Oracle® Fusion Middleware

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

12*c* Release 1 (12.1.1)

E24967-01

December 2011

Documentation for software developers that describes how to develop WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

ORACLE

Oracle Fusion Middleware Getting Started With JAX-RPC Web Services for Oracle WebLogic Server, 12*c* Release 1 (12.1.1)

E24967-01

Contents

4 Programming the JWS File

Preface

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

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs 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.
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 chapter provides a summary table of getting started topics for software developers who program WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

JAX-RPC is a specification that defines the Java APIs for making XML-based remote procedure calls (RPC). In particular, these APIs are used to invoke and get a response from a Web service using SOAP 1.1, and XML-based protocol for exchange of information in a decentralized and distributed environment. For more information, see `http://java.net/projects/jax-rpc/`.

> **Note:** 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*.

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"	Review and 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, "Understanding Data Binding"	Use the Java Architecture for XML Binding (JAXB) data binding.
Chapter 6, "Invoking Web Services"	Invoke your Web service from a Java client or another Web service.
Chapter 7, "Administering Web Services"	Administer WebLogic Web services using the Administration Console.

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*.

2

Use Cases and Examples

This chapter describes common use cases and examples for WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

- 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 Java SE 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="JAXRPC"/>
  </jwsc>
</target>
```

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

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

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

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

9. Start the WebLogic Server instance to which the Web service will be deployed.
10. Deploy the Web service, packaged in an enterprise application, to WebLogic Server, using either the Administration Console or the `wldeploy` Ant task. In either case, you deploy the `helloWorldEar` Enterprise application, located in the `output` directory.

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

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

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

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

    ```
    prompt> ant deploy
    ```

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

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

    You construct the URL using the values of the `contextPath` and `serviceUri` attributes of the `WLHttpTransport` JWS annotation; however, because the JWS file in this use case does not include the `WLHttpTransport` annotation, use the default values for the `contextPath` and `serviceUri` attributes: the name of the Java class in the JWS file. 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 Java SE 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="JAXRPC"/>
    </jwsc>
  </target>
  <target name="deploy">
     <wldeploy action="deploy" name="${ear.deployed.name}"
       source="${ear-dir}" user="${wls.username}"
       password="${wls.password}" verbose="true"
       adminurl="t3://${wls.hostname}:${wls.port}"
       targets="${wls.server.name}" />
   </target>
  <target name="undeploy">
     <wldeploy action="undeploy" name="${ear.deployed.name}"
```

```
        failonerror="false"
        user="${wls.username}" password="${wls.password}" verbose="true"
        adminurl="t3://${wls.hostname}:${wls.port}"
        targets="${wls.server.name}" />
    </target>
   <target name="client">
     <clientgen

wsdl="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldImpl?WSDL"
        destDir="${clientclass-dir}"
        packageName="examples.webservices.hello_world.client"
        type="JAXRPC"/>
     <javac
        srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
        includes="**/*.java"/>
     <javac
        srcdir="src" destdir="${clientclass-dir}"
        includes="examples/webservices/hello_world/client/**/*.java"/>
   </target>
   <target name="run">
      <java classname="examples.webservices.hello_world.client.Main"
           fork="true" failonerror="true" >
        <classpath refid="client.class.path"/>
        <arg
   line="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldImpl"  />
      </java> </target>
</project>
```

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 JAX-RPC type mapping file, and so on.

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; `@SOAPBinding` to specify the type of Web service; and `@WLHttpTransport` to specify the URI used to invoke the Web service. The `ComplexImpl.java` JWS file also imports the `examples.webservice.complex.BasicStruct` class and then uses the `BasicStruct` user-defined data type as both a parameter and return value of the `echoStruct()` method.

   For more in-depth information about creating a JWS file, see Chapter 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="JAXRPC">
          <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. This value overrides the value specified in the JWS file using the `@WLHttpTransport` attribute. 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 Java SE 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 WebLogic-specific JWS annotation interface
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct JavaBean
import examples.webservices.complex.BasicStruct;
// Standard JWS annotation that specifies that the portType name of the Web
 // Service is "ComplexPortType", its public service name is "ComplexService",
 // and the targetNamespace used in the generated WSDL is "http://example.org"
@WebService(serviceName="ComplexService", name="ComplexPortType",
             targetNamespace="http://example.org")
// Standard JWS annotation that specifies this is a document-literal-wrapped
 // Web Service
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
              use=SOAPBinding.Use.LITERAL,
              parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```

```
// WebLogic-specific JWS annotation that specifies the context path and service
 // URI used to build the URI of the Web Service is "complex/ComplexService"
@WLHttpTransport(contextPath="complex", serviceUri="ComplexService",
                 portName="ComplexServicePort")
/**
  * This JWS file forms the basis of a WebLogic Web Service.  The Web Services
  * has two public operations:
  *
  *  - echoInt(int)
  *  - echoComplexType(BasicStruct)
  *
  * The Web Service is defined as a "document-literal" service, which means
  * that the SOAP messages have a single part referencing an XML Schema element
  * that defines the entire body.
  */
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="JAXRPC">
       <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="JAXRPC"/>
  <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.
- Data binding artifacts used by WebLogic Server to convert between the XML and Java representations of the Web service parameters and return values.
- 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" />
   </target>
   ```

   The `wsdlc` task in the examples generates the JAR file that contains the JWS SEI and data binding artifacts into the `output/compiledWsdl` directory under the current directory. It also generates a partial implementation file (`TemperaturePortTypeImpl.java`) of the JWS SEI into the `output/impl/examples/webservices/wsdlc` directory (which is a combination of the output directory specified by `destImplDir` and the directory hierarchy specified by the package name). All generated JWS files will be packaged in the `examples.webservices.wsdlc` package.

