when invoking the Web Service, must attach a security token to the SOAP request. For example, a Web Service might require that the client application present a SAML authorization token issued by a trusted authorization authority for the Web Service to be able to access sensitive data. If this element is part of <Confidentiality>, then it specifies the token used for encryption.

The specific type of the security token is determined by the value of its TokenType attribute, as well as its parent element.

By default, a security token for a secure conversation has a lifetime of 12 hours. To change this default value, add a <Claims> child element that itself has a <TokenLifeTime> child element, as described in “Claims” on page D-6.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivedFromTokenType</td>
<td>.</td>
<td>No.</td>
</tr>
</tbody>
</table>
SecurityTokenReference

For internal use only.

You should never include this security assertion in your custom WS-Policy file; it is described in this section for informational purposes only. The WebLogic Web Services runtime automatically inserts this security assertion in the WS-Policy file that is published in the dynamic WSDL of the deployed Web Service. The security assertion specifies WebLogic Server’s public key; the client application that invokes the Web Service then uses it to encrypt the parts of the SOAP message specified by the WS-Policy file. The Web Services runtime then uses the server’s private key to decrypt the message.

SignatureAlgorithm

Specifies the cryptographic algorithm used to compute the digital signature.

Table D-9 Attributes of `<SecurityToken>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IncludeInMessage</td>
<td>Specifies whether to include the token in the SOAP message. Valid values are true or false.</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>The default value of this attribute is false when used in the <code>&lt;Confidentiality&gt;</code> assertion and true when used in the <code>&lt;Integrity&gt;</code> assertion.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The value of this attribute is always true when used in the <code>&lt;Identity&gt;</code> assertion, even if you explicitly set it to false.</td>
<td></td>
</tr>
<tr>
<td>TokenType</td>
<td>Specifies the type of security token. Valid values:</td>
<td>Yes.</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3</a> (To specify a binary X.509 token)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken</a> (To specify a username token)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <a href="http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID">http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID</a> (To specify a SAML token)</td>
<td></td>
</tr>
</tbody>
</table>
SupportedTokens

Specifies the list of supported security tokens that can be used for authentication, encryption, or digital signatures, depending on the parent element.

This element has no attributes.

Target

Encapsulates information about which targets of a SOAP message are to be encrypted or signed, depending on the parent element.

The child elements also depend on the parent element; for example, when used in `<Integrity>`, you can specify the `<DigestAlgorithm>`, `<Transform>`, and `<MessageParts>` child elements. When used in `<Confidentiality>`, you can specify the `<EncryptionAlgorithm>`, `<Transform>`, and `<MessageParts>` child elements.

You can have one or more targets.

---

Table D-10 Attributes of `<SignatureAlgorithm>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>Specifies the cryptographic algorithm used to compute the signature.</td>
<td>Yes.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Be sure that you specify an algorithm that is compatible with the certificates you are using in your enterprise.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Valid values are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.w3.org/2000/09/xmldsig#rsa-sha1">http://www.w3.org/2000/09/xmldsig#rsa-sha1</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.w3.org/2000/09/xmldsig#dsa-sha1">http://www.w3.org/2000/09/xmldsig#dsa-sha1</a></td>
<td></td>
</tr>
</tbody>
</table>
TokenLifeTime

Specifies the lifetime, in seconds, of the security context token or derived key token. This element is used only when configuring WS-SecurityConversation security contexts.

The default lifetime of a security token is 12 hours (43,200 seconds).

This element has no attributes.

Transform

Specifies the URI of a transformation algorithm that is applied to the parts of the SOAP message that are signed or encrypted, depending on the parent element.

You can specify zero or more transforms, which are executed in the order they appear in the <Target> parent element.

---

Table D-11 Attributes of <Target>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>encryptContent</td>
<td>Specifies whether to encrypt an entire element, or just its content.</td>
<td>No.</td>
</tr>
<tr>
<td></td>
<td>This attribute can be specified only when &lt;Target&gt; is a child element of &lt;Confidentiality&gt;.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default value of this attribute is true, which means that only the content is encrypted.</td>
<td></td>
</tr>
</tbody>
</table>
UsePassword

Specifies that whether the plaintext or the digest of the password appear in the SOAP messages. This element is used only with username tokens.

