Oracle® Fusion Middleware

Getting Started With JAX-WS Web Services for Oracle WebLogic Server

11*g* Release 1 (10.3.3)

E13758-02

April 2010

This document describes how to develop WebLogic Web services using the Java API for XML-based Web services (JAX-WS).

ORACLE

Oracle Fusion Middleware Getting Started With JAX-WS Web Services for Oracle WebLogic Server, 11*g* Release 1 (10.3.3)

E13758-02

Contents

4 Programming the JWS File

5 Using JAXB Data Binding

6 Invoking Web Services

7 Administering Web Services

8 Migrating JAX-RPC Web Services and Clients to JAX-WS

Preface

This preface describes the document accessibility features and conventions used in this guide—*Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction

This document describes how to program WebLogic Web services using the 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 Oracle WebLogic Server*. For information about migrating a JAX-RPC Web service to JAX-WS, see Chapter 8, "Migrating JAX-RPC Web Services and Clients to JAX-WS."

The following table summarizes the contents of this guide.

Table 1–1 Content Summary

This section . . .	Describes how to . . .
Chapter 2, "Use Cases and Examples"	Run common use cases and examples.
Chapter 3, "Developing WebLogic Web Services"	Develop Web services using the WebLogic development environment.
Chapter 4, "Programming the JWS File"	Program the JWS file that implements your Web service.
Chapter 5, "Using JAXB Data Binding"	Use the Java Architecture for XML Binding (JAXB) data binding.
Chapter 6, "Invoking Web Services"	Invoke your Web service from a stand-alone client or another Web service.
Chapter 7, "Administering Web Services"	Administer WebLogic Web services using the Administration Console.
Chapter 8, "Migrating JAX-RPC Web Services and Clients to JAX-WS"	Migrate a JAX-RPC Web service to JAX-WS.

Note: The JAX-WS implementation in Oracle WebLogic Server is extended from the JAX-WS Reference Implementation (RI) developed by the Glassfish Community (see https://jax-ws.dev.java.net/). All features defined in the JAX-WS specification (JSR-224) are fully supported by Oracle WebLogic Server.

The JAX-WS RI also contains a variety of extensions, provided by Glassfish contributors. Unless specifically documented, JAX-WS RI extensions are not supported for use in Oracle WebLogic Server.

For an overview of WebLogic Web services, standards, samples, and related documentation, see *Introducing WebLogic Web Services for Oracle WebLogic Server*. For information about WebLogic Web service security, see *Securing WebLogic Web Services for Oracle WebLogic Server*.

A Note About Upgrading Existing WebLogic Web Services

There are no steps required to upgrade a 10.*x* WebLogic Web service to Release 10.3.1; you can redeploy a 10.*x* Web service to WebLogic Server Release 10.3.1 without making any changes or recompiling it.

2

Use Cases and Examples

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

- Section 2.1, "Creating a Simple HelloWorld Web Service"
- Section 2.2, "Creating a Web Service With User-Defined Data Types"
- Section 2.3, "Creating a Web Service from a WSDL File"
- Section 2.4, "Invoking a Web Service from a Stand-alone Java Client"
- Section 2.5, "Invoking a Web Service from a WebLogic Web Service"

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.

2.1 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 for Oracle WebLogic Server*.

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 *`MW_HOME`*`/user_projects/domains/`*`domainName`*, where *`MW_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 Section 2.1.1, "Sample HelloWorldImpl.java JWS File."

   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 Section 2.1.2, "Sample Ant Build File for HelloWorldImpl.java" 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. You specify the type of Web service (JAX-WS) that you want to create using `type="JAXWS"`.

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, Section 2.2, "Creating a Web Service With User-Defined Data Types." 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 Section 2.4, "Invoking a Web Service from a Stand-alone Java Client" for an example of creating a Java client application that invokes a Web service.

2.1.1 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 + "'";
  }
}
```

2.1.2 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>
```

2.2 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 Section 2.1, "Creating a Simple HelloWorld Web Service." 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 *`MW_HOME`*`/user_projects/domains/`*`domainName`*, where *`MW_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 Section 2.2.1, "Sample BasicStruct JavaBean."

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 Section 2.2.2, "Sample ComplexImpl.java JWS File."

   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 Section 2.2.3, "Sample Ant Build File for ComplexImpl.java JWS File" 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 for Oracle WebLogic Server*.

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 Section 2.4, "Invoking a Web Service from a Stand-alone Java Client" for an example of creating a Java client application that invokes a Web service.

2.2.1 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;
  }
}
```

2.2.2 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;
  }
}
```

2.2.3 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>
```

2.3 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 Section 2.3.1, "Sample WSDL File." 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 *MW_HOME*/user_projects/domains/*domainName*, where *MW_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 Section 2.3.1, "Sample WSDL File" 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 Section 2.3.3, "Sample Ant Build File for TemperatureService" 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"
      type="JAXWS"/>
</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/TemperaturePortTypeImp`

`1.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 Section 2.3.2, "Sample TemperaturePortType Java Implementation File" 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 Oracle WebLogic Server* 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 Section 2.4, "Invoking a Web Service from a Stand-alone Java Client" for an example of creating a Java client application that invokes a Web service.

2.3.1 Sample WSDL File

```
<?xml version="1.0"?>
<definitions
   name="TemperatureService"
   targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
   xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/" >
        <message name="getTempRequest">
```

```
                <part name="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="urn:xmethods-Temperature" />
                        </input>
                        <output>
                                <soap:body use="literal"
                                 namespace="urn:xmethods-Temperature" />
                        </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>
```

2.3.2 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.xmethods.net/sd/TemperatureService.wsdl"
  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;
```

```
  }
}
```

2.3.3 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>
```

2.4 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 Section 2.2, "Creating a Web Service With User-Defined Data Types." 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 *`MW_HOME`*`/user_projects/domains/`*`domainName`*, where *`MW_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 Section 2.4.2, "Sample Ant Build File For Building Stand-alone Client Application" 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 for Oracle WebLogic Server*.

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 Section 2.4.1, "Sample Java Client Application."

   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.

2.4.1 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());
  }
}
```

2.4.2 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>
```

2.5 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 Section 2.4, "Invoking a Web Service from a Stand-alone Java Client" 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 Section 2.2, "Creating a Web Service With User-Defined Data Types."

> **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 *`MW_HOME`*`/user_projects/domains/`*`domainName`*, where *`MW_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 Section 2.5.1, "Sample ClientServiceImpl.java JWS File."

   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 Section 2.5.2, "Sample Ant Build File For Building ClientService" 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 Section 2.4, "Invoking a Web Service from a Stand-alone Java Client" for an example of creating a Java client application that invokes a Web service.

2.5.1 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() + "'";
  }
}
```

