BEA WebLogic Server®

WebLogic Web Services: Reference

Version 10.0
Revised: April 28, 2008

Contents

E. WebLogic Web Service Deployment Descriptor Element Reference

CHAPTER 1

Introduction and Roadmap

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

Document Scope and Audience

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

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

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

It is assumed that the reader is familiar with Java Platform, Enterprise Edition (Java EE) Version 5 and Web Services concepts, the Java programming language, and Web technologies. This document emphasizes the value-added features provided by WebLogic Web Services and key information about how to use WebLogic Server features and facilities to get a WebLogic Web Service application up and running.

WebLogic Web Services Documentation Set

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

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

Guide to This Document

This document is organized as follows:

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide and the features of WebLogic Web Services.

- Appendix A, “Ant Task Reference,” provides reference documentation about the WebLogic Web Services Ant tasks.
- Appendix B, “JWS Annotation Reference,”provides reference information about the WebLogic-specific JWS annotations that you can use in the JWS file that implements your Web Service.
- Appendix C, “Web Service Reliable Messaging Policy Assertion Reference,” provides reference information about the policy assertions you can add to a WS-Policy file to configure the Web Service reliable messaging feature of a WebLogic Web Service.
- Appendix D, “BEA Web Services Security Policy Assertion Reference,” provides reference information about the policy assertions you can add to a WS-Policy file to configure the message-level (digital signatures and encryption) security of a WebLogic Web Service, using a proprietary BEA security policy schema. Note that you may prefer to use files that conform to the OASIS WS-SecurityPolicy specification, as described in Configuring Message-Level Security.
- Appendix E, “WebLogic Web Service Deployment Descriptor Element Reference,” provides reference information about the elements in the WebLogic-specific Web Services deployment descriptor `weblogic-webservices.xml`.

Related Documentation

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

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server 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.

For a full description and location of the available code samples, see Samples for the Web Services Developer in the *WebLogic Web Services: Getting Started* document.

Release-Specific WebLogic Web Services Information

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

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

Summary of WebLogic Web Services Features

For a full list of WebLogic Web Services features, including advanced features, see Summary of WebLogic Web Services Features in the *WebLogic Web Services: Getting Started* document.

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-19
- "wsdlc" on page A-59

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
- Iterative Development of WebLogic Web Services Starting From a WSDL File: Main Steps
- 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 A-1 WebLogic Web Services Ant Tasks

Ant Task	Description
clientgen	Generates the `Service` stubs and other client-side artifacts used to invoke a Web Service. Can be used against both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.
jwsc	Compiles a *JWS*-annotated file into a Web Service. JWS refers to Java Web Service. Can generate both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.
wsdlc	Generates a partial Web Service implementation based on a WSDL file.

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`:

   ```
   <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:

   ```
     <taskdef name="jwsc"
         classname="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:

```
<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:

```
<jwsc ...>
   <classpath>
       <pathelement path="${java.class.path}" />
       <pathelement path="my_fab_directory" />
   </classpath>
...
</jwsc>
```

The following example shows how you can build your CLASSPATH variable outside of the WebLogic Web Service Ant task declarations, then specify the variable from within the task using the `<classpath>` element:

```
<path id="myClassID">
   <pathelement path="${java.class.path}"/>
   <pathelement path="${additional.path1}"/>
   <pathelement path="${additional.path2}"/>
</path>

<jwsc ....>
   <classpath refid="myClassID" />
...
</jwsc>
```

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
public class MYRETURNTYPE
       ^
```

To work around this problem rewrite the XML Schema so that this type of naming conflict does not occur, or if that is not possible, run the Ant task on a case sensitive operating system, such as Unix.

clientgen

The `clientgen` Ant task generates, from an existing WSDL file, the client component files that client applications use to invoke both WebLogic and non-WebLogic Web Services. These files include:

- The Java source code for the `Stub` and `Service` interface implementations for the particular Web Service you want to invoke. The Web Service can be based on either JAX-RPC 1.1 or JAX-WS 2.0.
- The Java source code for any user-defined XML Schema data types included in the WSDL file.

- For JAX-RPC 1.1 Web Services, 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 Java Platform, Enterprise Edition (Java EE) Version 5 client container.
- Java EE clients, such as EJBs, JSPs, and Web Services, that use the Java EE 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

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

Examples

```
<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`:

```
<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:

```
wsdl="jar:file:archive_file!WSDL_file"
```

where *archive_file* refers to the full (or relative to the current directory) name of the archive file and *WSDL_file* refers to the full pathname of the WSDL file, relative to the root directory of the archive file. For example:

```
    <clientgen

wsdl="jar:file:output/myEAR/examples/webservices/simple/SimpleImpl.war!/WE
B-INF/SimpleService.wsdl"
      destDir="/output/clientclasses"
      packageName="myapp.myservice.client"/>
```

The preceding example shows how to execute `clientgen` on a static WSDL file called `SimpleService.wsdl`, which is packaged in the `WEB-INF` directory of a WAR file called `SimpleImpl.war`, which is located in the `output/myEAR/examples/webservices/simple` sub-directory of the directory that contains the `build.xml` file.

You can use the standard Ant `<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:

```
<clientgen
    wsdl="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>
```

Finally, in the preceding examples, it is assumed that the Web Service for which you are generating client artifacts is based on JAX-RPC 1.1; the following example shows how to use the `type` attribute to specify that the Web Service is based on JAX-WS 2.0:

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

Child Elements

The `clientgen` Ant task has one WebLogic-specific child element: `<binding>`.

See"Standard Ant Attributes and Elements That Apply To clientgen" on page A-18 for the list of elements associated with the standard Ant `javac` task that you can also set for the `clientgen` Ant task.

binding

Use the `<binding>` child element to specify one or more XMLBeans configuration files, which by convention end in `.xsdconfig`. Use this element if your Web Service uses Tylar data types as parameters or return values.

The `<binding>` 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.

You can use this child element when generating client files for both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

Note: The `<binding>` element replaces the `<xsdConfig>` element, which is deprecated as of version 10.0 of WebLogic Server.

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-18 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

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
autoDetectWrapped	Specifies whether the `clientgen` 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 `clientgen` Ant task parses a WSDL file to create the client 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 `clientgen` 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 `autoDetectWrapped` attribute to `False`. Valid values for this attribute are `True` or `False`. The default value is `True`.	Boolean	No.	JAX-RPC

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
destDir	Directory into which the `clientgen` Ant task generates the client source code, WSDL, and client deployment descriptor files. You can set this attribute to any directory you want. However, if you are generating the client component files to invoke a Web Service from an EJB, JSP, or other Web Service, you typically set this attribute to the directory of the Java EE 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.	String	You must specify either the `destFile` or `destDir` attribute, but not both.	Both
destFile	Name of a JAR file or exploded directory into which the `clientgen` task packages the client source code, compiled classes, WSDL, and client deployment descriptor files. If you specify this attribute, the `clientgen` Ant task also compiles all Java code into classes. To create or update a JAR file, use a `.jar` suffix when specifying the JAR file, such as `myclientjar.jar`. If the attribute value does not have a `.jar` suffix, then the `clientgen` task assumes you are referring to a directory name. If you specify a JAR file or directory that does not exist, the `clientgen` task creates a new JAR file or directory.	String	You must specify either the `destFile` or `destDir` attribute, but not both.	Both

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
failonerror	Specifies whether the `clientgen` Ant task continues executing in the event of an error. Valid values for this attribute are `True` or `False`. The default value is `True`, which means `clientgen` continues executing even after it encounters an error.	Boolean	No.	Both

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
generateAsyncMethods	Specifies whether the `clientgen` Ant task should include methods in the generated 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 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 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 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.	Boolean	No.	JAX-RPC

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
generatePolicyMethods	Specifies whether the `clientgen` Ant task should include WS-Policy-loading methods in the generated 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 `get`*`XXX`*`SoapPort()` are added as extensions to the `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 for more information.	Boolean	No.	JAX-RPC

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
handlerChainFile	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 for details and examples about creating client-side SOAP message handlers.	String	No	JAX-RPC
includeGlobalTypes	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.	Boolean	No.	JAX-RPC

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
jaxRPCWrappedArray Style	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 `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`.	Boolean	No.	JAX-RPC
packageName	Package name into which the generated 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 `http://example.org`, 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.	String	No.	Both

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
serviceName	Name of the Web Service in the WSDL file for which the corresponding client component files should be generated. The Web Service name corresponds to the `<service>` element in the WSDL file. The generated mapping file and client-side copy of the WSDL file will use this name. For example, if you set `serviceName` to `CuteService`, the 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`.	String	This attribute is required *only* if the WSDL file contains more than one `<service>` element. The Ant task returns an error if you do not specify this attribute and the WSDL file contains more than one `<service>` element.	JAX-RPC
type	Specifies the type of Web Service for which you are generating client artifacts: JAX-RPC 1.1 or JAX-WS 2.0. Valid values are: • `JAXWS` • `JAXRPC` Default value is `JAXRPC`.	String	No.	Both.

Table A-2 Attributes of the clientgen Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
wsdl	Full path name or URL of the WSDL that describes a Web Service (either WebLogic or non-WebLogic) for which the client component files should be generated. The generated stub factory classes in the client JAR file use the value of this attribute in the default constructor.	String	Yes.	Both

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`
- `verbose`

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>`

jwsc

The `jwsc` Ant task takes as input one or more Java Web Service (JWS) files that contains both standard 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-109 Web Service, such as the service endpoint interface (called `*JWS_ClassName*PortType.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. (For JAX-RPC only. For information about generating the WSDL and including it in the WAR file for JAX-WS Web Services, see "Generating a WSDL for JAX-WS Web Services" on page A-24.)

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.

By default, the `jwsc` Ant task generates a Web Service that follows the JAX-RPC 1.1 specification. However, by using the `type="JAXWS"` attribute of the `<jws>` child element, you can alternatively specify that the Ant task generate a JAX-WS 2.0 Web Service.

Note: Although not typical, you can code your JWS file to explicitly implement `javax.ejb.SessionBean`. See Should You Implement a Stateless Session EJB? 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-21

- "How to Determine the Final Context Root of a WebLogic Web Service" on page A-22
- "Generating Client Artifacts for an Invoked Web Service" on page A-23
- "Updating an Existing Enterprise Application or Web Application" on page A-23
- "Generating a WSDL for JAX-WS Web Services" on page A-24

See "Examples" on page A-25 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.