Table D-12 Attributes of <Transform>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI</td>
<td>Specifies the URI of the transformation algorithm. Valid URIs are:</td>
<td>Yes.</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.w3.org/2000/09/xmldsig#base64">http://www.w3.org/2000/09/xmldsig#base64</a> (Base64 decoding transforms)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.w3.org/TR/1999/REC-xpath-19991116">http://www.w3.org/TR/1999/REC-xpath-19991116</a> (XPath filtering)</td>
<td></td>
</tr>
</tbody>
</table>

For detailed information about these transform algorithms, see XML-Signature Syntax and Processing.
Using MessageParts To Specify Parts of the SOAP Messages that Must Be Encrypted or Signed

When you use either the Integrity or Confidentiality assertion in your WS-Policy file, you are required to also use the Target child assertion to specify the targets of the SOAP message to digitally sign or encrypt. The Target assertion in turn requires that you use the MessageParts child assertion to specify the actual parts of the SOAP message that should be digitally signed or encrypted. This section describes various ways to use the MessageParts assertion.

See “Example of a Policy File With Security Elements” on page D-5 for an example of a complete WS-Policy file that uses the MessageParts assertion within a Confidentiality assertion. The example shows how to specify that the entire body, as well as the Assertion security header, of the SOAP messages should be encrypted.

You use the Dialect attribute of MessageParts to specify the dialect used to identify the SOAP message parts. The WebLogic Web Services security runtime supports the following three dialects:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Specifies the type of password. Valid values are:</td>
<td>Yes.</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText</a>: Specifies that cleartext passwords should be used in the SOAP messages.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest</a>: Specifies that password digests should be used in the SOAP messages.</td>
<td></td>
</tr>
</tbody>
</table>

Note: For backward compatibility reasons, the two preceding URIs can also be specified with an initial "www." For example:

- http://www.docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText
- http://www.docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest

Using MessageParts To Specify Parts of the SOAP Messages that Must Be Encrypted or Signed

When you use either the Integrity or Confidentiality assertion in your WS-Policy file, you are required to also use the Target child assertion to specify the targets of the SOAP message to digitally sign or encrypt. The Target assertion in turn requires that you use the MessageParts child assertion to specify the actual parts of the SOAP message that should be digitally signed or encrypted. This section describes various ways to use the MessageParts assertion.

See “Example of a Policy File With Security Elements” on page D-5 for an example of a complete WS-Policy file that uses the MessageParts assertion within a Confidentiality assertion. The example shows how to specify that the entire body, as well as the Assertion security header, of the SOAP messages should be encrypted.

You use the Dialect attribute of MessageParts to specify the dialect used to identify the SOAP message parts. The WebLogic Web Services security runtime supports the following three dialects:
**XPath 1.0**

This dialect enables you to use an XPath 1.0 expression to specify the part of the SOAP message that should be signed or encrypted. The value of the `Dialect` attribute to enable this dialect is `http://www.w3.org/TR/1999/REC-xpath-19991116`.

You typically want to specify that the parts of a SOAP message that should be encrypted or digitally signed are child elements of either the `soap:Body` or `soap:Header` elements. For this reason, BEA provides the following two functions that take as parameters an XPath expression:

- `wsp:GetBody(xpath_expression)`—Specifies that the root element from which the XPath expression starts searching is `soap:Body`.
- `wsp:GetHeader(xpath_expression)`—Specifies that the root element from which the XPath expression starts searching is `soap:Header`.

You can also use a plain XPath expression as the content of the `MessageParts` assertion, without one of the preceding functions. In this case, the root element from which the XPath expression starts searching is `soap:Envelope`.

The following example specifies that the `AddInt` part, with namespace prefix `n1` and located in the SOAP message body, should be signed or encrypted, depending on whether the parent `Target` parent is a child of `Integrity` or `Confidentiality` assertion:

```xml
<wssp:MessageParts
  Dialect="http://www.w3.org/TR/1999/REC-xpath-19991116"
  xmlns:n1="http://www.bea.com/foo">
```

**Pre-Defined wsp:Body() Function**

**WebLogic-Specific Header Functions**

Be sure that you specify a message part that actually exists in the SOAP messages that result from a client invoke of a message-secured Web Service. If the Web Services security runtime encounters an inbound SOAP message that does not include a part that the WS-Policy file indicates should be signed or encrypted, then the Web Services security runtime returns an error and the invoke fails. The only exception is if you use the WebLogic-specific `wls:SystemHeader()` function to specify that any WebLogic-specific SOAP header in a SOAP message should be signed or encrypted; if the Web Services security runtime does not find any of these headers in the SOAP message, the runtime simply continues with the invoke and does not return an error.
Using MessageParts To Specify Parts of the SOAP Messages that Must Be Encrypted or Signed

```
wsp:GetBody(./n1:AddInt)
</wssp:MessageParts>
```

The preceding example shows that you should define the namespace of a part specified in the XPath expression (n1 in the example) as an attribute to the MessageParts assertion, if you have not already defined the namespace elsewhere in the WS-Policy file.