2.5.2 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>
```

3 Developing WebLogic Web Services

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

3.1 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 at `http://www.jcp.org/en/jsr/detail?id=175`) and include standard annotations defined by *Web Services Metadata for the Java Platform* specification (JSR-181), described at `http://www.jcp.org/en/jsr/detail?id=181`, the JAX-WS specification (JSR-224), described at `https://jax-ws.dev.java.net`, as well as additional ones. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *Introducing WebLogic Web Services for Oracle WebLogic Server*. For additional detailed information about this programming model, see "Anatomy of a WebLogic Web Service" in *Introducing WebLogic Web Services for Oracle WebLogic Server*.

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:

- Section 3.3, "Developing WebLogic Web Services Starting From Java: Main Steps"
- Section 3.4, "Developing WebLogic Web Services Starting From a WSDL File: Main Steps"

Iterative development refers to setting up your development environment in such a way so that you can repeatedly code, compile, package, deploy, and test a Web service until it works as you want. The WebLogic Web service programming model uses Ant tasks to perform most of the steps of the iterative development 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.

In addition to the command-line tools described in this section, you can use an IDE, such as Oracle JDeveloper or Oracle Enterprise Pack for Eclipse (OEPE), to develop Web services. For more information, see "Using Oracle IDEs to Build Web Services" in *Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server*.

3.2 Configuring Your Domain For Advanced Web Services Features

When creating or extending a domain, you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservices_jaxws.jar`) to configure automatically the resources required to support the following advanced Web service features:

- Web services atomic transactions
- Security using WS-SecureConversation

Although use of this extension template is not required, it makes the configuration of the required resources much easier. Alternatively, you can manually configure the resources required for these advanced features using the Oracle WebLogic Administration Console or WLST.

The following procedures describe how to configure a domain automatically for the advanced Web services features. For detailed instructions about using the Configuration Wizard to create and update WebLogic Server domains, see *Creating Domains Using the Configuration Wizard*.

- Section 3.2.1, "Resources Required by Advanced Web Service Features"
- Section 3.2.2, "Configuring a Domain for Advanced Web Service Features Using the Configuration Wizard"
- Section 3.2.3, "Using WLST to Extend a Domain With the Web Services Extension Template"
- Section 3.2.4, "Updating Resources Added After Extending Your Domain"

3.2.1 Resources Required by Advanced Web Service Features

Table 3–1 lists the resources that are defined automatically when using the WebLogic Advanced Web Services for JAX-WS Extension template. If you do not apply the extension template, you need to configure the resources manually using the Oracle WebLogic Administration Console or WLST.

The following variables are used in the table:

- *server_designator* specifies an ID that is generated automatically by the configuration framework. Typically, this ID is of the format `auto_number`.
- *uniqueID* specifies unique numeric ID that is generated automatically by the configuration framework. Typically, this ID is a numeric value, such as `1234`.
- *server_name* specifies the user-specified name of the server.

Note: At runtime, you should not change the name of resources; otherwise, you may experience runtime errors or data loss.

Several resources are reserved for future use, as indicated in the table.

Table 3–1 Resources Required by Advanced Web Services Features

Resource Name	Resource Type	Description
`WseeJaxwsJmsModule`	JMS Module	Defines a JMS module that defines the JMS resources needed for advanced Web services. All associated targets (JMS servers targeted to a server) on this JMS module will be used to support JAX-WS Web services. All servers to which this module is targeted must have the proper Web services resources configured. **Note**: By default, the resource is configured as a weighted distributed destination (WDD). It is strongly recommended that you update the configuration to use a uniform distributed destination (UDD). To configure distributed destinations manually and for more information, see "Using Distributed Destination" in *Programming JMS for Oracle WebLogic Server*.
`WseeJaxwsFileStore_`*`server_designator`*	File store	Specifies the file store. A separate file store is configured on each Managed Server targeted by the `WseeJaxwsJmsModule`, as specified by *`server_designator`*. In a single server domain, the file store is named `WseeJaxwsFileStore`. To configure the file stores manually, see "Using the WebLogic Persistent Store" in *Configuring Server Environments for Oracle WebLogic Server*.
`WseeJaxwsJmsServer_`*`server_designator`*	JMS server	Specifies the JMS server management container. A separate JMS Server is configured on each Managed Server targeted by `WseeJaxwsJmsModule`, as specified by *`server_designator`*. The JMS server uses `WseeFileStore_`*`server_designator`* as the file store. To configure the JMS server manually, see "JMS Configuration" in *Configuring and Managing JMS for Oracle WebLogic Server*.
`WseeJaxwsJmsServer`*`uniqueID`*	JMS subdeployment	Specifies the JMS subdeployment targeting the JMS servers defined on all Managed Servers in the cluster. To configure the JMS subdeployment manually, see "Configure subdeployments in JMS system modules" in *Oracle WebLogic Server Administration Console Help*.

Table 3–1 (Cont.) Resources Required by Advanced Web Services Features

Resource Name	Resource Type	Description
`weblogic.wsee.jaxws.mdb.DispatchPolicy`	Work Manager	Enables an application to execute multiple work items concurrently within a container. One Work Manager is generated for the domain and targeted to all servers to which the `WseeJaxwsJmsModule` is targeted. **Note**: You should not change the name of the Work Manager resource. To configure Work Managers manually, see "Description of the Work Manager API" in *Timer and Work Manager API (CommonJ) Programmer's Guide for Oracle WebLogic Server*.
`ReliableWseeJaxwsSAFAgent_`*`server_name`*	Store-and-forward (SAF) service agent	Provides highly available JMS message production. A separate SAF agent is configured on each Managed Server, as specified by *`server_name`*. The SAF agent uses `WseeFileStore_`*`server_name`* as the file store. In a single server domain, the SAF agent is named `ReliableWseeJaxwsSAFAgent`. To configure SAF service agents, see "Understanding the Store-and-Forward Service" in *Configuring and Managing Store-and-Forward for Oracle WebLogic Server*.
`dist_WseeBufferedRequestQueue_`*`server_designator`*	JMS queue	**Note**: Reserved for future use. Specifies the queue used for buffered requests. A separate queue is configured on each Managed Server, as specified by *`server_name`*. To configure the queues manually, see "Configure queues" in *Oracle WebLogic Server Administration Console Help*.
`dist_WseeBufferedRequestErrorQueue_`*`server_designator`*	JMS queue	**Note**: Reserved for future use. Specifies the error queue used for `WseeBufferedRequestQueue` for buffered requests that cannot be processed within the maximum number of retries. A separate queue is configured on each Managed Server, as specified by *`server_name`*. To configure the queues manually, see "Configure queues" in *Oracle WebLogic Server Administration Console Help*.

Table 3–1 (Cont.) Resources Required by Advanced Web Services Features

Resource Name	Resource Type	Description
`dist_WseeBufferedResponseQueue_server_designator`	JMS queue	**Note**: Reserved for future use. Specifies the queue used for buffered responses. A separate queue is configured on each Managed Server, as specified by *`server_name`*. To configure the queues manually, see "Configure queues" in *Oracle WebLogic Server Administration Console Help*.
`dist_WseeBufferedResponseErrorQueue_server_designator`	JMS queue	**Note**: Reserved for future use. Specifies the error queue used for `WseeBufferedResponseQueue` for buffered responses that cannot be delivered within the maximum number of retries. A separate queue is configured on each Managed Server, as specified by *`server_name`*. To configure the queues manually, see "Configure queues" in *Oracle WebLogic Server Administration Console Help*.
`WseeStore`	Logical store	**Note**: Reserved for future use. Defines the logical store. A separate logical store is configured on each Managed Server targeted by `WseeJaxwsJmsModule`. The logical store points to the `WseeBufferedRequestQueue` queue for its configuration and file store.

3.2.2 Configuring a Domain for Advanced Web Service Features Using the Configuration Wizard

The following sections describe how to configure a domain for advanced Web service features.

- Section 3.2.2.1, "Creating a Domain With the Web Services Extension Template"
- Section 3.2.2.2, "Extending a Domain With the Web Services Extension Template"

3.2.2.1 Creating a Domain With the Web Services Extension Template

To create a domain that is automatically configured for the advanced Web service features:

1. Start the Configuration Wizard.
2. In the Welcome window, select **Create a new WebLogic domain**.
3. Click **Next**.
4. Select **Generate a domain configured automatically to support the following products** and select **WebLogic Advanced Web Services for JAX-WS Extension**.
5. Click **Next**.
6. Enter the name and location of the domain and click **Next**.
7. Configure the administrator user name and password and click **Next**.
8. Configure the server start mode and JDK and click **Next**.
9. To configure additional servers and clusters:

a. On the Select Optional Configuration screen, at a minimum select **JMS Distributed Destination** and **Managed Servers, Clusters, and Machines** to update the distributed destination to a UDD (recommended for clustered domains) and define the Managed Servers and clusters. Select any other items, as desired, and click **Next**.

b. Select **UDD** for the `WseeJaxwsJmsModule` and click **Next**.

c. Configure the Managed Servers in your environment and click **Next**.

d. Configure the clusters in your environment and click **Next**.

e. Assign the managed servers to the clusters on the Assign to Clusters screen and click **Next**.

f. Configure the machines in your environment and click **Next**.

g. Target the services defined in the environment to clusters or servers on the Target Services to Clusters or Servers screen and click **Next**.

   Note: Target the `WseeJaxwsJmsModule` JMS module and `weblogic.wsee.jaxws.mdb.DispatchPolicy` Work Manager to the same servers in the cluster.

   Servers targeted on this screen will be fully configured for use with advanced Web services.

h. Configure additional information on additional configuration screens (if selected in step 9a) and click **Next**.

10. When you reach the Configuration Summary screen, verify the domain details and click **Create**.

3.2.2.2 Extending a Domain With the Web Services Extension Template

To extend an existing domain so that it is automatically configured for these Web Services features:

1. Start the Configuration Wizard.
2. In the Welcome window, select **Extend an Existing WebLogic Domain**.
3. Click **Next**.
4. Select the domain to which you want to apply the extension template.
5. Click **Next**.
6. Select **Extend my domain automatically to support the following added products** and select **WebLogic Advanced Web Services for JAX-WS Extension**.
7. Click **Next**.
8. To configure additional servers and clusters:

   a. On the Select Optional Configuration screen, at a minimum select **JMS Distributed Destination** and **Managed Servers, Clusters, and Machines** to update the distributed destination to a UDD (recommended for clustered domains) and define the Managed Servers and clusters. Select any other items, as desired, and click **Next**.

   b. Select **UDD** for the `WseeJaxwsJmsModule` and click **Next**.

   c. Configure the Managed Servers in your environment and click **Next**.

   d. Configure the clusters in your environment and click **Next**.

e. Assign the managed servers to the clusters on the Assign to Clusters screen and click **Next**.

f. Configure the machines in your environment and click **Next**.

g. Target the services defined in the environment to clusters or servers on the Target Services to Clusters or Servers screen and click **Next**.

   Note: Target the `WseeJaxwsJmsModule` JMS module and `weblogic.wsee.jaxws.mdb.DispatchPolicy` Work Manager to the same servers in the cluster.

   Servers targeted on this screen will be fully configured for use with advanced Web services.

h. Configure additional information on additional configuration screens (if selected in step 9a) and click **Next**.

9. Verify that you are extending the correct domain, then click **Extend**.
10. Click **Done** to exit.

3.2.3 Using WLST to Extend a Domain With the Web Services Extension Template

The following provides an example of how to use WLST to extend a domain using the Web services extension template. Specifically, this example demonstrates how to extend a single server domain. It is assumed that you have already created a single server domain. You can add additional servers and clusters to the domain in the location noted in the example script below.

After updating the script and executing it against your domain, all resources will be configured for advanced Web service features.

Review the comments provided in the sample for more information. For more information about the WLST commands described, see the *Oracle WebLogic Scripting Tool*.

Note: The `wls_webservice_fixup_utils.py` script used at the end of this example is added to the domain directory when you extend the domain using the Web services extension template.

Example 3–1 WLST Script to Extend a Domain With the Web Services Extension Template