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 can be the same when specify both `<WLJMSTransport>` and `<WLHttpTransport>`.
- 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:

`http://`*`hostname`*`:7001/financial/GetQuote?WSDL`

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 `<WL`*`XXX`*`Transport>` child elements of `<jws>`.
3. The `contextPath` attribute of the `@WL`*`XXX`*`Transport` 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`. The context root of the Web Service would now be `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 `<clientgen>` element of `jwsc` to generate and compile the required client component files, such as the `Stub` and `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, `jwsc` generates a new Enterprise Application exploded directory at the location specified by the `destDir` attribute. However, if you specify an *existing* Enterprise Application as the destination directory, `jwsc` updates any existing `application.xml` file with the new Web Services information.

Similarly, `jwsc` typically generates new Web application deployment descriptors (`web.xml` and `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 `<descriptor>` child element of the `<module>` element to specify existing `web.xml` and `weblogic.xml` files; in this case, `jwsc` copies these files to the `destDir` directory and adds new information to them. Use the standard Ant `<fileset>` element to copy the other existing Web application files to the destDir directory.

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.

Generating a WSDL for JAX-WS Web Services

For JAX-WS Web Services, the WSDL file is not generated automatically by the `jwsc` Ant task. To generate a WSDL for JAX-WS Web Services, you need to include two instances of the `<jws>` child element, as follows:

- Set up the first instance of the `<jws>` child element to generate the WSDL. Set the `wsdlOnly` attribute to `true` to generate the WSDL file (but no other artifacts). In this case, the `jwsc` Ant task only generates the WSDL. It does not generate any other artifacts or compile the JWS file.
- Set up the second instance of the `<jws>` child element to compile and package your JWS file and include the WSDL in the WAR. Set the `wsdlOnly` attribute to `false` (the default) and set all other attributes as required to compile and package your JWS file.

The following provides an example of a `jwsc` Ant task that generates a WSDL for a JAX-WS Web Service and subsequently compiles the JWS file:

```
<jwsc
   srcdir="src"
   destdir="${ear-dir}">
   <jws file="hello_world.java" wsdlOnly="true" type="JAXWS">
      <WLHttpTransport serviceUri="myServiceURI" />
   </jws>
   <jws file="hello_world.java" wsdlOnly="false" type="JAXWS">
      <WLHttpTransport serviceUri="myServiceURI" />
   </jws>
</jwsc>
```

Taskdef Classname

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

Examples

The following examples show how to use the `jwsc` Ant task by including it in a `build-service` target of the `build.xml` Ant file that iteratively develops your Web Service. See Common Web Services Use Cases and Examples and Iterative Development of WebLogic Web Services for samples of complete `build.xml` files that contain many other targets that are useful when iteratively developing a WebLogic Web Service, such as `clean`, `deploy`, `client`, and `run`.

The following sample shows a very simple usage of `jwsc`:

```
<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`:

```
  <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:

```
<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):

```
<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"
          queue="JMSTransportQueue"/>
       <clientgen
          wsdl="http://examples.org/complex/ComplexService?WSDL"
          serviceName="ComplexService"
          packageName="examples.webservices.simple_client"/>
     </jws>
   </jwsc>
 </target>
```

The preceding example also shows how to use the `<clientgen>` element to generate and include the client-side artifacts (such as the `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" />
      <clientgen
         wsdl="http://examples.org/complex/ComplexService?WSDL"
        serviceName="ComplexService"
        packageName="examples.webservices.simple_client" />
    </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:

```
  <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 the `jwsc` Ant task not create new Web application deployment descriptors, but rather, add to existing ones:

```
<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.

All preceding examples generated JAX-RPC 1.1 Web Services by default; the following simple example shows how to generate a JAX-WS 2.0 Web Service by specifying the `type="JAXWS"` attribute of the `<jws>` child element:

```
  <target name="build-service8">

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

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

    </jwsc>

  </target>
```

You can specify the type attribute for the `<jws>` or `<jwsfileset>` elements.

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.

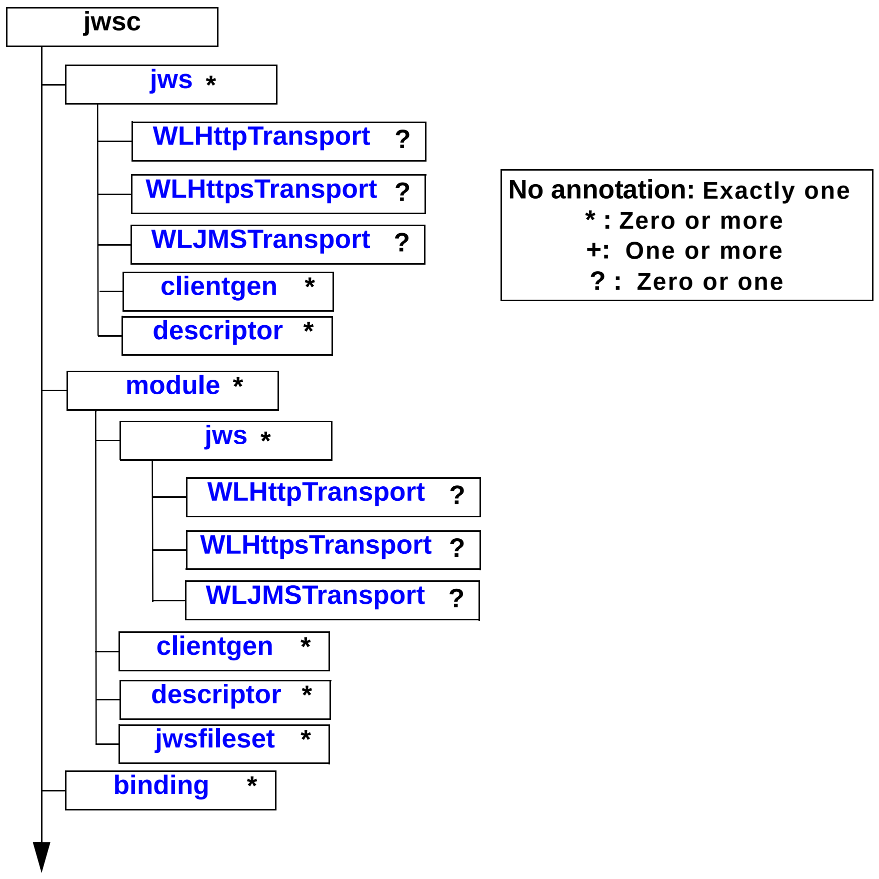
The `<jws>` element (when used as either a child element of `<jwsc>` or `<module>`) has three optional child elements: `<WLHttpTransport>`, `<WLHttpsTransport>`, and `<WLJMSTransport>`. See "Specifying the Transport Used to Invoke the Web Service" on page A-21 for common information about using the transport 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.

Figure A-1 Element Hierarchy of 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-36 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

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
applicationXml	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.	No.	Both
destdir	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.	Yes.	Both

Table A-3 Attributes of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
destEncoding	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.	No.	Both
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.	No	JAX-RPC

Table A-3 Attributes of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
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.	No.	Deprecated attribute so not applicable.
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`.	No.	Both

Table A-3 Attributes of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
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.	No.	Both
srcdir	The full pathname of top-level directory that contains the JWS file you want to compile (specified with the `file` attribute of the `<jws>` 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.	Yes.	Both
srcEncoding	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.	No.	Both

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`
- `failonerror`
- `fork`
- `includeantruntime`
- `includejavaruntime`
- `listfiles`
- `memoryInitialSize`
- `memoryMaximumSize`
- `nowarn`
- `optimize`
- `proceed`
- `source`
- `sourcepath`
- `sourcepathRef`
- `tempdir`

- `verbose`

You can also use the following standard Ant child elements with the `jwsc` Ant task:

- `<SourcePath>`
- `<Classpath>`
- `<Extdirs>`

You can use the following standard Ant elements with the `<jws>` and `<module>` child elements of the `jwsc` Ant task:

- `<FileSet>`
- `<ZipFileSet>`

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-25 for examples of using the element.

You can use the standard Ant `<FileSet>` child element with the `<jws>` element of `jwsc`.

You can use the `<jws>` child element when generating both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

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

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
compiledWsdl	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 `compiledWsdl` 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 `<jws>` when used as a child of both `jwsc` and `<module>`.	Only required for the "starting from WSDL" use case.	Both

Table A-4 Attributes of the <jws> Element of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
contextPath	Context root of the Web Service. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `http://hostname:7001/financial/GetQuote?WSDL` 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.	Both
explode	Specifies whether the generated WAR file that contains the deployable Web Service 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. Applies only when `<jws>` is a direct child of `jwsc`.	No.	Both
file	The name of the JWS file that you want to compile. The `jwsc` Ant task looks for the file in the `srcdir` directory. Applies to `<jws>` when used as a child of both `jwsc` and `<module>`.	Yes.	Both

Table A-4 Attributes of the <jws> Element of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?

includeSchemas	The full pathname of the XML Schema file that describes an `XMLBeans` parameter or return value of the Web Service.
	
	To specify more than one XML Schema file, use either a comma or semi-colon as a delimiter:
	`includeSchemas="po.xsd,customer.xsd"`
	
	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.
	
	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.ParameterSt`
	`yle.BARE)`
	
	For more information on `XMLBeans`, see http://dev2dev.bea.com/technologies/xmlbeans/index.jsp.
	
	Applies to `<jws>` when used as a child of both `jwsc` and `<module>`.
	
	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.
		Required if you are using an `XMLBean s` data type as a parameter or return value.	JAX-RPC

Table A-4 Attributes of the <jws> Element of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
name	The name of the generated WAR file (or exploded directory, if the `explode` attribute is set to `true`) that contains the deployable Web Service. If an actual JAR archive file is generated, the name of the file will have a `.war` extension. The default value of this attribute is the name of the JWS file, specified by the `file` attribute. Applies only when `<jws>` is a direct child of `jwsc`.	No.	Both
type	Specifies the type of Web Service to generate: JAX-RPC 1.1 or JAX-WS 2.0. Valid values are: • `JAXWS` • `JAXRPC` Default value is `JAXRPC`.	No.	Both.

Table A-4 Attributes of the <jws> Element of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
wsdlOnly	Specifies that *only* a WSDL file should be generated for this JWS file. **Note:** Although the other artifacts, such as the deployment descriptors and service endpoint interface, are not generated, data binding artifacts *are* generated because the WSDL must include the XML Schema that describes the data types of the parameters and return values of the Web Service operations. The WSDL is generated into the `destDir` directory. The name of the file is `*JWS_ClassName*Service.wsdl`, where `*JWS_ClassName*` refers to the name of the JWS class. `*JWS_ClassNameService*` is also the name of Web Service in the generated WSDL file. If you set this attribute to `true` but also set the `explode` attribute to `false` (which is also the default value), then jwsc ignores the `explode` attribute and always generates the output in exploded format. Valid values for this attribute are `true` or `false`. The default value is `false`, which means that all artifacts are generated by default, not just the WSDL file. Applies only when `<jws>` is a child of `jwsc`.	No.	Both

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 can not 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>.

You can use the <module> child element when generating both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

See Figure A-1 for a visual description of where this element fits in the jwsc element hierarchy. See "Examples" on page A-25 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

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
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: `http://hostname:7001/financial/GetQuote?WSDL` 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-22	Both.
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.	Both

Table A-5 Attributes of the <module> Element of the jwsc Ant Task (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
name	The name of the generated WAR file (or exploded directory, if the `explode` attribute is set to `true`) that contains the deployable Web Service(s). If an actual WAR archive file is generated, the name of the file will have a `.war` extension. The default value of this attribute is `jws`.	No.	Both.
wsdlOnly	Specifies that *only* a WSDL file should be generated for each JWS file specified by the `<jws>` child element of `<module>`. **Note:** Although the other artifacts, such as the deployment descriptors and service endpoint interface, are not generated, data binding artifacts *are* generated because the WSDL must include the XML Schema that describes the data types of the parameters and return values of the Web Service operations. The WSDL is generated into the `destDir` directory. The name of the file is *`JWS_ClassName`*`Service.wsdl`, where *`JWS_ClassName`* refers to the name of the JWS class. *`JWS_ClassNameService`* is also the name of Web Service in the generated WSDL file. If you set this attribute to `true` but also set the `explode` attribute to `false` (which is also the default value), then jwsc ignores the `explode` attribute and always generates the output in exploded format. Valid values for this attribute are `true` or `false`. The default value is `false`, which means that all artifacts are generated by default, not just the WSDL file.	No.	Both.

WLHttpTransport

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-21 for guidelines to follow when specifying this element.

You can use the `<WlHttpTransport>` child element when generating both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

See Figure A-1 for a visual description of where this element fits in the `jwsc` element hierarchy. See "Examples" on page A-25 for examples of using the element.

The following table describes the attributes of `<WLHttpTransport>`.

Table A-6 Attributes of the <WLHttpTransport> Child Element of the <jws> Element

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
contextPath	Context root of the Web Service. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `http://hostname:7001/financial/GetQuote?WSDL` The contextPath for this Web Service is `financial`. 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`.	No.	Both.

Table A-6 Attributes of the <WLHttpTransport> Child Element of the <jws> Element (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
serviceUri	Web Service URI portion of the URL. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `http://hostname:7001/financial/GetQuote?WSDL` The serviceUri for this Web Service is `GetQuote`. 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`.	No.	Both.
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`.	No.	Both.

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.

You can use the `<WlHttpsTransport>` child element *only* for generating JAX-RPC 1.1 Web Services.

See "Specifying the Transport Used to Invoke the Web Service" on page A-21 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>`.

Table A-7 Attributes of the <WLHttpsTransport> Child Element of the <jws> Element

Attribute	Description	Required?
contextPath	Context root of the Web Service. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `https://hostname:7001/financial/GetQuote?WSDL` The contextPath for this Web Service is `financial`. 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`.	No.

Table A-7 Attributes of the <WLHttpsTransport> Child Element of the <jws> Element (Continued)

Attribute	Description	Required?
serviceUri	Web Service URI portion of the URL. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `https://hostname:7001/financial/GetQuote?WSDL` The serviceUri for this Web Service is `GetQuote`. 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`.	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`.	No.

WLJMSTransport

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. 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 `<WLJmsTransport>` elements for a given JWS file.

You can use the `<WLJmsTransport>` child element *only* for generating JAX-RPC 1.1 Web Services.

See "Specifying the Transport Used to Invoke the Web Service" on page A-21 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-25 for examples of using the element.

The following table describes the attributes of `<WLJmsTransport>`.

Table A-8 Attributes of the <WLJMSTransport> Child Element of the <jws> Element

Attribute	Description	Required?
contextPath	Context root of the Web Service. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `http://hostname:7001/financial/GetQuote?WSDL` The contextPath for this Web Service is `financial`. 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`.	No.
serviceUri	Web Service URI portion of the URL. For example, assume the deployed WSDL of a WebLogic Web Service is as follows: `http://hostname:7001/financial/GetQuote?WSDL` The serviceUri for this Web Service is `GetQuote`. 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`.	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`.	No.

Table A-8 Attributes of the <WLJMSTransport> Child Element of the <jws> Element (Continued)

Attribute	Description	Required?
queue	The JNDI name of the JMS queue that you have configured for the JMS transport. See Using JMS Transport as the Connection Protocol for details about using JMS transport. The default value of this attribute, if you do not specify it, is `weblogic.wsee.DefaultQueue`. You must still create this JMS queue in the WebLogic Server instance to which you deploy your Web Service.	No.
connectionFactory	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.	No.

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 `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.

See Figure A-1 for a visual description of where this element fits in the `jwsc` element hierarchy. See "Examples" on page A-25 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>`.

You can use the `<clientgen>` child element for generating both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

The following table describes the attributes of the `<clientgen>` element.

Table A-9 Attributes of the <clientgen> Element

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
autoDetectWrapped	Specifies whether the `jwsc` 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 `jwsc` Ant task parses a WSDL file to create the 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 `jwsc` 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 `autoDetectWrapped` attribute to `False`. Valid values for this attribute are `True` or `False`. The default value is `True`.	No.	JAX-RPC

Table A-9 Attributes of the <clientgen> Element (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
handlerChainFile	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 for details and examples about creating client-side SOAP message handlers.	No.	JAX-RPC

Table A-9 Attributes of the <clientgen> Element (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
generateAsyncMethods	Specifies whether the `jwsc` Ant task should include methods in the generated 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 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 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 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.	No.	JAX-RPC

Table A-9 Attributes of the <clientgen> Element (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
generatePolicyMethods	Specifies whether the `jwsc` Ant task should include WS-Policy-loading methods in the generated 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 `get`*`XXX`*`SoapPort()` are added as extensions to the `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 for more information.	No.	JAX-RPC
includeGlobalTypes	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.		JAX-RPC

Table A-9 Attributes of the <clientgen> Element (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
jaxRPCWrappedArrayStyle	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`	No.	JAX-RPC
packageName	Package name into which the generated client interfaces and stub files are packaged. BEA recommends you use all lower-case letters for the package name.	Yes.	Both.
serviceName	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 `<service>` element in the WSDL file. The generated JAX-RPC mapping file and client-side copy of the WSDL file will use this name. For example, if you set `serviceName` to `CuteService`, the JAX-RPC mapping file will be called `cuteService_java_wsdl_mapping.xml` and the client-side copy of the WSDL will be called `CuteService_saved_wsdl.wsdl`.	This attribute is required *only* if the WSDL file contains more than one `<service>` element. The Ant task returns an error if you do not specify this attribute and the WSDL file contains more than one `<service>` element.	JAX-RPC

Table A-9 Attributes of the <clientgen> Element (Continued)

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
wsdl	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 stub factory classes use the value of this attribute in the default constructor.	Yes.	Both.

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.

You can use the `<descriptor>` child element *only* for generating JAX-RPC 1.1 Web Services.

See Figure A-1 for a visual description of where this element fits in the `jwsc` element hierarchy. See "Examples" on page A-25 for examples of using the element.

The following table describes the attributes of the `<descriptor>` element.

Table A-10 Attributes of the <descriptor> Element

Attribute	Description	Required?
file	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 `jwsc` Ant task does not update this file directly, but rather, copies it to the newly-generated Web application.	Yes.

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.

You can use the `<jwsfileset>` child element for generating both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

See Figure A-1 for a visual description of where this element fits in the `jwsc` element hierarchy. See "Examples" on page A-25 for examples of using the element.

The following table describes the attributes of the `<jwsfileset>` element.

Table A-11 Attributes of the <jwsfileset> Element

Attribute	Description	Required?	JAX-RPC, JAX-WS, or Both?
srcdir	Specifies the directories (separated by semi-colons) that the `jwsc` Ant task should search for JWS files to compile.	Yes.	Both.
type	Specifies the type of Web Service to generate for each found JWS file: JAX-RPC 1.1 or JAX-WS 2.0. Valid values are: • `JAXWS` • `JAXRPC` Default value is `JAXRPC`.	No.	Both.

binding

Use the `<binding>` child element to specify one or more XMLBeans configuration files, which by convention end in `.xsdconfig`. Use this element if your Web Service uses Tylar data types as parameters or return values.

The `<binding>` 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 use the `<binding>` child element for generating both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

Note: The `<binding>` element replaces the `<xsdConfig>` element, which is deprecated as of version 10.0 of WebLogic Server.

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 specifying the `type` attribute, you can generate a partial implementation based on either JAX-RPC 1.1 or JAX-WS 2.0.

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.

Specifically, the `wsdlc` Ant task generates:

- 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.

  For JAX-RPC, the Tylar binding is used by default. For JAX-WS, the JAXB binding is used by default.

  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 `wsdlc` Ant task obviously does not know how you want your Web Service to function, although it does know the *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 *only* standard JSR-181 JWS annotations you can include in the JWS implementation file are `@WebService`, `@HandlerChain`, `@SOAPMessageHandler`, and

`@SOAPMessageHandlers`. If you specify any other JWS-181 JWS annotations, the `jwsc` Ant task will return an error when you try to compile the JWS file into a Web Service.

- Additionally, 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.
- For JAX-RPC 1.1 Web Services, you can specify any WebLogic-specific JWS annotation that you want. You cannot use any WebLogic-specific JWS annotations in a JAX-WS 2.0 Web Service.

Finally, after you have coded the JWS file so that it works as you want, iteratively run the `jwsc` Ant task to generate a complete Java implementation of the Web Service. Use the `compiledWsdl` attribute of `jwsc` to specify the JAR file generated by the `wsdlc` Ant task which contains the JWS interface file and data binding artifacts. By specifying this attribute, the `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.

Note: The only potential difference between the original and deployed WSDL is the value of the `location` attribute of the `<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.

Depending on the type of partial implementation you generate (JAX-RPC 1.1 or JAX-WS 2.0), the Java package name of the generated complex data types differs, as described in the following guidelines:

- For JAX-RPC 1.1, if you specify the `packageName` attribute of the `wsdlc` Ant task, only the generated JWS interface and implementation are in this package. The package name of the generated Java complex data types, however, always corresponds to the XSD Schema type namespace, whether you specify the `packageName` attribute or not.
- For JAX-WS 2.0, if you specify the `packageName` attribute, then *all* artifacts (Java complex data types, JWS interface, and the JWS interface implementation) are generated into this package. If you want to change the package name of the generated Java complex data types in this case, use the `<binding>` child element of the `wsdlc` Ant task to specify a custom binding file.

See Creating a Web Service from a WSDL File for a complete example of using the `wsdlc` Ant task in conjunction with `jwsc`.

Taskdef Classname

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

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:

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

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

  <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 Elements

The `wsdlc` Ant task has one WebLogic-specific child element: `<binding>`.

See“Standard Ant javac Attributes That Apply To wsdlc” on page A-70 for the list of elements associated with the standard Ant `javac` task that you can also set for the `wsdlc` Ant task.

binding

Use the `<binding>` child element to specify one or more XMLBeans configuration files, which by convention end in `.xsdconfig`. Use this element if your Web Service uses Tylar data types as parameters or return values.

The `<binding>` 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.

You can use the `<binding>` child element when generating a partial implemention of both JAX-RPC 1.1 and JAX-WS 2.0 Web Services.

Note: The `<binding>` element replaces the `<xsdConfig>` element, which is deprecated as of version 10.0 of WebLogic Server.

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-70 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

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
autoDetectWrapped	Specifies whether the `wsdlc` 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 `wsdlc` 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 `wsdlc` 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 `autoDetectWrapped` attribute to `False`. Valid values for this attribute are `True` or `False`. The default value is `True`.	Boolean	No.	JAX-RPC

Table A-12 Attributes of the wsdlc Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
destImplDir	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 `jwsc` Ant task to generate a deployable Web Service.	String	No.	Both
destJavadocDir	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.	String	No.	Both
destJwsDir	Directory into which the JAR file that contains the JWS interface and data binding artifacts should be generated. The name of the generated JAR file is `*WSDLFile*_wsdl.jar`, where `*WSDLFile*` refers to the root name of the WSDL file. For example, if the name of the WSDL file you specify to the `file` attribute is `MyService.wsdl`, then the generated JAR file is `MyService_wsdl.jar`.	String	Yes.	Both

Table A-12 Attributes of the wsdlc Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
explode	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 `true` or `false`. Default value is `false`, which means that `wsdlc` generates an actual JAR archive file, and not an exploded directory.	Boolean	No.	Both
jaxRPCWrapped ArrayStyle	When the `wsdlc` 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`.	Boolean	No.	JAX-RPC
packageName	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.	String	No.	Both

Table A-12 Attributes of the wsdlc Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?

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	JAX-RPC

Table A-12 Attributes of the wsdlc Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
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`.	String.	No.	Both

Table A-12 Attributes of the wsdlc Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
srcServiceName	Name of the Web Service from which the JWS interface file should be generated. Set the value of this attribute to the value of the `name` attribute of the `<service>` element that corresponds to the Web Service for which you want to generate a JWS interface file. 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 `<service>` element contains more than one `<port>` element, the following must be true: • The bindings for each port must be the same or equivalent to each other. • The transport for each port must be different. The `wsdlc` Ant task determines the transport for a port from the address listed in its `<address>` child element. Because WebLogic Web Services support only three transports (JMS, HTTP, and HTTPS), this means that there can be at most three `<port>` child elements for the `<service>` element specified by this attribute. The generated JWS implementation file will then include the corresponding `@WLXXXTransport` annotations. If you specify this attribute, you cannot also specify `srcPortName`. If you do not specify either this or the `srcPortName` attribute, the WSDL file must include only *one* `<service>` element. The `wsdlc` Ant task generates the JWS interface file and data binding JAR file from this single Web Service.	String	No.	Both

Table A-12 Attributes of the wsdlc Ant Task (Continued)

Attribute	Description	Data Type	Required?	JAX-RPC, JAX-WS, or Both?
srcWsdl	Name of the WSDL from which to generate the JAR file that contains the JWS interface and data binding artifacts. The name must include its pathname, either absolute or relative to the directory which contains the Ant `build.xml` file.	String	Yes.	Both
type	Specifies the type of Web Service for which you are generating a partial implementation: JAX-RPC 1.1 or JAX-WS 2.0. Valid values are: • `JAXWS` • `JAXRPC` Default value is `JAXRPC`.	String	No.	Both
wlw81CallbackGen	Specifies whether to generate a WebLogic Workshop 8.1 style callback. Valid values for this attribute are `True` or `False`. The default value is `False`.	Boolean	No.	JAX-RPC

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`
- `verbose`

You can also use the following standard Ant child elements with the `wsdlc` Ant task:

- `<FileSet>`
- `<SourcePath>`
- `<Classpath>`
- `<Extdirs>`

APPENDIX B

JWS Annotation Reference

The following sections provide reference documentation about the 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-4

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 the WebLogic-specific annotations.

WARNING: Although this release of WebLogic Server supports both JAX-RPC 1.1 and JAX-WS 2.0 based Web Services, you can use the WebLogic-specific annotations *only* with JAX-RPC-based Web Services.

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:

```
package examples.webservices.complex;

// Import the standard JWS annotation interfaces

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

// Import the WebLogic-specific JWS annotation interface

import weblogic.jws.WLHttpTransport;

// Import the BasicStruct JavaBean

import examples.webservices.complex.BasicStruct;

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

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

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

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

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

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

/**
 * This JWS file forms the basis of a WebLogic Web Service.  The Web Services
 * has two public operations:
 *
 *  - echoInt(int)
 *  - echoComplexType(BasicStruct)
 *
 * The Web Service is defined as a "document-literal" service, which means
 * that the SOAP messages have a single part referencing an XML Schema element
 * that defines the entire body.
 *
 * @author Copyright (c) 2005 by BEA Systems. All Rights Reserved.
 */

public class ComplexImpl {

  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: echoInt.
  //
  // The WebResult annotation specifies that the name of the result of the
  // operation in the generated WSDL is "IntegerOutput", rather than the
  // default name "return".   The WebParam annotation specifies that the input
  // parameter name in the WSDL file is "IntegerInput" rather than the Java
  // name of the parameter, "input".

  @WebMethod()
  @WebResult(name="IntegerOutput",
             targetNamespace="http://example.org/complex")
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/complex")
      int input)
  {
    System.out.println("echoInt '" + input + "' to you too!");
    return input;
  }

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

  @WebMethod(operationName="echoComplexType")
  @WebResult(name="EchoStructReturnMessage",
             targetNamespace="http://example.org/complex")
  public BasicStruct echoStruct(BasicStruct struct)
  {
    System.out.println("echoComplexType called");
    return struct;
  }
}
```

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; see this specification for full reference information on each annotation.

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-5
- "weblogic.jws.AsyncResponse" on page B-8
- "weblogic.jws.Binding" on page B-11
- "weblogic.jws.BufferQueue" on page B-13
- "weblogic.jws.Callback" on page B-14
- "weblogic.jws.CallbackMethod" on page B-16
- "weblogic.jws.CallbackService" on page B-17
- "weblogic.jws.Context" on page B-19
- "weblogic.jws.Conversation" on page B-21
- "weblogic.jws.Conversational" on page B-23
- "weblogic.jws.FileStore" on page B-26
- "weblogic.jws.MessageBuffer" on page B-26
- "weblogic.jws.Policies" on page B-29
- "weblogic.jws.Policy" on page B-30
- "weblogic.jws.ReliabilityBuffer" on page B-33
- "weblogic.jws.ReliabilityErrorHandler" on page B-35
- "weblogic.jws.ServiceClient" on page B-37

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 stub for the Web Service you are invoking and the name of the operation that you are invoking asynchronously. The 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 `on`*`MethodName`*`AsyncFailure`, 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:

  ```
  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 (`on`*`MethodName`*`AsyncFailure`) 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 for detailed information and examples of using this annotation.

Attributes

Table B-1 Attributes of the weblogic.jws.AsyncFailure JWS Annotation Tag

Name	Description	Data Type	Required?
target	The name of the 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.	String	Yes
operation	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="getQuote"`, then in the JWS file you invoke it asynchronously as follows: `port.getQuoteAsync (apc, symbol);`	String	Yes.

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:

```
package examples.webservices.async_req_res;

...

public class StockQuoteClientImpl {

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

  }

...

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

}
```

The example shows a stub called `port`, used to invoke the Web Service located at `http://localhost:7001/async/StockQuote`. The `getQuote` operation is invoked asynchronously, and any exception from this invocation is handled by the `onGetQuoteAsyncFailure` method, as specified by the `@AsyncFailure` annotation.

weblogic.jws.AsyncResponse

Description

Target: Method

Specifies the method that handles the response 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 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. Use the `@AsyncResponse` annotation to specify the method in the JWS file that will handle the response of an asynchronous operation invocation.

The `@AsyncResponse` annotation takes two parameters: the name of the stub for the Web Service you are invoking and the name of the operation that you are invoking asynchronously. The stub is the one that has been annotation with the `@ServiceClient` annotation.