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

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

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

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

   See 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` and `@WLHttpTransport` JWS annotations that correspond to the values 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="JAXRPC">
        <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. This value overrides the value specified in the JWS file using the `@WLHttpTransport` attribute.

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 Java SE 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 weblogic.jws.*;
/**
  * TemperaturePortTypeImpl class implements web service endpoint
  * interface TemperaturePortType */
@WebService(
   serviceName="TemperatureService",
   targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
   endpointInterface="examples.webservices.wsdlc.TemperaturePortType)
@WLHttpTransport(
   contextPath="temp",
   serviceUri="TemperatureService",
   portName="TemperaturePort")
 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="JAXRPC">
          <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="JAXRPC">
     <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 Java SE Client

> **Note:** As described in this section, you can invoke a Web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). For information about support for *stand-alone* Java applications that are running in an environment where WebLogic Server libraries are not available, see Section 6.4, "Using a Stand-Alone Client JAR File When Invoking Web Services".

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 JAX-RPC `Stub` and `Service` interface implementations for the particular Web service you want to invoke.
- The Java class for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

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 Java 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="JAXRPC"/>
    <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.

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

The `build-client` target also specifies the standard `javac` Ant task, in addition to `clientgen`, to compile all the Java code, including the simple 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 `Main` client application takes a single argument: the WSDL URL of the Web service. The application then follows standard JAX-RPC guidelines to invoke an operation of the Web service using the Web service-specific implementation of the `Service` interface generated by `clientgen`. The application also imports and uses the `BasicStruct` user-defined type, generated by the `clientgen` Ant task, that is used as a parameter and return value for the `echoStruct` operation. For details, see Chapter 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"/>
     <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService" />
   </java>
</target>
```

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

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

```
prompt> ant run
```

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

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

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

2.4.1 Sample Java Client Application

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

```
package examples.webservices.simple_client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
// import the BasicStruct class, used as a param and return value of the
 // echoComplexType operation. The class is generated automatically by
 // the clientgen Ant task.
import examples.webservices.complex.BasicStruct;
/**
  * This is a simple Java client application that invokes the
  * echoComplexType operation of the ComplexService Web service.
  */
public class Main {
  public static void main(String[] args)
       throws ServiceException, RemoteException {
    ComplexService service = new ComplexService_Impl (args[0] + "?WSDL" );
     ComplexPortType port = service.getComplexServicePort();
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
     in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
     System.out.println("echoComplexType called. Result: " + result.getIntValue()
+ ", " + result.getStringValue());
   }
 }
```

2.4.2 Sample Ant Build File For Building Java Client Application

The following `build.xml` file defines tasks to build the Java 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="JAXRPC"
      wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
       destDir="${clientclass-dir}"
       packageName="examples.webservices.simple_client"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/simple_client/*.java"/>
   </target>
  <target name="run" >
     <java fork="true"
           classname="examples.webservices.simple_client.Main"
           failonerror="true" >
       <classpath refid="client.class.path"/>
    <arg line="http://${wls.hostname}:${wls.port}/complex/ComplexService" />
     </java>
   </target>
</project>
```

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.

The procedure for invoking a Web service from a WebLogic Web service is similar to that described in Section 2.4, "Invoking a Web Service from a Java SE 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-RPC programming guidelines in the JWS file that implements the Web service that invokes the other Web service.

The following example shows how to write a JWS file that invokes the `echoComplexType` operation of the `ComplexService` Web service described in 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-RPC stubs, generated later on by the `jwsc` Ant task, as well as the `BasicStruct` Java Bean (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 two parameters: a `BasicStruct` which is passed on to the `echoComplexType` operation of the `ComplexService` Web service, and the URL of the `ComplexService` Web service. The method then uses the standard JAX-RPC APIs to get the `Service` and `PortType` of the `ComplexService`, using the stubs generated by `jwsc`, and then invokes the `echoComplexType` operation.