```
# Read the domain.
readDomain(single_server_domain_dir)

# Apply the template to the domain to configure the servers for advanced Web service features.
installDir = install_directory/wlserver_10.3
templateLocation = installDir + '/common/templates/applications/wls_webservice_jaxws.jar'
addTemplate(templateLocation)

# Save and close the domain
updateDomain()
closeDomain()

# Read the domain
readDomain(domain_dir)

# Optionally create any servers and clusters required in your domain environment.
# <Include create calls here . . . >
# For example: create('server1','Server') or create('cluster1','Cluster')
```

```
# Optionally configure the JMS module as a Uniform Distributed Destination (Recommended)
setDistDestType('WseeJaxwsJmsModule', 'UDD')

# Target WseeJaxwsJmsModule to the desired servers and clusters.
assign('JMSSystemResource', 'WseeJaxwsJmsModule', 'Target', server_or_cluster)
# Repeat assign call for other servers and clusters in the environment.

# Unassign the resource from the Administration Server.
unassign('JMSSystemResource', 'WseeJaxwsJmsModule', 'Target', Administration_Server)

# Target JMSServer, SAFAgent, FileStore to migratable targets
sys.path.append(domain_dir)

# Import the wls_webservice_fixup_utils.py script. This script is added to the domain directory
# when you extend the domain using the Web services extension template.
import wls_webservice_fixup_utils as fixup
fixup.doWseeFixup(globals())

# Save and close the domain
updateDomain()
closeDomain()
```

3.2.4 Updating Resources Added After Extending Your Domain

Once you have created or extended a domain using the Weblogic Advanced Web Services for JAX-WS Extension template, if you then modify the resources in your domain, you can update the configuration of those resources quickly and easily using the following WLST script.

After updating the script and executing it against your domain, all resources will be configured for advanced Web service features.

> **Note:** The `wls_webservice_fixup_utils.py` script used at the end of this example is added to the domain directory when you extend the domain using the Web services extension template.

Review the comments provided in the sample for more information. For more information about the WLST commands described, see the *Oracle WebLogic Scripting Tool*.

Example 3–2 WLST Script for Updating Resources Added After Extending Your Domain

```
# Read the domain.
readDomain(domain_dir)

# Optionally configure the JMS module as a Uniform Distributed Destination (Recommended)
setDistDestType('WseeJaxwsJmsModule', 'UDD')

# Target WseeJaxwsJmsModule to the desired servers and clusters.
assign('JMSSystemResource', 'WseeJaxwsJmsModule', 'Target', server_or_cluster_name)
# Repeat assign call for other servers and clusters in the environment.

# Unassign the resource from the Administration Server.
unassign('JMSSystemResource', 'WseeJaxwsJmsModule', 'Target', Administration_Server_name)

# Target JMSServer, SAFAgent, FileStore to migratable targets.
```

```
sys.path.append(domain_dir)

# Import the wls_webservice_fixup_utils.py script. This script is added to the domain directory
# when you extend the domain using the Web services extension template.
import wls_webservice_fixup_utils as fixup
fixup.doWseeFixup(globals())

# Save and close the domain.
updateDomain()
```

3.3 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 Chapter 2, "Use Cases and Examples" 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 Section 3.13, "Integrating Web Services Into the WebLogic Split Development Directory Environment" 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 *`MW_HOME`*`/user_projects/domains/`*`domainName`*, where *`MW_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 Chapter 4, "Programming the JWS File."
4	Create user-defined data types. (Optional)	If your Web service uses user-defined data types, create the JavaBeans that describes them. See Section 4.6, "Programming the User-Defined Java Data Type."
5	Create a basic Ant build file, `build.xml`.	See Section 3.5, "Creating the Basic Ant build.xml File."
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 Section 3.6, "Running the jwsc WebLogic Web Services Ant Task."
7	Deploy the Web service to WebLogic Server.	See Section 3.9, "Deploying and Undeploying WebLogic Web Services."

Table 3–2 (Cont.) Steps to Develop Web Services Starting From Java

#	Step	Description
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 Section 3.10, "Browsing to the WSDL of the Web Service."
9	Test the Web service.	See Section 3.12, "Testing the Web Service."
10	Edit the Web service. (Optional)	To make changes to the Web service, update the JWS file, undeploy the Web service as described in Section 3.9, "Deploying and Undeploying WebLogic Web Services," then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See Chapter 6, "Invoking Web Services" for information on writing client applications that invoke a Web service.

3.4 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 Chapter 2, "Use Cases and Examples," 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 Section 3.13, "Integrating Web Services Into the WebLogic Split Development Directory Environment" for details.

Table 3–3 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 *`MW_HOME`*`/user_projects/domains/`*`domainName`*, where *`MW_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 Section 3.5, "Creating the Basic Ant build.xml File."
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 Section 3.7, "Running the wsdlc WebLogic Web Services Ant Task."

Table 3–3 (Cont.) Steps to Develop Web Services Starting From Java

#	Step	Description
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 Section 3.8, "Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc."
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 Section 3.6, "Running the jwsc WebLogic Web Services Ant Task."
8	Deploy the Web service to WebLogic Server.	See Section 3.9, "Deploying and Undeploying WebLogic Web Services."
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 Section 3.10, "Browsing to the WSDL of the Web Service."
10	Test the Web service.	See Section 3.12, "Testing the Web Service."
11	Edit the Web service. (Optional)	To make changes to the Web service, update the JWS file, undeploy the Web service as described in Section 3.9, "Deploying and Undeploying WebLogic Web Services," then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See Chapter 6, "Invoking Web Services" for information on writing client applications that invoke a Web service.

3.5 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>
```

3.6 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: 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 Section 3.4, "Developing WebLogic Web Services Starting From a WSDL File: Main Steps."

- *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 Oracle WebLogic Server* for more information.

3.6.1 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
```

3.6.2 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 Oracle WebLogic Server* for complete documentation and examples about the `jwsc` Ant task.

3.7 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_PortTypeImpl.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 Oracle WebLogic Server* for more information.

3.8 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_PortTypeImpl.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`, `@SOAPMessageHandlers`, `@Policies`, and `@Policy`. 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`, `endpointInterface`, and `targetNamespace` 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. Use the `targetNamespace` attribute to specify the namespace of a WSDL service, which can be different from the on in JWS SEI.

- You can specify JAX-WS—JSR 224, JAXB (JSR 222)—or Common (JSR 250) annotations, as required. For more information about the annotations that are supported, see "JWS Annotation Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

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 Section 2.3.1, "Sample WSDL File"; 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;
  }
}
```

3.9 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 Oracle WebLogic Server*.

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

- Section 3.9.1, "Using the wldeploy Ant Task to Deploy Web Services"
- Section 3.9.2, "Using the Administration Console to Deploy Web Services"

3.9.1 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.

3.9.2 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 *Oracle WebLogic Server Administration Console Help*.

3.10 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 Oracle WebLogic Server* 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
```

3.11 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 Section 3.10, "Browsing to the WSDL of the Web Service." 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.

- Section 3.11.1, "Web service is not a callback service and can be invoked using HTTP/S"
- Section 3.11.2, "Web service is a callback service"
- Section 3.11.3, "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.

3.11.1 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 *Oracle WebLogic Server Administration Console 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 *Oracle WebLogic Server Administration Console 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.

3.11.2 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 *Oracle WebLogic Server Administration Console 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 *Oracle WebLogic Server Administration Console 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 *Oracle WebLogic Server Administration Console 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 *Oracle WebLogic Server Administration Console Help*.

3.11.3 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 *Oracle WebLogic Server Administration Console Help*.

3.12 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 *Oracle WebLogic Server Administration Console Help*.

3.13 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 Environment" in *Developing Applications for Oracle 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 Section 3.6, "Running the jwsc WebLogic Web Services Ant Task."

   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 you JWS file defines an EJB via the `@Stateless` annotation. 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 Section 3.10, "Browsing to the WSDL of the Web Service."

4 Programming the JWS File

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

4.1 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), described at http://www.jcp.org/en/jsr/detail?id=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 for Oracle WebLogic Server*.

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.

4.2 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) at http://www.jcp.org/en/jsr/detail?id=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`, `@WebService.serviceName`, and `@WebService.targetNamespace`.
- 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.

4.3 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 for Oracle WebLogic Server*.

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`, `javax.jws.soap`, or `javax.xml.ws` package. For example:
		`import javax.jws.WebMethod;`
		`import javax.jws.WebService;`
		`import javax.jws.soap.SOAPBinding;`
		`import javax.xml.ws.BindingType;`

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 for Oracle WebLogic Server*.
3	Add the standard required `@WebService` JWS annotation at the class level to specify that the Java class exposes a Web service.	See Section 4.3.2, "Specifying that the JWS File Implements a Web Service (@WebService Annotation)."
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 Section 4.3.3, "Specifying the Mapping of the Web Service to the SOAP Message Protocol (@SOAPBinding Annotation)." 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 Section 4.3.7, "Specifying the Binding to Use for an Endpoint (@BindingType Annotation)."
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 Section 4.3.4, "Specifying That a JWS Method Be Exposed as a Public Operation (@WebMethod and @OneWay Annotations)."
7	Add `@WebParam` annotation to customize the name of the input parameters of the exposed operations. (Optional)	See Section 4.3.5, "Customizing the Mapping Between Operation Parameters and WSDL Elements (@WebParam Annotation)."
8	Add `@WebResult` annotations to customize the name and behavior of the return value of the exposed operations. (Optional)	See Section 4.3.6, "Customizing the Mapping Between the Operation Return Value and a WSDL Element (@WebResult Annotation)."
9	Add your business code.	Add your business code to the methods to make the WebService behave as required.

4.3.1 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 + "'";
  }
}
```