The method that handles the asynchronous response must follow these guidelines:

- Return `void`.
- Be named `on`*`MethodName`*`AsyncResponse`, 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:

  ```
  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 (`on`*`MethodName`*`AsyncResponse`) 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 for detailed information and examples of using this annotation.

Attributes

Table B-2 Attributes of the weblogic.jws.AsyncResponse JWS Annotation Tag

Name	Description	Data Type	Required?
target	The name of the 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.	String	Yes
operation	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="getQuote"`, then in the JWS file you invoke it asynchronously as follows: `port.getQuoteAsync (apc, symbol);`	String	Yes.

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:

```
package examples.webservices.async_req_res;

...

public class StockQuoteClientImpl {

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

  @WebMethodpublic 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();
    }

  }

...

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

}
```

The example shows a stub called `port`, used to invoke the Web Service located at `http://localhost:7001/async/StockQuote`. The `getQuote` operation is invoked asynchronously, and the response from this invocation is handled by the `onGetQuoteAsyncResponse` method, as specified by the `@AsyncResponse` annotation.

weblogic.jws.Binding

Description

Target: Class

Specifies whether the Web Service uses version 1.1 or 1.2 of the Simple Object Access Protocol (SOAP) implementation when accepting or sending SOAP messages. By default, WebLogic Web Services use SOAP 1.1.

Attributes

Table B-3 Attributes of the weblogic.jws.Binding JWS Annotation Tag

Name	Description	Data Type	Required?
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:

```
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 `@BuffereQueue` 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 and Using Web Service Reliable Messaging for detailed information and examples of creating buffered or reliable Web Services.

Attributes

Table B-4 Attributes of the weblogic.jws.BufferQueue JWS Annotation Tag

Name	Description	Data Type	Required?
name	The JNDI name of the JMS queue to which the buffered or reliable operation invocation is queued.	String	Yes

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.buffere.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 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:

```
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 for additional overview and procedural information about programming callbacks.

Attributes

Table B-5 Attributes of the weblogic.jws.CallbackMethod JWS Annotation Tag

Name	Description	Data Type	Required?
operation	Specifies the name of the callback method in the callback interface for which this method will handle callback messages.	String	Yes
target	Specifies the name of the stub for which you want to receive callbacks. The stub is the one that has been annotated with the `@ServiceClient` field-level annotation.	String	Yes

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 stub information and that the annotated method will handle messages received from a callback operation called `callbackOperation()`.

```
  @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 for additional overview and procedural information about programming callbacks.

Attributes

Table B-6 Attributes of the weblogic.jws.CallbackService JWS Annotation Tag

Name	Description	Data Type	Required?
name	Name of the callback Web Service. Maps to the `<wsdl:portType>` element in the WSDL file. Default value is the unqualified name of the Java class in the JWS file.	String	No.
serviceName	Service name of the callback Web Service. Maps to the `<wsdl:service>` element in the WSDL file. Default value is the unqualified name of the Java class in the JWS file, appended with the string `Service`.	String	No.

Example

The following example shows a very simple callback interface. The resulting callback Web Service has one callback method, `callbackOperation()`.

```
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.

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:

```
...

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 for detailed information and examples of using this annotation.

Attributes

Table B-7 Attributes of the weblogic.jws.Conversation JWS Annotation Tag

Name	Description	Data Type	Required?
value	Specifies the phase of a conversation that the annotated method implements. 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`	enum	No.

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.

```
...
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 for detailed information and examples of using this annotation.

Attributes

Table B-8 Attributes of the weblogic.jws.Conversational JWS Annotation Tag

Name	Description	Data Type	Required?
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

Table B-8 Attributes of the weblogic.jws.Conversational JWS Annotation Tag

Name	Description	Data Type	Required?
runAsStartUser	Specifies whether the continue and finish phases of an existing conversation are run as the user who started the conversation. Typically, the same user executes the start, continue, and finish methods of a conversation, so that changing the value of this attribute has no effect. However, if you set the `singlePrincipal` attribute to `false`, which allows users different from the user who initiated the conversation to execute the continue and finish phases of an existing conversation, then the `runAsStartUser` attribute specifies which user the methods are actually "run as": the user who initiated the conversation or the different user who executes subsequent phases of the conversation. Valid values are `true` and `false`. Default value is `false`.	boolean	No.
singlePrincipal	Specifies whether users other than the one who started a conversation are allowed to execute the continue and finish phases of the conversation. Typically, the same user executes all phases of a conversation. However, if you set this attribute to `false`, then other users can obtain the conversation ID of an existing conversation and use it to execute later phases of the conversation. Valid values are `true` and `false`. Default value is `false`.	boolean	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",
                 runAsStartUser=false,
                 singlePrincipal=false )
public class ConversationalServiceImpl implements Serializable {
...
```

weblogic.jws.FileStore

Description

Target: Class

Specifies that the Web Service does not use the default WebLogic Server default filestore to store internal state information, such as conversational state, but rather uses one specified by the programmer. If you do not specify this JWS annotation in your JWS file, the Web Service uses the default filestore configured for WebLogic Server.

You can also use this JWS annotation for reliable Web Services to store internal state.

If you deploy the Web Service in a cluster, be sure you specify the *logical name* of the filestore so that the same name of the filestore can be used on all servers in the cluster

WARNING: This annotation applies only to filestores, not to JDBC stores.

Attributes

Table B-9 Attributes of the weblogic.jws.FileStore JWS Annotation Tag

Name	Description	Data Type	Required?
storeName	Specifies the name of the filestore.	String	Yes.

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 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 for detailed information and examples for using this annotation.

Attributes

Table B-10 Attributes of the weblogic.jws.MessageBuffer JWS Annotation Tag

Name	Description	Data Type	Required?
retryCount	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.	int	No
retryDelay	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="2 days")` Default value is `5 seconds.`	String	No

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 and Configuring Message-Level Security (Digital Signatures and Encryption) for detailed information and examples of using this annotation.

This JWS annotation does not have any attributes.

Example

```
@Policies({
    @Policy(uri="policy:firstPolicy.xml"),
    @Policy(uri="policy:secondPolicy.xml")
  })
```

weblogic.jws.Policy

Description

Target: Class, Method

Specifies that a WS-Policy file, which contains information about digital signatures, encryption, or 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.

WARNING: If the WS-Policy file you are specifying with the `@Policy` annotation contains information about Web Service reliable messaging, then you can set the `direction` attribute *only* to its default value: `Policy.Direction.both`. This is because Web Service reliable messaging is always applied to both the request and response SOAP message.

The two pre-packaged WS-Policy files that specify reliable messaging information are `DefaultReliability.xml` and `LongRunningReliability.xml`.

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 and Configuring Message-Level Security (Digital Signatures and Encryption) 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-11 Attributes of the weblogic.jws.Policies JWS Annotation Tag

Name	Description	Data Type	Required?
uri	Specifies the location from which to retrieve the WS-Policy file. Use the `http:` prefix to specify the URL of a WS-Policy file on the Web. Use the `policy`: prefix to specify that the WS-Policy file is packaged in the Web Service archive file or in a shareable Java EE library of WebLogic Server, as shown in the following example: `@Policy(uri="policy:MyPolicyFile.xml")` If you are going to publish the WS-Policy file in the Web Service archive, the WS-Policy XML file must be located in either the `META-INF/policies` or `WEB-INF/policies` directory of the EJB JAR file (for EJB implemented Web Services) or WAR file (for Java class implemented Web Services), respectively. For information on publishing the WS-Policy file in a library, see Creating Shared J2EE Libraries and Optional Packages.	String	Yes.

Table B-11 Attributes of the weblogic.jws.Policies JWS Annotation Tag

Name	Description	Data Type	Required?
direction	Specifies when to apply the policy: on the inbound request SOAP message, the outbound response SOAP message, or both (default). Valid values for this attribute are: • `Policy.Direction.both` • `Policy.Direction.inbound` • `Policy.Direction.outbound` The default value is `Policy.Direction.both`. **Note:** If the WS-Policy file you are specifying with the `@Policy` annotation contains information about Web Service reliable messaging, then you can set the `direction` attribute *only* to its default value: `Policy.Direction.both`. This is because Web Service reliable messaging is always applied to both the request and response SOAP message. The two pre-packaged WS-Policy files that specify reliable messaging information are `DefaultReliability.xml` and `LongRunningReliability.xml`.	enum	No.
attachToWsdl	Specifies whether the WS-Policy file should be attached to the WSDL that describes the Web Service. Valid values are `true` and `false`. Default value is `false`.	boolean	No.

Example

```
@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 for detailed information about enabling Web Services reliable messaging for your Web Service.

Attributes

Table B-12 Attributes of the weblogic.jws.ReliabilityBuffer JWS Annotation Tag

Name	Description	Data Type	Required?
retryCount	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.	int	No
retryDelay	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="2 days")` Default value is `5 seconds.`	String	No

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:

```
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.ReliabilityErrorContext`. 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 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 B-13 Attributes of the weblogic.jws.ReliabilityErrorHandler JWS Annotation Tag

Name	Description	Data Type	Required?
target	Specifies the target stub name for which this method handles reliability failures.	String	Yes

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:

```
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(

wsdlLocation="http://localhost:7001/ReliableHelloWorld/ReliableHelloWorld?
WSDL",
     serviceName="ReliableHelloWorldService",
     portName="ReliableHelloWorldServicePort")

  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 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` 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 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 port type for the Web Service described by the `@ServiceClient` attributes. The Enterprise Application that contains the client Web Service must also include the stubs of the Web Service you are invoking; you generate the stubs with the `clientgen` Ant task.

See WebLogic Web Service: Advanced Programming for additional information and examples of using the `@ServiceClient` annotation.

Attributes

Table B-14 Attributes of the weblogic.jws.ServiceClient JWS Annotation Tag

Name	Description	Data Type	Required?
serviceName	Specifies the name of the Web Service that you are invoking. Corresponds to the `name` attribute of the `<service>` 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.	String	Yes
portName	Specifies the name of the port of the Web Service you are invoking. Corresponds to the `name` attribute of the `<port>` child element of the `<service>` 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 `<service>` element in the WSDL contains only one `<port>` child element, which `@ServiceClient` uses. If there is more than one port, the client Web Service returns a runtime exception.	String	No.
wsdlLocation	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 `Service` implementation of the Web Service to be invoked.	String	No.
endpointAddress	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.	String	No.

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 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:

```
package examples.webservices.reliable;

import javax.jws.WebService;

...

import weblogic.jws.ServiceClient;

import examples.webservices.reliable.ReliableHelloWorldPortType;

@WebService(...

public class ReliableClientImpl

{

  @ServiceClient(

wsdlLocation="http://localhost:7001/ReliableHelloWorld/ReliableHelloWorld?
WSDL",
     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.

```
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")
```

```
@WLHttpTransport(contextPath="stream_attach",
                 serviceUri="StreamAttachService",
                 portName="StreamAttachServicePort")

@StreamAttachments

/**
 * Example of stream attachments
 */

public class StreamAttachImpl {

  @WebMethod()
 public DataHandler echoAttachment(DataHandler dh) throws RemoteException {

        return dh;

  }
}
```

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 B-15 Attributes of the weblogic.jws.Transactional JWS Annotation Tag

Name	Description	Data Type	Required?
value	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`.	boolean	No.
timeout	Specifies a timeout value, in seconds, for the current transaction. The default value for this attribute is `30` seconds.	int	No

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:

```
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-16 Attributes of the weblogic.jws.Types JWS Annotation Tag

Name	Description	Data Type	Required?
value	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.	String[]	Yes

Example

The following example shows a simple JWS file that uses the `@Types` annotation, with relevant Java code shown in bold:

```
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");
    return struct;
  }

}
```

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 B-17 Attributes of the weblogic.jws.WildcardBinding JWS Annotation Tag

Name	Description	Data Type	Required?
className	Specifies the fully qualified name of the wildcard class for which this binding applies. Typical values are `javax.xml.soap.SOAPElement` and `org.apache.xmlbeans.XmlObject`.	String	Yes.
binding	Specifies the XML Schema data type to which the wildcard class should bind. You can specify one of the following values: • `WildcardParticle.ANY` • `WildcardParticle.ANYTYPE`	enum	Yes.

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:

```
@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-46 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-18 Attributes of the weblogic.jws.WLHttpTransport JWS Annotation Tag

Name	Description	Data Type	Required?
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: `http://hostname:7001/financial/GetQuote?WSDL` 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: `http://hostname:7001/financial/GetQuote?WSDL` 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

```
@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-19 Attributes of the weblogic.jws.WLHttpsTransport JWS Annotation Tag

Name	Description	Data Type	Required?
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: `https://hostname:7001/financial/GetQuote?WSDL` 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: `https://hostname:7001/financial/GetQuote?WSDL` 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

```
@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 B-20 Attributes of the weblogic.jws.WLJmsTransport JWS Annotation Tag

Name	Description	Data Type	Required?
contextPath	Context root of the Web Service. You use this value in the URL that invokes the Web Service.	String	No.
serviceUri	Web Service URI portion of the URL used by client applications to invoke the Web Service.	String	No.
queue	The JNDI name of the JMS queue that you have configured for the JMS transport. See Using JMS Transport as the Connection Protocol for details about using JMS transport. The default value of this attribute, if you do not specify it, is `weblogic.wsee.DefaultQueue`. You must still create this JMS queue in the WebLogic Server instance to which you deploy your Web Service.	String	No.

Table B-20 Attributes of the weblogic.jws.WLJmsTransport JWS Annotation Tag

Name	Description	Data Type	Required?
portName	The name of the port in the generated WSDL. This attribute maps to the `name` attribute of the `<port>` element in the WSDL. If you do not specify this attribute, the `jwsc` generates a default name based on the name of the class that implements the Web Service.	String	No.
connectionFactory	The JNDI name of the JMS connection factory that you have configured for the JMS transport. See Using JMS Transport as the Connection Protocol for details about using JMS transport.	String	Yes.

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.

```
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:

```
@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:

```
http://ariel:7001/hello/SimpleImpl?WSDL
```

Attributes

Table B-21 Attributes of the weblogic.jws.WSDL JWS Annotation Tag

Name	Description	Data Type	Required?
exposed	Specifies whether to expose the WSDL of a deployed Web Service. Valid values are `true` and `false`. Default value is `true`, which means that by default the WSDL *is* exposed.	boolean	No.

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:

```
package examples.webservices;

import....

@WebService(name="WsdlAnnotationPortType",
            serviceName="WsdlAnnotationService",
            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 B-22 Attributes of the weblogic.jws.security.CallbackRolesAllowed JWS Annotation Tag

Name	Description	Data Type	Required?
value	Array of `@weblogic.jws.security.RolesAllowed` that list the roles allowed to invoke the callback methods.	String[]	Yes.

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:

```
  @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 B-23 Attributes of the weblogic.jws.security.RolesAllowed JWS Annotation Tag

Name	Description	Data Type	Required?
value	Array of @`weblogic.jws.security.RolesAllowed` that list the roles allowed to invoke the Web Service methods.	String[]	Yes.

Example

```
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.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

```
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 B-24 Attributes of the weblogic.jws.security.RunAs JWS Annotation

Name	Description	Data Type	Required?
role	Specifies the role which the Web Service should be run as.	String	Yes.
mapToPrincipal	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.	String	Yes.

Example

```
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-25 Attributes of the weblogic.jws.security.SecurityRole JWS Annotation

Name	Description	Data Type	Required?
role	The name of the role that is allowed to invoke the Web Service.	String	Yes
mapToPrincipals	An array of user names that map to the role. If you do not specify this attribute, it is assumed that you have externally defined the mapping between users and the role, typically using the Administration Console.	String[]	No

Example

```
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 B-26 Attributes of the weblogic.jws.security.SecurityRoleRef JWS Annotation

Name	Description	Data Type	Required?
role	Name of the role reference.	String	Yes.
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.	String	Yes.

Example

```
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.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-27 Attributes of the weblogic.jws.security.UserDataConstraint JWS Annotation

Name	Description	Data Type	Required?
transport	Specifies whether the client is required to use the HTTPS transport when invoking the Web Service. Valid values are: • `Transport.NONE`—Specifies that the Web Service does not require any transport guarantees. • `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. • `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. Default value is `Transport.NONE`.	enum	No

Example

```
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).

WARNING: All WebLogic Web Services packaged in a single Web Application must be associated with the same security configuration when using the `@WssConfiguration` annotation. This means, for example, that if a `@WssConfiguration` annotation exists in all the JWS files that implement the Web Services contained in a given Web Application, then the `value` attribute of each `@WssConfiguration` must be the same.

To specify that more than one Web Service be contained in a single Web Application when using the `jwsc` Ant task to compile the JWS files into Web Services, group the corresponding `<jws>` elements under a single `<module>` element.

Attributes

Table B-28 Attributes of the weblogic.jws.security.WssConfiguration JWS Annotation Tag

Name	Description	Data Type	Required?
value	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.	String	Yes.

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:

```
package examples.webservices.wss_configuration;

import javax.jws.WebService;
...

import weblogic.jws.security.WssConfiguration;

@WebService(...
...

@WssConfiguration(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-29 Attributes of the weblogic.jws.soap.SOAPBinding JWS Annotation

Name	Description	Data Type	Required?
style	Specifies the message style of the request and response SOAP messages of the invoked annotated operation. Valid values are: • `SOAPBinding.Style.RPC` • `SOAPBinding.Style.DOCUMENT`. Default value is `SOAPBinding.Style.DOCUMENT`.	enum	No.

Table B-29 Attributes of the weblogic.jws.soap.SOAPBinding JWS Annotation

Name	Description	Data Type	Required?
use	Specifies the formatting style of the request and response SOAP messages of the invoked annotated operation. Valid values are: • `SOAPBinding.Use.LITERAL` • `SOAPBinding.Use.ENCODED` Default value is `SOAPBinding.Use.LITERAL`.	enum	No.
parameterStyle	Determines whether method parameters represent the entire message body, or whether the parameters are elements wrapped inside a top-level element named after the operation. Valid values are: • `SOAPBinding.ParameterStyle.BARE` • `SOAPBinding.ParameterStyle.WRAPPED` Default value is `SOAPBinding.ParameterStyle.WRAPPED` **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 `style` attribute to `SOAPBinding.Style.DOCUMENT` and the `use` attribute to `SOAPBinding.Use.LITERAL`.	enum	No.

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.

```
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? for information about explicitly implementing an EJB in a JWS file.

Attributes

Table B-30 Attributes of the weblogic.jws.security.SecurityRoles JWS Annotation

Name	Description	Data Type	Required?
rolesAllowed	Specifies the list of roles that are allowed to access the Web Service. This annotation is the equivalent of the `<method-permission>` element in the `ejb-jar.xml` deployment descriptor of the stateless session EJB that implements the Web Service.	Array of String	No.
rolesReferenced	Specifies a list of roles referenced by the Web Service. The Web Service may access other resources using the credentials of the listed roles. This annotation is the equivalent of the `<security-role-ref>` element in the `ejb-jar.xml` deployment descriptor of the stateless session EJB that implements the Web Service.	Array of String	No.