The following example is similar, except that the part that will be signed or encrypted is `wsu:Timestamp`, which is a child element of `wsee:Security` and is located in the SOAP message header:

```
<wssp:MessageParts
    Dialect="http://www.w3.org/TR/1999/REC-xpath-19991116">
  wsp:GetHeader(./wsse:Security/wsu:Timestamp)
</wssp:MessageParts>
```

In the preceding example, it is assumed that the `wsee:` and `wse:` namespaces have been defined elsewhere in the WS-Policy file.

**Note:** It is beyond the scope of this document to describe how to create XPath expressions. For detailed information, see the XML Path Language (XPath), Version 1.0, specification.

**Pre-Defined wsp:Body() Function**

The XPath dialect described in “XPath 1.0” on page D-22 is flexible enough for you to pinpoint any part of the SOAP message that should be encrypted or signed. However, sometimes you might just want to specify that the entire SOAP message body be signed or encrypted. In this case using an XPath expression is unduly complicated, so BEA recommends you use the dialect that pre-defines the `wsp:Body()` function for just this purpose, as shown in the following example:

```
<wssp:MessageParts
    Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
  wsp:Body()
</wssp:MessageParts>
```

**WebLogic-Specific Header Functions**

BEA provides its own dialect that pre-defines a set of functions to easily specify that some or all of the WebLogic security or system headers should be signed or encrypted. Although you can achieve the same goal using the XPath dialect, it is much simpler to use this WebLogic dialect. You enable this dialect by setting the Dialect attribute to

```
```
The `wls:SystemHeaders()` function specifies that all of the WebLogic-specific headers should be signed or encrypted. These headers are used internally by the WebLogic Web Services runtime for various features, such as reliable messaging and addressing. The headers are:

- `wsrm:SequenceAcknowledgement`
- `wsrm:AckRequested`
- `wsrm:Sequence`
- `wsa:Action`
- `wsa:FaultTo`
- `wsa:From`
- `wsa:MessageID`
- `wsa:RelatesTo`
- `wsa:ReplyTo`
- `wsa:To`
- `wsax:SetCookie`

The following example shows how to use the `wls:SystemHeader()` function:

```xml
<wssp:MessageParts
    Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
    wls:SystemHeaders()
</wssp:MessageParts>
```

Use the `wls:SecurityHeader(header)` function to specify a particular part in the security header that should be signed or encrypted, as shown in the following example:

```xml
<wssp:MessageParts
    Dialect="http://www.bea.com/wls90/security/policy/wsee#part">
    wls:SecurityHeader(wsa:From)
</wssp:MessageParts>
```

In the example, only the `wsa:From` security header is signed or encrypted. You can specify any of the preceding list of headers to the `wls:SecurityHeader()` function.
WebLogic Web Service Deployment Descriptor Element Reference

The following sections provide information about the WebLogic-specific Web Services deployment descriptor file, `weblogic-webservices.xml`:

- “Overview of weblogic-webservices.xml” on page E-1
- “Graphical Representation” on page E-2
- “XML Schema” on page E-4
- “Example of a weblogic-webservices.xml Deployment Descriptor File” on page E-4
- “Element Description” on page E-4

**Overview of weblogic-webservices.xml**

The standard J2EE deployment descriptor for Web Services is called `webservice.xml`. This file specifies the set of Web Services that are to be deployed to WebLogic Server and the dependencies they have on container resources and other services. See the Web Services XML Schema for a full description of this file.

The WebLogic equivalent to the standard J2EE `webservice.xml` deployment descriptor file is called `weblogic-webservices.xml`. This file contains WebLogic-specific information about a WebLogic Web Service, such as the URL used to invoke the deployed Web Service, and so on.

Both deployment descriptor files are located in the same location on the J2EE archive that contains the Web Service. In particular:
For Java class-implemented Web Services, the Web Service is packaged as a Web application WAR file and the deployment descriptors are located in the WEB-INF directory.

For stateless session EJB-implemented Web Services, the Web Service is packaged as an EJB JAR file and the deployment descriptors are located in the META-INF directory.

The structure of the `weblogic-webservices.xml` file is similar to the structure of the J2EE `webservices.xml` file in how it lists and identifies the Web Services that are contained within the archive. For example, for each Web Service in the archive, both files have a `<webservice-description>` child element of the appropriate root element (`<webservices>` for the J2EE `webservices.xml` file and `<weblogic-webservices>` for the `weblogic-webservices.xml` file).

Typically users never need to update either deployment descriptor files, because the jwsc Ant task automatically generates the files for you based on the value of the JWS annotations in the JWS file that implements the Web Service. For this reason, this section is published for informational purposes only.