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

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

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

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

See 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="JAXRPC">
            <WLHttpTransport
              contextPath="ClientService" serviceUri="ClientService"
              portName="ClientServicePort"/>
     <clientgen
        type="JAXRPC"
 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 Java SE 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 java.rmi.RemoteException;
 import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
 import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct data type, generated by clientgen and used
 // by the ComplexService Web Service
import examples.webservices.complex.BasicStruct;
// Import the JAX-RPC Stubs for invoking the ComplexService Web Service.
 // Stubs generated by clientgen
import examples.webservices.service_to_service.ComplexPortType;
 import examples.webservices.service_to_service.ComplexService_Impl;
 import examples.webservices.service_to_service.ComplexService;
@WebService(name="ClientPortType", serviceName="ClientService",
             targetNamespace="http://examples.org")
@WLHttpTransport(contextPath="ClientService", serviceUri="ClientService",
                  portName="ClientServicePort")
public class ClientServiceImpl {
  @WebMethod()
   public String callComplexService(BasicStruct input, String serviceUrl)
       throws ServiceException, RemoteException
   {
    // Create service and port stubs to invoke ComplexService
     ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
     ComplexPortType port = service.getComplexServicePort();
    // Invoke the echoComplexType operation of ComplexService
     BasicStruct result = port.echoComplexType(input);
     System.out.println("Invoked ComplexPortType.echoComplexType." );
    return "Invoke went okay!  Here's the result: '" + result.getIntValue() + ",
" + result.getStringValue() + "'";
  }
```

```
}
```

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="JAXRPC">
         <WLHttpTransport
           contextPath="ClientService" serviceUri="ClientService"
           portName="ClientServicePort"/>
         <clientgen
type="JAXRPC"
           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="JAXRPC"/>
     <javac
        srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
        includes="**/*.java"/>
     <javac
        srcdir="src" destdir="${clientclass-dir}"
        includes="examples/webservices/service_to_service/client/**/*.java"/>
   </target>
   <target name="run">
      <java classname="examples.webservices.service_to_service.client.Main"
            fork="true"
            failonerror="true" >
            <classpath refid="client.class.path"/>
            <arg

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

3 Developing WebLogic Web Services

This chapter describes the iterative development process for WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

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, as well as additional ones. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*. For additional detailed information about this programming model, see *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, to develop Web services. For more information, see "Using Oracle IDEs to Build Web Services" in *Introducing WebLogic Web Services for Oracle WebLogic Server*.

3.2 Configuring Your Domain For Web Services Features

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

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

- Web Services Reliable Messaging
- Buffering
- JMS Transport

> **Note:** A domain that does not contain Web Services resources will still boot and operate correctly for non-Web services scenarios, and any Web Services scenario that does not involve asynchronous request and response. You will, however, see INFO messages in the server log indicating that asynchronous resources have not been configured and that the asynchronous response service for Web services has not been completely deployed.

The following procedures describe how to create and extend a domain so that it is automatically configured 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*.

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-RPC 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. If you want to further configure the JMS services, file stores, or any other feature, select the items on the Select Optional Configuration screen. This is not typical.

   Otherwise, leave all items deselected and click **Next**.
10. When you reach the Configuration Summary screen, verify the domain details and click **Create**.
11. Click **Done** to exit.

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-RPC Extension**.
7. Click **Next**.
8. If you want to further configure the JMS services or file stores, select the items on the Select Optional Configuration screen. This is not typical.

   Otherwise, leave all items deselected and click **Next**.
9. Verify that you are extending the correct domain, then click **Extend**.
10. Click **Done** to exit.

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 3–1 Steps to Develop Web Services Starting From Java

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`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 Section 4.3, "Programming the JWS File: Typical Steps."
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."
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 3, "Developing WebLogic Web Services" 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 3–2 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 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."
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."

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

#	Step	Description
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." The URL used to invoke the WSDL of the deployed Web Service is essentially the same as the value of the `location` attribute of the `<address>` element in the original WSDL (except for the host and port values which now correspond to the host and port of the WebLogic Server instance to which you deployed the service.) This is because the `wsdlc` Ant task generated values for the `contextPath` and `serviceURI` of the `@WLHttpTransport` annotation in the JWS implementation file so that together they create the same URI as the endpoint address specified in the original WSDL.
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.
- All required deployment descriptors, including:
  - Standard and WebLogic-specific Web Services deployment descriptors: `webservices.xml` and `weblogic-webservices.xml`.
  - JAX-RPC mapping files.
  - Java class-implemented Web Services: `web.xml` and `weblogic.xml`.
  - EJB-implemented Web Services: `ejb-jar.xml` and `weblogic-ejb-jar.xml`.
  - Ear deployment descriptor files: `application.xml` and `weblogic-application.xml`.
- The XML Schema representation of any Java user-defined types used as parameters or return values to the Web Service operations.
- The WSDL file that publicly describes the Web Service.

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

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

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

```
<taskdef name="jwsc"
          classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
    <jwsc
       srcdir="src_directory"
       destdir="ear_directory"
       >
       <jws file="JWS_file"
            compiledWsdl="WSDLC_Generated_JAR"
             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="JAXRPC"/>
    </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-RPC 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="JAXRPC"/>
  </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/JMS) that client applications can use when invoking the Web Service, possibly overriding any existing `@WLXXXTransport` annotations.
- Automatically generate the JAX-RPC client stubs of any other Web Service that is invoked within the JWS file.
- Update an existing Enterprise Application or Web application, rather than generate a completely new one.