4.3.2 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) at `http://www.jcp.org/en/jsr/detail?id=181` for the default values of each attribute.

4.3.3 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. In general, document-literal-wrapped Web services are the most interoperable type of Web service.

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`

4.3.4 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.

To exclude a method as a Web service operation, specify @WebMethod(exclude="true").

> **Note:** For JAX-WS, the service endpoint interface (SEI) defines the public methods. If no SEI exists, then *all* public methods are exposed as Web service operations, unless they are tagged explicitly with @WebMethod(exclude="true").

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) at `http://www.jcp.org/en/jsr/detail?id=181` for the default values of each attribute, as well as additional information about the `@WebMethod` and `@Oneway` annotations.

4.3.5 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) at `http://www.jcp.org/en/jsr/detail?id=181` for the default value of each attribute.

4.3.6 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) at `http://www.jcp.org/en/jsr/detail?id=181` for the default value of each attribute.

4.3.7 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 at https://jax-ws.dev.java.net/nonav/2.1.4/docs/annotations.html.

4.4 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` (http://java.sun.com/javase/6/docs/api/javax/xml/ws/BindingProvider.html)—From the client application, access the request and response context of the protocol binding. See Section 4.4.1, "Accessing the Protocol Binding Context."
- `javax.xml.ws.WebServiceContext` (http://java.sun.com/javase/6/docs/api/javax/xml/ws/WebServiceContext.html)—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 Section 4.4.2, "Accessing the Web Service Context."
- `javax.xml.ws.handler.MessageContext` (http://java.sun.com/javase/6/docs/api/javax/xml/ws/handler/MessageContext.html)—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 Section 4.4.3, "Using the MessageContext Property Values."

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

4.4.1 Accessing the Protocol Binding Context

> **Note:** The `com.sun.xml.ws.developer.JAXWSProperties` (https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/developer/JAXWSProperties.html) and `com.sun.xml.ws.client.BindingProviderProperties` (https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/client/BindingProviderProperties.html) 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, see http://java.sun.com/javase/6/docs/api/javax/xml/ws/BindingProvider.html. 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 Section 4.4.3, "Using the MessageContext Property Values."

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 at https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/developer/JAXWSProperties.html.
- 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 at https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/client/BindingProviderProperties.html.

4.4.2 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. For more information, see http://java.sun.com/javase/6/docs/api/javax/xml/ws/WebServiceContext.html.

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 Section 4.4.3, "Using the MessageContext Property Values."

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. For more information, see `http://java.sun.com/javase/6/docs/api/javax/xml/ws/WebServiceContext.html`.

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 Section 4.4.3, "Using the MessageContext Property Values."
`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`.

4.4.3 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 at http://java.sun.com/javase/6/docs/api/javax/xml/ws/handler/class-use/MessageContext.html.

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.
`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.

4.5 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 *Programming WebLogic Enterprise JavaBeans, Version 3.0 for Oracle WebLogic Server*.

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 at `http://java.sun.com/javaee/5/docs/api/javax/ejb/package-summary.html` 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://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;
}
```

4.6 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.

> **Note:** You can use JAXB to provide custom mapping. For more information, see "Customizing Java-to-XML Schema Mapping Using JAXB Annotations" on page 5-9.

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 Section 5.3.2, "Supported User-Defined Data Types." See Section 5.3.1, "Supported Built-In Data Types" 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 Section 2.2.2, "Sample ComplexImpl.java JWS File":

```
package examples.webservices.complex;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
// Import the WebLogic-specific JWS annotation interface
// Import 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;
```

```
  }
}
```

4.7 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 Section 2.4, "Invoking a Web Service from a Stand-alone Java Client," 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 Section 6.3, "Invoking a Web Service from Another Web Service" for detailed instructions.

4.8 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.BindingTyp` 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.

4.9 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.

4.9.1 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. For more information, see https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/developer/SchemaValidation.html.

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)
```

4.9.2 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. For more information, see https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/developer/SchemaValidationFeature.html.

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);
```

4.10 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.

5 Using JAXB Data Binding

The following sections provide information about using JAXB data binding:

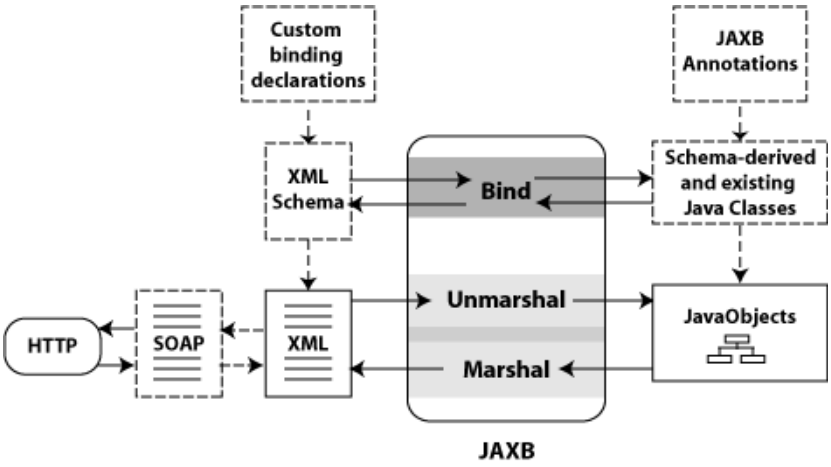
5.1 Overview of Data Binding Using JAXB

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), described at `http://jcp.org/en/jsr/detail?id=222`, 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:

- Section 5.2, "Developing the JAXB Data Binding Artifacts"—Describes how to develop the JAXB data binding artifacts using WebLogic Server.
- Section 5.3, "Standard Data Type Mapping"—Describes the standard built-in and user-defined data types that are supported.
- Section 5.4, "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.
- Section 5.5, "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.

5.2 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 Section 5.4, "Customizing Java-to-XML Schema Mapping Using JAXB Annotations." If no customizations are required, JAXB uses the standard built-in and user-defined data type mapping as described in the following sections: Section 5.3.1.2, "Java-to-XML Mapping for Built-In Data Types" and Section 5.3.2.2, "Supported Java User-Defined Data Types."

  For more information about this programming model, see Section 3.3, "Developing WebLogic Web Services Starting From Java: Main Steps."

- **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 Section 5.5, "Customizing XML Schema-to-Java Mapping Using Binding Declarations." If no customizations are required, the standard built-in and user-defined data type mapping as described in the following sections: Section 5.3.1.1, "XML-to-Java Mapping for Built-in Data Types" and Section 5.3.2.1, "Supported XML User-Defined Data Types."

  For more information about this programming model, see Section 3.4, "Developing WebLogic Web Services Starting From a WSDL File: Main Steps."

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 Section 5.5, "Customizing XML Schema-to-Java Mapping Using Binding Declarations." If no customizations are required, JAXB uses the standard built-in and user-defined data type mappings described in Section 5.3, "Standard Data Type Mapping."

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

5.3 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 at `http://jcp.org/en/jsr/detail?id=222`, 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:

- Section 5.3.1, "Supported Built-In Data Types"
- Section 5.3.2, "Supported User-Defined Data Types"

5.3.1 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 Section 5.3.2, "Supported User-Defined Data Types" for a list of supported XML and Java data types.

5.3.1.1 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 Section 5.3.1.2, "Java-to-XML Mapping for Built-In Data Types."

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
double	double
duration	javax.xml.datatype.Duration
float	float
g	java.xml.datatype.XMLGregorianCalendar
hexBinary	byte[]

Table 5–1 (Cont.) 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)
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.

5.3.1.1.1 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>
```

