Oracle® WebLogic Server

Getting Started With WebLogic Web Services Using JAX-WS

10*g* Release 3 (10.1.3)

July 2008

Oracle WebLogic Server Getting Started With WebLogic Web Services Using JAX-WS, 10*g* Release 3 (10.1.3)

Contents

4. Programming the JWS File

Introduction

This document describes how to program WebLogic Web Services using Java API for XML-based Web Services (JAX-WS). JAX-WS is a standards-based API for coding, assembling, and deploying Java Web Services.

JAX-WS is designed to take the place of JAX-RPC in Web services and Web applications. To compare the features that are supported for JAX-WS and JAX-RPC, see "How Do I Choose Between JAX-WS and JAX-RPC?" in *Introducing WebLogic Web Services*. For information about migrating a JAX-RPC Web Service to JAX-WS, see "Migrating JAX-RPC Web Services and Clients to JAX-WS" on page 8-1.

The following table summarizes the contents of this guide.

Table 1-1 Content Summary

This section . . .	Describes how to . . .
Use Cases and Examples	Run common use cases and examples.
Developing WebLogic Web Services	Develop Web Services using the WebLogic development environment.
Programming the JWS File	Program the JWS file that implements your Web Service.
Using JAXB Data Binding	Use the Java Architecture for XML Binding (JAXB) data binding.
Invoking Web Services	Invoke your Web Service from a stand-alone client or another Web Service.

Table 1-1 Content Summary (Continued)

This section . . .	Describes how to . . .
Administering Web Services	Administer WebLogic Web Services using the Administration Console.
Migrating JAX-RPC Web Services and Clients to JAX-WS	Migrate a JAX-RPC Web Service to JAX-WS.

For an overview of WebLogic Web Services, standards, samples, and related documentation, see *Introducing WebLogic Web Services*.

JAX-WS supports Web Services Security (WS-Security) 1.1 (except for WS-Secure Conversation). For information about WebLogic Web Service security, see *Securing WebLogic Web Services*.

A Note About Upgrading Existing WebLogic Web Services

There are no steps required to upgrade a 10.0 WebLogic Web Service to 10g Release 3 (10.1.3); you can redeploy a 10.0 Web Service to WebLogic Server 10g Release 3 (10.1.3) without making any changes or recompiling it.

Use Cases and Examples

The following sections describe common Web Service use cases and examples:

- "Creating a Simple HelloWorld Web Service" on page 2-1
- "Creating a Web Service With User-Defined Data Types" on page 2-7
- "Creating a Web Service from a WSDL File" on page 2-16
- "Invoking a Web Service from a Stand-alone Java Client" on page 2-25
- "Invoking a Web Service from a WebLogic Web Service" on page 2-30

Each use case provides step-by-step procedures for creating simple WebLogic Web Services and invoking an operation from a deployed Web Service. The examples include basic Java code and Ant `build.xml` files that you can use in your own development environment to recreate the example, or by following the instructions to create and run the examples in an environment that is separate from your development environment.

The use cases do not go into detail about the processes and tools used in the examples; later chapters are referenced for more detail.

Creating a Simple HelloWorld Web Service

This section describes how to create a very simple Web Service that contains a single operation. The *Java Web Service (JWS)* file that implements the Web Service uses just the one required *JWS annotation*: `@WebService`. A JWS file is a standard Java file that uses JWS metadata annotations to specify the shape of the Web Service. Metadata annotations were introduced with JDK 5.0, and the set of annotations used to annotate Web Service files are called JWS annotations. WebLogic Web Services use standard JWS annotations. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference*.

The following example shows how to create a Web Service called `HelloWorldService` that includes a single operation, `sayHelloWorld`. For simplicity, the operation returns the inputted String value.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.

2. Create a project directory, as follows:

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

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

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

4. Create the JWS file that implements the Web Service.

   Open your favorite Java IDE or text editor and create a Java file called `HelloWorldImpl.java` using the Java code specified in "Sample HelloWorldImpl.java JWS File" on page 2-4.

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

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

6. Create a standard Ant `build.xml` file in the project directory (`myExamples/hello_world/src`) and add a `taskdef` Ant task to specify the full Java classname of the `jwsc` task:

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

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

</project>
```

See “Sample Ant Build File for HelloWorldImpl.java” on page 2-5 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

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

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

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

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

    </jwsc>

  </target>
```

The `jwsc` WebLogic Web Service Ant task generates the supporting artifacts, compiles the user-created and generated Java code, and archives all the artifacts into an Enterprise Application EAR file that you later deploy to WebLogic Server.

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

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

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

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

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

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

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

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

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

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

```
prompt> ant deploy
```

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

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

    You construct the URL using the default values for the `contextPath` and `serviceUri` attributes. The default value for the `contextPath` is the name of the Java class in the JWS file. The default value of the `serviceURI` attribute is the `serviceName` element of the `@WebService` annotation if specified. Otherwise, the name of the JWS file, without its extension, followed by `Service`. For example, if the `serviceName` element of the `@WebService` annotation is not specified and the name of the JWS file is `HelloWorldImpl.java`, then the default value of its `serviceUri` is `HelloWorldImplService`.

    These attributes will be set explicitly in the next example, "Creating a Web Service With User-Defined Data Types" on page 2-7. Use the hostname and port relevant to your WebLogic Server instance.

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

To run the Web Service, you need to create a client that invokes it. See "Invoking a Web Service from a Stand-alone Java Client" on page 2-25 for an example of creating a Java client application that invokes a Web Service.

Sample HelloWorldImpl.java JWS File

```
package examples.webservices.hello_world;

// Import the @WebService annotation

import javax.jws.WebService;
```

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

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

public class HelloWorldImpl {
  // By default, all public methods are exposed as Web Services operation
  public String sayHelloWorld(String message) {
  try {
    System.out.println("sayHelloWorld:" + message);
  } catch (Exception ex) { ex.printStackTrace(); }

    return "Here is the message: '" + message + "'";
  }
}
```

Sample Ant Build File for HelloWorldImpl.java

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

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

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

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

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

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

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

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

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

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

<target name="build-service">

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

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

  </jwsc>

</target>

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

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

<target name="client">

  <clientgen
```

```
wsdl="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldService?
WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.hello_world.client"
      type="JAXWS"/>

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

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

  </target>

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

line="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldService"
/>
    </java> </target>

</project>
```

Creating a Web Service With User-Defined Data Types

The preceding use case uses only a simple data type, `String`, as the parameter and return value of the Web Service operation. This next example shows how to create a Web Service that uses a user-defined data type, in particular a JavaBean called `BasicStruct`, as both a parameter and a return value of its operation.

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

The following procedure is very similar to the procedure in "Creating a Simple HelloWorld Web Service" on page 2-1. For this reason, although the procedure does show all the needed steps, it provides details only for those steps that differ from the simple HelloWorld example.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.

2. Create a project directory:

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

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

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

4. Create the source for the `BasicStruct` JavaBean.

   Open your favorite Java IDE or text editor and create a Java file called `BasicStruct.java`, in the project directory, using the Java code specified in "Sample BasicStruct JavaBean" on page 2-11.

5. Save the `BasicStruct.java` file in the `src/examples/webservices/complex` subdirectory of the project directory.

6. Create the JWS file that implements the Web Service using the Java code specified in "Sample ComplexImpl.java JWS File" on page 2-12.

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

   For more in-depth information about creating a JWS file, see Chapter 4, "Programming the JWS File."

7. Save the `ComplexImpl.java` file in the `src/examples/webservices/complex` subdirectory of the project directory.

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

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

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

</project>
```

   See “Sample Ant Build File for ComplexImpl.java JWS File” on page 2-13 for a full sample `build.xml` file.

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

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

  <jwsc
    srcdir="src"
    destdir="output/ComplexServiceEar" >
    <jws file="examples/webservices/complex/ComplexImpl.java"
       type="JAXWS">

       <WLHttpTransport
        contextPath="complex" serviceUri="ComplexService"
        portName="ComplexServicePort"/>
     </jws>
    </jwsc>
</target>
```

   In the preceding example:

   – The `type` attribute of the `<jws>` element specifies the type of Web Service (JAX-WS or JAX-RPC).

   – The `<WLHttpTransport>` child element of the `<jws>` element of the `jwsc` Ant task 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. For more information about defining the context path, see “Defining the Context Path of a WebLogic Web Service” in *WebLogic Web Services Reference*.

10. Execute the `jwsc` Ant task:

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

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

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

12. Deploy the Web Service, packaged in the `ComplexServiceEar` Enterprise Application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. For example:

```
prompt> ant deploy
```

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

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

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

<target name="deploy">

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

</target>
```

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

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

```
prompt> ant deploy
```

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

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

To run the Web Service, you need to create a client that invokes it. See "Invoking a Web Service from a Stand-alone Java Client" on page 2-25 for an example of creating a Java client application that invokes a Web Service.

Sample BasicStruct JavaBean

```
package examples.webservices.complex;

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

public class BasicStruct {

  // Properties

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

  // Getter and setter methods

  public int getIntValue() {
    return intValue;
  }

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

  public String getStringValue() {
    return stringValue;
  }

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

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

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

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

Sample ComplexImpl.java JWS File

```
package examples.webservices.complex;

// Import the standard JWS annotation interfaces

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

// Import the 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)

/**
 * 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.
 */

public class ComplexImpl {

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

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

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

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

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

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

Sample Ant Build File for ComplexImpl.java JWS File

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

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

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

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

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

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

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

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

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

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

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

<target name="build-service">

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

       <WLHttpTransport
        contextPath="complex" serviceUri="ComplexService"
        portName="ComplexServicePort"/>
    </jws>

  </jwsc>

</target>
```

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

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

  <target name="client">

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

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

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

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

```
    </java>
  </target>

</project>
```

Creating a Web Service from a WSDL File

Another common use case of creating a Web Service is to start from an existing WSDL file, often referred to as the *golden WSDL*. A WSDL file is a public contract that specifies what the Web Service looks like, such as the list of supported operations, the signature and shape of each operation, the protocols and transports that can be used when invoking the operations, and the XML Schema data types that are used when transporting the data. Based on this WSDL file, you generate the artifacts that implement the Web Service so that it can be deployed to WebLogic Server. You use the `wsdlc` Ant task to generate the following artifacts.

- JWS service endpoint interface (SEI) that implements the Web Service described by the WSDL file.
- JWS implementation file that contains a partial (stubbed-out) implementation of the generated JWS SEI. This file must be customized by the developer.
- JAXB data binding artifacts.
- Optional Javadocs for the generated JWS SEI.

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

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

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

The following simple example shows how to create a Web Service from the WSDL file shown in "Sample WSDL File" on page 2-20. The Web Service has one operation, `getTemp`, that returns a temperature when passed a zip code.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.

2. Create a working directory:

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

3. Put your WSDL file into an accessible directory on your computer.

   For the purposes of this example, it is assumed that your WSDL file is called `TemperatureService.wsdl` and is located in the `/myExamples/wsdlc/wsdl_files` directory. See "Sample WSDL File" on page 2-20 for a full listing of the file.

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

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

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

   </project>
   ```

   See "Sample Ant Build File for TemperatureService" on page 2-22 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

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

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

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

     </target>
   ```

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

JWS SEI into the `output/impl/examples/webservices/wsdlc` directory (which is a combination of the output directory specified by `destImplDir` and the directory hierarchy specified by the package name). All generated JWS files will be packaged in the `examples.webservices.wsdlc` package.

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

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

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

7. Update the generated `output/impl/examples/webservices/wsdlc/TemperaturePortTypeImpl.java` JWS implementation file using your favorite Java IDE or text editor to add Java code to the methods so that they behave as you want.

   See "Sample TemperaturePortType Java Implementation File" on page 2-21 for an example; the added Java code is in **bold**. The generated JWS implementation file automatically includes values for the `@WebService` JWS annotation that corresponds to the value in the original WSDL file.

   Note: There are restrictions on the JWS annotations that you can add to the JWS implementation file in the "starting from WSDL" use case. See "wsdlc" in the *WebLogic Web Services Reference* for details.

   For simplicity, the sample `getTemp()` method in `TemperaturePortTypeImpl.java` returns a hard-coded number. In real life, the implementation of this method would actually look up the current temperature at the given zip code.

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

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

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

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

   <target name="build-service">
  <jwsc
    srcdir="src"
    destdir="${ear-dir}">

    <jws file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
         compiledWsdl="${compiledWsdl-dir}/TemperatureService_wsdl.jar"
         type="JAXWS">
       <WLHttpTransport
        contextPath="temp" serviceUri="TemperatureService"
        portName="TemperaturePort">
       </WLHttpTransport>

    </jws>

  </jwsc>

   </target>
```

In the preceding example:

– The `type` attribute of the `<jws>` element specifies the type of Web Services (JAX-WS or JAX-RPC).

– The `<WLHttpTransport>` child element of the `<jws>` element of the `jwsc` Ant task 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.

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

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

    You can re-run this target if you want to update and then re-build the JWS file.

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

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

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

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

<target name="deploy">

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

</target>
```

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

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

```
prompt> ant deploy
```

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

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

    The context path and service URI section of the preceding URL are specified by the original golden WSDL. Use the hostname and port relevant to your WebLogic Server instance. Note that the deployed and original WSDL files are the same, except for the host and port of the endpoint address.

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

To run the Web Service, you need to create a client that invokes it. See "Invoking a Web Service from a Stand-alone Java Client" on page 2-25 for an example of creating a Java client application that invokes a Web Service.

Sample WSDL File

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

<definitions
   name="TemperatureService"
   targetNamespace="http://www.bea.com/wls103"
   xmlns:tns="http://www.bea.com/wls103"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/" >
```

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

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

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

        <binding name="TemperatureBinding" type="tns:TemperaturePortType">
                <soap:binding style="document"
                             transport="http://schemas.xmlsoap.org/soap/http"/>
                <operation name="getTemp">
                        <soap:operation soapAction=""/>
                        <input>
                                <soap:body use="literal"
                                 namespace="http://www.bea.com/wls103" />
                        </input>
                        <output>
                                <soap:body use="literal"
                                 namespace="http://www.bea.com/wls103" />
                        </output>
                </operation>
        </binding>

        <service name="TemperatureService">
            <documentation>
                Returns current temperature in a given U.S. zipcode
            </documentation>
            <port name="TemperaturePort" binding="tns:TemperatureBinding">
               <soap:address
                     location="http://localhost:7001/temp/TemperatureService"/>
            </port>
        </service>

</definitions>
```

Sample TemperaturePortType Java Implementation File

```
package examples.webservices.wsdlc;

import javax.jws.WebService;
import javax.xml.ws.BindingType;
```

```
/**
 * examples.webservices.wsdlc.TemperatureServiceImpl class implements web
 * service endpoint interface
 * examples.webservices.wsdlc.TemperaturePortType */

@WebService(
  portName="TemperaturePort"
  serviceName="TemperatureService",
  targetNamespace="http://www.bea.com/wls103"
  endpointInterface="examples.webservices.wsdlc.TemperaturePortType"
  wsdlLocation="/wsdls/TemperatureServices.wsdl")

@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")

public class TemperaturePortTypeImpl implements
examples.webservices.wsdlc.TemperaturePortType {

  public TemperaturePortTypeImpl() { }

  public float getTemp(java.lang.String zip) {
     return 1.234f;
  }
}
```

Sample Ant Build File for TemperatureService

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

```
<project default="all">

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

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

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

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

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

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

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

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

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

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

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

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

  </target>

  <target name="build-service">

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

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

```
         <WLHttpTransport
          contextPath="temp" serviceUri="TemperatureService"
          portName="TemperaturePort"/>
      </jws>

    </jwsc>

  </target>

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

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

  <target name="client">

    <clientgen

wsdl="http://${wls.hostname}:${wls.port}/temp/TemperatureService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.wsdlc.client"
      type="JAXWS">
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>

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

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

  </target>

</project>
```

Invoking a Web Service from a Stand-alone Java Client

When you invoke an operation of a deployed Web Service from a client application, the Web Service could be deployed to WebLogic Server or to any other application server, such as .NET. All you need to know is the URL to its public contract file, or WSDL.

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

- The Java class for the `Service` interface implementation for the particular Web Service you want to invoke.
- JAXB data binding artifacts.
- The Java class for any user-defined XML Schema data types included in the WSDL file.

The following example shows how to create a Java client application that invokes the `echoComplexType` operation of the `ComplexService` WebLogic Web Service described in "Creating a Web Service With User-Defined Data Types" on page 2-7. The `echoComplexType` operation takes as both a parameter and return type the `BasicStruct` user-defined data type.

Note: It is assumed in this procedure that you have created and deployed the `ComplexService` Web Service.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is

BEA_HOME/user_projects/domains/*domainName*, where *BEA_HOME* is the top-level installation directory of the Oracle products and *domainName* is the name of your domain.

2. Create a project directory:

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

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

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

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

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

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

</project>
```

See "Sample Ant Build File For Building Stand-alone Client Application" on page 2-29 for a full sample `build.xml` file. The full `build.xml` file uses properties, such as `${clientclass-dir}`, rather than always using the hard-coded name output directory for client classes.

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

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

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

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

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

</target>
```

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

In this example, the package name is set to the same package name as the client application, `examples.webservices.simple_client`. If you set the package name to one that is different from the client application, you would need to import the appropriate class files. For example, if you defined the package name as `examples.webservices.complex`, you would need to import the following class files in the client application:

```
import examples.webservices.complex.BasicStruct;
import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;
```

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

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

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

6. Create the Java client application file that invokes the `echoComplexType` operation.

   Open your favorite Java IDE or text editor and create a Java file called `Main.java` using the code specified in "Sample Java Client Application" on page 2-28.

   The application follows standard JAX-WS guidelines to invoke an operation of the Web Service using the Web Service-specific implementation of the `Service` interface generated by `clientgen`. For details, see Chapter 6, "Invoking Web Services."

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

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

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

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

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

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

   <target name="run" >

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

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

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

    ```
      prompt> ant run
    ```

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

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

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

Sample Java Client Application

The following provides a simple Java client application that invokes the `echoComplexType` operation. Because the `<clientgen>` `packageName` attribute was set to the same package name as the client application, we are not required to import the `<clientgen>`-generated files.

```
package examples.webservices.simple_client;

/**
 * This is a simple stand-alone client application that invokes the
 * echoComplexType operation of the ComplexService Web service.
 */

public class Main {

  public static void main(String[] args) {

    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();
```

```
    BasicStruct in = new BasicStruct();

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

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

Sample Ant Build File For Building Stand-alone Client Application

The following `build.xml` file defines tasks to build the stand-alone client application. The example uses properties to simplify the file.