See "jwsc" 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.
- Data binding artifacts used by WebLogic Server to convert between the XML and Java representations of the Web Service parameters and return values.
- 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 `PortTypeImpl.java`, where `PortType` refers to the `name` attribute of the `<portType>` element in the WSDL file for which you are generating a Web Service. For example, if the port type name is `MyServicePortType`, then the JWS implementation file is called `MyServicePortTypeImpl.java`.
- *Package_name* refers to the package into which the generated JWS 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="JAXRPC" />
</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 port type name in the WSDL file is `TemperaturePortType`, then the name of the JWS implementation file is `TemperaturePortTypeImpl.java`.
- A JAX-RPC 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 `PortTypeImpl.java`, where `PortType` is the name of the portType in the original WSDL. The class file includes everything you need to compile it into a Web Service, except for your own business logic.

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

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

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

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

- You can include the following annotations from the standard (JSR-181) `javax.jws` package in the JWS implementation file: `@WebService`, `@HandlerChain`, `@SOAPMessageHandler`, and `@SOAPMessageHandlers`. If you specify any other JWS annotation from the `javax.jws` package, the `jwsc` Ant task returns error when you try to compile the JWS file into a Web Service.
- You can specify *only* the `serviceName`, `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 WebLogic-specific JWS annotations, as required.

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

The following example shows the `wsdlc`-generated JWS implementation file from the WSDL shown in 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;
 import weblogic.jws.*;
/**
  * TemperaturePortTypeImpl class implements web service endpoint interface
  * TemperaturePortType */
@WebService(
   serviceName="TemperatureService",
   endpointInterface="examples.webservices.wsdlc.TemperaturePortType")
@WLHttpTransport(
   contextPath="temp",
   serviceUri="TemperatureService",
   portName="TemperaturePort")
public class TemperaturePortTypeImpl implements TemperaturePortType {
  public TemperaturePortTypeImpl() {
  }
  public float getTemp(java.lang.String zipcode)
  {
    //replace with your impl here
     return 0;
  }
}
```

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 `contextPath` attribute of the `@WLHttpTransport` annotation, 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` JWS annotation of the JWS file that implements your Web Service or `<WLHttpTransport>` child element of the `jwsc` Ant task; the second takes precedence over the first. If you do not specify *any* `serviceUri` attribute in

either the JWS file or the `jwsc` Ant task, then the `serviceUri` of the Web Service is the default value: the name of the JWS file without its `*.java` extension.

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

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

Further assume that you do *not* override the `contextPath` or `serviceURI` values by setting equivalent attributes for the `<WLHttpTransport>` element of the `jwsc` Ant task. Then the URL to view the WSDL of the Web Service, assuming the service is running on a host called `ariel` at the default port number (`7001`), is:

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

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 `@WLXXXTransport` annotations or `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 not a callback service and can be invoked using JMS Transport"
- Section 3.11.3, "Web Service is a callback service"
- Section 3.11.4, "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 the *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 the *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 not a callback service and can be invoked using JMS Transport

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

   See "Configure Clusters" in the *Oracle WebLogic Server Administration Console Help*.

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

   See "Configure Listen Addresses" in the *Oracle WebLogic Server Administration Console Help*.

3.11.3 Web Service is a callback service

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

   See "Configure HTTP Settings for a Cluster" in the *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 the *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 the *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 the *Oracle WebLogic Server Administration Console Help*.

3.11.4 Web Service is invoked using a proxy server

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

See "Configure HTTP Settings for a Cluster" or "Configure HTTP Protocol" in the *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 the *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 for an Application" 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="JAXRPC" />
     </jwsc>
   </target>
   ```

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

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

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

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

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

```
<target name="compile">
   <wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
            excludes="appStartup,helloWebService">
    ...
    </wlcomplile>
...
 </target>
<target name="appc">
   <wlappc source="${dest.dir}" deprecation="yes" debug="false"
            excludes="helloWebService"/>
</target>
```

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

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

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

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

Your split development directory environment is now updated to include Web Service development. When you rebuild and deploy the entire Enterprise Application, the Web Service will also be deployed as part of the EAR. You invoke the Web Service in the standard way described in Section 3.10, "Browsing to the WSDL of the Web Service."

4 Programming the JWS File

This chapter describes how to program the JWS file that implements the WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

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 run time 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*. See *Programming Advanced Features of JAX-RPC Web Services for Oracle WebLogic Server* for information on using other JWS annotations to program more advanced features, such as Web service reliable messaging, conversations, SOAP message handlers, and so on.