5.3.1.1.2 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){...}
}
```

5.3.1.2 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 Section 5.3.2.2, "Supported Java User-Defined Data Types."

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
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

Table 5–2 (Cont.) Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	XML Schema Data Type
`javax.xml.namespace.QName`	`Qname`
`javax.xml.transform.Source`	`base64Binary`
`short`	`short`

5.3.2 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 Section 5.3.1, "Supported Built-In Data Types," then you must create the user-defined data type artifacts manually.

5.3.2.1 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>`	`java.lang.Object`
`<xsd:any[]>`	`java.lang.Object`
`<xsd:union>`	Common parent type of union members.

Table 5–3 (Cont.) Supported User-defined XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
<xsi:nil> and <xsd:nillable> attribute	Java null value. If the XML data type is built-in and usually maps to a Java primitive data type (such as int or short), then the XML data type is actually mapped to the equivalent object wrapper type (such as java.lang.Integer or java.lang.Short).
Derivation of complex types	Mapped using Java inheritance.
Abstract types	Abstract Java data type.

5.3.2.2 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

5.4 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, described at http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/package-tree.html.

Table 5–5 JAXB Mapping Annotations

Annotation	Description
`@XmlAccessorType`	Specifies whether fields or properties are mapped by default. See Section 5.4.2, "Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)."
`@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 Section 5.4.3, "Mapping Properties to Local Elements (@XmlElement)."
`@XMLMimeType`	Associates the MIME type that controls the XML representation of the property with a textual representation, such as `image/jpeg`. See Section 5.4.4, "Specifying the MIME Type (@XmlMimeType Annotation)."
`@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 Section 5.4.5, "Mapping a Top-level Class to a Global Element (@XmlRootElement)."
`@XmlSeeAlso`	Binds other classes when binding the current class. See Section 5.4.6, "Binding a Set of Classes (@XmlSeeAlso)."
`@XmlType`	Maps a class or enum type to an XML Schema type.See Section 5.4.7, "Mapping a Value Class to a Schema Type (@XmlType)."

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

- Section 5.3.1.2, "Java-to-XML Mapping for Built-In Data Types"
- Section 5.3.2.2, "Supported Java User-Defined Data Types"

5.4.1 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;
    }
}
```

5.4.2 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` (see Section 5.4.7, "Mapping a Value Class to a Schema Type (@XmlType)") and `@XmlRootElement` (see Section 5.4.5, "Mapping a Top-level Class to a Global Element (@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 at `http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlAccessorType.html`. An example is provided in Section 5.4.1, "Example of JAXB Annotations."

5.4.3 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 at `http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlElement.html`.

5.4.4 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 at `http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlMimeType.html`.

5.4.5 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` (see Section 5.4.7, "Mapping a Value Class to a Schema Type (@XmlType)") and `@XmlAccessorType` (see Section 5.4.2, "Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)") 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.

Table 5–9 (Cont.) Optional Elements for @XmlRootElement Annotation

Element	Description
`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 at `http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlRootElement.html`. An example is provided in Section 5.4.1, "Example of JAXB Annotations."

5.4.6 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.

5.4.7 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` (see Section 5.4.5, "Mapping a Top-level Class to a Global Element (@XmlRootElement)") and `@XmlAccessorType` (see Section 5.4.2, "Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)") 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 at `http://java.sun.com/javase/6/docs/api/javax/xml/bind/annotation/XmlType.html`. An example is provided in Section 5.4.1, "Example of JAXB Annotations."

5.5 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 Section 5.5.1, "Creating an External Binding Declarations File."

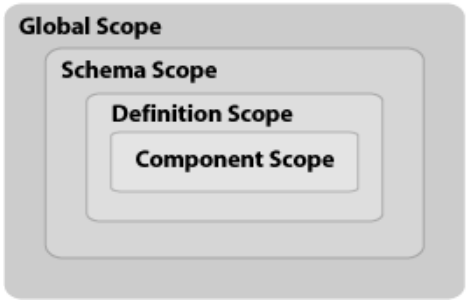
  Note: If customizations are required, Oracle recommends this method to maintain flexibility by keeping the customizations separate from the WSDL or XML Schema document.

- Embed binding declarations within the WSDL or XML Schema document. See Section 5.5.2, "Embedding Binding Declarations."

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 Scope 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 Section 5.5.1.1.1, "Specifying the Root Element." ■ 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.
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:

- Section 5.5.1, "Creating an External Binding Declarations File"
- Section 5.5.2, "Embedding Binding Declarations"
- Section 5.5.3, "JAX-WS Custom Binding Declarations"
- Section 5.5.4, "JAXB Custom Binding Declarations"

For more information about using custom binding declarations, see:

- *JAX-WS WSDL Customizations* at `https://jax-ws.dev.java.net/nonav/2.1.2m1/docs/customizations.html`
- "Customizing XML Schema to Java Representation Binding" in the JAXB specification at `http://jcp.org/en/jsr/detail?id=222`.

5.5.1 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:

- Section 5.5.1.1, "Creating an External Binding Declarations File Using JAX-WS Binding Declarations"
- Section 5.5.1.2, "Creating an External Binding Declarations File Using JAXB Binding Declarations"

5.5.1.1 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 Section 5.5.3, "JAX-WS Custom Binding Declarations."

5.5.1.1.1 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 5–5, 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>
```

5.5.1.1.2 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* at https://jax-ws.dev.java.net/nonav/2.1.2m1/docs/customizations.html. For

information about the custom JAXB binding declarations that you can define, see Section 5.5.4, "JAXB Custom Binding Declarations."

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>
```

5.5.1.2 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 Section 5.5.4, "JAXB Custom Binding Declarations."

5.5.1.2.1 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.

5.5.1.2.2 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>
```

5.5.2 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 Section 5.5.2.1, "Embedding JAX-WS or JAXB Binding Declarations in the WSDL File."
- Embed a JAXB binding declaration in the XML Schema as part of an `<appinfo>` element. See Section 5.5.2.2, "Embedding JAXB Binding Declarations in the XML Schema."

5.5.2.1 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 Section 5.5.3, "JAX-WS Custom Binding Declarations."

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*, which is available at https://jax-ws.dev.java.net/nonav/2.1.2m1/docs/customizations.html. For information about the custom JAXB binding declarations that you can define, see Section 5.5.4, "JAXB Custom Binding Declarations."

5.5.2.2 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>
```

5.5.3 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* at https://jax-ws.dev.java.net/nonav/2.1.2/docs/customizations.html .

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 "wsdlc" in the *WebLogic Web Services Reference for Oracle WebLogic Server*.
	
	This binding declaration can be specified as part of the root binding element, as described in Section 5.5.1, "Creating an External Binding Declarations File," 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>`

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 Section 5.5.1, "Creating an External Binding Declarations File," 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/SimpleServic`
	`e?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 (Cont.) JAX-WS Custom Binding Declarations

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 Section 5.5.1, "Creating an External Binding Declarations File," 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 asynchronous polling and callback operations:
	
	`<bindings`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"`
	`wsdlLocation="http://localhost:7001/simple/SimpleServic`
	`e?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.

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='SimpleServic`
	`e']">`
	`    <class name="myService"></class>`
	`</bindings>`

Table 5–13 (Cont.) JAX-WS Custom Binding Declarations

Customization	Description
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='SimpleServiceImpl']/wsdl:operation[@name='EchoHello']"> <method name="Greeting"></method> </bindings>`
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='SimpleServiceImpl']/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='SimpleServiceImpl']/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) at http://www.jcp.org/en/jsr/detail?id=181.

5.5.4 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 (http://jcp.org/en/jsr/detail?id=222) 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 setter 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 the description of "Package name" in this table.

Table 5–14 (Cont.) JAXB Custom Binding Declarations

Customization	Description

Package name	Use the `<package>` element of the `<schemaBindings>` binding declaration (see Table 5–12) 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 "wsdlc" in the *WebLogic Web Services Reference for Oracle WebLogic Server*.
	
	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>`

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: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 shows how to define the Java property name:
	
	`<jaxb:bindindgs`
	`xmlns:xs="http://www.w3.org/2001/XMLSchema"`
	`node="//xs:schema/">`
	`<jaxb:schemaBindings>`
	`<jaxb:property generateIsSetMethod="true"/>`
	`</jaxb:schemaBindings>`
	`</jaxb:bindings>`

Table 5–14 (Cont.) JAXB Custom Binding Declarations

Customization	Description
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 (above).

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>`

6 Invoking Web Services

The following sections describe how to invoke WebLogic Web services:

- Section 6.1, "Overview of Web Services Invocation"
- Section 6.2, "Invoking a Web Service from a Stand-alone Client: Main Steps"
- Section 6.3, "Invoking a Web Service from Another Web Service"
- Section 6.4, "Using a Proxy Server When Invoking a Web Service"
- Section 6.5, "Client Considerations When Redeploying a Web Service"

> **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 for Oracle WebLogic Server*.

6.1 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.

This chapter focuses on how to generate a static Java class of the `Service` interface implementation for the particular Web service you want to invoke. For information about generating dynamic proxy clients, see "Creating Dynamic Proxy Clients" in *Programming Advanced Features of JAX-WS Web Services for Oracle 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.