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

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

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

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

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

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

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

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

  <target name="build-client">

    <clientgen
     type="JAXWS"
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
```

```
      destDir="${clientclass-dir}"
      packageName="examples.webservices.simple_client"/>

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

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

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

</project>
```

Invoking a Web Service from a WebLogic Web Service

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

The procedure is similar to that described in "Invoking a Web Service from a Stand-alone Java Client" on page 2-25 except that instead of running the `clientgen` Ant task to generate the client stubs, you use the `<clientgen>` child element of `<jws>`, inside of the `jwsc` Ant task. The `jwsc` Ant task automatically packages the generated client stubs in the invoking Web Service WAR file so that the Web Service has immediate access to them. You then follow standard JAX-WS programming guidelines in the JWS file that implements the Web Service that invokes the other Web Service.

The following example shows how to write a JWS file that invokes the `echoComplexType` operation of the `ComplexService` Web Service described in "Creating a Web Service With User-Defined Data Types" on page 2-7.

Note: It is assumed that you have successfully deployed the `ComplexService` Web Service.

1. Set your WebLogic Server environment.

Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.

2. Create a project directory:

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

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

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

4. Create the JWS file that implements the Web Service that invokes the `ComplexService` Web Service.

   Open your favorite Java IDE or text editor and create a Java file called `ClientServiceImpl.java` using the Java code specified in “Sample ClientServiceImpl.java JWS File” on page 2-33.

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

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

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

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

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

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

   </project>
   ```

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

See "Sample Ant Build File For Building ClientService" on page 2-34 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, `deploy`, `undeploy`, `client`, and `run`. The full `build.xml` file also uses properties, such as `${ear-dir}`, rather than always using the hard-coded name for the EAR directory.

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

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

  <jwsc
    srcdir="src"
    destdir="output/ClientServiceEar" >
    <jws

file="examples/webservices/service_to_service/ClientServiceImpl.java"
     type="JAXWS">

     <WLHttpTransport
      contextPath="ClientService" serviceUri="ClientService"
      portName="ClientServicePort"/>

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

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

In this example, the package name is set to `examples.webservices.complex`, which is different from the client application package name, `examples.webservices.simple_client`. As a result, you need to import the appropriate class files in the client application:

```
import examples.webservices.complex.BasicStruct;
import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;
```

If the package name is set to the same package name as the client application, the import calls would be optional.

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

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

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

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

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

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

    <target name="deploy">

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

    </target>
    ```

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

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

    ```
    prompt> ant deploy
    ```

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

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

See "Invoking a Web Service from a Stand-alone Java Client" on page 2-25 for an example of creating a Java client application that invokes a Web Service.

Sample ClientServiceImpl.java JWS File

The following provides a simple Web Service client application that invokes the `echoComplexType` operation.

```
package examples.webservices.service_to_service;

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

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

import examples.webservices.complex.BasicStruct;
```

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

import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;

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

public class ClientServiceImpl {

  @WebMethod()
  public String callComplexService(BasicStruct input)
  {

    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();

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

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

  }
}
```

Sample Ant Build File For Building ClientService

The following `build.xml` file defines tasks to build the client application. The example uses properties to simplify the file.

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

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

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

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

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

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

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

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

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

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

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

<target name="build-service">

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

     <jws
       file="examples/webservices/service_to_service/ClientServiceImpl.java"
       type="JAXWS">

       <WLHttpTransport
        contextPath="ClientService" serviceUri="ClientService"
        portName="ClientServicePort"/>

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

  </jwsc>

</target>

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

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

  <target name="client">

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

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

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

  </target>

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

  </target>

</project>
```

CHAPTER 3

Developing WebLogic Web Services

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

Overview of the WebLogic Web Service Programming Model

The WebLogic Web Services programming model centers around *JWS files*—Java files that use *JWS annotations* to specify the shape and behavior of the Web Service—and Ant tasks that execute on the JWS file. JWS annotations are based on the metadata feature, introduced in Version 5.0 of the JDK (specified by JSR-175), and include standard annotations defined by the Web Services Metadata for the Java Platform specification (JSR-181), the JAX-WS specification (JSR-224), as well as additional ones. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference*. For additional detailed information about this programming model, see "Anatomy of a WebLogic Web Service" in *Introducing WebLogic Web Services*.

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

- "Developing WebLogic Web Services Starting From Java: Main Steps" on page 3-2
- "Developing WebLogic Web Services Starting From a WSDL File: Main Steps" on page 3-4

Iterative development refers to setting up your development environment in such a way so that you can repeatedly code, compile, package, deploy, and test a Web Service until it works as you want. The WebLogic Web Service programming model uses Ant tasks to perform most of the steps of the iterative development process. Typically, you create a single `build.xml` file that contains targets for all the steps, then repeatedly run the targets, after you have updated your JWS file with new Java code, to test that the updates work as you expect.

Developing WebLogic Web Services Starting From Java: Main Steps

This section describes the general procedure for developing WebLogic Web Services starting from Java—in effect, coding the JWS file from scratch and later generating the WSDL file that describes the service. See "Use Cases and Examples" on page 2-1 for specific examples of this process.

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

Note: This procedure does not use the WebLogic Web Services split development directory environment. If you are using this development environment, and would like to integrate Web Services development into it, see "Integrating Web Services Into the WebLogic Split Development Directory Environment" on page 3-21 for details.

Table 3-1 Steps to Develop Web Services Starting From Java

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.
2	Create a project directory.	The project directory will contain the JWS file, Java source for any user-defined data types, and the Ant `build.xml` file. You can name the project directory anything you want.
3	Create the JWS file that implements the Web Service.	See "Programming the JWS File" on page 4-1.
4	Create user-defined data types. (Optional)	If your Web Service uses user-defined data types, create the JavaBeans that describes them. See "Programming the User-Defined Java Data Type" on page 4-20.
5	Create a basic Ant build file, `build.xml`.	See "Creating the Basic Ant build.xml File" on page 3-6.
6	Run the `jwsc` Ant task against the JWS file.	The `jwsc` Ant task generates source code, data binding artifacts, deployment descriptors, and so on, into an output directory. The `jwsc` Ant task generates an Enterprise application directory structure at this output directory; later you deploy this exploded directory to WebLogic Server as part of the iterative development process. See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.
7	Deploy the Web Service to WebLogic Server.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.
8	Browse to the WSDL of the Web Service.	Browse to the WSDL of the Web Service to ensure that it was deployed correctly. See "Browsing to the WSDL of the Web Service" on page 3-17.

Table 3-1 Steps to Develop Web Services Starting From Java (Continued)

#	Step	Description
9	Test the Web Service.	See "Testing the Web Service" on page 3-20.
10	Edit the Web Service. (Optional)	To make changes to the Web Service, update the JWS file, undeploy the Web Service as described in "Deploying and Undeploying WebLogic Web Services" on page 3-14, then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See "Invoking Web Services" on page 6-1 for information on writing client applications that invoke a Web Service.

Developing WebLogic Web Services Starting From a WSDL File: Main Steps

This section describes the general procedure for developing WebLogic Web Services based on an existing WSDL file. See "Developing WebLogic Web Services" on page 3-1 for a specific example of this process.

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

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

Note: This procedure does not use the WebLogic Web Services split development directory environment. If you are using this development environment, and would like to integrate Web Services development into it, see "Integrating Web Services Into the WebLogic Split Development Directory Environment" on page 3-21 for details.

Table 3-2 Steps to Develop Web Services Starting From Java

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.
2	Create a project directory.	The project directory will contain the generated artifacts and the Ant `build.xml` file.
3	Create a basic Ant build file, `build.xml`.	See "Creating the Basic Ant build.xml File" on page 3-6.
4	Put your WSDL file in a directory that the `build.xml` Ant build file is able to read.	For example, you can put the WSDL file in a `wsdl_files` child directory of the project directory.
5	Run the `wsdlc` Ant task against the WSDL file.	The `wsdlc` Ant task generates the JWS service endpoint interface (SEI), the stubbed-out JWS class file, JavaBeans that represent the XML Schema data types, and so on, into output directories. See "Running the wsdlc WebLogic Web Services Ant Task" on page 3-10.
6	Update the stubbed-out JWS file generated by the `wsdlc` Ant task.	The `wsdlc` Ant task generates a stubbed-out JWS file. You need to add your business code to the Web Service so it behaves as you want. See "Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc" on page 3-13.
7	Run the `jwsc` Ant task against the JWS file.	Specify the artifacts generated by the `wsdlc` Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web Service. See "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.
8	Deploy the Web Service to WebLogic Server.	See "Deploying and Undeploying WebLogic Web Services" on page 3-14.
9	Browse to the WSDL of the Web Service.	Browse to the WSDL of the Web Service to ensure that it was deployed correctly. See "Browsing to the WSDL of the Web Service" on page 3-17.

Table 3-2 Steps to Develop Web Services Starting From Java (Continued)

#	Step	Description
10	Test the Web Service.	See "Testing the Web Service" on page 3-20.
11	Edit the Web Service. (Optional)	To make changes to the Web Service, update the JWS file, undeploy the Web Service as described in "Deploying and Undeploying WebLogic Web Services" on page 3-14, then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See "Invoking Web Services" on page 6-1 for information on writing client applications that invoke a Web Service.

Creating the Basic Ant build.xml File

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

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

```
<project default="all">

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

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

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

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

```
</project>
```

Running the jwsc WebLogic Web Services Ant Task

The `jwsc` Ant task takes as input a JWS file that contains JWS annotations and generates all the artifacts you need to create a WebLogic Web Service. The JWS file can be either one you coded yourself from scratch or one generated by the `wsdlc` Ant task. The `jwsc`-generated artifacts include:

- JSR-109 Web Service class file.
- JAXB data binding artifact class file.
- All required deployment descriptors, including:
  - Servlet-based Web Service deployment descriptor file: `web.xml`.
  - Ear deployment descriptor files: `application.xml` and `weblogic-application.xml`.

  Note: No EJB deployment descriptors are required for EJB 3.0-based Web Services.

Note: The WSDL file is generated when the service endpoint is deployed.

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

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

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

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

<target name="build-service">

    <jwsc
      srcdir="src_directory"
      destdir="ear_directory"
```

```
        >
        <jws file="JWS_file"
             compiledWsdl="WSDLC_Generated_JAR"
             type="WebService_type"/>
      </jwsc>
    </target>
```

where:

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

  Note: You specify this attribute only in the "starting from WSDL" use case; this procedure is described in "Developing WebLogic Web Services Starting From a WSDL File: Main Steps" on page 3-4.

- *WebService_type* specifies the type of Web Service. This value can be set to JAXWS or JAXRPC.

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

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

Examples of Using jwsc

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

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

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

    <jwsc
      srcdir="src"
      destdir="output/helloWorldEar">
      <jws
         file="examples/webservices/hello_world/HelloWorldImpl.java"
         type="JAXWS"/>

    </jwsc>

</target>
```

In the example:

- The Enterprise application will be generated, in exploded form, in `output/helloWorldEar`, relative to the current directory.
- The JWS file is called `HelloWorldImpl.java`, and is located in the `src/examples/webservices/hello_world` directory, relative to the current directory. This implies that the JWS file is in the package `examples.webservices.helloWorld`.
- A JAX-WS Web Service is generated.

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

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

<target name="build-service">

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

    <jws

        file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
        compiledWsdl="output/compiledWsdl/TemperatureService_wsdl.jar"
        type="JAXWS"/>

  </jwsc>

</target>
```

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

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

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

Advanced Uses of jwsc

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

- Process multiple JWS files at once. You can choose to package each resulting Web Service into its own Web application WAR file, or group all of the Web Services into a single WAR file.
- Specify the transports (HTTP/HTTPS) that client applications can use when invoking the Web Service.
- Update an existing Enterprise Application or Web application, rather than generate a completely new one.

See "jwsc" in the *WebLogic Web Services Reference* for complete documentation and examples about the `jwsc` Ant task.

Running the wsdlc WebLogic Web Services Ant Task

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

- JWS service endpoint interface (SEI) that implements the Web Service described by the WSDL file.
- JWS implementation file that contains a partial (stubbed-out) implementation of the generated JWS SEI. This file must be customized by the developer.
- JAXB data binding artifacts.
- Optional Javadocs for the generated JWS SEI.

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

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

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

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

  <wsdlc
      srcWsdl="WSDL_file"
      destJwsDir="JWS_interface_directory"
      destImplDir="JWS_implementation_directory"
      packageName="Package_name"
      type="WebService_type"/>
</target>
```

where:

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

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

  The name of the generated JWS file is *`Service_PortType`*`Impl.java`, where *`Service`* and *`PortType`* refer to the `name` attribute of the `<service>` element and its inner `<port>` element, respectively, in the WSDL file for which you are generating a Web Service. For example, if the service name is `MyService` and the port name is `MyServicePortType`, then the JWS implementation file is called `MyService_MyServicePortTypeImpl.java`.
- *`Package_name`* refers to the package into which the generated JWS SEI 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.

- `WebService_type` specifies the type of Web Service. This value can be set to JAXWS or JAXRPC.

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

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

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

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

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

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

</target>
```

In the example:

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

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

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

See "wsdlc" in *WebLogic Web Services Reference* for more information.

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

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

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

The `wsdlc` Ant task automatically includes the `@WebService` annotation in the JWS implementation class; the value corresponds to the equivalent value in the WSDL. For example, the `serviceName` attribute of `@WebService` is the same as the `name` attribute of the `<service>` element in the WSDL file.

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

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

In addition, you can add additional JWS annotations to the file, with the following restrictions:

- You can include the following annotations from the standard (JSR-181) `javax.jws` package in the JWS implementation file: `@WebService`, `@HandlerChain`, `@SOAPMessageHandler`, and `@SOAPMessageHandlers`. If you specify any other JWS annotation from the `javax.jws` package, the `jwsc` Ant task returns error when you try to compile the JWS file into a Web Service.
- You can specify *only* the `serviceName` and `endpointInterface` attributes of the `@WebService` annotation. Use the `serviceName` attribute to specify a different `<service>` WSDL element from the one that the `wsdlc` Ant task used, in the rare case that the WSDL

file contains more than one `<service>` element. Use the `endpointInterface` attribute to specify the JWS SEI generated by the `wsdlc` Ant task.

- You can specify JAX-WS (JSR 224), JAXB (JSR 222), or Common (JSR 250) annotations, as required.

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

The following example shows the `wsdlc`-generated JWS implementation file from the WSDL shown in "Sample WSDL File" on page 2-20; the text in **bold** indicates where you would add Java code to implement the single operation (`getTemp`) of the Web Service:

```
package examples.webservices.wsdlc;

import javax.jws.WebService;
/**
 * TemperaturePortTypeImpl class implements web service endpoint interface
 * TemperaturePortType */

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

public class TemperaturePortTypeImpl implements TemperaturePortType {

  public TemperaturePortTypeImpl() {

  }

  public float getTemp(java.lang.String zipcode)

  {

    //replace with your impl here

     return 0;

  }

}
```

Deploying and Undeploying WebLogic Web Services

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

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

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

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

Using the wldeploy Ant Task to Deploy Web Services

The easiest way to deploy a Web Service as part of the iterative development process is to add a target that executes the `wldeploy` WebLogic Ant task to the same `build.xml` file that contains the `jwsc` Ant task. You can add tasks to both deploy and undeploy the Web Service so that as you add more Java code and regenerate the service, you can redeploy and test it iteratively.

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

```
<target name="deploy">

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

</target>
```

where:

- *DeploymentName* refers to the deployment name of the Enterprise Application, or the name that appears in the Administration Console under the list of deployments.
- *Source* refers to the name of the Enterprise Application EAR file or exploded directory that is being deployed. By default, the `jwsc` Ant task generates an exploded Enterprise Application directory.
- *AdminUser* refers to administrative username.
- *AdminPassword* refers to the administrative password.

- *AdminServerURL* refers to the URL of the Administration Server, typically `t3://localhost:7001`.
- *ServerName* refers to the name of the WebLogic Server instance to which you are deploying the Web Service.

For example, the following `wldeploy` task specifies that the Enterprise Application exploded directory, located in the `output/ComplexServiceEar` directory relative to the current directory, be deployed to the `myServer` WebLogic Server instance. Its deployed name is `ComplexServiceEar`.

```
<target name="deploy">

  <wldeploy action="deploy"
    name="ComplexServiceEar"
    source="output/ComplexServiceEar" user="weblogic"
    password="weblogic" verbose="true"
    adminurl="t3://localhost:7001"
    targets="myserver"/>

</target>
```

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

```
prompt> ant deploy
```

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

```
<target name="undeploy">

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

</target>
```

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

Using the Administration Console to Deploy Web Services

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

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

where:

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

Then use the deployment assistants to help you deploy the Enterprise application. For more information on the Administration Console, see the Administration Console Online Help.

Browsing to the WSDL of the Web Service

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

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

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

where:

- `host` refers to the computer on which WebLogic Server is running (for example, `localhost`).
- `port` refers to the port number on which WebLogic Server is listening (default value is `7001`).
- `contextPath` refers to the context root of the Web Service. There are many places to set the context root (the `<WLHttpTransport>`, `<module>`, or `<jws>` element of `jwsc`) and certain methods take precedence over others. See "Defining the Context Path of a WebLogic Web Service" in *WebLogic Web Services Reference* for a complete explanation.
- `serviceUri` refers to the value of the `serviceUri` attribute of the `<WLHttpTransport>` child element of the `jwsc` Ant task. If you do not specify *any* `serviceUri` attribute in the `jwsc` Ant task, then the `serviceUri` of the Web Service is the default value: the `serviceName` element of the `@WebService` annotation if specified; otherwise, the name of the JWS file, without its extension, followed by `Service`.

For example, assume that you specified the following `<WLHttpTransport>` child element in the `jwsc` task that you use to build your Web Service:

```
<target name="build-service">
  <jwsc
     srcdir="src"
     destdir="${ear-dir}"
     keepGenerated="true">
    <jws file="examples/webservices/complex/ComplexImpl.java"
       type="JAXWS">
    <WLHttpTransport
       contextPath="complex" serviceUri="ComplexService"
       portName="ComplexServicePort"/>
    </jws>
  </jwsc>
</target>
```

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

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

Configuring the Server Address Specified in the Dynamic WSDL

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

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

...