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:

```
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
```

```
@SecurityRoles(rolesAllowed="Admnin")
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? for information about explicitly implementing an EJB in a JWS file.

Attributes

Table B-31 Attributes of the weblogic.jws.security.SecurityIdentity JWS Annotation

Name	Description	Data Type	Required?
value	Specifies the role which the Web Service assumes when it is invoked. The role must map to a user or group in the WebLogic Server security realm.	String	Yes.

Example

The following example shows how to specify that the Web Service, when invoked, runs as the `Admin` role:

```
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 {

...
```

APPENDIX C

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:

WARNING: This section applies *only* to JAX-RPC 1.1-based Web Services, and not to JAX-WS 2.0 Web Services.

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.

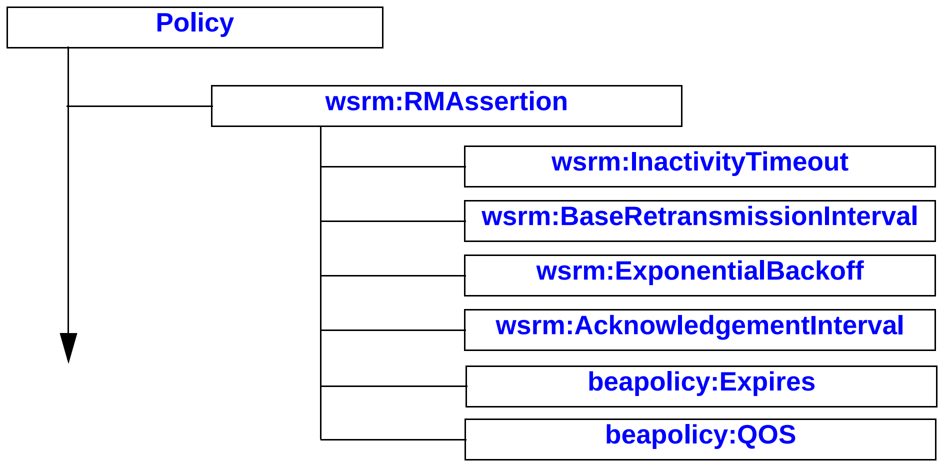
WebLogic Server includes two WS-Policy files (`DefaultReliability.xml` and `LongRunningReliability.xml`) that contain typical reliable messaging assertions that you can use if you do not want to create your own WS-Policy file. For details about these two files, see Use of WS-Policy Files for Web Service Reliable Messaging Configuration.

See Using Web Service Reliable Messaging 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 1-1 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 version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/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.

Table C-1 Attributes of <beapolicy:Expires>

Attribute	Description	Required?
Expires	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 `Expires="P3H"`.	Yes

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 C-2 Attributes of <beapolicy:QOS>

Attribute	Description	Required?
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: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-3 Attributes of <wsrm:AcknowledgementInterval>

Attribute	Description	Required?
Milliseconds	Specifies the maximum interval, in milliseconds, in which the destination endpoint must transmit a stand alone acknowledgement.	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 conjunctions 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-4 Attributes of <wsrm:BaseRetransmissionInterval>

Attribute	Description	Required?
Milliseconds	Number of milliseconds the source endpoint waits to retransmit message.	Yes.

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 C-5 Attributes of <wsrm:InactivityTimeout>

Attribute	Description	Required?
Milliseconds	The number of milliseconds that defines a period of inactivity.	Yes.

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>

Attribute	Description	Required?
optional	Specifies whether the Web Service requires the operations to be invoked reliably. Valid values for this attribute are `true` and `false`. Default value is `false`.	No.

BEA Web Services Security Policy Assertion Reference

Previous releases of WebLogic Server, released before the formulation of the OASIS WS-SecurityPolicy specification, used security policy files written under the WS-Policy specification, using a proprietary BEA schema for Web Services security policy. This release of WebLogic Server supports both security policy files that conform to the WS-SecurityPolicy 1.2 specification and the BEA Web Services security policy files first included in WebLogic Server 9. The following sections provide reference information about the security assertions you can configure in a Web Services security policy file using the proprietary BEA schema:

WARNING: This section applies *only* to JAX-RPC 1.1-based Web Services using policies written under the BEA Web Services security policy schema, and not to JAX-WS 2.0 Web Services or to policies written under the OASIS WS-SecurityPolicy 1.2 specification.

Overview of a Policy File That Contains Security Assertions

You can use 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 security policy file that is associated with a WebLogic Web Service.

A security 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 security policy files that use the BEA Web Services security policy schema (`Auth.xml`, `Sign.xml`, `Encrypt.xml`, `Wssc-dk.xml`, and `Wssc-sct.xml`). These packaged policy files contain typical security assertions that you can use if you do not want to create your own security policy file. For details about these files, see BEA Web Services Security Policy Files.

Policy files using the BEA Web Services security policy schema have the following namespace

```
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
  >
```

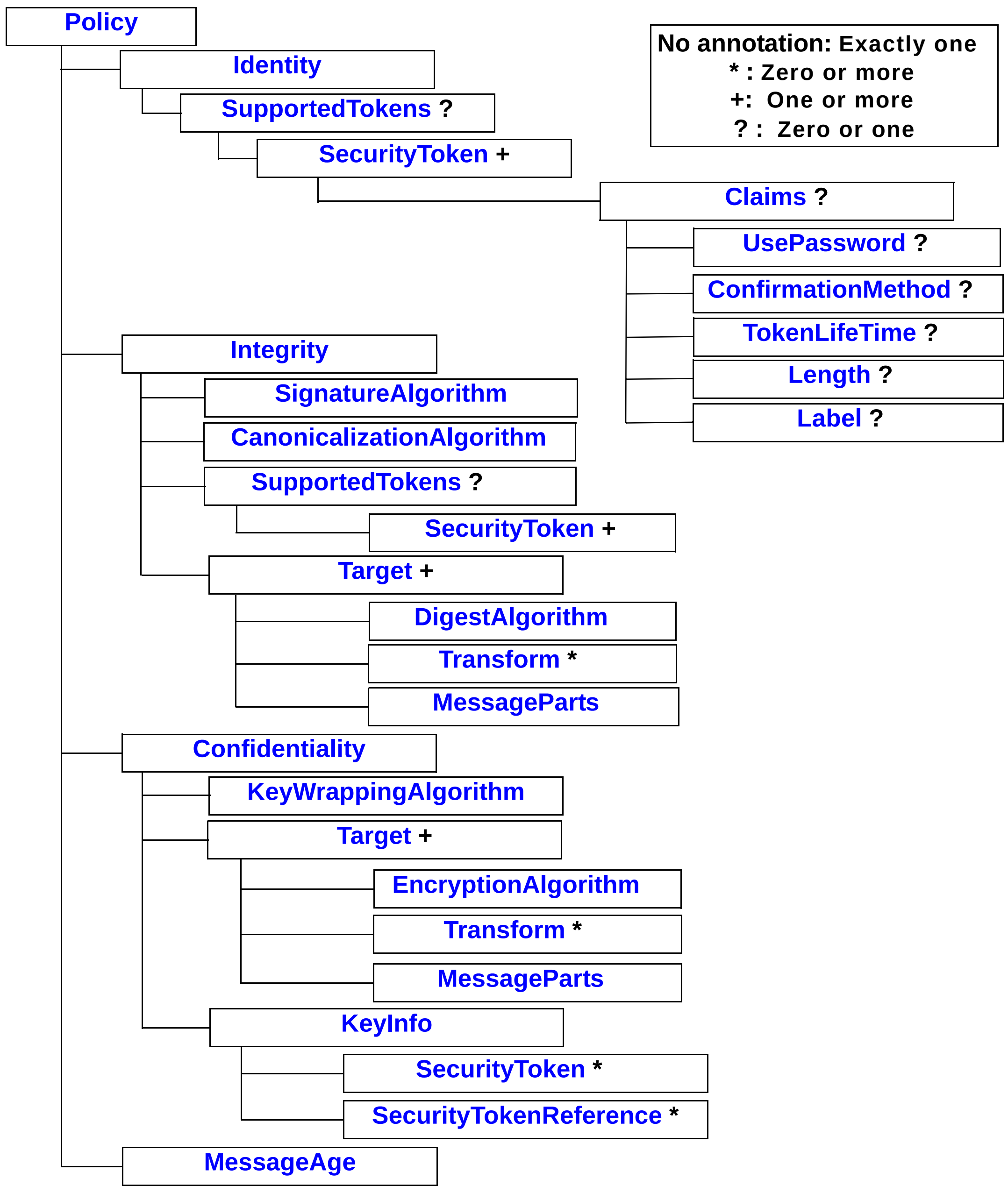
This release of WebLogic Server also includes a large number of packaged policy files that conform to the OASIS WS-SecurityPolicy 1.2 specification. WS-SecurityPolicy 1.2 policy files and BEA proprietary Web Services security policy schema files are not mutually compatible; you cannot use both types of policy file in the same Web Services security configuration. For information about using WS-SecurityPolicy 1.2 security policy files, see Using WS-SecurityPolicy Policy Files.

See Configuring Message-Level Security (Digital Signatures and Encryption) 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 BEA security policy file.

Figure 1-2 Element Hierarchy of BEA Security Policy Assertions


Policy
Identity
SupportedTokens ?
SecurityToken +
Claims ?
UsePassword ?
ConfirmationMethod ?
TokenLifeTime ?
Length ?
Label ?
Integrity
SignatureAlgorithm
CanonicalizationAlgorithm
SupportedTokens ?
SecurityToken +
Target +
DigestAlgorithm
Transform *
MessageParts
Confidentiality
KeyWrappingAlgorithm
Target +
EncryptionAlgorithm
Transform *
MessageParts
KeyInfo
SecurityToken *
SecurityTokenReference *
MessageAge
No annotation: Exactly one
* : Zero or more
+: One or more
? : Zero or one

Example of a Policy File With Security Elements

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

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/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-pro
file-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:MessageParts>
    </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:MessageParts>
    </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>

Attribute	Description	Required?
URI	The algorithm used to canonicalize the SOAP message being signed. You can specify only the following canonicalization algorithm: `http://www.w3.org/2001/10/xml-exc-cl4n#`	Yes.

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.

Table D-2 Attributes of <Confidentiality>

Attribute	Description	Required?
SupportTrust10	. The valid values for this attribute are `true` and `false`. The default value is `false`.	No.

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:

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

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"
```

```
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecur
ity-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>
```

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 version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecur
ity-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: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>

Attribute	Description	Required?
URI	The digest algorithm that is used when digitally signing the specified parts of a SOAP message. You can specify only the following digest algorithm: `http://www.w3.org/2000/09/xmldsig#sha1`	Yes.

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>

Attribute	Description	Required?

URI	The encryption algorithm used to encrypt specified parts of the SOAP message.
	Valid values are:
	`http://www.w3.org/2001/04/xmlenc#tripledes-cbc`
	`http://www.w3.org/2001/04/xmlenc#kw-tripledes`
	`http://www.w3.org/2001/04/xmlenc#aes128-cbc`
	When interoperating between Web Services built with WebLogic Workshop 8.1, you *must* specify `http://www.w3.org/2001/04/xmlenc#aes128-cbc` as the encryption algorithm.
		Yes.