In addition to the command-line tools described in this section, you can use an IDE such as Oracle JDeveloper or Oracle Enterprise Pack for Eclipse (OEPE) for Web service proxy generation and testing. For more information, see "Using Oracle IDEs to Build Web Services" in *Introducing WebLogic Web Services for Oracle WebLogic Server*.

6.2 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.

> **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 Section 3.5, "Creating the Basic Ant build.xml File." For a full example of a build.xml file used in this section, see Section 6.2.5, "Sample Ant Build File for a Stand-Alone Java Client."

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 *MW_HOME*/user_projects/domains/*domainName*, where *MW_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 Section 6.2.1, "Using the clientgen Ant Task To Generate Client Artifacts."
3	Get information about the Web service, such as the signature of its operations and the name of the ports.	See Section 6.2.2, "Getting Information About a Web Service."
4	Write the client application Java code that includes code for invoking the Web service operation.	See Section 6.2.3, "Writing the Java Client Application Code to Invoke a Web Service."
5	Create a basic Ant build file, build.xml.	See Section 3.5, "Creating the Basic Ant build.xml File."
6	Compile and run your Java client application.	See Section 6.2.4, "Compiling and Running the Client Application."

6.2.1 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 for Oracle WebLogic Server*.

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 for Oracle WebLogic Server*.

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 Section 6.2.5, "Sample Ant Build File for a Stand-Alone Java Client."

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.

6.2.2 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 `getPortName()` methods for getting the Web service port, where *`ServiceName`* refers to the name of the Web service and *`PortName`* refers to the name of the port. If the Web service was implemented with a JWS file, the name of the Web service is the value of the `serviceName` attribute of the `@WebService` JWS annotation and the name of the port is the value of the `portName` attribute of the `<WLHttpTransport>` 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 Section 3.10, "Browsing to the WSDL of the Web Service" 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>
```

6.2.3 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`.

6.2.4 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 Section 6.2.5, "Sample Ant Build File for a Stand-Alone Java Client" 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.

6.2.5 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 Section 6.2.1, "Using the clientgen Ant Task To Generate Client Artifacts" and Section 6.2.4, "Compiling and Running the Client Application" 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>
```

6.3 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 Section 6.2, "Invoking a Web Service from a Stand-alone Client: Main Steps," 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 Section 6.3.3, "Defining a Web Service Reference Using the @WebServiceRef Annotation."

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 Section 6.2, "Invoking a Web Service from a Stand-alone Client: Main Steps." 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 Section 6.3.1, "Sample build.xml File for a Web Service Client" 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 Section 6.3.2, "Sample JWS File That Invokes a Web Service" 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 Section 6.2.3, "Writing the Java Client Application Code to Invoke a Web Service" for details.

6.3.1 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>
```

6.3.2 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);
```

6.3.3 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);
```

6.4 Using a Proxy Server When Invoking a Web Service

You can use a proxy server to proxy requests from a client application to an application server (either WebLogic or non-WebLogic) that hosts the invoked Web service. You typically use a proxy server when the application server is behind a firewall. You can specify the proxy server in your client application using Java system properties. There are two ways to specify the proxy server in your client application: programmatically using the WebLogic `ClientProxyFeature` API or using system properties.

6.4.1 Using the ClientProxyFeature API to Specify the Proxy Server

You can programmatically specify within the Java client application itself the details of the proxy server that will proxy the Web service invoke using the `weblogic.wsee.jaxws.proxy.ClientProxyFeature` API. For more about the `ClientProxyFeature` API, see the *Oracle WebLogic Server API Reference*.

The proxy server settings defined by the `ClientProxyFeature` override the settings defined at the JVM-level, as described in Section 6.4.2, "Using System Properties to Specify the Proxy Server".

> **Note:** The `ClientProxyFeature` configures the port for WebLogic HTTP over SSL. It is recommended that you configure SSL for WebLogic Server. For more information, see "Configuring SSL" in *Securing Oracle WebLogic Server*.

You can configure the proxy server information using the `ClientProxyFeature` and pass the feature as an argument when creating the Web service port, as shown in the following example.

Example 6–1 Pass ClientProxyFeature as an Argument When Creating Port

```
package examples.webservices.simple_client;
import weblogic.wsee.jaxws.proxy
public class Main {
  public static void main(String[] args) {
    ComplexService test = new ComplexService();
    ClientProxyFeature cpf = new ClientProxyFeature();
    cpf.setProxyHost("localhost");
    cpf.setProxyPort(8888);
    cpf.setProxyUserName("proxyu");
    cpf.setProxyPassword("proxyp");
    ComplexPortType port = test.getComplexPortTypePort(cpf);
    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());
  }
}
```

Alternatively, you can configure the proxy server information after the port is created, as shown in the following example. In this case, you execute the `attachsPort()` method to attach the `ClientProxyFeature` to the existing port.

Example 6–2 Configuring the ClientProxyFeature After Creating the Port

```
package examples.webservices.simple_client;
import weblogic.wsee.jaxws.proxy
public class Main {
  public static void main(String[] args) {
    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();
    ClientProxyFeature cpf = new ClientProxyFeature();
    cpf.setProxyHost("localhost");
    cpf.setProxyPort(8888);
    cpf.setProxyUserName("proxyu");
    cpf.setProxyPassword("proxyp");
    cpf.attachsPort(port);
    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());
  }
}
```

If after configuring the `ClientProxyFeature` and attaching it to the port you want to disable the client proxy settings, you set the proxy port to a negative value. For example:

Example 6–3 Disabling Client Proxy Settings

```
. . .
    ClientProxyFeature cpf = new ClientProxyFeature();
    cpf.setProxyPort(-1);\
    cpf.attachsPort(port);
. . .
```

6.4.2 Using System Properties to Specify the Proxy Server

To use system properties to specify the proxy server, write your client application in the standard way, and then specify Java system properties when you execute the client application.

The following table summarizes the Java system properties.

Note: In this case, the proxySet system property must not be set. If the proxySet system property is set to (proxySet=false), proxy properties will be ignored and no proxy will be used.

Table 6–2 Java System Properties Used to Specify Proxy Server

Property	Description
http.proxyHost=*proxyHost* or https.proxyHost=*proxyHost*	Name of the host computer on which the proxy server is running. Use https.proxyHost for HTTP over SSL.
http.proxyPort=*proxyPort* or https.proxy.Port=*proxyPort*	Port to which the proxy server is listening. Use https.proxyPort for HTTP over SSL.
http.nonProxyHosts=*hostname* | *hostname* | ...	List of hosts that should be reached directly, bypassing the proxy. Separate each host name using a | character. This property applies to both HTTP and HTTPS.

The following excerpt from an Ant build script shows an example of setting Java system properties when invoking a client application called `clients.InvokeMyService`:

```
<target name="run-client">
   <java fork="true"
         classname="clients.InvokeMyService"
         failonerror="true">
     <classpath refid="client.class.path"/>
     <arg line="${http-endpoint}"/>
     <jvmarg line=
       "-Dhttp.proxyHost=${proxy-host}
       -Dhttp.proxyPort=${proxy-port}
       -Dhttp.nonProxyHosts=${mydomain}"
     />
   </java>
 </target>
```

6.5 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.

7 Administering Web Services

The following sections describe how to administer WebLogic Web services:

7.1 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.

- Viewing the SOAP handlers associated with the Web service.
- Viewing the WSDL of the Web service.
- Creating a Web service security configuration.

7.2 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:

- Section 7.3, "Using the WebLogic Server Administration Console"
- Section 7.4, "Using the Oracle Enterprise Manager Fusion Middleware Control"
- Section 7.5, "Using the WebLogic Scripting Tool"
- Section 7.6, "Using WebLogic Ant Tasks"
- Section 7.7, "Using the Java Management Extensions (JMX)"
- Section 7.8, "Using the Java EE Deployment API"

7.3 Using the WebLogic Server 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 WebLogic 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

For more information about using the Administration Console to administer Web services, see the *Oracle WebLogic Server Administration Console Help*.

The following sections provide more details on the following topics:

- Section 7.3.1, "Invoking the Administration Console"
- Section 7.3.2, "How Web Services Are Displayed In the Administration Console"
- Section 7.3.3, "Creating a Web Services Security Configuration"
- Section 7.3.4, "Monitoring Web Services and Clients"

7.3.1 Invoking the Administration Console

To invoke the Administration Console in your browser, enter the following URL:

`http://host:port/console`

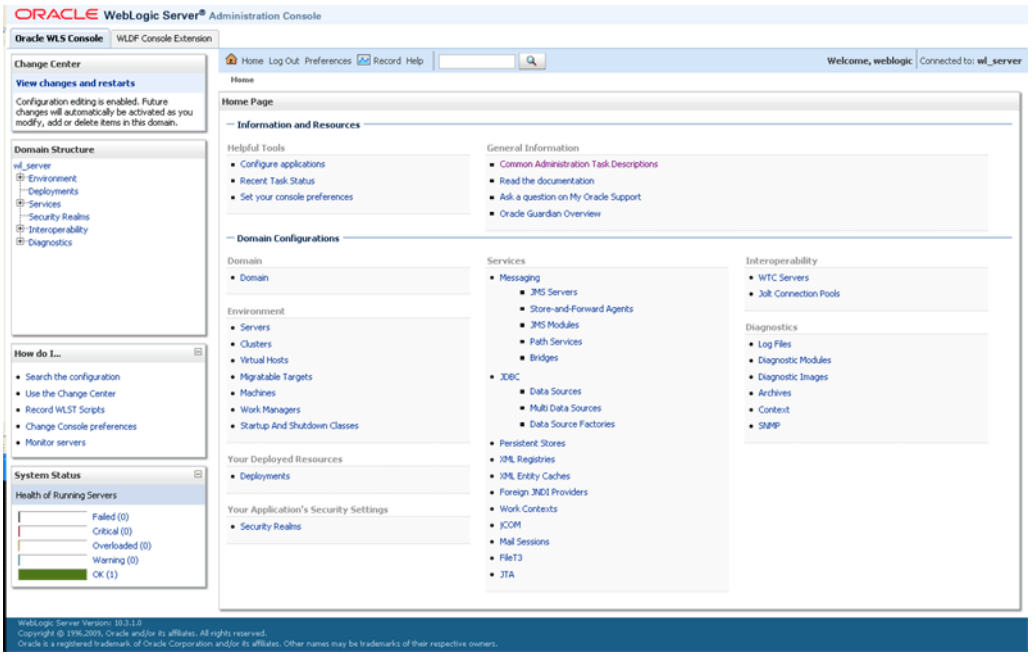
where

- *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



7.3.2 How Web Services Are Displayed In the Administration Console

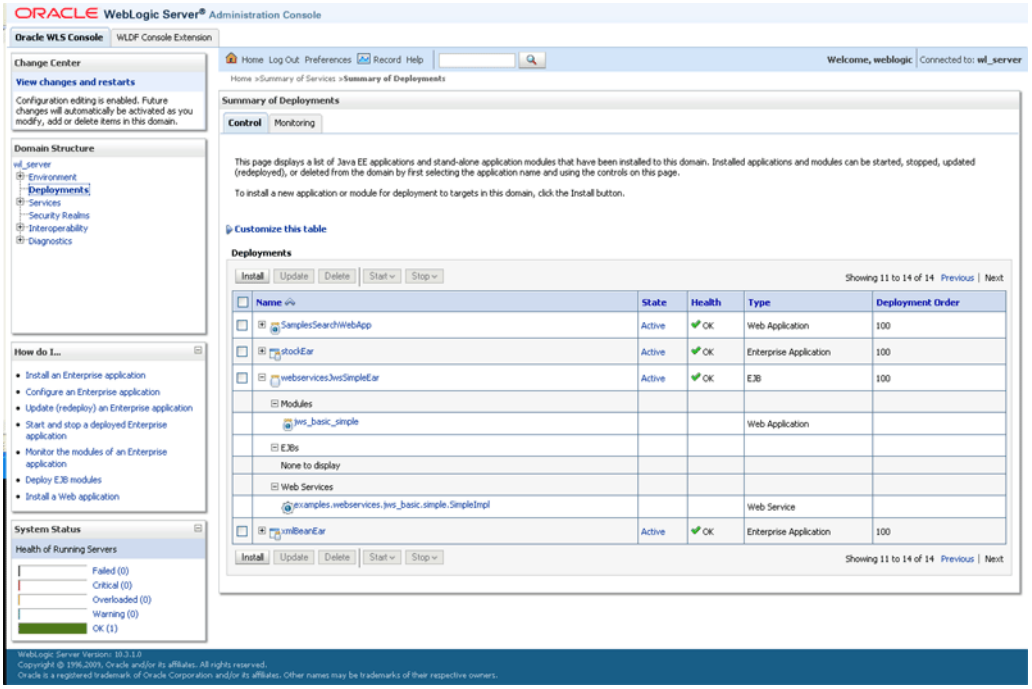
Web services are typically deployed to WebLogic Server as part of an Enterprise Application. The Enterprise Application can be either archived as an EAR, or be in exploded directory format. The Web service itself is almost always packaged as a Web Application; the only exception is if your JWS file implements 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 WebLogic Server Administration Console Main Window



showing how the `HelloWorldService` Web service, packaged inside the `helloWorldEar` Enterprise application, is displayed in the **Deployments** table.

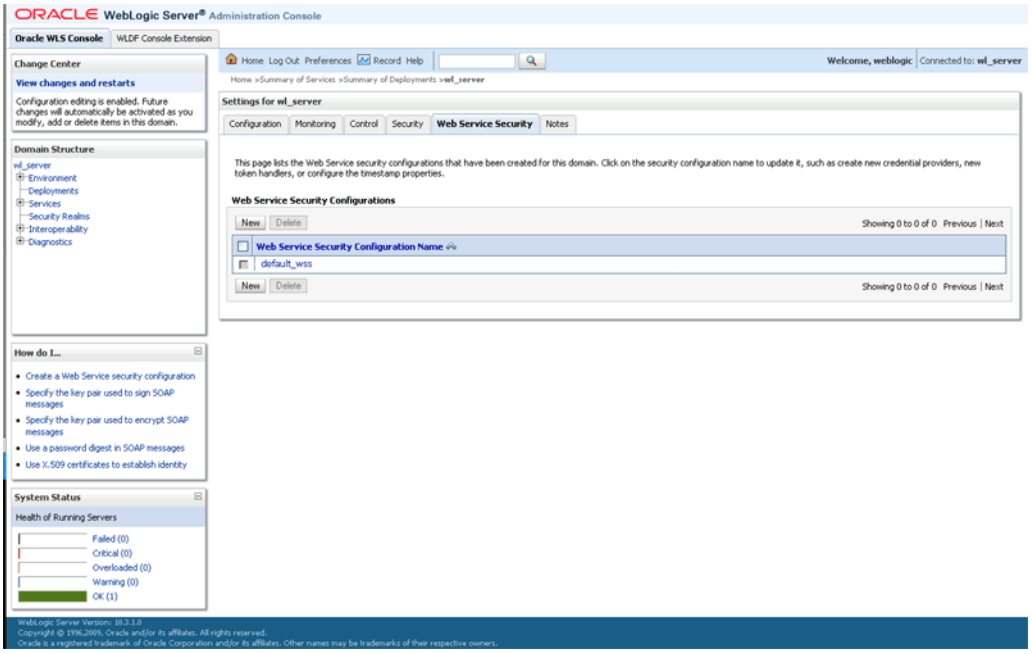
****

7.3.3 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



7.3.4 Monitoring Web Services and Clients

You can monitor runtime information for Web service and client such as number of invocations, errors, faults, and so on from the Administration Console.

To monitor 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 and click the **Monitoring** tab.

The following table lists the tabs that you can select to monitor Web service information. The pages aggregate the statistics of all the servers on which the Web service is running.

Table 7–1 Monitoring Web Services

Click this tab . . .	To view . . .
Monitoring> General	General statistics about the Web services, including total error and invocations counts.
Monitoring> Invocations	Invocation statistics, such as dispatch and execution times and averages.
Monitoring> WS-Policy	Policies that are attached to the Web service, organized into the following categories: authentication, authorization, confidentiality, and integrity.

Table 7–1 (Cont.) Monitoring Web Services

Click this tab . . .	To view . . .
Monitoring> Ports	Table listing the Web service endpoints (ports). The table provides a summary of information for each port. Click a port name to view the public operations that can be invoked by client applications. For each operation, runtime monitoring information is displayed, such as the number of times the operation has been invoked since the WebLogic Server instance started, the average time it took to invoke the Web service, the average time it took to respond, and so on. You can customize the information that is shown in the table by clicking **Customize this table**.
Monitoring> Ports > General	General statistics about the Web service endpoint. The page displays information such as the Web service endpoint name, its URI, and its associated Web service, Enterprise application, and application module. Error and invocations counts are aggregated for all Web service endpoint operations.
Monitoring> Ports > Invocations	Invocation statistics for the Web service endpoint, such as success, fault, and violation counts.
Monitoring> Ports > Cluster Routing	Cluster routing statistics for the Web service endpoint, such as request and response, and routing failures.
Monitoring> Ports > WS-Policy	Statistics related to the policies that are attached to the Web service endpoint, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Ports > Operations	List of operations for the Web service endpoint. For each operation, runtime monitoring information is displayed, such as average response, execution, and dispatch times, response, invocation and error counts, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. **Note**: For JAX-WS Web services, the built-in Ws-Protocol operation displays statistics that are relevant to the underlying WS-* protocols. This information is helpful in evaluating the application performance. Click the name of an operation to view more information. **Click the Genera**l or **Invocations** tab to display general statistics or invocation statistics, respectively, for the selected operation.

To monitor a Web service client 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 client is packaged. Expand the application by clicking the **+** node and click on the application module within which the Web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab. The table provides a summary of runtime information for each Web service client. Click the client name in the table to view more information.

Table 7–2

Click this tab . . .	To view . . .
Monitoring> General	General statistics about the Web service clients, including total error and invocations counts. The page displays the Web service client name, its associated Enterprise application and application module, and context root. Error and invocations statistics are aggregated for all servers on which the Web service is running.
Monitoring> Invocations	Invocation statistics, such as dispatch and execution times and averages.
Monitoring> WS-Policy	Policies that are attached to the Web service client, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Servers	Table listing the server on which the client is currently running. Click the client name and then use the tabs in the following steps to view more information about the Web service client on that server.