The data type definitions of two elements in the `weblogic-webservices.xml` file (login-config and transport-guarantee) are imported from the J2EE Schema for the `web.xml` file. See the Servlet Deployment Descriptor Schema for details about these elements and data types.

**Graphical Representation**

The following graphic describes the element hierarchy of the `weblogic-webservices.xml` deployment descriptor file.
Figure 13-3  Element Hierarchy of weblogic-webservices.xml

weblogic-webservices
  + webservice-description
    - webservice-description-name
    - wsdl-publish-file
    + port-component
      - port-component-name
      - service-endpoint-address
        - webservice-contextpath
        - webservice-serviceuri
      ? deployment-listener-list
        + deployment-listener
      ? wsdl
        - exposed
      ? login-config
      ? transport-guarantee
  ? webservice-security
    - mbean-name

No annotation: Exactly one
* : Zero or more
+ : One or more
? : Zero or one
XML Schema

For the XML Schema file that describes the `weblogic-webservices.xml` deployment descriptor, see http://www.bea.com/ns/weblogic/90/weblogic-wsee.xsd.

Example of a `weblogic-webservices.xml` Deployment Descriptor File

The following example shows a simple `weblogic-webservices.xml` deployment descriptor:

```xml
<?xml version='1.0' encoding='UTF-8'?>
<weblogic-webservices xmlns="http://www.bea.com/ns/weblogic/90">
  <webservice-description>
    <webservice-description-name>MyService</webservice-description-name>
    <port-component>
      <port-component-name>MyServiceServicePort</port-component-name>
      <service-endpoint-address>
        <webservice-contextpath>/MyService</webservice-contextpath>
        <webservice-serviceuri>/MyService</webservice-serviceuri>
      </service-endpoint-address>
    </port-component>
  </webservice-description>
</weblogic-webservices>
```

Element Description

**deployment-listener-list**

For internal use only.

**deployment-listener**

For internal use only.

**exposed**

Boolean attribute indicating whether the WSDL should be exposed to the public when the Web Service is deployed.
login-config

The j2ee:login-config element specifies the authentication method that should be used, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism.

The XML Schema data type of the j2ee:login-config element is j2ee:login-configType, and is defined in the J2EE Schema that describes the standard web.xml deployment descriptor. For the full reference information, see [http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd](http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd).

mbean-name

Specifies the name of the Web Service security configuration (specifically an instantiation of the WebserviceSecurityMBean) that is associated with the Web Services described in the deployment descriptor file. The default configuration is called default_wss.

The associated 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 digital signatures, and so on.

You must create the security configuration (even the default one) using the Administration Console before you can successfully invoke the Web Service.

Note: The Web Service security configuration described by this element applies to all Web Services contained in the weblogic-webservices.xml file. The jwsc Ant task always packages a Web Service in its own JAR or WAR file, so this limitation is not an issue if you always use the jwsc Ant task to generate a Web Service. However, if you update the weblogic-webservices.xml deployment descriptor manually and add additional Web Service descriptions, you cannot associate different security configurations to different services.

port-component

The <port-component> element is a holder of other elements used to describe a Web Service port.

The child elements of the <port-component> element specify WebLogic-specific characteristics of the Web Service port, such as the context path and service URI used to invoke the Web Service after it has been deployed to WebLogic Server.
**port-component-name**

The `<port-component-name>` child element of the `<port-component>` element specifies the internal name of the WSDL port.

The value of this element must be unique for all `<port-component-name>` elements within a single `weblogic-webservices.xml` file.

**service-endpoint-address**

The `<service-endpoint-address>` element groups the WebLogic-specific context path and service URI values that together make up the Web Service endpoint address, or the URL that invokes the Web Service after it has been deployed to WebLogic Server.

These values are specified with the `<webservice-contextpath>` and `<webservice-serviceuri>` child elements.

**Note:** During a Web Service deployment, modifying the values of `<webservice-contextpath>` and `<webservice-serviceuri>` elements in `weblogic-webservices.xml` file does not change `<service-endpoint-address>`. At deployment time if you need to change the context path and service URI, set values for the following elements:

- `<context-root>` in `weblogic-application.xml` file to change context path. See `weblogic-application.xml` Deployment Descriptor Elements.
- `<url-pattern>` in `web.xml` file to change service URI. See `web.xml` Deployment Descriptor Elements.

**transport-guarantee**

The `j2ee:transport-guarantee` element specifies the type of communication between the client application invoking the Web Service and WebLogic server.