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

</definitions>
```

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

WebLogic Server determines the `complex/ComplexService` section of this address by examining the `contextPath` and `serviceURI` attributes of the `jwsc` elements, as described in "Browsing to the WSDL of the Web Service" on page 3-17. However, the method WebLogic Server uses to determine the protocol and host section of the address (`http://myhost:7101`, in the example) is more complicated, as described below. For clarity, this section uses the term *server address* to refer to the protocol and host section of the address.

The server address that WebLogic Server publishes in a dynamic WSDL of a deployed Web Service depends on whether the Web Service can be invoked using HTTP/S or JMS, whether you have configured a proxy server, whether the Web Service is deployed to a cluster, or whether the Web Service is actually a callback service.

The following sections reflect these different configuration options, and provide links to procedural information about changing the configuration to suit your needs.

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

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

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

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

   See "Configure HTTP Settings for a Cluster" in the Administration Console Online Help.

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

   See "Configure HTTP Protocol" in the Administration Console Online Help.

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

Web Service is a callback service

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

   See "Configure HTTP Settings for a Cluster' in the Administration Console Online Help.

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

   See "Configure HTTP Protocol" in the Administration Console Online Help.

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

   See "Configure Clusters" in the Administration Console Online Help.

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

   See "Configure Listen Addresses" in the Administration Console Online Help.

Web Service is invoked using a proxy server

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

See "Configure HTTP Settings for a Cluster" or "Configure HTTP Protocol" in the Administration Console Online Help.

Testing the Web Service

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

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

1. Invoke the Administration Console in your browser using the following URL:

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

   where:

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

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

2. Follow the procedure described in "Test a Web Service" in the Administration Console Online Help.

Integrating Web Services Into the WebLogic Split Development Directory Environment

This section describes how to integrate Web Services development into the WebLogic split development directory environment. It is assumed that you understand this WebLogic feature and have set up this type of environment for developing standard Java Platform, Enterprise Edition (Java EE) Version 5 applications and modules, such as EJBs and Web applications, and you want to update the single `build.xml` file to include Web Services development.

For detailed information about the WebLogic split development directory environment, see "Creating a Split Development Directory for an Application" in *Developing Applications With WebLogic Server* and the `splitdir/helloWorldEar` example installed with WebLogic Server, located in the *`WL_HOME`*`/samples/server/examples/src/examples` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Server installation.

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

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

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

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

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

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

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

4. In the `build.xml` file that builds the Enterprise application, create a new target to build the Web Service, adding a call to the `jwsc` WebLogic Web Service Ant task, as described in "Running the jwsc WebLogic Web Services Ant Task" on page 3-7.

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

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

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

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

    </jwsc>

  </target>
```

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

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

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

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

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

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

```
<target name="compile">

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

...
</target>

<target name="appc">

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

</target>
```

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

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

```
<application>

...

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

...

</application>
```

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

Your split development directory environment is now updated to include Web Service development. When you rebuild and deploy the entire Enterprise Application, the Web Service will also be deployed as part of the EAR. You invoke the Web Service in the standard way described in “Browsing to the WSDL of the Web Service” on page 3-17.

CHAPTER 4

Programming the JWS File

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

Overview of JWS Files and JWS Annotations

There are two ways to program a WebLogic Web Service from scratch:

1. Annotate a standard EJB or Java class with Web Service Java annotations, as defined by JSR-181, the JAX-WS specification, and by the WebLogic Web Services programming model.
2. Combine a standard EJB or Java class with the various XML descriptor files and artifacts specified by JSR-109 (such as, deployment descriptors, WSDL files, data mapping descriptors, data binding artifacts for user-defined data types, and so on).

Oracle strongly recommends using option 1 above. Instead of authoring XML metadata descriptors yourself, the WebLogic Ant tasks and runtime will generate the required descriptors and artifacts based on the annotations you include in your JWS. Not only is this process much easier, but it keeps the information about your Web Service in a central location, the JWS file, rather than scattering it across many Java and XML files.

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 Java metadata annotations to specify the shape and characteristics of the Web Service. The JWS annotations you can use in a JWS file include the standard ones defined by the Web Services Metadata for the Java Platform specification (JSR-181) plus a set of additional annotations based on the type of Web Service you are building—JAX-WS or JAX-RPC. For a complete list of JWS annotations that are supported for JAX-WS and JAX-RPC Web Services, see "Web Service Annotation Support" in *WebLogic Web Services Reference*.

When programming the JWS file, you include annotations to program basic Web Service features. The annotations are used at different levels, or targets, in your JWS file. Some are used at the class-level to indicate that the annotation applies to the entire JWS file. Others are used at the method-level and yet others at the parameter level.

Java Requirements for a JWS File

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

- Must be an outer public class, must not be declared `final`, and must not be `abstract`.
- Must have a default public constructor.

- Must not define a `finalize()` method.
- Must include, at a minimum, a `@WebService` JWS annotation at the class level to indicate that the JWS file implements a Web Service.
- May reference a service endpoint interface by using the `@WebService.endpointInterface` annotation. In this case, it is assumed that the service endpoint interface exists and you cannot specify any other JWS annotations in the JWS file other than `@WebService.endpointInterface` and `@WebService.serviceName`.
- If JWS file does not implement a service endpoint interface, all public methods other than those inherited from `java.lang.Object` will be exposed as Web Service operations. This behavior can be overridden by using the `@WebMethod` annotation to specify explicitly the public methods that are to be exposed. If a `@WebMethod` annotation is present, only the methods to which it is applied are exposed.

Programming the JWS File: Typical Steps

The following procedure describes the typical steps for programming a JWS file that implements a Web Service.

Note: It is assumed that you have created a JWS file and now want to add JWS annotations to it.

For more information about each of the JWS annotations, see "JWS Annotation Reference" in *WebLogic Web Services Reference*.

Table 4-1 Steps to Program the JWS File

#	Step	Description

1	Import the standard JWS annotations that will be used in your JWS file.	The standard JWS annotations are in either the `javax.jws` or `javax.jws.soap` package. For example:
		`import javax.jws.WebMethod;`
		`import javax.jws.WebService;`
		`import javax.jws.soap.SOAPBinding;`

2	Import additional annotations, as required.	For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference*.
3	Add the standard required `@WebService` JWS annotation at the class level to specify that the Java class exposes a Web Service.	See "Specifying that the JWS File Implements a Web Service (@WebService Annotation)" on page 4-6.

Table 4-1 Steps to Program the JWS File (Continued)

#	Step	Description
4	Add the standard `@SOAPBinding` JWS annotation at the class level to specify the mapping between the Web service and the SOAP message protocol. (Optional)	In particular, use this annotation to specify whether the Web Service is document-literal, document-encoded, and so on. See "Specifying the Mapping of the Web Service to the SOAP Message Protocol (@SOAPBinding Annotation)" on page 4-6. Although this JWS annotation is not required, Oracle recommends you explicitly specify it in your JWS file to clarify the type of SOAP bindings a client application uses to invoke the Web Service.
5	Add the JAX-WS `@BindingType` JWS annotation at the class level to specify the binding type to use for a Web Service endpoint implementation class. (Optional)	See "Specifying the Binding to Use for an Endpoint (@BindingType Annotation)" on page 4-10.
6	Add the standard `@WebMethod` annotation for each method in the JWS file that you want to expose as a public operation. (Optional)	Optionally specify that the operation takes only input parameters but does not return any value by using the standard `@Oneway` annotation. See "Specifying That a JWS Method Be Exposed as a Public Operation (@WebMethod and @OneWay Annotations)" on page 4-7.
7	Add `@WebParam` annotation to customize the name of the input parameters of the exposed operations. (Optional)	See "Customizing the Mapping Between Operation Parameters and WSDL Elements (@WebParam Annotation)" on page 4-8.
8	Add `@WebResult` annotations to customize the name and behavior of the return value of the exposed operations. (Optional)	See "Customizing the Mapping Between the Operation Return Value and a WSDL Element (@WebResult Annotation)" on page 4-9.
9	Add your business code.	Add your business code to the methods to make the WebService behave as required.

Example of a JWS File

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

```
package examples.webservices.simple;

// Import the standard JWS annotation interfaces

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

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

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

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

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

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

public class SimpleImpl {

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

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

Specifying that the JWS File Implements a Web Service (@WebService Annotation)

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

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

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

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

- `endpointInterface`—Fully qualified name of an existing service endpoint interface file. This annotation allows the separation of interface definition from the implementation. If you specify this attribute, the `jwsc` Ant task does not generate the interface for you, but assumes you have created it and it is in your CLASSPATH.
- `portname`—Name that is used in the `wsdl:port`.

None of the attributes of the `@WebService` annotation is required. See the Web Services Metadata for the Java Platform (JSR 181) for the default values of each attribute.

Specifying the Mapping of the Web Service to the SOAP Message Protocol (@SOAPBinding Annotation)

It is assumed that you want your Web Service to be available over the SOAP message protocol; for this reason, your JWS file should include the standard `@SOAPBinding` annotation, at the class level, to specify the SOAP bindings of the Web Service (such as, document-encoded or document-literal-wrapped), as shown in the following code excerpt:

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

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

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

The following table lists the possible and default values for the three attributes of the `@SOAPBinding` (either the standard or WebLogic-specific) annotation.

Table 4-2 Attributes of the @SOAPBinding Annotation

Attribute	Possible Values	Default Value
`style`	`SOAPBinding.Style.RPC` `SOAPBinding.Style.DOCUMENT`	`SOAPBinding.Style.DOCUMENT`
`use`	`SOAPBinding.Use.LITERAL`	`SOAPBinding.Use.LITERAL`
`parameterStyle`	`SOAPBinding.ParameterStyle.BARE` `SOAPBinding.ParameterStyle.WRAPPED`	`SOAPBinding.ParameterStyle.WRAPPED`

Specifying That a JWS Method Be Exposed as a Public Operation (@WebMethod and @OneWay Annotations)

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

```
public class SimpleImpl {

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

In the example, the `sayHello()` method of the `SimpleImpl` JWS file is exposed as a public operation of the Web Service. The `operationName` attribute specifies, however, that the public name of the operation in the WSDL file is `sayHelloOperation`. If you do not specify the `operationName` attribute, the public name of the operation is the name of the method itself.

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

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

```
 public class OneWayImpl {

   @WebMethod()
   @Oneway()

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

...
```

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

None of the attributes of the `@WebMethod` annotation is required. See the Web Services Metadata for the Java Platform (JSR 181) for the default values of each attribute, as well as additional information about the `@WebMethod` and `@Oneway` annotations.

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

Customizing the Mapping Between Operation Parameters and WSDL Elements (@WebParam Annotation)

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

```
 public class SimpleImpl {

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

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

```
  }
...
```

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

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

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

None of the attributes of the `@WebParam` annotation is required. See the Web Services Metadata for the Java Platform (JSR 181) for the default value of each attribute.

Customizing the Mapping Between the Operation Return Value and a WSDL Element (@WebResult Annotation)

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

```
public class Simple {

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

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

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

None of the attributes of the `@WebResult` annotation is required. See the Web Services Metadata for the Java Platform (JSR 181) for the default value of each attribute.

Specifying the Binding to Use for an Endpoint (@BindingType Annotation)

Use the JAX-WS `@BindingType` annotation to customize the binding to use for a web service endpoint implementation class, as shown in the following code excerpt:

```
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;
 public class Simple {

  @WebService()
  @BindingType(value=SOAPBinding.SOAP12HTTP_BINDING)
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/docLiteralBare")
      int input)

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

In the example, the deployed endpoint would use the SOAP1.2 over HTTP binding. If not specified, the binding defaults to SOAP 1.1 over HTTP.

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

- `features`—An array of features to enable/disable on the specified binding. If not specified, features are enabled based on their own rules.

For more information about the `@BindingType` annotation, see JAX-WS 2.1 Annotations.

Accessing Runtime Information About a 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 or client can use to access, and sometimes change, runtime information about the service.

To access runtime information, you can use one of the following methods:

- `javax.xml.ws.BindingProvider`—From the client application, access the request and response context of the protocol binding. See "Accessing the Protocol Binding Context" on page 4-11.
- `javax.xml.ws.WebServiceContext`—From the Web Service, access runtime message context and security information relative to a request being served. Typically, a `WebServiceContext` is injected into an endpoint using the `@Resource` annotation. See "Accessing the Web Service Context" on page 4-14.
- `javax.xml.ws.handler.MessageContext`—Access a set of runtime properties from a message handler—from the client application or Web Service—or directly from the `WebServiceContext` from a Web Service. See "Using the MessageContext Property Values" on page 4-16

The following sections describe how to use the `BindingProvider`, `WebServiceContext`, and `MessageContext` to access runtime information in more detail.

Accessing the Protocol Binding Context

Note: The `com.sun.xml.ws.developer.JAXWSProperties` and `com.sun.xml.ws.client.BindingProviderProperties` APIs are supported as an extension to the JDK 6.0, provided by Sun Microsystems. Because the APIs are not provided as part of the JDK 6.0 kit, they are subject to change.

The `javax.xml.ws.BindingProvider` interface enables you to access from the client application the request and response context of the protocol binding. For more information about developing Web Service client files, see "Invoking Web Services" on page 6-1.

The following example shows a simple Web Service client application that uses the context to access HTTP request header information. The code in **bold** is discussed in the programming guidelines described following the example.

```
package examples.webservices.hello_world.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.MessageContext;
import com.sun.xml.ws.developer.JAXWSProperties;
import com.sun.xml.ws.client.BindingProviderProperties;

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHelloWorld</code> operation of the Simple Web service.
 */

public class Main {
  public static void main(String[] args) {
    HelloWorldService service;
    try {
       service = new HelloWorldService(new URL(args[0] + "?WSDL"),
           new QName("http://hello_world.webservices.examples/",
           "HelloWorldService") );
    } catch (MalformedURLException murl) { throw new RuntimeException(murl); }
      HelloWorldPortType port = service.getHelloWorldPortTypePort();
      String result = null;
      result = port.sayHelloWorld("Hi there!");
      System.out.println( "Got result: " + result );
      Map requestContext = ((BindingProvider)port).getRequestContext();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                    "http://examples.com/HelloWorldImpl/HelloWorldService");
      requestContext.put(JAXWSProperties.CONNECT_TIMEOUT, 300);
      requestContext.put(BindingProviderProperties.REQUEST_TIMEOUT, 300);
      Map responseContext = ((BindingProvider)port).getResponseContext();
      Integer responseCode =
              (Integer)responseContext.get(MessageContext.HTTP_RESPONSE_CODE);
...
  }
}
```

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

- Import the `javax.xml.ws.BindingProvider` API, as well as any other related APIs that you might use:

  ```
  import java.util.Map;
  import javax.xml.ws.BindingProvider;
  import javax.xml.ws.handler.MessageContext;
  import com.sun.xml.ws.developer.JAXWSProperties;
  import com.sun.xml.ws.client.BindingProviderProperties;
  import com.sun.xml.ws.client.BindingProviderProperties;
  ```

- Use the methods of the `BindingProvider` class to access the binding protocol context information. The following example shows how to get the request and response context for the protocol binding and subsequently set the target service endpoint address used by the client for the request context, set the connection and read timeouts (in milliseconds) for the request context, and set the HTTP response status code for the response context:

  ```
  Map requestContext = ((BindingProvider)port).getRequestContext();
  requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                 "http://examples.com/HelloWorldImpl/HelloWorldService");
  requestContext.put(JAXWSProperties.CONNECT_TIMEOUT, 300);
  requestContext.put(BindingProviderProperties.REQUEST_TIMEOUT, 300);
  Map responseContext = ((BindingProvider)port).getResponseContext();
  Integer responseCode =
         (Integer)responseContext.get(MessageContext.HTTP_RESPONSE_CODE);
  ```

The following table summarizes the methods of the `javax.xml.ws.BindingProvider` that you can use in your JWS file to access runtime information about the Web Service.

Table 4-3 Methods of the BindingProvider

Method	Returns	Description
`getBinding()`	`Binding`	Returns the binding for the binding provider.
`getRequestContext()`	`java.Util.Map`	Returns the context that is used to initialize the message and context for request messages.
`getResponseContext()`	`java.Util.Map`	Returns the response context.

One you get the request or response context, you can access the `BindingProvider` property values defined in the following table and the `MessageContext` property values defined in "Using the MessageContext Property Values" on page 4-16.

Table 4-4 Properties of BindingProvider

Property	Type	Description
`ENDPOINT_ADDRESS_PROPERTY`	`java.lang.String`	Target service endpoint address.
`PASSWORD_PROPERTY`	`java.lang.String`	Password used for authentication.
`SESSION_MAINTAIN_PROPERTY`	`java.lang.Boolean`	Flag that specifies whether a service client wants to participate in a session with a service endpoint. Defaults to `false`, indicating that the service client does not want to participate.
`SOAPACTION_URI_PROPERTY`	`java.lang.String`	Property for SOAPAction specifying the SOAPAction URI. This property is valid only if `SOAPACTION_USE_PROPERTY` is set to `true`.
`SOAPACTION_USE_PROPERTY`	`java.lang.Boolean`	Property for SOAPAction specifying whether or not SOAPAction should be used.
`USERNAME_PROPERTY`	`java.lang.String`	User name used for authentication.

In addition, in the previous example:

- The `JAXWSProperties.CONNECT_TIMEOUT` property is used to define the connection timeout. For a complete list of `JAXWSProperties` that you can set, see the `com.sun.xml.ws.developer.JAXWSProperties` Javadoc.
- The `BindingProviderProperties.REQUEST_TIMEOUT` property is used to define the request timeout. For a complete list of `BindingProviderProperties` that you can set, see the `com.sun.xml.ws.client.BindingProviderProperties` Javadoc.

Accessing the Web Service Context

The `javax.xml.ws.WebServiceContext` interface enables you to access from the Web Service runtime message context and security information relative to a request being served. Typically, a `WebServiceContext` is injected into an endpoint using the `@Resource` annotation.

The following example shows a simple JWS file that uses the context to access HTTP request header information. The code in **bold** is discussed in the programming guidelines described following the example.