Table 4–1 Steps to Program the JWS File

#	Step	Description

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

2	Import the WebLogic-specific annotations used in your JWS file.	The WebLogic-specific annotations are in the `weblogic.jws` package. For example: `import weblogic.jws.WLHttpTransport;`
3	Add the standard required `@WebService` JWS annotation at the class level to specify that the Java class exposes a Web service.	See 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, RPC-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 WebLogic-specific `@WLHttpTransport` JWS annotation at the class level to specify the context path and service URI used in the URL that invokes the Web service. (Optional)	See Section 4.3.4, "Specifying the Context Path and Service URI of the Web Service (@WLHttpTransport Annotation)." Although this JWS annotation is not required, Oracle recommends you explicitly specify it in your JWS file so that it is clear what URL a client application uses to invoke the Web service.
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 4.3.5, "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 4.3.6, "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 4.3.7, "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;
// Import the WebLogic-specific JWS annotation interfaces
import weblogic.jws.WLHttpTransport;
// Standard JWS annotation that specifies that the porType name of the Web
 // Service is "SimplePortType", the service name is "SimpleService", and the
 // targetNamespace used in the generated WSDL is "http://example.org"
@WebService(name="SimplePortType", serviceName="SimpleService",
            targetNamespace="http://example.org")
// Standard JWS annotation that specifies the mapping of the service onto the
 // SOAP message protocol.  In particular, it specifies that the SOAP messages
 // are document-literal-wrapped.
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
// WebLogic-specific JWS annotation that specifies the context path and
 // service URI used to build the URI of the Web Service is
 // "simple/SimpleService"
@WLHttpTransport(contextPath="simple", serviceUri="SimpleService",
                 portName="SimpleServicePort")
/**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHello
  *
  */
public class SimpleImpl {
  // Standard JWS annotation that specifies that the method should be exposed
   // as a public operation.  Because the annotation does not include the
   // member-value "operationName", the public name of the operation is the
  // same as the method name: sayHello.
  @WebMethod()
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 }
```

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, RPC-encoded or document-literal-wrapped), as shown in the following code excerpt:

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

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

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

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

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.ENCODED`	`SOAPBinding.Use.LITERAL`
`parameterStyle`	`SOAPBinding.ParameterStyle.BARE` `SOAPBinding.ParameterStyle.WRAPPED`	`SOAPBinding.ParameterStyle.WRAPPED`

4.3.4 Specifying the Context Path and Service URI of the Web Service (@WLHttpTransport Annotation)

Use the WebLogic-specific `@WLHttpTransport` annotation to specify the context path and service URI sections of the URL used to invoke the Web service over the HTTP transport, as well as the name of the port in the generated WSDL, as shown in the following code excerpt:

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

In the example, the name of the port in the WSDL (in particular, the `name` attribute of the `<port>` element) file generated by the `jwsc` Ant task is `SimpleServicePort`. The URL used to invoke the Web service over HTTP includes a context path of `simple` and a service URI of `SimpleService`, as shown in the following example:

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

For reference documentation on this and other WebLogic-specific annotations, see "JWS Annotation Reference" in the *WebLogic Web Services Reference*.

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

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

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

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

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

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

```
public class OneWayImpl {
 @WebMethod()
  @Oneway()
 public void ping() {
    System.out.println("ping operation");
  }
...
```

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

None of the attributes of the `@WebMethod` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) 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.

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

4.3.6 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`). The OUT and INOUT modes may be specified only for parameter types that conform to the JAX-RPC definition of `Holder` types. OUT and INOUT modes are only supported for RPC-style operations or for parameters that map to headers.
- `header`—Boolean attribute that, when set to `true`, specifies that the value of the parameter should be retrieved from the SOAP header, rather than the default body.

None of the attributes of the `@WebParam` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) at `http://www.jcp.org/en/jsr/detail?id=181` for the default value of each attribute.

4.3.7 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.

4.4 Accessing Run-Time Information About a Web Service

The following sections describe how to access run-time information about a Web service:

- Section 4.4.1, "Using JwsContext to Access Run-Time Information"—Use the Web service context to access and change run-time information about the service in your JWS file.
- Section 4.4.2, "Using the Stub Interface to Access Run-Time Information"—Get and set properties on the Stub interface in the client file.

4.4.1 Using JwsContext to Access Run-Time Information

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

You can use annotations and WebLogic Web service APIs in your JWS file to access run-time context information, as described in the following sections.

4.4.1.1 Guidelines for Accessing the Web Service Context

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

```
package examples.webservices.jws_context;
import javax.jws.WebMethod;
 import javax.jws.WebService;
```