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 D-5 Attributes of <Integrity>

Attribute	Description	Required?
SignToken	Specifies whether the security token, specified using the `<SecurityToken>` child element of `<Integrity>`, 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`.	No.
SupportTrust10	. The valid values for this attribute are `true` and `false`. The default value is `false`.	No.
X509AuthConditional		

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>

Attribute	Description	Required?
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 security 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 security policy file is associated. The `<Timestamp>` header indicates to the recipient of the SOAP message when the message expires.

For example, assume that your security policy file includes the following `<MessageAge>` assertion:

```
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecur
ity-utility-1.0.xsd"
  >
...
   <wssp:MessageAge Age="300" />
</wsp:Policy>
```

The resulting generated SOAP message will have a `<Timestamp>` header similar to the following excerpt:

```
<wsu:Timestamp
     wsu:Id="Dy2PFsX3ZQacqNKEANpXbNMnMhm2BmGOA2WDc2E0JpiaaTmbYNwT"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecur
ity-utility-1.0.xsd">
   <wsu:Created>2005-11-09T17:46:55Z</wsu:Created>
   <wsu:Expires>2005-11-09T17:51:55Z</wsu:Expires>
</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 D-7 Attributes of <MessageAge>

Attribute	Description	Required?
Age	Specifies the actual maximum age time-out for a SOAP message, in seconds.	No.

The following table lists the properties that describe the timestamp behavior of the WebLogic Web Services security runtime, along with their default values.

Table 1-1 Timestamp Behavior Properties

Property	Description	Default Value
Clock Synchronized	Specifies whether the Web Service assumes synchronized clocks.	true
Clock Precision	If clocks are synchronized, describes the accuracy of the synchronization. **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.	60000 milliseconds
Clock Skew	Specifies the allowable difference, in milliseconds, between the sender and receiver of the message.	60000 milliseconds
Lax Precision	Allows you to relax the enforcement of the clock precision property. **Note:** This property is deprecated as of release 9.2 of WebLogic Web Services. Use the Clock Skew property instead.	false
Max Processing Delay	Specifies the freshness policy for received messages.	-1
Validity Period	Represents the length of time the sender wants the outbound message to be valid.	60 seconds

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-22 for detailed information about using this assertion, along with a variety of examples.

Table D-8 Attributes of <MessageParts>

Attribute	Description	Required?
Dialect	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. 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-22 for examples of using these dialects.	Yes.

SecurityToken

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 is 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-7.

Table D-9 Attributes of <SecurityToken>

Attribute	Description	Required?
DerivedFromTokenType	.	No.

Table D-9 Attributes of <SecurityToken>

Attribute	Description	Required?
IncludeInMessage	Specifies whether to include the token in the SOAP message. Valid values are `true` or `false`. The default value of this attribute is `false` when used in the `<Confidentiality>` assertion and `true` when used in the `<Integrity>` assertion. The value of this attribute is *always* `true` when used in the `<Identity>` assertion, even if you explicitly set it to `false`.	No.
TokenType	Specifies the type of security token. Valid values are: • `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3` (To specify a binary X.509 token) • `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken` (To specify a username token) • `http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-saml-token-profile-1.0#SAMLAssertionID` (To specify a SAML token)	Yes.

SecurityTokenReference

For internal use only.

You should never include this security assertion in your custom security policy file; it is described in this section for informational purposes only. The WebLogic Web Services runtime automatically inserts this security assertion in the security 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 security 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-10 Attributes of <SignatureAlgorithm>

Attribute	Description	Required?

URI	Specifies the cryptographic algorithm used to compute the signature.
	Note: Be sure that you specify an algorithm that is compatible with the certificates you are using in your enterprise.
	Valid values are:
	`http://www.w3.org/2000/09/xmldsig#rsa-sha1`
	`http://www.w3.org/2000/09/xmldsig#dsa-sha1`
		Yes.

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-11 Attributes of <Target>

Attribute	Description	Required?
encryptContent Only	Specifies whether to encrypt an entire element, or just its content. This attribute can be specified only when `<Target>` is a child element of `<Confidentiality>`. Default value of this attribute is `true`, which means that only the content is encrypted.	No.

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-12 Attributes of <Transform>

Attribute	Description	Required?
URI	Specifies the URI of the transformation algorithm. Valid URIs are: • `http://www.w3.org/2000/09/xmldsig#base64` (Base64 decoding transforms) • `http://www.w3.org/TR/1999/REC-xpath-19991116` (XPath filtering) For detailed information about these transform algorithms, see XML-Signature Syntax and Processing.	Yes.

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-13 Attributes of <UsePassword>

Attribute	Description	Required?
Type	Specifies the type of password. Valid values are: • `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText` : Specifies that cleartext passwords should be used in the SOAP messages. • `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest` : Specifies that password digests should be used in the SOAP messages. **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`	Yes.

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 security 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-6 for an example of a complete security 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
- 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 security 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.

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:

```
<wssp:MessageParts
     Dialect="http://www.w3.org/TR/1999/REC-xpath-19991116"
     xmlns:n1="http://www.bea.com/foo">
```

```
        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 security 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 security 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-23 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 `http://www.bea.com/wls90/security/policy/wsee#part`.

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:

```
<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:

```
<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.

APPENDIX E

WebLogic Web Service Deployment Descriptor Element Reference

The following sections provide information about the WebLogic-specific Web Services deployment descriptor file, `weblogic-webservices.xml`:

WARNING: This section applies *only* to JAX-RPC 1.1-based Web Services, and not to JAX-WS 2.0 Web Services.

Overview of weblogic-webservices.xml

The standard Java EE deployment descriptor for Web Services is called `webservices.xml`. This file specifies the set of Web Services that are to be deployed to WebLogic Server and the dependencies they have on container resources and other services. See the Web Services XML Schema for a full description of this file.

The WebLogic equivalent to the standard Java EE `webservices.xml` deployment descriptor file is called `weblogic-webservices.xml`. This file contains WebLogic-specific information about a WebLogic Web Service, such as the URL used to invoke the deployed Web Service, and so on.

Both deployment descriptor files are located in the same location on the Java EE 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 Java EE `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 Java EE `webservices.xml` file and `<weblogic-webservices>` for the `weblogic-webservices.xml` file)

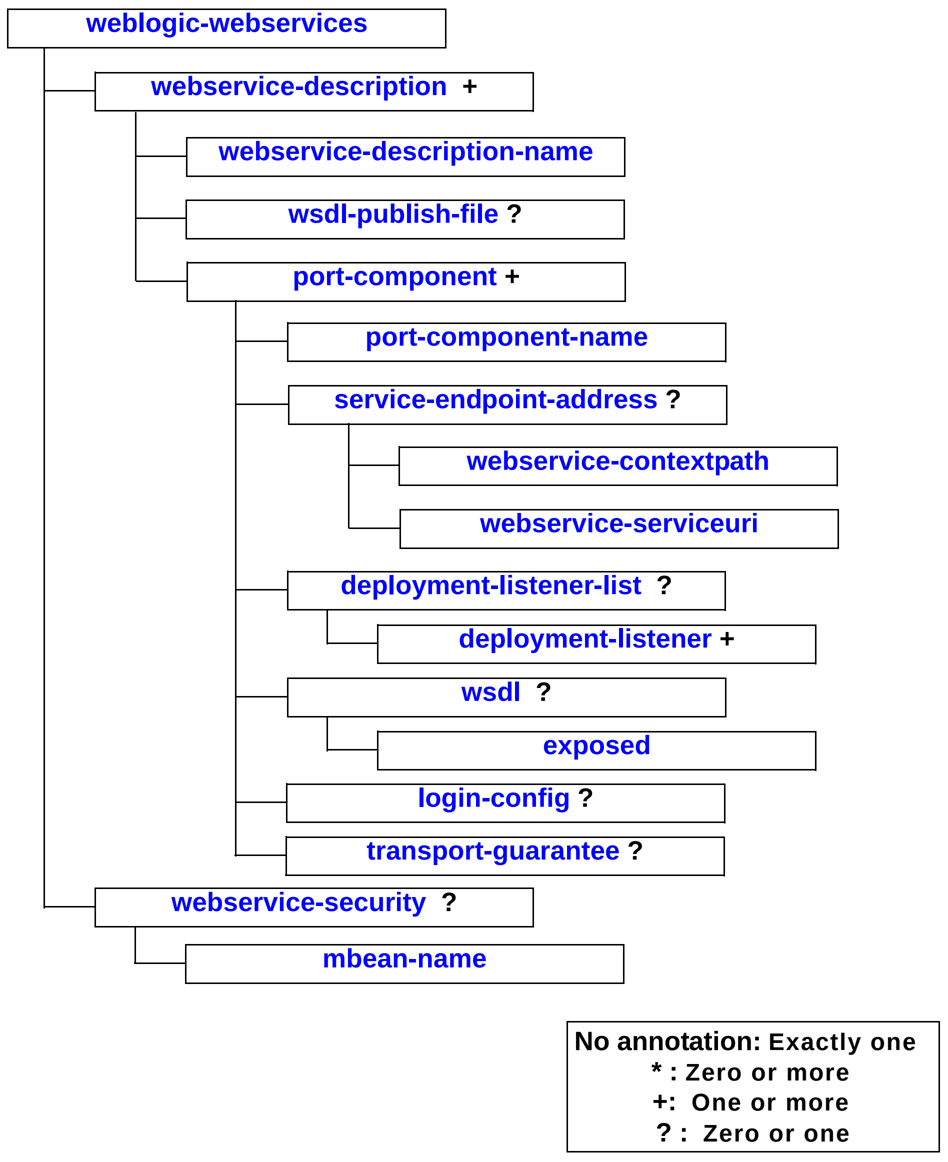
Typically users *never* need to update either deployment descriptor files, because the `jwsc` Ant task automatically generates the files for you based on the value of the JWS annotations in the JWS file that implements the Web Service. For this reason, this section is published for informational purposes only.

The data type definitions of two elements in the `weblogic-webservices.xml` file (login-config and transport-guarantee) are imported from the Java EE 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 1-3 Element Hierarchy of weblogic-webservices.xml



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 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 Java EE 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.

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 `<webserivce-serviceuri>` child 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 Java EE 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 Java EE 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.

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.