```
package examples.webservices.jws_context;

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

import java.util.Map;
import javax.xml.ws.WebServiceContext;
import javax.annotation.Resource;
import javax.xml.ws.handler.MessageContext;

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

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

public class JwsContextImpl {

  @Resource
  private WebServiceContext ctx;

  @WebMethod()
  public String msgContext(String msg) {
    MessageContext context=ctx.getMessageContext();
    Map requestHeaders = (Map)context.get(MessageContext.HTTP_REQUEST_HEADERS);
  }

}
```

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

- Import the `@javax.annotation.Resource` JWS annotation:

  ```
  import javax.annotation.Resource;
  ```

- Import the `javax.xml.ws.WebServiceContext` API, as well as any other related APIs that you might use:

  ```
  import java.util.Map;
  import javax.xml.ws.WebServiceContext;
  import javax.xml.ws.handler.MessageContext;
  ```

- Annotate a private variable, of data type `javax.xml.ws.WebServiceContext`, with the field-level `@Resource` JWS annotation:

  ```
  @Resource
  private WebServiceContext ctx;
  ```

- Use the methods of the `WebServiceContext` class to access runtime information about the Web Service. The following example shows how to get the message context for the current service request and subsequently access the HTTP request headers:

  ```
  MessageContext context=ctx.getMessageContext();
  Map requestHeaders =
  (Map)context.get(MessageContext.HTTP_REQUEST_HEADERS)
  ```

  For more information about the `MessageContext` property values, see "Using the MessageContext Property Values" on page 4-16.

The following table summarizes the methods of the `javax.xml.ws.WebServiceContext` that you can use in your JWS file to access runtime information about the Web Service.

Table 4-5 Methods of the WebServiceContext

Method	Returns	Description
`getMessageContext()`	`MessageContext`	Returns the MessageContext for the current service request. You can access properties that are application-scoped only, such as `HTTP_REQUEST_HEADERS`, `MESSAGE_ATTACHMENTS`, and so on, as defined in "Using the MessageContext Property Values" on page 4-16.
`getUserPrincipal()`	`java.security.Principal`	Returns the Principal that identifies the sender of the current service request. If the sender has not been authenticated, the method returns `null`.
`isUserInRole(java.lang.String role)`	`boolean`	Returns a boolean value specifying whether the authenticated user is included in the specified logical role. If the user has not been authenticated, the method returns `false`.

Using the MessageContext Property Values

The following table defined the `javax.xml.ws.handler.MessageContext` property values that you can access from a message handler—from the client application or Web Service—or directly from the `WebServiceContext` from the Web Service. For more information, see the `javax.xml.ws.handler.MessageContext` Javadocs.

Table 4-6 Properties of MessageContext

Property	Type	Description
`HTTP_REQUEST_HEADERS`	`java.util.Map`	Map of HTTP request headers for the request message.
`HTTP_REQUEST_METHOD`	`java.lang.String`	HTTP request method for example GET, POST, or PUT.
`HTTP_RESPONSE_CODE`	`java.lang.Integer`	HTTP response status code for the last invocation.
`HTTP_RESPONSE_HEADERS`	`java.util.Map`	HTTP response headers.
`INBOUND_MESSAGE_ATTACHMENTS`	`java.util.Map`	Map of attachments for the inbound messages.
`MESSAGE_OUTBOUND_PROPERTY`	`java.lang.Boolean`	Message direction. This property is `true` for outbound messages and `false` for inbound messages.
`OUTBOUND_MESSAGE_ATTACHMENTS`	`java.util.Map`	Map of attachments for the outbound messages.
`PATH_INFO`	`java.lang.String`	Request path information.
`QUERY_STRING`	`java.lang.String`	Query string for request.
`REFERENCE_PARAMETERS`	`java.awt.List`	WS-Addressing reference parameters. The list must include all SOAP headers marked with the `wsa:IsReferenceParameter="true"` attribute.
`SERVLET_CONTEXT`	`javax.servlet.ServletContext`	Servlet context object associated with request.
`SERVLET_REQUEST`	`javax.servlet.http.HttpServletRequest`	Servlet request object associated with request.
`SERVLET_RESPONSE`	`javax.servlet.http.HttpServletResponse`	Servlet response object associated with request.

Table 4-6 Properties of MessageContext (Continued)

Property	Type	Description
`WSDL_DESCRIPTION`	`org.xml.sax.InputSource`	Input source (resolvable URI) for the WSDL document.
`WSDL_INTERFACE`	`javax.xml.namespace.QName`	Name of the WSDL interface or port type.
`WSDL_OPERATION`	`javax.xml.namespace.QName`	Name of the WSDL operation to which the current message belongs.
`WSDL_PORT`	`javax.xml.namespace.QName`	Name of the WSDL port to which the message was received.
`WSDL_SERVICE`	`javax.xml.namespace.QName`	Name of the service being invoked.

Should You Implement a Stateless Session EJB?

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

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

EJB 3.0 introduced metadata annotations that enable you to automatically generate, rather than manually create, the EJB Remote and Home interface classes and deployment descriptor files needed when implementing an EJB. For more information about EJB 3.0, see *Enterprise JavaBeans (EJB) 3.0*.

To implement an EJB in your JWS file, perform the following steps:

- Import the EJB 3.0 annotations, all of which are in the `javax.ejb` package. At a minimum you need to import the `@Stateless` annotation. You can also specify additional EJB annotations in your JWS file to specify the shape and behavior of the EJB, see the `javax.ejb` Javadoc for more information.

  For example:

```
import javax.ejb.Stateless;
```

- At a minimum, use the `@Stateless` annotation at the class level to identify the EJB:

```
@Stateless
public class SimpleEjbImpl {
```

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

```
package examples.webservices.jaxws;

import weblogic.transaction.TransactionHelper;
import javax.ejb.Stateless;
import javax.ejb.SessionContext;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.annotation.Resource;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.transaction.SystemException;
import javax.transaction.Status;
import javax.transaction.Transaction;
import javax.xml.ws.WebServiceContext;

/**
* A transaction-awared stateless EJB-implemented JWS
*/

// Standard JWS annotation that specifies that the portName,serviceName and
// target Namespace of the Web Service.
@WebService(
       name = "Simple",
       portName = "SimpleEJBPort",
       serviceName = "SimpleEjbService",
       targetNamespace = "http://www.bea.com/wls/samples")

//Standard EJB annotation
@Stateless
public class SimpleEjbImpl {

   @Resource
   private WebServiceContext context;
   private String constructed = null;

   // The WebMethod annotation exposes the subsequent method as a public
   // operation on the Web Service.
   @WebMethod()
   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public String sayHello(String s) throws SystemException {
      Transaction transaction =
         TransactionHelper.getTransactionHelper().getTransaction();
      int status = transaction.getStatus();
      if (Status.STATUS_ACTIVE != status)
           throw new IllegalStateException("transaction did not start,
           status is: " + status + ", check ejb annotation processing");

   return constructed + ":" + s;
}
```

Programming the User-Defined Java Data Type

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

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

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

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

The `jwsc` Ant task can generate data binding artifacts for most common XML and Java data types. For the list of supported user-defined data types, see "Supported User-Defined Data Types" on page 5-10. See "Supported Built-In Data Types" on page 5-5 for the full list of supported built-in data types.

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

```
package examples.webservices.complex;

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

  // Properties

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

  // Getter and setter methods

  public int getIntValue() {
    return intValue;
  }

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

  public String getStringValue() {
    return stringValue;
  }

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

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

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

}
```

The following snippets from a JWS file show how to import the `BasicStruct` class and use it as both a parameter and return value for one of its methods; for the full JWS file, see "Sample ComplexImpl.java JWS File" on page 2-12:

```
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 the BasicStruct JavaBean

import examples.webservices.complex.BasicStruct;

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

...

public class ComplexImpl {

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

  {
    return struct;
  }
}
```

Invoking Another Web Service from the JWS File

From within your JWS file you can invoke another Web Service, either one deployed on WebLogic Server or one deployed on some other application server, such as .NET. The steps to do this are similar to those described in "Invoking a Web Service from a Stand-alone Java Client" on page 2-25, except that rather than running the `clientgen` Ant task to generate the client stubs, you include a `<clientgen>` child element of the `jwsc` Ant task that builds the invoking Web Service to generate the client stubs instead. You then use the standard JAX-WS APIs in your JWS file the same as you do in a stand-alone client application.

See "Invoking a Web Service from Another Web Service" on page 6-10 for detailed instructions.

Using SOAP 1.2

WebLogic Web Services use, by default, Version 1.1 of Simple Object Access Protocol (SOAP) as the message format when transmitting data and invocation calls between the Web Service and its client. WebLogic Web Services support both SOAP 1.1 and the newer SOAP 1.2, and you are free to use either version.

To specify that the Web Service use Version 1.2 of SOAP, use the class-level `@javax.xml.ws.BindingType` annotation in your JWS file and set its single attribute to the value `SOAPBinding.SOAP12HTTP_BINDING`, as shown in the following example (relevant code shown in bold):

```
package examples.webservices.soap12;

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

import javax.xml.ws.BindingType;
import javax.xml.ws.SOAPBinding;

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

@BindingType(value = SOAPBinding.SOAP12HTTP_BINDING)
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello.  The class uses SOAP 1.2
 * as its binding.
 *
 */

public class SOAP12Impl {

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

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

Validating the XML Schema

By default, SOAP messages are not validated against their XML schemas. You can enable XML schema validation for document-literal Web Services on the server or client, as described in the following sections.

Note: This feature adds a small amount of extra processing to a Web Service request.

Enabling Schema Validation on the Server

Note: The `com.sun.xml.ws.developer.SchemaValidation` API is supported as an extension to the JDK 6.0, provided by Sun Microsystems. Because this API is not provided as part of the JDK 6.0 kit, it is subject to change.

To enable schema validation on the server, add the `@SchemaValidation` annotation on the endpoint implementation. For example:

```
import com.sun.xml.ws.developer.SchemaValidation;
import javax.jws.WebService;

@SchemaValidation
@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
public class HelloWorldImpl {
  public String sayHelloWorld(String message) {
    System.out.println("sayHelloWorld:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

You can pass your own validation error handler class as an argument to the annotation, if you want to manage errors within your application. For example:

```
@SchemaValidation(handler=ErrorHandler.class)
```

Enabling Schema Validation on the Client

Note: The `com.sun.xml.ws.developer.SchemaValidationFeature` API is supported as an extension to the JDK 6.0, provided by Sun Microsystems. Because this API is not provided as part of the JDK 6.0 kit, it is subject to change.

To enable schema validation on the client, create a `SchemaValidationFeature` object and pass this as an argument when creating the `PortType` stub implementation.

```
package examples.webservices.hello_world.client;

import com.sun.xml.ws.developer.SchemaValidationFeature;
import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
public class Main {

  public static void main(String[] args) {
    HelloWorldService service;
    try {
      service = new HelloWorldService(new URL(args[0] + "?WSDL"),
                new QName("http://example.org", "HelloWorldService") );
   } catch (MalformedURLException murl) { throw new RuntimeException(murl);
}
      SchemaValidationFeature feature =
           new SchemaValidationFeature();
      HelloWorldPortType port = service.getHelloWorldPortTypePort(feature);
      String result = null;
      result = port.sayHelloWorld("Hi there!");
      System.out.println( "Got result: " + result );
  }
}
```

You can pass your own validation error handler as an argument to the `SchemaValidationFeature` object, if you want to manage errors within your application. For example:

```
      SchemaValidationFeature feature =
           new SchemaValidationFeature(MyErrorHandler.class);
      HelloWorldPortType port = service.getHelloWorldPortTypePort(feature);
```

JWS Programming Best Practices

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

- When you create a document-literal-bare Web Service, use the `@WebParam` JWS annotation to ensure that all input parameters for all operations of a given Web Service have a unique name. Because of the nature of document-literal-bare Web Services, if you do not explicitly use the `@WebParam` annotation to specify the name of the input parameters, WebLogic Server creates one for you and run the risk of duplicating the names of the parameters across a Web Service.

- In general, document-literal-wrapped Web Services are the most interoperable type of Web Service.
- Use the `@WebResult` JWS annotation to explicitly set the name of the returned value of an operation, rather than always relying on the hard-coded name `return`, which is the default name of the returned value if you do not use the `@WebResult` annotation in your JWS file.

Using JAXB Data Binding

The following sections provide information about using JAXB data binding:

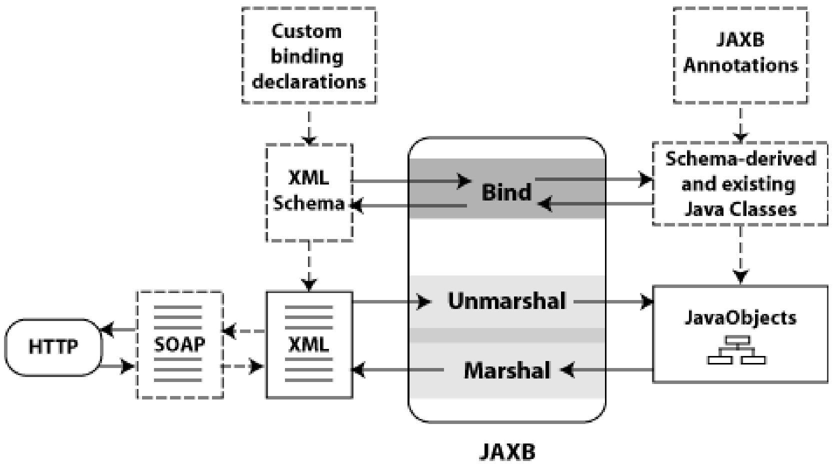
Overview of Data Binding Using JAXB

With the emergence of XML as the standard for exchanging data across disparate systems, Web Service applications need a way to access data that are in XML format directly from the Java application. Specifically, the XML content needs to be converted to a format that is readable by the Java application. *Data binding* describes the conversion of data between its XML and Java representations.

JAX-WS uses Java Architecture for XML Binding (JAXB) to manage all of the data binding tasks. Specifically, JAXB binds Java method signatures and WSDL messages and operations and allows you to customize the mapping while automatically handling the runtime conversion. This makes it easy for you to incorporate XML data and processing functions in applications based on Java technology without having to know much about XML.

The following figure shows the JAXB data binding process.

Figure 5-1 Data Binding With JAXB



As shown in the previous figure, the JAXB data binding process consists of the following tasks:

- **Bind**—Binds XML Schema to *schema-derived JAXB Java classes*, or value classes. Each class provides access to the content via a set of JavaBean-style access methods (that is, get and set). Binding is managed by the JAXB *schema compiler.*
- **Unmarshal**—Converts the XML document to create a tree of Java program elements, or objects, that represents the content and organization of the document that can be accessed by your Java code. In the content tree, complex types are mapped to value classes. Attribute declarations or elements with simple types are mapped to properties or fields within the value class and you can access the values for them using `get` and `set` methods. Unmarshalling is managed by the JAXB *binding framework*.
- **Marshal**—Converts the Java objects back to XML content. In this case, the Java methods that are deployed as WSDL operations determine the schema components in the `wsdl:types` section. Marshalling is managed by the JAXB binding framework.

You can use the JAXB binding language to define custom binding declarations or specify JAXB annotations to control the conversion of data between XML and Java.

This following sections describe:

- Developing the JAXB Data Binding Artifacts—Describes how to develop the JAXB data binding artifacts using WebLogic Server.

- Standard Data Type Mapping—Describes the standard built-in and user-defined data types that are supported.
- Customizing Java-to-XML Schema Mapping Using JAXB Annotations—Describes how you can control and customize the Java-to-XML Schema mapping using JAXB annotations in the JWS file.
- Customizing XML Schema-to-Java Mapping Using Binding Declarations—Describes how you can control and customize the XML Schema-to-Java mapping using binding declarations that are defined in a separate file or embedded inline.

Developing the JAXB Data Binding Artifacts

The steps to develop the JAXB data binding artifacts using WebLogic Server depend on whether you are starting from a Java class file or a WSDL.

- **Start from Java**: Using this programming model, you create the Java classes. At run-time, JAXB *marshals* the Java objects to generate the XML content which is then packaged in a SOAP message and sent as a Web Service request or response.

  To control the Java-to-XML mapping, you include JAXB annotations in your JWS file, as described in "Customizing Java-to-XML Schema Mapping Using JAXB Annotations" on page 5-13. If no customizations are required, JAXB uses the standard built-in and user-defined data type mapping as described in the following sections: "Java-to-XML Mapping for Built-In Data Types" on page 5-9 and "Supported Java User-Defined Data Types" on page 5-12.

  For more information about this programming model, see "Developing WebLogic Web Services Starting From Java: Main Steps" on page 3-2.

- **Start from WSDL**: Using this programming model, the XML Schemas exist and JAXB *unmarshals* the XML document to generate the Java objects.

  To control the XML-to-Java mapping, you can define custom binding declarations within the WSDL or XML Schema, or in an external file, as described in "Customizing XML Schema-to-Java Mapping Using Binding Declarations" on page 5-18. If no customizations are required, the standard built-in and user-defined data type mapping as described in the following sections: "XML-to-Java Mapping for Built-in Data Types" on page 5-5 and "Supported XML User-Defined Data Types" on page 5-10.

  For more information about this programming model, see "Developing WebLogic Web Services Starting From a WSDL File: Main Steps" on page 3-4.

Please note, when invoking the `jwsc`, `wsdlc`, or `clientgen` Ant tasks described in these procedures:

- You must specify the `type="JAXWS"` attribute to generate a JAX-WS Web Service and JAXB binding artifacts. For `jwsc`, you specify the type attribute as part of the `<jws>` child element.
- You can optionally specify the `<binding>` child element to specify a customizations file that contains JAX-WS and JAXB data binding customizations. For information about creating a customizations file, see "Customizing XML Schema-to-Java Mapping Using Binding Declarations" on page 5-18. If no customizations are required, JAXB uses the standard built-in and user-defined data type mappings described in "Standard Data Type Mapping" on page 5-4.

For more information about the `jwsc`, `wsdlc`, or `clientgen` Ant tasks, see "Ant Task Reference" in *WebLogic Web Services Reference*.

Standard Data Type Mapping

WebLogic Web Services support a full set of built-in XML Schema, Java, and SOAP types, as specified by the JAXB 2.0 (JSR 222) specification, that you can use in your Web Service operations without performing any additional programming steps. Built-in data types are those such as `integer`, `string`, and `time`.

Additionally, you can use a variety of user-defined XML and Java data types as input parameters and return values of your Web Service. User-defined data types are those that you create from XML Schema or Java building blocks, such as `<xsd:complexType>` or JavaBeans. The WebLogic Web Services Ant tasks, such as `jwsc` and `clientgen`, automatically generate the data binding artifacts needed to convert the user-defined data types between their XML and Java representations. The XML representation is used in the SOAP request and response messages, and the Java representation is used in the JWS that implements the Web Service.

The following sections describe the built-in and user-defined data types that are supported by JAXB:

- Supported Built-In Data Types
- Supported User-Defined Data Types

Supported Built-In Data Types

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

When using user-defined data types, then you must create the data binding artifacts that convert the data between XML and Java. WebLogic Server includes the `jwsc` and `wsdlc` Ant tasks that can automatically generate the data binding artifacts for most user-defined data types. See "Supported User-Defined Data Types" on page 5-10 for a list of supported XML and Java data types.

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

The following table lists alphabetically the supported XML Schema data types (target namespace `http://www.w3.org/2001/XMLSchema`) and their corresponding Java data types. For a list of the supported user-defined XML data types, see "Java-to-XML Mapping for Built-In Data Types" on page 5-9.

Table 5-1 Mapping XML Schema Built-in Data Types to Java Data Types

XML Schema Data Type	Java Data Type (lower case indicates a primitive data type)
`anySimpleType (for xsd:element of this type)`	`java.lang.Object`
`anySimpleType (for xsd:attribute of this type)`	`java.lang.String`
`base64Binary`	`byte[]`
`boolean`	`boolean`
`byte`	`byte`
`date`	`java.xml.datatype.XMLGregorianCalendar`
`dateTime`	`javax.xml.datatype.XMLGregorianCalendar`
`decimal`	`java.math.BigDecimal`

Table 5-1 Mapping XML Schema Built-in Data Types to Java Data Types (Continued)

XML Schema Data Type	Java Data Type (lower case indicates a primitive data type)
`double`	`double`
`duration`	`javax.xml.datatype.Duration`
`float`	`float`
`g`	`java.xml.datatype.XMLGregorianCalendar`
`hexBinary`	`byte[]`
`int`	`int`
`integer`	`java.math.BigInteger`
`long`	`long`
`NOTATION`	`javax.xml.namespace.QName`
`Qname`	`javax.xml.namespace.QName`
`short`	`short`
`string`	`java.lang.String`
`time`	`java.xml.datatype.XMLGregorianCalendar`
`unsignedByte`	`short`
`unsignedInt`	`long`
`unsignedShort`	`int`

The following example, borrowed from the JAXB specification, shows an example of the default XML-to-Java binding.

XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
     <xsd:sequence>
          <xsd:element name="shipTo" type="USAddress"/>
          <xsd:element name="billTo" type="USAddress"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="items" type="Items"/>
     </xsd:sequence>
     <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
     <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street" type="xsd:string"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zip" type="xsd:decimal"/>
     </xsd:sequence>
<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
     <xsd:sequence>
          <xsd:element name="item" minOccurs="1" maxOccurs="unbounded">
               <xsd:complexType>
                    <xsd:sequence>
                         <xsd:element name="productName" type="xsd:string"/>
                         <xsd:element name="quantity">
                              <xsd:simpleType>
                                   <xsd:restriction base="xsd:positiveInteger">
                                        <xsd:maxExclusive value="100"/>
                                   </xsd:restriction>
                              </xsd:simpleType>
                         </xsd:element>
                         <xsd:element name="USPrice" type="xsd:decimal"/>
                         <xsd:element ref="comment" minOccurs="0"/>
                         <xsd:element name="shipDate" type="xsd:date"
                              minOccurs="0"/>
                    </xsd:sequence>
                    <xsd:attribute name="partNum" type="SKU" use="required"/>
               </xsd:complexType>
          </xsd:element>
     </xsd:sequence>
</xsd:complexType>

<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
     <xsd:restriction base="xsd:string">
          <xsd:pattern value="\d{3}-[A-Z]{2}"/>
     </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Default Java Binding