```
import weblogic.jws.WLHttpTransport;
import weblogic.jws.Context;
import weblogic.wsee.jws.JwsContext;
 import weblogic.wsee.jws.Protocol;
@WebService(name="JwsContextPortType", serviceName="JwsContextService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="contexts", serviceUri="JwsContext",
                 portName="JwsContextPort")
/**
  * Simple web service to show how to use the @Context annotation.
  */
public class JwsContextImpl {
  @Context
   private JwsContext ctx;
  @WebMethod()
   public String getProtocol() {
    Protocol protocol = ctx.getProtocol();
    System.out.println("protocol: " + protocol);
     return "This is the protocol: " + protocol;
   }
}
```

Use the following guidelines in your JWS file to access the run-time context of the Web service, as shown in the code in **bold** in the preceding example:

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

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

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

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

  See the `weblogic.wsee.*` packages in the *Oracle WebLogic Server API Reference* for more documentation about the context-related APIs.

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

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

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

  Use the methods of the `JwsContext` class to access run-time information about the Web service. The following example shows how to get the protocol that was used to invoke the Web service.

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

  See Section 4.4.1.2, "Methods of the JwsContext" for the full list of available methods.

4.4.1.2 Methods of the JwsContext

The following table summarizes the methods of the `JwsContext` that you can use in your JWS file to access run-time information about the Web service. See

`weblogic.wsee.*` packages in the *Oracle WebLogic Server API Reference* for detailed reference information about `JwsContext`, and other context-related APIs, as `Protocol` and `ServiceHandle`.

Table 4–3 Methods of JwsContext

Method	Returns	Description
`isFinished()`	`boolean`	Returns a boolean value specifying whether the current conversation is finished, or if it is still continuing. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`finishConversation()`	`void`	Finishes the current conversation. This method is equivalent to a client application invoking a method that has been annotated with the `@Conversation (Conversation.Phase.FINISH)` JWS annotation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxAge(java.util.Date)`	`void`	Sets a new maximum age for the conversation to an absolute `Date`. If the date parameter is in the past, WebLogic Server immediately finishes the conversation. This method is equivalent to the `maxAge` attribute of the `@Conversational` annotation, which specifies the *default* maximum age of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxAge(String)`	`void`	Sets a new maximum age for the conversation by specifying a `String` duration, such as `1 day`. Valid values for the `String` parameter are a number and one of the following terms: ■ `seconds` ■ `minutes` ■ `hours` ■ `days` ■ `years` For example, to specify a maximum age of ten minutes, use the following syntax: `ctx.setMaxAge("10 minutes")` This method is equivalent to the `maxAge` attribute of the `@Conversational` annotation, which specifies the *default* maximum age of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 4–3 (Cont.) Methods of JwsContext

Method	Returns	Description
`getMaxAge()`	`long`	Returns the maximum allowed age, in seconds, of a conversation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`getCurrentAge()`	`long`	Returns the current age, in seconds, of the conversation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`resetIdleTime()`	`void`	Resets the timer which measures the number of seconds since the last activity for the current conversation. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxIdleTime(long)`	`void`	Sets the number of seconds that the conversation can remain idle before WebLogic Server finishes it due to client inactivity. This method is equivalent to the `maxIdleTime` attribute of the `@Conversational` annotation, which specifies the *default* idle time of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`setMaxIdleTime(String)`	`void`	Sets the number of seconds, specified as a `String`, that the conversation can remain idle before WebLogic Server finishes it due to client inactivity. Valid values for the `String` parameter are a number and one of the following terms: ■ `seconds` ■ `minutes` ■ `hours` ■ `days` ■ `years` For example, to specify a maximum idle time of ten minutes, use the following syntax: `ctx.setMaxIdleTime("10 minutes")` This method is equivalent to the `maxIdleTime` attribute of the `@Conversational` annotation, which specifies the *default* idle time of a conversation. Use this method to override this default value at run time. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.

Table 4–3 (Cont.) Methods of JwsContext

Method	Returns	Description
`getMaxIdleTime()`	`long`	Returns the number of seconds that the conversation is allowed to remain idle before WebLogic Server finishes it due to client inactivity. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`getCurrentIdleTime()`	`long`	Gets the number of seconds since the last client request, or since the conversation's maximum idle time was reset. Use this method only in conversational Web services, or those that have been annotated with the `@Conversation` or `@Conversational` annotation.
`getCallerPrincipal()`	`java.security.Principal`	Returns the security principal associated with the operation that was just invoked, assuming that basic authentication was performed.
`isCallerInRole(String)`	`boolean`	Returns `true` if the authenticated principal is within the specified security role.
`getService()`	`weblogic.wsee.jws.ServiceHandle`	Returns an instance of `ServiceHandle`, a WebLogic Web service API, which you can query to gather additional information about the Web service, such as the conversation ID (if the Web service is conversational), the URL of the Web service, and so on.
`getLogger(String)`	`weblogic.wsee.jws.util.Logger`	Gets an instance of the `Logger` class, which you can use to send messages from the Web service to a log file.
`getInputHeaders()`	`org.w3c.dom.Element[]`	Returns an array of the SOAP headers associated with the SOAP request message of the current operation invoke.
`setUnderstoodInputHeaders(boolean)`	`void`	Indicates whether input headers should be understood.
`getUnderstoodInputHeaders()`	`boolean`	Returns the value that was most recently set by a call to setUnderstoodInputHeader.
`setOutputHeaders(Element[])`	`void`	Specifies an array of SOAP headers that should be associated with the outgoing SOAP response message sent back to the client application that initially invoked the current operation.
`getProtocol()`	`weblogic.wsee.jws.Protocol`	Returns the protocol (such as HTTP/S or JMS) used to invoke the current operation.

4.4.2 Using the Stub Interface to Access Run-Time Information

The`javax.xml.rpc.Stub` interface enables you to dynamically configure the Stub instance in your Web service client file. For more information, see http://download.oracle.com/javaee/5/api/javax/xml/rpc/Stub.html. For example, you can set the target service endpoint dynamically for the `port` Stub instance, as follows:

```
ComplexService service = new ComplexService_Impl (args[0] + "?WSDL" );
 ComplexPortType port = service.getComplexServicePort();
 ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
    "http://localhost:8010/MyContext/MyService");
```

For more information about developing Web service clients, see Chapter 6, "Invoking Web Services."

The following table summarizes the methods of the `Stub` interface that you can use in your JWS file to access run-time information about the Web service.

Table 4–4 Methods of Stub Interface

Method	Returns	Description
`_getProperty()`	`java.lang.Object`	Gets the value of the specified configuration property.
`_getPropertyNames()`	`java.util.Iterator`	Returns an Iterator view of the names of the properties that can be configured on the `Stub` instance.
`_setProperty()`	`void`	Sets the name and value of a configuration property for the `Stub` instance.

The following table defined the `javax.xml.rpc.Stub` property values that you can access from the Stub instance.

Table 4–5 Properties of Stub Interface

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.String`	Flag specifying whether to participate in a session with a service endpoint.
`USERNAME_PROPERTY`	`java.lang.String`	User name used for authentication.

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.

Note: JAX-RPC supports EJB 2.x only; it does not support EJB 3.0.

4.5.1 Programming Guidelines When Implementing an EJB in Your JWS File

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

For more information on EJBGen, see "EJBGen Reference" in *Programming Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

Follow these guidelines when explicitly implementing a stateless session EJB in your JWS file. See Section 4.5.2, "Example of a JWS File That Implements an EJB" for an example; the relevant sections are shown in **bold**:

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

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

- Import the EJBGen annotations, all of which are in the `weblogic.ejbgen` package. At a minimum you need to import the `@Session` annotation; if you want to use additional EJBGen annotations in your JWS file to specify the shape and behavior of the EJB, see the "EJBGen Reference" in *Programming Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server* for the name of the annotation you should import.

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

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

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

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

- Ensure that the JWS class implements `SessionBean`:

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

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

```
  public void ejbCreate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
```

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

4.5.2 Example of a JWS File That Implements an EJB

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

```
package examples.webservices.transactional;
import javax.ejb.SessionBean;
 import javax.ejb.SessionContext;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Transactional;
import weblogic.ejbgen.Session;
@Session(ejbName="TransactionEJB")
@WebService(name="TransactionPortType", serviceName="TransactionService",
```

```
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="transactions", serviceUri="TransactionService",
                 portName="TransactionPort")
/**
  * This JWS file forms the basis of simple EJB-implemented WebLogic
  * Web Service with a single operation: sayHello.  The operation executes
  * as part of a transaction.
  *
  */
public class TransactionImpl implements SessionBean {
  @WebMethod()
   @Transactional(value=true)
  public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
  // Standard EJB methods.  Typically there's no need to override the methods.
  public void ejbCreate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
 }
```

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, such as the corresponding XML Schema representation of the Java user-defined data type, the JAX-RPC type mapping file, and so on.

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

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

These requirements are specified by JAX-RPC; for more detailed information and the complete list of requirements, see the JAX-RPC specification at `http://java.net/projects/jax-rpc/`.

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, "Supported User-Defined Data Types." See Section 5.2, "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 weblogic.jws.WLHttpTransport;
// Import the BasicStruct JavaBean
import examples.webservices.complex.BasicStruct;
@WebService(serviceName="ComplexService", name="ComplexPortType",
             targetNamespace="http://example.org")
...
public class ComplexImpl {
  @WebMethod(operationName="echoComplexType")
  public BasicStruct echoStruct(BasicStruct struct)
  {
     return struct;
   }
 }
```

4.7 Throwing Exceptions

When you write the error-handling Java code in methods of the JWS file, you can either throw your own user-defined exceptions or throw a `javax.xml.rpc.soap.SOAPFaultException` exception. If you throw a `SOAPFaultException`, WebLogic Server maps it to a SOAP fault and sends it to the client application that invokes the operation.

If your JWS file throws any type of Java exception other than `SOAPFaultException`, WebLogic Server tries to map it to a SOAP fault as best it can. However, if you want to control what the client application receives and send it the best possible exception information, you should explicitly throw a `SOAPFaultException` exception or one that extends the exception. See the JAX-RPC specification at http://java.net/projects/jax-rpc/ for detailed information about creating and throwing your own user-defined exceptions.

The following excerpt describes the `SOAPFaultException` class:

```
public class SOAPFaultException extends java.lang.RuntimeException {
     public SOAPFaultException (QName faultcode,
                                String faultstring,
                                String faultactor,
                                javax.xml.soap.Detail detail ) {...}
     public Qname getFaultCode() {...}
     public String getFaultString() {...}
     public String getFaultActor() {...}
     public javax.xml.soap.Detail getDetail() {...}
 }
```

Use the SOAP with Attachments API for Java 1.1 (SAAJ) `javax.xml.soap.SOAPFactory.createDetail()` method to create the `Detail` object, which is a container for `DetailEntry` objects that provide detailed application-specific information about the error.