Table 7–2 (Cont.)

Click this tab . . .	To view . . .
Monitoring> Servers > General	General statistics about the Web service client. The page displays information such as the Web service client port, its associated Enterprise application, and application module, context root, and so on. Error and invocations counts are aggregated for all Web service client operations.
Monitoring> Servers > Invocations	Invocation statistics for the Web service client, such as success, fault, and violation counts.
Monitoring> Servers > Cluster Routing	Cluster routing statistics for the Web service client, such as request and response, and routing failures. For more information, see "Monitoring Cluster Router Performance" in *Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server*.
Monitoring> Servers > WS-Policy	Statistics related to the policies that are attached to the Web service client, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Servers > Operations	List of operations for the Web service client. For each operation, runtime monitoring information is displayed, such as average response, execution, and dispatch times, response, invocation and error counts, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. **Note**: For JAX-WS Web services, the Web services runtime creates system-defined client instances within a Web service endpoint that are used to send protocol-specific messages as required by that endpoint. These client instances are named after the Web service endpoint that they serve wth the following suffix: `-SystemClient`. Monitoring information relevant to the system-defined client instances is provided to assist in evaluating the application. Click the name of an operation to view more information. **Click the Genera**l or **Invocations** tab to display general statistics or invocation statistics, respectively, for the selected operation.

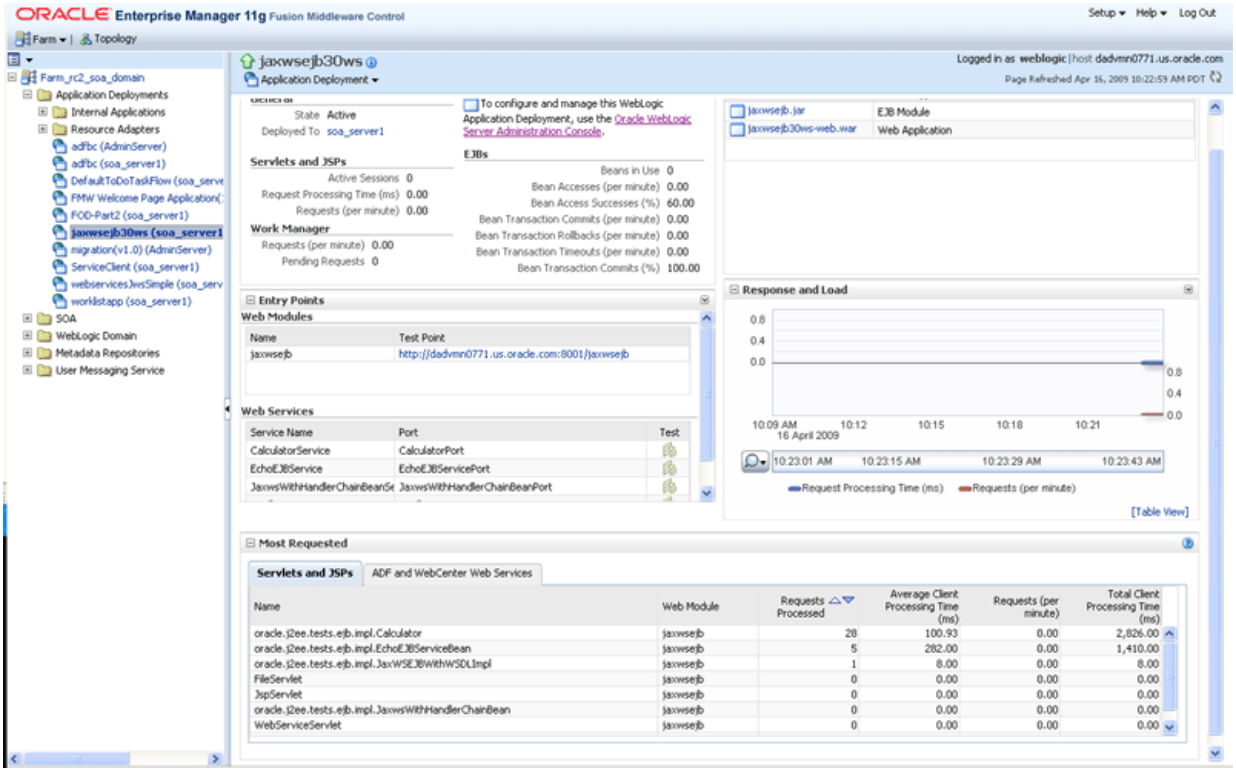
7.4 Using the Oracle Enterprise Manager Fusion Middleware Control

The Oracle Enterprise Manager Fusion Middleware Control (Fusion Middleware Control) Fusion Middleware Control is a Web browser-based, graphical user interface that you can use to monitor and administer a farm. A *farm* is a collection of components managed by Fusion Middleware Control. It can contain Oracle WebLogic Server domains, one or more Managed Servers and the Oracle Fusion Middleware system components that are installed, configured, and running in the domain.

Fusion Middleware Control organizes a wide variety of performance data and administrative functions into distinct, Web-based home pages for the farm, Oracle WebLogic Server domain, components, and applications. The Fusion Middleware Control home pages make it easy to locate the most important monitoring data and the most commonly used administrative functions—all from your Web browser.

The following figure shows Fusion Middleware Control.

Figure 7–4 Oracle Enterprise Manager Fusion Middleware Control



For more information about monitoring and testing Web services using the Enterprise Manager, see "Securing and Administering WebLogic Web Services" in *Security and Administrator's Guide for Web Services*.

Fusion Middleware Control is available as part of the Oracle Fusion Middleware product; it is not available to you if you purchase the standalone version of Oracle WebLogic Server. For more information about Fusion Middleware Control, see "Getting Started Using Oracle Enterprise Manager Fusion Middleware Control" in *Oracle Application Server Administrator's Guide*.

7.5 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 *Oracle WebLogic Scripting Tool*.

7.6 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 for Oracle WebLogic Server* for specific information about the non-Web services related WebLogic Ant tasks.

7.7 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 *Oracle WebLogic Server MBean Reference* and the following sections in *Developing Custom Management Utilities With JMX for Oracle WebLogic Server*:

- "Understanding WebLogic Server MBeans"
- "Accessing WebLogic Server MBeans with JMX"
- "Managing a Domain's Configuration with JMX"

7.8 Using the Java EE Deployment API

In Java EE 5, the *J2EE Application Deployment* specification (JSR-88), described at `http://jcp.org/en/jsr/detail?id=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 Oracle WebLogic Server* for more information.

7.9 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 *Oracle WebLogic Server Administration Console 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 Server Environments for Oracle WebLogic Server*.

8

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.

When migrating your JAX-RPC Web services, to preserve the original WSDL file, use the top-down approach, starting from a WSDL file, to generate the JAX-WS Web service. For more information, see "Developing WebLogic Web Services Starting From a WSDL File: Main Steps" on page 3-10.

> **Note:** In some cases, a JAX-RPC feature may not be supported currently by JAX-WS. 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
Section 8.1, "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.
Section 8.2, "Using WebLogic-specific Annotations"	Describes the WebLogic-specific annotations that are supported by JAX-WS.
Section 8.3, "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.
Section 8.4, "Using JAXB Custom Types"	Describes the use of Java Architecture for XML Binding (JAXB) for managing all of the data binding tasks.
Section 8.5, "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.
Section 8.6, "Migrating from RPC Style SOAP Binding"	Provides guidelines for setting the SOAP binding. RPC style is supported, but not recommended for JAX-WS.
Section 8.7, "Updating SOAP Message Handlers"	Explains how you must re-write your JAX-RPC SOAP message handlers when migrating to JAX-WS.
Section 8.8, "Invoking JAX-WS Clients"	Explains how you must re-write your JAX-RPC client to invoke JAX-WS clients.

8.1 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 "Defining the Context Path of a WebLogic Web Service" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

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.

8.2 Using WebLogic-specific Annotations

JAX-WS supports the following WebLogic-specific annotations:

- @Policy
- @Policies
- @SecurityPolicy
- @SecurityPolicies
- @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 for Oracle WebLogic Server*.

8.3 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 for Oracle WebLogic Server*.

8.4 Using JAXB Custom Types

JAX-WS uses Java Architecture for XML Binding (JAXB), described at `http://jcp.org/en/jsr/detail?id=222`, 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 Chapter 5, "Using JAXB Data Binding."

8.5 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*.

8.6 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.

8.7 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 JAX-WS Web Services for Oracle WebLogic Server*.

8.8 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*.