```
import javax.xml.datatype.XMLGregorianCalendar; import java.util.List;
public class PurchaseOrderType {
     USAddress getShipTo(){...}
     void setShipTo(USAddress){...}
     USAddress getBillTo(){...}
     void setBillTo(USAddress){...}
     /** Optional to set Comment property. */
     String getComment(){...}
     void setComment(String){...}
     Items getItems(){...}
     void setItems(Items){...}
     XMLGregorianCalendar getOrderDate()
     void setOrderDate(XMLGregorianCalendar)
};

public class USAddress {
     String getName(){...}
     void setName(String){...}
     String getStreet(){...}
     void setStreet(String){...}
     String getCity(){...}
     void setCity(String){...}
     String getState(){...}
     void setState(String){...}
     int getZip(){...}
     void setZip(int){...}
     static final String COUNTRY="USA";
};

public class Items {
     public class ItemType {
          String getProductName(){...}
          void setProductName(String){...}
          /** Type constraint on Quantity setter value 0..99.*/
          int getQuantity(){...}
          void setQuantity(int){...}
          float getUSPrice(){...}
          void setUSPrice(float){...}
          /** Optional to set Comment property. */
          String getComment(){...}
          void setComment(String){...}
          XMLGregorianCalendar getShipDate();
          void setShipDate(XMLGregorianCalendar);
          /** Type constraint on PartNum setter value "\d{3}-[A-Z]{2}".*/
          String getPartNum(){...} void setPartNum(String){...}
     };
     /** Local structural constraint 1 or more instances of Items.ItemType.*/
     List<Items.ItemType> getItem(){...}
}

public class ObjectFactory {
     // type factories
     Object newInstance(Class javaInterface){...}
     PurchaseOrderType createPurchaseOrderType(){...}
     USAddress createUSAddress(){...}
     Items createItems(){...}
     Items.ItemType createItemsItemType(){...}
     // element factories
JAXBElement<PurchaseOrderType>createPurchaseOrder(PurchaseOrderType){...}
     JAXBElement<String> createComment(String value){...}
}
```

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

The following table lists alphabetically the supported Java data types and their equivalent XML Schema data types. For a list of the supported user-defined Java data types, see "Supported Java User-Defined Data Types" on page 5-12.

Table 5-2 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	XML Schema Data Type
`boolean`	`boolean`
`byte`	`byte`
`double`	`double`
`float`	`float`
`long`	`long`
`int`	`int`
`javax.activation.DataHandler`	`base64Binary`
`java.awt.Image`	`base64Binary`
`java.lang.Object`	`anyType`
`java.lang.String`	`string`

Table 5-2 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	XML Schema Data Type
`java.math.BigInteger`	`integer`
`java.math.BigDecimal`	`decimal`
`java.net.URI`	`string`
`java.util.Calendar`	`dateTime`
`java.util.Date`	`dateTime`
`java.util.UUID`	`string`
`javax.xml.datatype.XMLGregorianCalendar`	`anySimpleType`
`javax.xml.datatype.Duration`	`duration`
`javax.xml.namespace.QName`	`Qname`
`javax.xml.transform.Source`	`base64Binary`
`short`	`short`

Supported User-Defined Data Types

The tables in the following sections list the user-defined XML and Java data types for which the `jwsc` and `wsdlc` Ant tasks can automatically generate data binding artifacts, such as the corresponding Java or XML representation.

If your XML or Java data type is not listed in these tables, and it is not one of the built-in data types listed in "Supported Built-In Data Types" on page 5-5, then you must create the user-defined data type artifacts manually.

Supported XML User-Defined Data Types

The following table lists the XML Schema data types supported by the `jwsc` and `wsdlc` Ant tasks and their equivalent Java data type or mapping mechanism.

Table 5-3 Supported User-Defined XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
`<xsd:complexType>` with elements of both simple and complex types.	JavaBean
`<xsd:complexType>` with simple content.	JavaBean
`<xsd:attribute>` in `<xsd:complexType>`	Property of a JavaBean
Derivation of new simple types by restriction of an existing simple type.	Equivalent Java data type of simple type.
Facets used with restriction element.	Facets not enforced during serialization and deserialization.
`<xsd:list>`	Array of the list data type.
Array derived from `soapenc:Array` by restriction using the `wsdl:arrayType` attribute.	Array of the Java equivalent of the `arrayType` data type.
Array derived from `soapenc:Array` by restriction.	Array of Java equivalent.
Derivation of a complex type from a simple type.	JavaBean with a property called `_value` whose type is mapped from the simple type according to the rules in this section.
`<xsd:anyType>`	`java.lang.Object`
`<xsd:any>`	`javax.xml.soap.SOAPElement`
`<xsd:any[]>`	`javax.xml.soap.SOAPElement[]`
`<xsd:union>`	Common parent type of union members.
`<xsi:nil>` and `<xsd:nillable>` attribute	Java `null` value. If the XML data type is built-in and usually maps to a Java primitive data type (such as `int` or `short`), then the XML data type is actually mapped to the equivalent object wrapper type (such as `java.lang.Integer` or `java.lang.Short`).
Derivation of complex types	Mapped using Java inheritance.
Abstract types	Abstract Java data type.

Supported Java User-Defined Data Types

The following table lists the Java user-defined data types supported by the `jwsc` and `wsdlc` Ant tasks and their equivalent XML Schema data type.

Table 5-4 Supported Java User-Defined Data Types

Java Data Type	Equivalent XML Schema Data Type
JavaBean whose properties are any supported data type.	`<xsd:complexType>` whose content model is a `<xsd:sequence>` of elements corresponding to JavaBean properties.
Array and multidimensional array of any supported data type (when used as a JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
`java.lang.Object` **Note:** The data type of the runtime object must be a known type.	`<xsd:anyType>`
`java.util.Collection`	Literal Array
`java.util.List`	Literal Array
`java.util.ArrayList`	Literal Array
`java.util.LinkedList`	Literal Array
`java.util.Vector`	Literal Array
`java.util.Stack`	Literal Array
`java.util.Set`	Literal Array
`java.util.TreeSet`	Literal Array
`java.utils.SortedSet`	Literal Array
`java.utils.HashSet`	Literal Array

Customizing Java-to-XML Schema Mapping Using JAXB Annotations

If required, you can override the default binding rules for Java-to-XML Schema mapping using JAXB annotations. Table 5-5 summarizes the JAXB mapping annotations that you can include in your JWS file to control how the Java objects are mapped to XML. Each of these annotations are available with the `javax.xml.bind.annotation` package.

Table 5-5 JAXB Mapping Annotations

Annotation	Description
`@XmlAccessorType`	Specifies whether fields or properties are mapped by default. See "Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)" on page 5-15.
`@XmlElement`	Maps a property contained in a class to a local element in the XML Schema complex type to which the containing class is mapped. See "Mapping Properties to Local Elements (@XmlElement)" on page 5-15.
`@XMLMimeType`	Associates the MIME type that controls the XML representation of the property with a textual representation, such as `image/jpeg`. See "Specifying the MIME Type (@XmlMimeType Annotation)" on page 5-16.
`@XmlRootElement`	Maps a top-level class to a global element in the XML Schema that is used by the WSDL of the Web Service. See "Mapping a Top-level Class to a Global Element (@XmlRootElement)" on page 5-16.
`@XmlSeeAlso`	Binds other classes when binding the current class. See "Binding a Set of Classes (@XmlSeeAlso)" on page 5-17.
`@XmlType`	Maps a class or enum type to an XML Schema type.See "Mapping a Value Class to a Schema Type (@XmlType)" on page 5-17.

The default mapping of Java objects to XML Schema for the supported built-in and user-defined types are listed in the following sections:

- "Java-to-XML Mapping for Built-In Data Types" on page 5-9
- "Supported Java User-Defined Data Types" on page 5-12

Example of JAXB Annotations

The following provides an example of the JAXB annotations.

```
@XmlRootElement(name = "ComplexService", namespace ="http://examples.org")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "basicStruct", propOrder = {
    "intValue",
    "stringArray",
    "stringValue"
)
public class BasicStruct {
    protected int intValue;
    @XmlElement(nillable = true)
    protected List<String> stringArray;
    protected String stringValue;

    public int getIntValue() {
        return intValue;
    }
    public void setIntValue(int value) {
        this.intValue = value;
    }
    public List<String> getStringArray() {
        if (stringArray == null) {
            stringArray = new ArrayList<String>();
        }
        return this.stringArray;
    }

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

Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)

The `@XmlAccessorType` annotation specifies whether fields or properties are mapped by default. The annotation can be specified for the following Java program elements:

- Package
- Top-level class

The `@XmlAccessorType` can be specified with the `@XmlType` and `@XmlRootElement` annotations.

The following table lists the optional element that can be passed to the `@XmlAccessorType` annotation.

Table 5-6 Optional Element for @XmlAccessorType Annotation

Element	Description
`value`	Specifies `XMLAccessType.`*`value`*, where `value` can be one of the following values: • `FIELD`—Fields are bound to XML. • `PROPERTY`—JavaBean properties (getter/setter pairs) are bound to XML. • `PUBLIC_MEMBER`—Public fields and JavaBean properties are bound to XML. This is the default. • `NONE`—Neither fields nor JavaBean properties are bound to XML.

For more information, see the `javax.xml.bind.annotation.XmlAccessorType` Javadoc. An example is provided in "Example of JAXB Annotations" on page 5-14.

Mapping Properties to Local Elements (@XmlElement)

The `@XmlElement` annotation maps a property contained in a class to a local element in the XML Schema complex type to which the containing class is mapped. The annotation can be specified for the following Java program elements:

- JavaBean property
- Non-static, non-transient field

The following table lists the annotation elements that can be passed to the `@XmlElement` annotation.

Table 5-7 Optional Element Summary for @XMLElement Annotation

Element	Description
`name`	Local name of the XML element that represents the property of a JavaBean. This element defaults to the JavaBean property name.
`namespace`	Namespace of the XML element that represents the property of a JavaBean. By default, the namespace is derived from the namespace of the containing class.
`nillable`	Customize the element declaration to be nillable.

For more information, see the `javax.xml.bind.annotation.XmlElement` Javadoc.

Specifying the MIME Type (@XmlMimeType Annotation)

The `@XmlMimeType` annotation specifies the MIME type that controls the XML representation of the property. The annotation can be specified for data types, such as `Image` or `Source`, that are bound to the `xsd:base64Binary` binary in XML.

The following table lists the required element that can be passed to the `@XmlMimeType` annotation.

Table 5-8 Required Element for @XmlMimeType Annotation

Element	Description
`value`	Specifies the textual representation of the MIME type, such as `image/jpeg`, `text/xml`, and so on.

For more information, see the `javax.xml.bind.annotation.XmlMimeType` Javadoc.

Mapping a Top-level Class to a Global Element (@XmlRootElement)

The `@XmlRootElement` annotation maps a top-level class to a global element in the XML Schema that is used by the WSDL of the Web Service. The annotation can be specified for the following Java program elements:

- Top-level class
- Enum type

The `@XmlRootElement` can be specified with the `@XmlType` and `@XmlAccessorType` annotations.

The following table lists the optional elements that can be passed to the`@XmlRootElement` annotation.

Table 5-9 Optional Elements for @XmlRootElement Annotation

Element	Description
`name`	Local name of the XML element. This element defaults to the class name.
`namespace`	Namespace of the XML element. By default, the namespace is derived from the package of the class.

For more information, see the `javax.xml.bind.annotation.XmlRootElement` Javadoc. An example is provided in "Example of JAXB Annotations" on page 5-14.

Binding a Set of Classes (@XmlSeeAlso)

The `@XmlSeeAlso` annotation binds a list of classes when binding the current class. The following table lists the optional element that can be passed to the `@XMLRootElement` annotation.

Table 5-10 Optional Element for @XmlSeeAlso Annotation

Element	Description
`value`	List of classes that JAXB uses when binding the current class.

Mapping a Value Class to a Schema Type (@XmlType)

The `@XmlType` annotation maps a class or enum type to an XML Schema type. The type can be a simple or complex type. The annotation can be specified for the following Java program elements:

- Top-level class

- Enum type

The `@XmlType` can be specified with the `@XmlRootElement` and `@XmlAccessorType` annotations.

The following table lists the optional elements that can be passed to the `@XmlType` annotation.

Table 5-11 Optional Elements for @XmlType Annotation

Element	Description
`name`	Name of the XML Schema type to which the class is mapped.
`namespace`	Name of the target namespace of the XML Schema type. By default, the target namespace to which the package containing the class is mapped.
`propOrder`	List of JavaBean property names defined in a class. The list defines an order for the XML Schema elements when the class is mapped to an XML Schema complex type. Each name in the list is the name of a Java identifier of the JavaBean property. All of the JavaBean properties must be listed.

For more information, see the `javax.xml.bind.annotation.XmlType` Javadoc. An example is provided in "Example of JAXB Annotations" on page 5-14.

Customizing XML Schema-to-Java Mapping Using Binding Declarations

Due to the distributed nature of a WSDL, you cannot always control or change its contents to meet the requirements of your application. For example, the WSDL may not be owned by you or it may already be in use by your partners, making changes impractical or impossible.

If directly editing the WSDL is not an option, you can customize how the WSDL components are mapped to Java objects by specifying custom *binding declarations*. You can use binding declarations to control specific features, as well, such as asynchrony, wrapper style, and so on, and to control the JAXB data binding artifacts that are produced by customizing the XML Schema.

You can define binding declarations in one of the following ways:

- Create an external binding declarations file that contains all binding declarations for a specific WSDL or XML Schema document. See "Creating an External Binding Declarations File" on page 5-21.

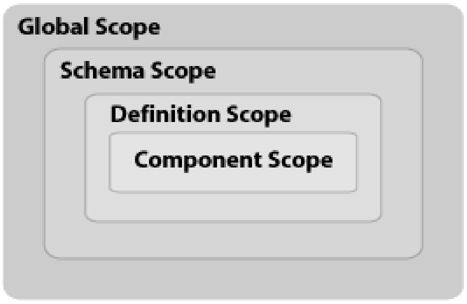
Note: If customizations are required, Oracle recommends this method to maintain flexibility by keeping the customizations separate from the WSDL or XML Schema document.

- Embed binding declarations within the WSDL or XML Schema document. See "Embedding Binding Declarations" on page 5-23.

The binding declarations are semantically equivalent regardless of which method you choose.

Custom binding declarations are associated with a scope, as shown in the following figure.

Figure 5-2 Scopes for Custom Binding Declarations



The following table describes the meaning of each scope.

Table 5-12 Scopes for Custom Binding Declarations

Scope	Definition
Global scope	Describes customization values with global scope. Specifically: • For JAX-WS binding declarations, describes customization values that are defined as part of the root element, as described in "Specifying the Root Element" on page 5-21. • For JAXB annotations, describes customization values that are contained within the `<globalBindings>` binding declaration. Global scope values apply to all of the schema elements in the source schema as well as any schemas that are included or imported.
Schema scope	Describes JAXB customization values that are contained within the `<schemaBindings>` binding declaration. Schema scope values apply to the elements in the target namespace of a schema. **Note:** This scope applies for JAXB binding declarations only.

Table 5-12 Scopes for Custom Binding Declarations (Continued)

Scope	Definition
Definition scope	Describes JAXB customization values that are defined in binding declarations of a type definition or global declaration. Definition scope values apply to elements that reference the type definition or global declaration. **Note:** This scope applies for JAXB binding declarations only.
Component scope	Describes customization values that apply to the WSDL or schema element that was annotated.

Scopes for custom binding declarations adhere to the following inheritance and overriding rules:

- Inheritance—Customization values are inherited from the top down. For example, a WSDL element (JAX-WS) in a component scope inherits a customization value defined in global scope. A schema element (JAXB) in a component scope inherits a customization value defined in global, schema, and definition scopes.
- Overriding—Customization values are overridden from the bottom up. For example, a WSDL element (JAX-WS) in a component scope overrides a customization value defined in global scope. A schema element (JAXB) in a component scope overrides a customization value defined in definition, schema, and global scopes.

The following sections describe how to create custom binding declarations and describe the standard custom binding declarations:

- Creating an External Binding Declarations File
- Embedding Binding Declarations
- JAX-WS Custom Binding Declarations
- JAXB Custom Binding Declarations

For more information about using custom binding declarations, see:

- *JAX-WS WSDL Customizations*
- "Customizing XML Schema to Java Representation Binding" in the JAXB specification

Creating an External Binding Declarations File

Create an external binding declarations file that contains all binding declarations for a specific WSDL or XML Schema document. Then, pass the binding declarations file to the `<binding>` child element of the `wsdlc`, `jwsc`, or `clientgen` Ant task.