The value of this element is either NONE, INTEGRAL, or CONFIDENTIAL. NONE means that the application does not require any transport guarantees. A value of INTEGRAL means that the application requires that the data sent between the client and server be sent in such a way that it cannot be changed in transit. CONFIDENTIAL means that the application requires that the data be transmitted in a way that prevents other entities from observing the contents of the transmission. In most cases, the presence of the INTEGRAL or CONFIDENTIAL flag indicates that the use of SSL is required.
The XML Schema data type of the j2ee:transport-guarantee element is j2ee:transport-guaranteeType, and is defined in the J2EE Schema that describes the standard web.xml deployment descriptor. For the full reference information, see http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd.

**weblogic-webservices**

The `<weblogic-webservices>` element is the root element of the WebLogic-specific Web Services deployment descriptor (weblogic-webservices.xml). The element specifies the set of Web Services contained in the J2EE component archive in which the deployment descriptor is also contained. The archive is either an EJB JAR file (for stateless session EJB-implemented Web Services) or a WAR file (for Java class-implemented Web Services).

**webservice-contextpath**

The `<webservice-contextpath>` element specifies the context path portion of the URL used to invoke the Web Service.

The URL to invoke a Web Service deployed to WebLogic Server is:

http://host:port/contextPath/serviceURI

where

- *host* is the host computer on which WebLogic Server is running.
- *port* is the port address to which WebLogic Server is listening.
- *contextPath* is the value of this element
- *serviceURI* is the value of the webservice-serviceuri element.

When using the jwsc Ant task to generate a Web Service from a JWS file, the value of the `<webservice-contextpath>` element is taken from the `contextPath` attribute of the WebLogic-specific `@WLHttpTransport` annotation or the `<WLHttpTransport>` child element of jwsc.

**webservice-description**

The `<webservice-description>` element is a holder of other elements used to describe a Web Service.
The `<webservice-description>` element defines a set of port components (specified using one or more `<port-component>` child elements) that are associated with the WSDL ports defined in the WSDL document.

There may be multiple `<webservice-description>` elements defined within a single `weblogic-webservices.xml` file, each corresponding to a particular stateless session EJB or Java class contained within the archive, depending on the implementation of your Web Service. In other words, an EJB JAR contains the EJBs that implement a Web Service, a WAR file contains the Java classes.

**webservice-description-name**

The `<webservice-description-name>` element specifies the internal name of the Web Service.

The value of this element must be unique for all `<webservice-description-name>` elements within a single `weblogic-webservices.xml` file.

**webservice-security**

Element used to group together all the security-related elements of the `weblogic-webservices.xml` deployment descriptor.

**webservice-serviceuri**

The `<webservice-serviceuri>` element specifies the Web Service URI portion of the URL used to invoke the Web Service.

The URL to invoke a Web Service deployed to WebLogic Server is:

```
http://host:port/contextPath/serviceURI
```

where

- *host* is the host computer on which WebLogic Server is running.
- *port* is the port address to which WebLogic Server is listening.
- *contextPath* is the value of the `<webservice-contextpath>` element
- *serviceURI* is the value of this element.

When using the `jwsc` Ant task to generate a Web Service from a JWS file, the value of the `<webservice-serviceuri>` element is taken from the `serviceURI` attribute of the
WebLogic-specific @WLHttpTransport annotation or the `<WLHttpTransport>` child element of `<jwsc>`.

### wsdl

Element used to group together all the WSDL-related elements of the `weblogic-webservices.xml` deployment descriptor.

### wsdl-publish-file

The `<wsdl-publish-file>` element specifies a directory (on the computer which hosts the Web Service) to which WebLogic Server should publish a hard-copy of the WSDL file of a deployed Web Service; this is in addition to the standard WSDL file accessible via HTTP.

For example, assume that your Web Service is implemented with an EJB, and its WSDL file is located in the following directory of the EJB JAR file, relative to the root of the JAR:

```
META-INF/wsdl/a/b/Fool.wsdl
```

Further assume that the `weblogic-webservices.xml` file includes the following element for a given Web Service:

```
<wsdl-publish-file>d:/bar</wsdl-publish-file>
```

This means that when WebLogic Server deploys the Web Service, the server publishes the WSDL file at the standard HTTP location, but also puts a copy of the WSDL file in the following directory of the computer on which the service is running:

```
d:/bar/a/b/Foo.wsdl
```

**WARNING:** Only specify this element if client applications that invoke the Web Service need to access the WSDL via the local file system or FTP; typically, client applications access the WSDL using HTTP, as described in “Browsing to the WSDL of the Web Service” on page 4-17.

The value of this element should be an absolute directory pathname. This directory must exist on **every** machine which hosts a WebLogic Server instance or cluster to which you deploy the Web Service.