The following sections describe:

- Creating an External Binding Declarations File Using JAX-WS Binding Declarations
- Creating an External Binding Declarations File Using JAXB Binding Declarations

Creating an External Binding Declarations File Using JAX-WS Binding Declarations

The following sections describe how to specify the root and child elements of the JAX-WS binding declarations file. For information about the custom binding declarations that you can define, see "JAX-WS Custom Binding Declarations" on page 5-24.

Specifying the Root Element

The `jaxws:bindings` declaration is the **root** of all other binding declarations and defines the location of the WSDL file and the namespace to which the XML Schema conforms: `http://java.sun.com/xml/ns/jaxws`.

The format of the root declaration is as follows:

```
<jaxws:bindings
    wsdlLocation="uri_of_wsdl"
    jaxws:xmlns="http://java.sun.com/xml/ns/jaxws">
```

`uri_of_wsdl` specifies the URI of the WSDL file.

The package, wrapper style, and asynchronous mapping customizations, defined in Table 5-13, can be *globally* defined as part of the root binding declaration in the external customization file. Global bindings apply to the entire scope of the `wsdl:definition` in the WSDL referenced by the `wsdlLocation` attribute.

The following provides an example of the root binding element that defines the package name, wrapper style, and asynchronous mapping customizations.

```
<jaxws:bindings
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="http://localhost:7001/simple/SimpleService?WSDL"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
```

```
    <package name="example.webservices.simple.simpleservice">
    <enableWrapperStyle>true</enableWrapperStyle>
    <enableAsyncMapping>false</enableAsyncMapping>
</jaxws:bindings>
```

Specifying Child Elements

The root `jaxws:bindings` element can contain **child elements**. You specify the WSDL node that is being customized by passing an XPath expression in the node attribute.

An XML Schema inlined inside a compiled WSDL file can be customized by using standard JAXB bindings. For more information, see "XML Schema Customization" in *JAX-WS WSDL Customizations*. For information about the custom JAXB binding declarations that you can define, see "JAXB Custom Binding Declarations" on page 5-30.

For example, the following example defines the package name as `examples.webservices.complex.complexservice` for the `wsdl:definitions` node of the WSDL document.

```
<jaxws:bindings
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="http://localhost:7001/simple/SimpleService?WSDL
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
  <jaxws:bindings node="wsdl:definitions"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <jaxws:package name="examples.webservices.simple.simpleservice"/>
</bindings>
```

Creating an External Binding Declarations File Using JAXB Binding Declarations

The JAXB binding declarations file is an XML document that conforms to the XML Schema for the following namespace: `http://java.sun.com/xml/ns/jaxb`. The following sections describe how to specify the root and child elements of the JAXB binding declarations file. For information about the custom binding declarations that you can define, see "JAXB Custom Binding Declarations" on page 5-30.

Specifying the Root Element

The `jaxb:bindings` declaration is the **root** of all other binding declarations. The format of the root declaration is as follows:

```
<jaxb:bindings
    schemaLocation="uri_of_schema">
```

`uri_of_schema` specifies the URI of the XML Schema file.

Specifying Child Elements

The root `jaxb:bindings` element can contain **child elements**. You specify the schema node that is being customized by passing an XPath expression in the node attribute.

For example, the following example defines the package name as `examples.webservices.simple.simpleservice`.

```
<jaxb:bindings
   schemaLocation="simpleservice.xsd">
   <jaxb:bindings node="//xs:simpleType[@name='value1']">
      <jaxb:package name="examples.webservices.simple.simpleservice"/>
   </jaxb:bindings>
</jaxb:bindings>
```

Embedding Binding Declarations

You can embed binding declarations in a WSDL file using one of the following methods:

- Embed a JAX-WS or JAXB binding declaration in the WSDL file using the `jaxws:bindings` element as a WSDL extension. See "Embedding JAX-WS or JAXB Binding Declarations in the WSDL File" on page 5-23.
- Embed a JAXB binding declaration in the XML Schema as part of an `<appinfo>` element. See "Embedding JAXB Binding Declarations in the XML Schema" on page 5-24.

Embedding JAX-WS or JAXB Binding Declarations in the WSDL File

You can embed a binding declaration in the WSDL file using the `jaxws:bindings` element as a WSDL extension. For information about the custom binding declarations that you can define, see "JAX-WS Custom Binding Declarations" on page 5-24.

For example, the following example defines the class name as `SimpleService` for the `SimpleServiceImpl` service endpoint interface (or port).

```
<wsdl:portType name="SimpleServiceImpl">
     <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
            <jaxws:class name="SimpleService"/>
     </jaxws:bindings>
</wsdl:portType>
```

If this binding declaration had not been specified, the class name of the service endpoint interface would be set to the `wsdl:portType` name—`SimpleServiceImpl`—by default.

An XML Schema inlined inside a compiled WSDL file can be customized by using standard JAXB bindings. For more information, see “XML Schema Customizations” in *JAX-WS WSDL Customizations*. For information about the custom JAXB binding declarations that you can define, see “JAXB Custom Binding Declarations” on page 5-30.

Embedding JAXB Binding Declarations in the XML Schema

You can embed a JAXB custom declaration within the `<appinfo>` element of the XML Schema, as illustrated below.

```
<xs:annotation>
    <xs:appinfo>
        <binding declaration>
    </xs:appinfo>
</xs:annotation>
```

For example, the following defines the package name for the schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.w3.org/2001/XMLSchema"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
    <annotation>
        <appinfo>
           <jaxb:schemaBindings>
           <jaxb:package name="example.webservices.simple.simpleservice"/>
           </jaxb:schemaBindings>
        </appinfo>
    </annotation>
</schema>
```

JAX-WS Custom Binding Declarations

The following table summarizes the typical JAX-WS customizations. For a complete list of JAX-WS custom binding declarations, see *JAX-WS WSDL Customization*.

Table 5-13 JAX-WS Custom Binding Declarations

Customization	Description

Package name	Use the `jaxws:package` binding declaration to define the package name.
	
	If you do not specify this customization, the `wsdlc` Ant task generates a package name based on the `targetNamespace` of the WSDL. This data binding customization is overridden by the `packageName` attribute of the `wsdlc`, `jwsc`, or `clientgen` Ant task. For more information, see "Ant Task Reference" in the *WebLogic Web Services Reference*.
	
	This binding declaration can be specified as part of the root binding element, as described in "Creating an External Binding Declarations File" on page 5-21, or on the `wsdl:definitions` node, as shown in the following example:
	
	`<bindings`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"`
	`  wsdlLocation=`
	
	`"http://localhost:7001/simple/SimpleService?WSDL"`
	`  xmlns="http://java.sun.com/xml/ns/jaxws">`
	`  <bindings node="wsdl:definitions"`
	
	`xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">`
	`    <package`
	`name="example.webservices.simple.simpleService"/>`
	`</bindings>`

Table 5-13 JAX-WS Custom Binding Declarations (Continued)

Customization	Description

Wrapper-style rules	Use the `jaxws:enablesWrapperStyle` binding declaration to enable or disable the wrapper style rules that control how the parameter types and return types of a WSDL operation are generated.
	
	This binding declaration can be specified as part of the root binding element, as described in "Creating an External Binding Declarations File" on page 5-21, or on one of the following nodes:
	• `wsdl:definitions`—Applies to all `wsdl:operations` of all `wsdl:portType` attributes.
	• `wsdl:portType`—Applies to all `wsdl:operations` in the `wsdl:portType`.
	• `wsdl:operation`—Applies to the `wsdl:operation` only.
	
	The following example disables the wrapper style rules for the `wsdl:definitions` node:
	
	`<bindings`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"`
	`wsdlLocation="http://localhost:7001/simple/Simple`
	`Service?WSDL"`
	`  xmlns="http://java.sun.com/xml/ns/jaxws">`
	`  <bindings node="wsdl:definitions"`
	
	`xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">`
	`    `**`<enableWrapperStyle>`**
	`       `**`false`**
	`    `**`</enableWrapperStyle>`**
	`</bindings>`

Table 5-13 JAX-WS Custom Binding Declarations (Continued)

Customization	Description

Asynchrony	Use the `jaxws:enableAsycMapping` binding declaration to instruct the `clientgen` Ant task to generate asynchronous polling and callback operations along with the normal synchronous methods when it compiles a WSDL file.
	
	This binding declaration can be specified as part of the root binding element, as described in "Creating an External Binding Declarations File" on page 5-21, or on one of the following nodes:
	• `wsdl:definitions`—Applies to all `wsdl:operations` of all `wsdl:portType` attributes.
	• `wsdl:portType`—Applies to all `wsdl:operations` in the `wsdl:portType`.
	• `wsdl:operation`—Applies to the `wsdl:operation` only.
	
	The following example disables the wrapper style rules for the `wsdl:definitions` node:
	
	`<bindings`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"`
	`wsdlLocation="http://localhost:7001/simple/Simple`
	`Service?WSDL"`
	`  xmlns="http://java.sun.com/xml/ns/jaxws">`
	`  <bindings node="wsdl:definitions"`
	
	`xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">`
	`    `**`<enableAsyncMapping>`**
	`       `**`false`**
	`    `**`</enableAsyncMapping>`**
	`</bindings>`

Provider	Use the `jaxws:provider` binding declaration to mark the part as a provider interface. This binding declaration can be specified as part of the `wsdl:portType`. This binding declaration applies when you are developing a service starting from a WSDL file.

Table 5-13 JAX-WS Custom Binding Declarations (Continued)

Customization	Description

Class name	Use the `jaxws:class` binding declaration to define the class name. This binding declaration can be specified for one of the following nodes:
	• `wsdl:portType`—Defines the interface class name.
	• `wsdl:fault`—Defines fault class names.
	• `soap:headerfault`—Defines exception class names.
	• `wsdl:service`—Defines the implementation class names.
	The following example defines the class name for the implementation class.
	`<bindings`
	`node="wsdl:definitions/wsdl:service[@name='Simple`
	`Service']">`
	    `<class name="myService"></class>`
	`</bindings>`

Method name	Use the `jaxws:method` binding declaration to customize the generated Java method name of a service endpoint interface or the port accessor method in the generated `Service` class.
	The following example defines the Java method name for the `wsdl:operation EchoHello`.
	`<bindings`
	`node="wsdl:definitions/wsdl:portType[@name='Simpl`
	`eServiceImpl']/wsdl:operation[@name='EchoHello']"`
	`>`
	    `<method name="Greeting"></method>`
	`</bindings>`

Table 5-13 JAX-WS Custom Binding Declarations (Continued)

Customization	Description

Java parameter name	Use the `jaxws:parameter` binding declaration to customize the parameter name of generated Java methods. This declaration can be used to change the method parameter of a `wsdl:operation` in a `wsdl:portType`.
	
	The following example defines the Java method name for the `wsdl:operation echoHello`.
	
	`<bindings`
	`node="wsdl:definitions/wsdl:portType[@name='Simpl`
	`eServiceImpl']/wsdl:operation[@name='EchoHello']"`
	`>`
	    `<parameter`
	`part="definitions/message[@name='EchoHello']/`
	    `part[@name='parameters']" element="hello"`
	    `name="greeting"/>`
	`</bindings>`

Javadoc	Use the `jaxws:javadoc` binding declaration to specify Javadoc text for a package, class, or method.
	
	For example, the following defines Javadoc at the method level.
	
	`<bindings`
	`node="wsdl:definitions/wsdl:portType[@name='Simpl`
	`eServiceImpl']/wsdl:operation[@name='EchoHello']"`
	`>`
	    `<method name="Hello">`
	       `<javadoc>Prints hello.</javadoc>`
	    `</method>`
	`</bindings>`

Handler chain	Use the `javaee:handlerchain` binding declaration to customize or add handlers. The inline handler must conform to the handler chain configuration defined in the Web Services Metadata for the Java Platform specification (JSR-181)

JAXB Custom Binding Declarations

The following table lists the typical JAXB customizations.

Note: The following table only summarizes the JAXB custom binding declarations, to help get you started. For a complete list and description of all JAXB custom binding declarations, see the JAXB specification or "Customizing JAXB Bindings" in the *Sun Java EE 5 Tutorial*.

Table 5-14 JAXB Custom Binding Declarations

Customization	Description

Global bindings	Use the `<globalBindings>` binding declaration to define binding declarations with global scope (see Figure 5-2).
	
	You can specify attributes and elements to the `<globalBindings>` binding declaration. For example, the following binding declaration defines:
	• `collectionType` attribute that specifies a type class, `myArray`, that implements the `java.util.List` interface and that is used to represent all lists in the generated implementation.
	• `generateIsSetMethod` attribute to generate the `isSet()` method corresponding to the getter and sestter property methods.
	• `javaType` element to customize the binding of an XML Schema atomic datatype to a Java datatype (built-in or application-specific).
	
	`<jaxb:globalBindings`
	`    collectionType ="java.util.myArray"`
	`    generateIsSetMethod="false">`
	`    <jaxb:javaType name="java.util.Date"`
	`        xmlType="xsd:date"`
	`    </jaxb:javaType>`
	`</jaxb:globalBindings>`

Schema bindings	Use the `<schemaBindings>` binding declaration to define binding declarations with schema scope (see Figure 5-2). For an example, see Package name.

Table 5-14 JAXB Custom Binding Declarations (Continued)

Customization	Description
Package name	Use the `<package>` element of the `<schemaBindings>` binding declaration to define the package name for the schema. If you do not specify this customization, the `wsdlc` Ant task generates a package name based on the `targetNamespace` of the WSDL. This data binding customization is overridden by the `packageName` attribute of the `wsdlc`, `jwsc`, or `clientgen` Ant task. For more information, see "Ant Task Reference" in the *WebLogic Web Services Reference*. For example, the following defines the package name for all JAXB classes generated from the `simpleservice.xsd` file: `<jaxb:bindings xmlns:xs="http://www.w3.org/2001/XMLSchema" schemaLocation="simpleservice.xsd" node="/xs:schema"> <jaxb:schemaBindings>` **`<jaxb:package name="examples.jaxb"/>`** `</jaxb:schemaBindings> </jaxb:bindings>` The following shows how to define the package name for an imported XML Schema: `<jaxb:bindindgs xmlns:xs="http://www.w3.org/2001/XMLSchema"` **`node="//xs:schema/xs:import[@namespace='http://examples.webservices.org/complexservice']"`**`> <jaxb:schemaBindings>` **`<jaxb:package name="examples.jaxb"/>`** `</jaxb:schemaBindings> </jaxb:bindings>`

Table 5-14 JAXB Custom Binding Declarations (Continued)

Customization	Description

Class name	Use the `<class>` binding declaration to define the class name for a schema element.
	
	The following example defines the class name for an `xsd:complexType`:
	
	`<xs:complexType name="ComplexType">`
	`    <xs:annotation><xs:appinfo>`
	`       <jaxb:class name="MyClass">`
	`           <jaxb:javadoc>This is my`
	`class.</jaxb:javadoc>`
	`       </jaxb:class>`
	`    </xs:appinfo></xs:annotation>`
	`</xs:complexType>`

Java property name	Use the `<property>` binding declaration to define the property name for a schema element.
	
	The following example
	
	`<jaxb:bindindgs`
	`    xmlns:xs="http://www.w3.org/2001/XMLSchema"`
	`    `**`node="//xs:schema/"`**`>`
	`    <jaxb:schemaBindings>`
	`      <jaxb:property generateIsSetMethod="true"/>`
	`    </jaxb:schemaBindings>`
	`</jaxb:bindings>`

Java datatype	Use the `<javaType>` binding declaration to customize the binding of an XML Schema atomic datatype to a Java datatype (built-in or application-specific). For example, see Global bindings.

Table 5-14 JAXB Custom Binding Declarations (Continued)

Customization	Description

Javadoc	Use the `<javadoc>` child element of the `<class>` or `<property>` binding declaration to specify Javadoc for the element.
	For example:
	`<xs:complexType name="ComplexType">`
	`    <xs:annotation><xs:appinfo>`
	`       <jaxb:class name="MyClass">`
	`           <jaxb:javadoc>This is my`
	`class.</jaxb:javadoc>`
	`       </jaxb:class>`
	`    </xs:appinfo></xs:annotation>`
	`</xs:complexType>`

Invoking Web Services

The following sections describe how to invoke WebLogic Web Services:

- "Overview of Web Services Invocation" on page 6-1
- "Invoking a Web Service from a Stand-alone Client: Main Steps" on page 6-2
- "Invoking a Web Service from Another Web Service" on page 6-10
- "Using a Stand-Alone Client JAR File When Invoking Web Services" on page 6-17
- "Client Considerations When Redeploying a Web Service" on page 6-18

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

Overview of Web Services Invocation

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

There are two types of client applications:

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

- A Java EE component deployed to WebLogic Server—In this type of client application, the Web Service runs inside a Java Platform, Enterprise Edition (Java EE) Version 5 component deployed to WebLogic Server, such as an EJB, servlet, or another Web Service. This type of client application, therefore, runs inside a WebLogic Server container.

The sections that follow describe how to use Oracle's implementation of the JAX-WS specification to invoke a Web Service from a Java client application. You can use this implementation to invoke Web Services running on any application server, both WebLogic and non-WebLogic. In addition, you can create a stand-alone client application or one that runs as part of a WebLogic Server.

WebLogic Server includes examples of creating and invoking WebLogic Web Services in the `WL_HOME/samples/server/examples/src/examples/webservices` directory, where `WL_HOME` refers to the main WebLogic Server directory. For detailed instructions on how to build and run the examples, open the `WL_HOME/samples/server/docs/index.html` Web page in your browser and expand the **WebLogic Server Examples->Examples->API->Web Services** node.

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

The following table summarizes the main steps to create a stand-alone client that invokes a Web Service. See also "Using a Stand-Alone Client JAR File When Invoking Web Services" on page 6-17.

Note: It is assumed that you use Ant in your development environment to build your client application, compile Java files, and so on, and that you have an existing `build.xml` file that you want to update with Web Services client tasks. For general information about using Ant in your development environment, see "Creating the Basic Ant build.xml File" on page 3-6. For a full example of a `build.xml` file used in this section, see "Sample Ant Build File for a Stand-Alone Java Client" on page 6-9.

Table 6-1 Steps to Invoke a Web Service from a Stand-alone Client

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the Oracle products and *`domainName`* is the name of your domain.
2	Update your `build.xml` file to execute the `clientgen` Ant task to generate the needed client-side artifacts to invoke a Web Service.	See "Using the clientgen Ant Task To Generate Client Artifacts" on page 6-3.
3	Get information about the Web Service, such as the signature of its operations and the name of the ports.	See "Getting Information About a Web Service" on page 6-5.
4	Write the client application Java code that includes code for invoking the Web Service operation.	See "Writing the Java Client Application Code to Invoke a Web Service" on page 6-6.
5	Create a basic Ant build file, `build.xml`.	See "Creating the Basic Ant build.xml File" on page 3-6.
6	Compile and run your Java client application.	See "Compiling and Running the Client Application" on page 6-7.

Using the clientgen Ant Task To Generate Client Artifacts

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

- The Java class for the `Service` interface implementation for the particular Web Service you want to invoke.

- JAXB data binding artifacts.
- The Java class for any user-defined XML Schema data types included in the WSDL file.

For additional information about the `clientgen` Ant task, such as all the available attributes, see "Ant Task Reference" in the *WebLogic Web Services Reference*.

Update your `build.xml` file, adding a call to the `clientgen` Ant task, as shown in the following example:

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

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

Before you can execute the `clientgen` WebLogic Web Service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task.

You must include the `wsdl` and `destDir` attributes of the `clientgen` Ant task to specify the WSDL file from which you want to create client-side artifacts and the directory into which these artifacts should be generated. The `packageName` attribute is optional; if you do not specify it, the `clientgen` task uses a package name based on the `targetNamespace` of the WSDL. The `type` is required in this example; otherwise, it defaults to `JAXRPC`.

In this example, the package name is set to the same package name as the client application, `examples.webservices.simple_client`. If you set the package name to one that is different from the client application, you would need to import the appropriate class files. For example, if you defined the package name as `examples.webservices.complex`, you would need to import the following class files in the client application:

```
import examples.webservices.complex.BasicStruct;
import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;
```

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

If the WSDL file specifies that user-defined data types are used as input parameters or return values of Web Service operations, `clientgen` automatically generates a JavaBean class that is the Java representation of the XML Schema data type defined in the WSDL. The JavaBean classes are generated into the `destDir` directory.

For a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, see "Sample Ant Build File for a Stand-Alone Java Client" on page 6-9.

To execute the `clientgen` Ant task, along with the other supporting Ant tasks, specify the `build-client` target at the command line:

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

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

Getting Information About a Web Service

You need to know the name of the Web Service and the signature of its operations before you write your Java client application code to invoke an operation. There are a variety of ways to find this information.

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

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

- The *`PortType`*`.java` file contains the method signatures that correspond to the public operations of the Web Service, where *`PortType`* refers to the port type of the Web Service. If the Web Service was implemented with a JWS file, the port type is the value of the `name` attribute of the `@WebService` JWS annotation.

You can also examine the actual WSDL of the Web Service; see "Browsing to the WSDL of the Web Service" on page 3-17 for details about the WSDL of a deployed WebLogic Web Service. The name of the Web Service is contained in the `<service>` element, as shown in the following excerpt of the `TraderService` WSDL:

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

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

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

Writing the Java Client Application Code to Invoke a Web Service

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

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

Because the `<clientgen>` `packageName` attribute was set to the same package name as the client application, we are not required to import the `<clientgen>`-generated files.

```
package examples.webservices.simple_client;

/**
 * This is a simple stand-alone client application that invokes the
 * the echoComplexType operation of the ComplexService Web service.
 */

public class Main {

  public static void main(String[] args) {

    ComplexService test = new ComplexService(),
    ComplexPortType port = test.getComplexPortTypePort();

    BasicStruct in = new BasicStruct();

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

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

In the preceding example:

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

  ```
  ComplexService test = new ComplexService(),
  ComplexPortType port = test.getComplexPortTypePort();
  ```

  The `ComplexService` class implements the JAX-WS `Service` interface. The `getComplexServicePortTypePort()` method is used to return an instance of the `ComplexPortType` stub implementation.

- The following code shows how to invoke the `echoComplexType` operation of the `ComplexService` Web Service:

  ```
  BasicStruct result = port.echoComplexType(in);
  ```

  The `echoComplexType` operation returns the user-defined data type called `BasicStruct.`

Compiling and Running the Client Application

Add `javac` tasks to the `build-client` target in the `build.xml` file to compile all the Java files (both of your client application and those generated by `clientgen`) into class files, as shown by the **bold** text in the following example:

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

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

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

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

  </target>
```

In the example, the first `javac` task compiles the Java files in the `clientclasses` directory that were generated by `clientgen,` and the second `javac` task compiles the Java files in the `examples/webservices/simple_client` subdirectory of the current directory; where it is assumed your Java client application source is located.

In the preceding example, the `clientgen`-generated Java source files and the resulting compiled classes end up in the same directory (`clientclasses`). Although this might be adequate for prototyping, it is often a best practice to keep source code (even generated code) in a different directory from the compiled classes. To do this, set the `destdir` for both `javac` tasks to a directory different from the `srcdir` directory.

To run the client application, add a `run` target to the `build.xml` that includes a call to the `java` task, as shown below:

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

<target name="run" >
    <java
       fork="true"
       classname="examples.webServices.simple_client.Main"
```

```
        failonerror="true" >
        <classpath refid="client.class.path"/>
</target>
```

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

See "Sample Ant Build File for a Stand-Alone Java Client" on page 6-9 for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`.

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

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

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

Sample Ant Build File for a Stand-Alone Java Client

The following example shows a complete `build.xml` file for generating and compiling a stand-alone Java client. See "Using the clientgen Ant Task To Generate Client Artifacts" on page 6-3 and "Compiling and Running the Client Application" on page 6-7 for explanations of the sections in **bold**.

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

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

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

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

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

  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <target name="clean" >
    <delete dir="${clientclass-dir}"/>
  </target>

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

  <target name="build-client">

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

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

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

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

</project>
```

Invoking a Web Service from Another Web Service

Invoking a Web Service from within a WebLogic Web Service is similar to invoking one from a stand-alone Java application, as described in "Invoking a Web Service from a Stand-alone Client: Main Steps" on page 6-2, with the following variations:

- Instead of using the `clientgen` Ant task to generate the JAX-WS `Service` interface of the Web Service to be invoked, you use the `<clientgen>` child element of the `<jws>` element, inside the `jwsc` Ant task that compiles the invoking Web Service. In the JWS file that invokes the other Web Service, however, you still use the same standard JAX-WS APIs to get `Service` and `PortType` instances to invoke the Web Service operations.

- You can use the `@WebServiceRef` annotation to define a reference to a Web Service, as described in “Defining a Web Service Reference Using the @WebServiceRef Annotation” on page 6-15.

This section describes the differences between invoking a Web Service from a client in a Java EE component and invoking from a stand-alone client. It is assumed that you have read and understood “Invoking a Web Service from a Stand-alone Client: Main Steps” on page 6-2. It is also assumed that you use Ant in your development environment to build your client application, compile Java files, and so on, and that you have an existing `build.xml` that builds a Web Service that you want to update to invoke another Web Service.

The following list describes the changes you must make to the `build.xml` file that builds your client Web Service, which will invoke another Web Service. See “Sample build.xml File for a Web Service Client” on page 6-12 for the full sample `build.xml` file:

- Add a `<clientgen>` child element to the `<jws>` element that specifies the JWS file that implements the Web Service that invokes another Web Service. Set the required `wsdl` attribute to the WSDL of the Web Service to be invoked. Set the required `packageName` attribute to the package into which you want the JAX-WS client stubs to be generated.

The following list describes the changes you must make to the JWS file that implements the client Web Service; see “Sample JWS File That Invokes a Web Service” on page 6-14 for the full JWS file example.

- Import the files generated by the `<clientgen>` child element of the `jwsc` Ant task. These include the JAX-WS `Service` interface of the invoked Web Service, as well as the Java representation of any user-defined data types used as parameters or return values in the operations of the invoked Web Service.

  Note: If the package name set using the `packageName` attribute of `<clientgen>` is set to the same package name as the client application, then you are not required to import the `<clientgen>`-generated files.

- Get the `Service` and `PortType` interface implementation and invoke the operation on the port as usual; see “Writing the Java Client Application Code to Invoke a Web Service” on page 6-6 for details.

Sample build.xml File for a Web Service Client

The following sample `build.xml` file shows how to create a Web Service that itself invokes another Web Service; the relevant sections that differ from the `build.xml` for building a simple Web Service that does not invoke another Web Service are shown in **bold**.

The `build-service` target in this case is very similar to a target that builds a simple Web Service; the only difference is that the `jwsc` Ant task that builds the invoking Web Service also includes a `<clientgen>` child element of the `<jws>` element so that `jwsc` also generates the required JAX-RPC client stubs.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    </jwsc>

  </target>

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

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

  <target name="client">

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

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

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

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

  </target>

</project>
```

Sample JWS File That Invokes a Web Service

The following sample JWS file, called `ClientServiceImpl.java`, implements a Web Service called `ClientService` that has an operation that in turn invokes the `echoComplexType` operation of a Web Service called `ComplexService`. This operation has a user-defined data type (`BasicStruct`) as both a parameter and a return value. The relevant code is shown in **bold** and described after the example.

```
package examples.webservices.service_to_service;

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

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

import examples.webservices.complex.BasicStruct;

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

import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;

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

public class ClientServiceImpl {

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

    // Create service and port stubs to invoke ComplexService
    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();
    // Invoke the echoComplexType operation of ComplexService
    BasicStruct result = port.echoComplexType(input);
    System.out.println("Invoked ComplexPortType.echoComplexType." );

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

  }
}
```

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

- Import any user-defined data types that are used by the invoked Web Service. In this example, the `ComplexService` uses the `BasicStruct` JavaBean:

  ```
  import examples.webservices.complex.BasicStruct;
  ```

- Import the JAX-WS interfaces of the `ComplexService` Web Service; the stubs are generated by the `<cliengen>` child element of `<jws>`:

  ```
  import examples.webservices.complex.ComplexPortType;
  import examples.webservices.complex.ComplexService;
  ```

- Create the JAX-WS `Service` and `PortType` instances for the `ComplexService`:

  ```
  ComplexService test = new ComplexService();
  ComplexPortType port = test.getComplexPortTypePort();
  ```

- Invoke the `echoComplexType` operation of `ComplexService` using the port you just instantiated:

  ```
  BasicStruct result = port.echoComplexType(input);
  ```

Defining a Web Service Reference Using the @WebServiceRef Annotation

The `@WebServiceRef` annotation enables you to define a reference to a Web Service. For example, in the following sample, a reference to the `ComplexService` is defined by passing the WSDL of the Web Service to the `@WebServiceRef` annotation.

```
package examples.webservices.service_to_service;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.WebServiceRef;

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

```
import examples.webservices.complex.BasicStruct;

// Import the JAX-WS interfaces for invoking the ComplexService Web Service.
// Interfaces generated by clientgen

import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;

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

public class ClientServiceImpl {
    @WebServiceRef()
    ComplexService service;

  @WebMethod()
  public String callComplexService(BasicStruct input)
  {

    // Create service and port stubs to invoke ComplexService
    ComplexPortType port = service.getComplexPortTypePort();

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

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

  }
}
```

In the preceding example:

- The `@WebServiceRef` annotation is used to define a reference to a Web Service and an injection target for it:

  ```
  @WebServiceRef()
  ComplexService service;
  ```

- The following code shows how to return an instance of the `ComplexPortType` stub implementation using the Web Service reference:

  ```
  ComplexPortType port = service.getComplexPortTypePort();
  ```

- The following code shows how to invoke the `sayHello` operation of the `ComplexService` Web Service:

  ```
  BasicStruct result = port.echoComplexType(input);
  ```

Using a Stand-Alone Client JAR File When Invoking Web Services

It is assumed in this document that, when you invoke a Web Service using the client-side artifacts generated by the `clientgen` or `wsdlc` Ant tasks, you have the entire set of WebLogic Server classes in your CLASSPATH. If, however, your computer does *not* have WebLogic Server installed, you can still invoke a Web Service by using the stand-alone WebLogic Web Services client JAR file, as described in this section.

The standalone client JAR file supports basic client-side functionality, such as:

- Use with client-side artifacts created by both the `clientgen` Ant tasks
- Processing SOAP messages
- Using client-side SOAP message handlers
- Using MTOM
- Invoking both JAX-WS and JAX-RPC Web Services
- Using SSL

The stand-alone client JAR file does *not*, however, support invoking Web Services that use the following advanced feature:

- Message-level security (WS-Security)

To use the stand-alone WebLogic Web Services client JAR file with your client application, follow these steps:

1. Copy the file *`WL_HOME`*`/server/lib/wseeclient.zip` from the computer hosting WebLogic Server to the client computer, where *`WL_HOME`* refers to the WebLogic Server installation directory, such as `/bea/wlserver_10.3`.
2. Unzip the `wseeclient.zip` file into the appropriate directory. For example, you might unzip the file into a directory that contains other classes used by your client application.
3. Add the `wseeclient.jar` file (unzipped from the `wseeclient.zip` file) to your CLASSPATH.

   Note: Also be sure that your CLASSPATH includes the JAR file that contains the Ant classes (`ant.jar`). This JAR file is typically located in the `lib` directory of the Ant distribution.

4. If you are using JDK 6 Update 2, note that this version of the Sun JDK only supports JAX-WS 2.0 and JAXB 2.0 APIs. In order to use JDK 6 Update 2 with this release, you need to update the API JARs in your JDK6 installation. To do so:

   a. Copy the following JARs from the WebLogic Server installation into your Sun JDK6 endorsed directory:

      - `$BEA_HOME/modules/javax.xml.bind_2.1.1 jar to JDK6_HOME/jre/lib/endorsed`
      - `$BEA_HOME/modules/javax.xml.ws_2.1.1 jar to JDK6_HOME/jre/lib/endorsed`

   b. Use the Java endorsed library to override the existing JDK 6 Update 2 library files, as described in *Java Endorsed Standards Override Mechanism*.

Client Considerations When Redeploying a Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated WebLogic Web Service alongside an older version of the same Web Service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web Service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web Service.

You can continue using the old client application with the new version of the Web Service, as long as the following Web Service artifacts have not changed in the new version:

- WSDL that describes the Web Service
- WS-Policy files attached to the Web Service

If any of these artifacts have changed, you must regenerate the JAX-WS stubs used by the client application by re-running the `clientgen` Ant task.

For example, if you change the signature of an operation in the new version of the Web Service, then the WSDL file that describes the new version of the Web Service will also change. In this case, you must regenerate the JAX-WS stubs. If, however, you simply change the implementation of an operation, but do not change its public contract, then you can continue using the existing client application.

Administering Web Services

The following sections describe how to administer WebLogic Web Services:

Overview of WebLogic Web Services Administration Tasks

When you use the `jwsc` Ant task to compile and package a WebLogic Web Service, the task packages it as part of an Enterprise Application. The Web Service itself is packaged inside the Enterprise application as a Web application WAR file, by default. However, if your JWS file implements a session bean then the Web Service is packaged as an EJB JAR file. Therefore, basic administration of Web Services is very similar to basic administration of standard Java Platform, Enterprise Edition (Java EE) Version 5 applications and modules. These standard tasks include:

- Installing the Enterprise application that contains the Web Service.
- Starting and stopping the deployed Enterprise application.
- Configuring the Enterprise application and the archive file which implements the actual Web Service. You can configure general characteristics of the Enterprise application, such as the deployment order, or module-specific characteristics, such as session time-out for Web applications or transaction type for EJBs.
- Creating and updating the Enterprise application's deployment plan.
- Monitoring the Enterprise application.
- Testing the Enterprise application.

The following administrative tasks are specific to Web Services:

- Configuring the WS-Policy files associated with a Web Service endpoint or its operations.

  Note: If you used the `@Policy` annotation in your Web Service to specify an associated WS-Policy file at the time you programmed the JWS file, you cannot change this association at run-time using the Administration Console or other administrative tools. You can only associate a *new* WS-Policy file, or disassociate one you added at run-time.

- Viewing the SOAP handlers associated with the Web Service.
- Viewing the WSDL of the Web Service.
- Creating a Web Service security configuration.

Administration Tools

There are a variety of ways to administer Java EE modules and applications that run on WebLogic Server, including Web Services; use the tool that best fits your needs:

- Using the Administration Console
- Using the WebLogic Scripting Tool
- Using WebLogic Ant Tasks
- Using the Java Management Extensions (JMX)

- Using the Java EE Deployment API

Using the Administration Console

The WebLogic Server Administration Console is a Web browser-based, graphical user interface you use to manage a WebLogic Server domain, one or more WebLogic Server instances, clusters, and applications, including Web Services, that are deployed to the server or cluster.

One instance of WebLogic Server in each domain is configured as an Administration Server. The Administration Server provides a central point for managing a WebLogic Server domain. All other WebLogic Server instances in a domain are called Managed Servers. In a domain with only a single WebLogic Server instance, that server functions both as Administration Server and Managed Server. The Administration Server hosts the Administration Console, which is a Web Application accessible from any supported Web browser with network access to the Administration Server.

You can use the System Administration Console to:

- Install an Enterprise application.
- Start and stop a deployed Enterprise application.
- Configure an Enterprise application.
- Configure Web applications.
- Configure EJBs.
- Create a deployment plan.
- Update a deployment plan.
- Test the modules in an Enterprise application.
- Associate the WS-Policy file with a Web Service.
- View the SOAP message handlers of a Web Service.
- View the WSDL of a Web Service.
- Create a Web Service security configuration

Invoking the Administration Console

To invoke the Administration Console in your browser, enter the following URL:

`http://`*`host`*`:`*`port`*`/console`

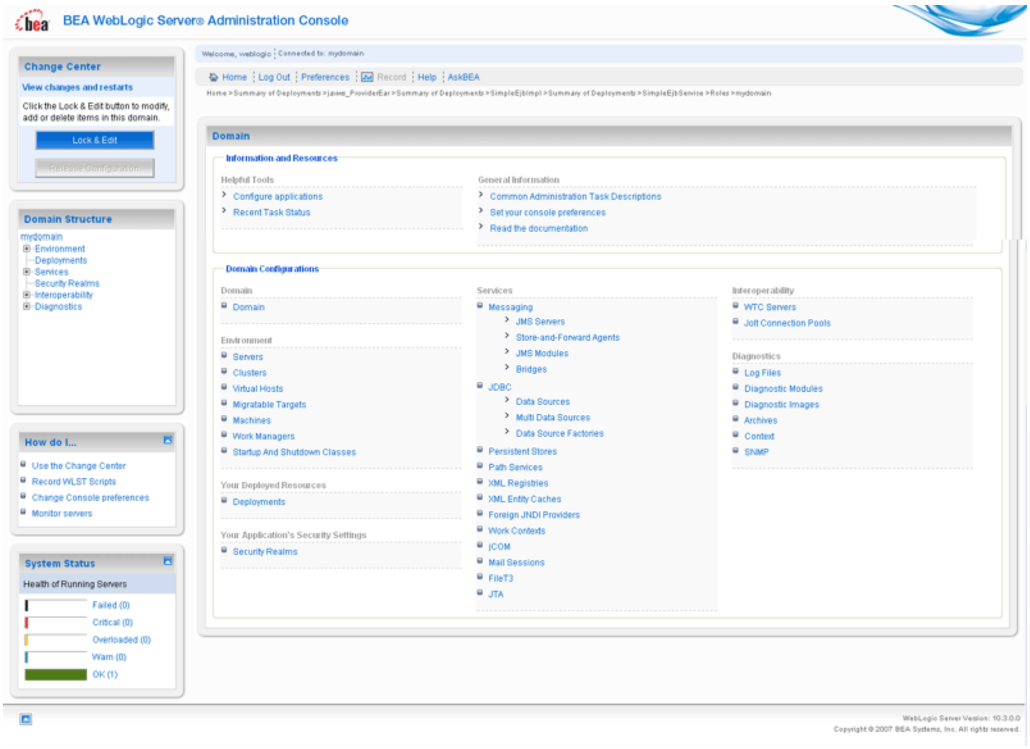
where

- *`host`* refers to the computer on which the Administration Server is running.
- *`port`* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001.

Click the **Help** button, located at the top right corner of the Administration Console, to invoke the Online Help for detailed instructions on using the Administration Console.

The following figure shows the main Administration Console window.

Figure 7-1 WebLogic Server Administration Console Main Window



Click the Lock & Edit button to modify, add or delete items in this domain.

How Web Services Are Displayed In the Administration Console

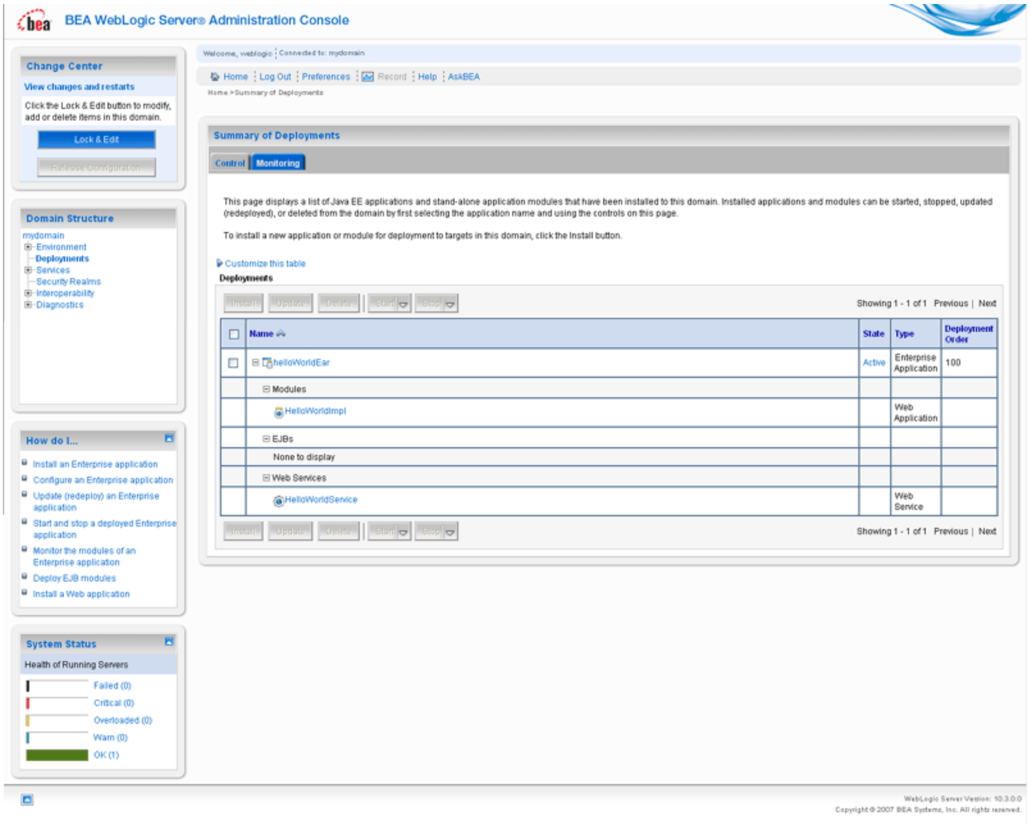
Web Services are typically deployed to WebLogic Server as part of an Enterprise Application. The Enterprise Application can be either archived as an EAR, or be in exploded directory format. The Web Service itself is almost always packaged as a Web Application; the only exception is if your JWS file implements a session bean in which case it is packaged as an EJB. The Web Service can be in archived format (WAR or EJB JAR file, respectively) or as an exploded directory.

It is not required that a Web Service be installed as part of an Enterprise application; it can be installed as just the Web Application or EJB. However, Oracle recommends that users install the Web Service as part of an Enterprise application. The WebLogic Ant task used to create a Web Service, `jwsc`, always packages the generated Web Service into an Enterprise application.

To view and update the Web Service-specific configuration information about a Web Service using the Administration Console, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the Web Service is packaged. Expand the application by clicking the **+** node; the Web Services in the application are listed under the **Web Services** category. Click on the name of the Web Service to view or update its configuration.

The following figure shows how the `HelloWorldService` Web Service, packaged inside the `helloWorldEar` Enterprise application, is displayed in the **Deployments** table of the Administration Console.

Figure 7-2 Web Service Displayed in Deployments Table of Administration Console



BEA WebLogic Server® Administration Console

Welcome, weblogic | Connected to: mydomain

Home | Log Out | Preferences | Record | Help | AskBEA

Home >Summary of Deployments

Change Center

View changes and restarts

Click the Lock & Edit button to modify, add or delete items in this domain.

Lock & Edit

Release Configuration

Domain Structure

mydomain
- Environment
- Deployments
- Services
- Security Realms
- Interoperability
- Diagnostics

How do I...

- Install an Enterprise application
- Configure an Enterprise application
- Update (redeploy) an Enterprise application
- Start and stop a deployed Enterprise application
- Monitor the modules of an Enterprise application
- Deploy EJB modules
- Install a Web application

System Status

Health of Running Servers

Failed (0)
Critical (0)
Overloaded (0)
Warn (0)
OK (1)

Summary of Deployments

Control | Monitoring

This page displays a list of Java EE applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redeployed), or deleted from the domain by first selecting the application name and using the controls on this page.

To install a new application or module for deployment to targets in this domain, click the Install button.

Customize this table

Deployments

Install | Update | Delete | Start | Stop — Showing 1 - 1 of 1 Previous | Next

	Name	State	Type	Deployment Order
☐	⊟ helloWorldEar	Active	Enterprise Application	100
	⊟ Modules			
	HelloWorldImpl		Web Application	
	⊟ EJBs			
	None to display			
	⊟ Web Services			
	HelloWorldService		Web Service	

Install | Update | Delete | Start | Stop — Showing 1 - 1 of 1 Previous | Next

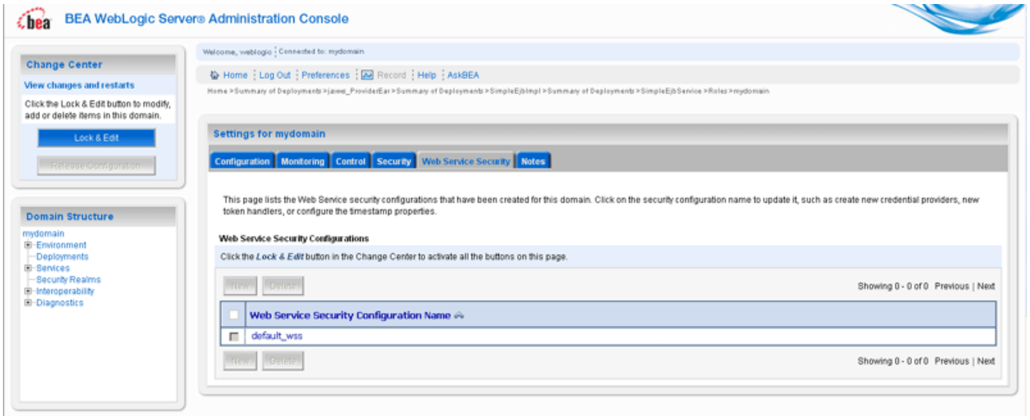
WebLogic Server Version: 10.3.0.0
Copyright © 2007 BEA Systems, Inc. All rights reserved.

Creating a Web Services Security Configuration

When a deployed WebLogic Web Service has been configured to use message-level security (encryption and digital signatures, as described by the WS-Security specification), the Web Services runtime determines whether a Web Service security configuration is also associated with the service. This security configuration specifies information such as whether to use an X.509 certificate for identity, whether to use password digests, the keystore to be used for encryption, and so on. A single security configuration can be associated with many Web Services.

Because Web Services security configurations are domain-wide, you create them from the ***domainName*** > **WebService Security** tab of the Administration Console, rather than the **Deployments** tab. The following figure shows the location of this tab.

Figure 7-3 Web Service Security Configuration in Administration Console



Welcome, weblogic | Connected to: mydomain

Home >Summary of Deployments >jaxws_ProviderEar >Summary of Deployments >SimpleEjbImpl >Summary of Deployments >SimpleEjbService >Roles >mydomain

Click the Lock & Edit button to modify, add or delete items in this domain.

This page lists the Web Service security configurations that have been created for this domain. Click on the security configuration name to update it, such as create new credential providers, new token handlers, or configure the timestamp properties.

Web Service Security Configurations

Click the *Lock & Edit* button in the Change Center to activate all the buttons on this page.

Showing 0 - 0 of 0

	Web Service Security Configuration Name
	default_wss

Showing 0 - 0 of 0

Using the WebLogic Scripting Tool

The WebLogic Scripting Tool (WLST) is a command-line scripting interface that you can use to interact with and configure WebLogic Server domains and instances, as well as deploy Java EE modules and applications (including Web Services) to a particular WebLogic Server instance. Using WLST, system administrators and operators can initiate, manage, and persist WebLogic Server configuration changes.

Typically, the types of WLST commands you use to administer Web Services fall under the Deployment category.

For more information on using WLST, see *WebLogic Scripting Tool*.

Using WebLogic Ant Tasks

WebLogic Server includes a variety of Ant tasks that you can use to centralize many of the configuration and administrative tasks into a single Ant build script. These Ant tasks can:

- Create, start, and configure a new WebLogic Server domain, using the `wlserver` and `wlconfig` Ant tasks.
- Deploy a compiled application to the newly-created domain, using the `wldeploy` Ant task.

See "Using Ant Tasks to Configure and Use a WebLogic Server Domain" and "wldeploy Ant Task Reference" in *Developing Applications With WebLogic Server* for specific information about the non-Web Services related WebLogic Ant tasks.

Using the Java Management Extensions (JMX)

A managed bean (MBean) is a Java bean that provides a Java Management Extensions (JMX) interface. JMX is the Java EE solution for monitoring and managing resources on a network. Like SNMP and other management standards, JMX is a public specification and many vendors of commonly used monitoring products support it.

WebLogic Server provides a set of MBeans that you can use to configure, monitor, and manage WebLogic Server resources through JMX. WebLogic Web Services also have their own set of MBeans that you can use to perform some Web Service administrative tasks.

There are two types of MBeans: runtime (for read-only monitoring information) and configuration (for configuring the Web Service after it has been deployed).

The configuration Web Services MBeans are:

- `WebserviceSecurityConfigurationMBean`
- `WebserviceCredentialProviderMBean`
- `WebserviceSecurityMBean`
- `WebserviceSecurityTokenMBean`
- `WebserviceTimestampMBean`
- `WebserviceTokenHandlerMBean`

The runtime Web Services MBeans are:

- `WseeRuntimeMBean`
- `WseeHandlerRuntimeMBean`
- `WseePortRuntimeMBean`
- `WseeOperationRuntimeMBean`
- `WseePolicyRuntimeMBean`

For more information on JMX, see the WebLogic Server MBean Reference and the following sections in *Developing Custom Management Utilities With JMX*:

- Understanding WebLogic Server MBeans
- Accessing WebLogic Server MBeans with JMX
- Managing a Domain's Configuration with JMX

Using the Java EE Deployment API

In Java EE 5, the J2EE Application Deployment specification (JSR-88) defines a standard API that you can use to configure an application for deployment to a target application server environment.

The specification describes the Java EE Deployment architecture, which in turn defines the contracts that enable tools or application programmers to configure and deploy applications on any Java EE platform product. The contracts define a uniform model between tools and Java EE platform products for application deployment configuration and deployment. The Deployment architecture makes it easier to deploy applications: Deployers do not have to learn all the features of many different Java EE deployment tools in order to deploy an application on many different Java EE platform products.

See *Deploying Applications to WebLogic Server* for more information.

Using Work Managers to Prioritize Web Services Work and Reduce Stuck Execute Threads

After a connection has been established between a client application and a Web Service, the interactions between the two are ideally smooth and quick, whereby the client makes requests and the service responds in a prompt and timely manner. Sometimes, however, a client application might take a long time to make a new request, during which the Web Service waits to respond, possibly for the life of the WebLogic Server instance; this is often referred to as a *stuck execute thread*. If, at any given moment, WebLogic Server has a lot of stuck execute threads, the overall performance of the server might degrade.

If a particular Web Service gets into this state fairly often, you can specify how the service prioritizes the execution of its work by configuring a Work Manager and applying it to the service. For example, you can configure a *response time request class* (a specific type of Work Manager component) that specifies a response time goal for the Web Service.

The following shows an example of how to define a response time request class in a deployment descriptor:

```
<work-manager>
    <name>responsetime_workmanager</name>
        <response-time-request-class>
            <name>my_response_time</name>
            <goal-ms>2000</goal-ms>
        </response-time-request-class>
</work-manager>
```

You can configure the response time request class using the Administration Console, as described in “Work Manager: Response Time: Configuration” in the *Administration Console Online Help*.

For more information about Work Managers in general and how to configure them for your Web Service, see “Using Work Managers to Optimize Scheduled Work” in *Configuring WebLogic Server Environments*.

Migrating JAX-RPC Web Services and Clients to JAX-WS

This section provides tips for migrating JAX-RPC Web Services and clients to JAX-WS. The following table summarizes the topics that are covered.

Note: In some cases, a JAX-RPC feature may not be supported currently by JAX-WS, such as WebLogic Reliable Messaging, conversational or buffered Web Services, and so on. In this case, the application cannot be migrated unless it is re-architected.

Table 8-1 Tips for Migrating JAX-RPC Web Services and Clients to JAX-WS

Topic	Description
Setting the Final Context Root of a WebLogic Web Service	Describes the methods that can be used to set the final context root of a WebLogic Web Service. The use of `@WLXXXTransport` JWS annotations is not supported for JAX-WS; these annotations are supported by JAX-RPC only.
Using WebLogic-specific Annotations	Describes the WebLogic-specific annotations that are supported by JAX-WS.
Generating a WSDL File	Describes how to generate a WSDL file when you are generating a JAX-WS Web Service using the `jwsc` Ant task.
Using JAXB Custom Types	Describes the use of Java Architecture for XML Binding (JAXB) for managing all of the data binding tasks.
Using EJB 3.0	Describes changes in EJB 3.0 from EJB 2.1. JAX-WS supports EJB 3.0. JAX-RPC supports EJB 2.1 only.
Migrating from RPC Style SOAP Binding	Provides guidelines for setting the SOAP binding. RPC style is supported, but not recommended for JAX-WS.

Table 8-1 Tips for Migrating JAX-RPC Web Services and Clients to JAX-WS (Continued)

Topic	Description
Updating SOAP Message Handlers	Explains how you must re-write your JAX-RPC SOAP message handlers when migrating to JAX-WS.
Invoking JAX-WS Clients	Explains how you must re-write your JAX-RPC client to invoke JAX-WS clients.

Setting the Final Context Root of a WebLogic Web Service

You can set the final context root of a WebLogic Web Service using a variety of methods, as described in "How to Determine the Final Context Root of a WebLogic Web Service" in *WebLogic Web Services Reference*.

As described in this section, when defining a JAX-RPC Web Service, you can use the `@WLXXXTransport` JWS annotations to specify the context root. For JAX-WS Web Services, the `@WLXXXTransport` JWS annotations are not valid. If used in the JAX-RPC Web Service, the JWS file needs to be updated to remove the annotations in favor of one of the other methods.

Using WebLogic-specific Annotations

JAX-WS supports the following WebLogic-specific annotations:

- `@Policy`
- `@Policies`
- `@WssConfiguration`

All other WebLogic-specific annotations must be removed from your JAX-RPC applications when migrating to JAX-WS. For more information, see "WebLogic-specific Annotations" in *WebLogic Web Services Reference*.

Generating a WSDL File

When you run the `jwsc` file on a JAX-RPC Web Service, a WSDL file is generated in the specified output directory. For JAX-WS Web Services, the WSDL file is generated when the service endpoint is deployed. In order to generate a WSDL file in the output directory, you must specify the `wsdlOnly` attribute of the `<jws>` child element of the `jwsc` Ant task. For more information, see "jwsc" in the *WebLogic Web Services Reference*.

Using JAXB Custom Types

JAX-WS uses Java Architecture for XML Binding (JAXB) to manage all of the data binding tasks. If your application supports custom types using XMLBeans or Tylar, you will need to modify them to use JAXB. For more information about using JAXB, see "Using JAXB Data Binding" on page 5-1.

Using EJB 3.0

JAX-WS supports EJB 3.0. JAX-RPC supports EJB 2.1 only.

EJB 3.0 introduced metadata annotations that enable you to automatically generate, rather than manually create, the EJB Remote and Home interface classes and deployment descriptor files needed when implementing an EJB.

For more information about EJB 3.0 bean class requirements and changes from 2.x, see "Programming the Bean File: Requirements and Changes from 2.X" in *Enterprise JavaBeans (EJB) 3.0*.

Migrating from RPC Style SOAP Binding

Use of the `SOAPBinding.Style.RPC` style, although supported, is not recommended with JAX-WS. It is recommended that you change the style to `SOAPBinding.Style.DOCUMENT`.

Updating SOAP Message Handlers

Although the SOAP APIs are similar, JAX-RPC SOAP handlers will need to be modified to run with JAX-WS. For more information, see "Creating and Using SOAP Message Handlers" in *Programming Advanced Features of WebLogic Web Services Using JAX-WS*.

Invoking JAX-WS Clients

JAX-RPC clients will need to be re-written as the JAX-RPC and JAX-WS client APIs are completely different. For more information about writing JAX-WS clients, see "Invoking Web Services" in *Getting Started With WebLogic Web Services Using JAX-WS*.