You can use your own implementation of the `SOAPFactory`, or use Oracle 's, which can be accessed in the JWS file by calling the static method `weblogic.wsee.util.WLSOAPFactory.createSOAPFactory()` which returns a `javax.xml.soap.SOAPFactory object`. Then at run time, use the `-Djavax.xml.soap.SOAPFactory` flag to specify Oracle's `SOAPFactory` implementation as shown:

```
-Djavax.xml.soap.SOAPFactory=weblogic.xml.saaj.SOAPFactoryImpl
```

The following JWS file shows an example of creating and throwing a `SOAPFaultException` from within a method that implements an operation of your Web service; the sections in bold highlight the exception code:

```
package examples.webservices.soap_exceptions;
import javax.xml.namespace.QName;
 import javax.xml.soap.Detail;
 import javax.xml.soap.SOAPException;
 import javax.xml.soap.SOAPFactory;
 import javax.xml.rpc.soap.SOAPFaultException;
// Import the @WebService annotation
import javax.jws.WebService;
// Import WLHttpTransport
import weblogic.jws.WLHttpTransport;
@WebService(serviceName="SoapExceptionsService",
             name="SoapExceptionsPortType",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="exceptions",
```

```
                serviceUri="SoapExceptionsService",
                portName="SoapExceptionsServicePort")
 /**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHelloWorld
  *
  */
public class SoapExceptionsImpl {
  public SoapExceptionsImpl() {
  }
  public void tirarSOAPException() {
    Detail detail = null;
    try {
      SOAPFactory soapFactory = SOAPFactory.newInstance();
       detail = soapFactory.createDetail();
    } catch (SOAPException e) {
         // do something
     }
    QName faultCode = null;
     String faultString = "the fault string";
     String faultActor = "the fault actor";
     throw new SOAPFaultException(faultCode, faultString, faultActor, detail);
   }
 }
```

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

> **Note:** If you create and throw your own exception (rather than use `SOAPFaultException`) and two or more of the properties of your exception class are of the same data type, then you *must* also create setter methods for these properties, even though the JAX-RPC specification does not require it. This is because when a WebLogic Web service receives the exception in a SOAP message and converts the XML into the Java exception class, there is no way of knowing which XML element maps to which class property without the corresponding setter methods.

4.8 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 Java SE 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-RPC APIs in your JWS file.

See Section 6.3, "Invoking a Web Service from Another Web Service" for detailed instructions.

4.9 Programming Additional Miscellaneous Features Using JWS Annotations and APIs

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

4.9.1 Sending Binary Data Using MTOM/XOP

SOAP Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM/XOP) describes a method for optimizing the transmission of XML data of type `xs:base64Binary` in SOAP messages. When the transport protocol is HTTP, MIME attachments are used to carry that data while at the same time allowing both the sender and the receiver direct access to the XML data in the SOAP message without having to be aware that any MIME artifacts were used to marshal the `xs:base64Binary` data. The binary data optimization process involves encoding the binary data, removing it from the SOAP envelope, compressing it and attaching it to the MIME package, and adding references to that package in the SOAP envelope.

The MTOM specification does not require that, when MTOM is enabled, the Web service run time use XOP binary optimization when transmitting `base64binary` data. Rather, the specification allows the run time to choose to do so. This is because in certain cases the run time may decide that it is more efficient to send `base64binary` data directly in the SOAP Message; an example of such a case is when transporting small amounts of data in which the overhead of conversion and transport consumes more resources than just inlining the data as is. The WebLogic Web services implementation for MTOM for JAX-RPC service, however, *always* uses MTOM/XOP when MTOM is enabled.

Support for MTOM/XOP in WebLogic JAX-RPC Web services is implemented using the pre-packaged WS-Policy file `Mtom.xml`. WS-Policy files follow the *WS-Policy* specification, described at `http://www.w3.org/TR/ws-policy`; this specification provides a general purpose model and XML syntax to describe and communicate the policies of a Web service, in this case the use of MTOM/XOP to send binary data. The installation of the pre-packaged `Mtom.xml` WS-Policy file in the `types` section of the Web service WSDL is as follows (provided for your information only; you cannot change this file):

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

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

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

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

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

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

In addition, you can attach the file at run time using by the Administration Console; for details, see "Associate a WS-Policy file with a Web Service" in the *Oracle WebLogic Server Administration Console Help*. This section describes how to use the JWS annotation.

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

In addition, this release of WebLogic Server does not support using MTOM with deprecated 9.*x* security policies.

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

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

```
package examples.webservices.mtom;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Policy;
@WebService(name="MtomPortType",
             serviceName="MtomService",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="mtom",
                  serviceUri="MtomService",
                  portName="MtomServicePort")
@Policy(uri="policy:Mtom.xml", attachToWsdl=true)
public class MtomImpl {
  @WebMethod
   public String echoBinaryAsString(byte[] bytes) {
     return new String(bytes);
   }
```

2. Use the Java `byte[]` data type in your Web service operations as either a return value or input parameter whenever you want the resulting SOAP message to use MTOM/XOP to send or receive the binary data. See the implementation of the `echoBinaryAsString` operation above for an example; this operation simply takes as input an array of `byte` and returns it as a `String`.
3. The WebLogic Web services run time has built in MTOM/XOP support which is enabled if the WSDL for which the `clientgen` Ant task generates client-side artifacts specifies MTOM/XOP support. In your client application itself, simply invoke the operations as usual, using `byte[]` as the relevant data type.

See the *SOAP Message Transmission Optimization Mechanism* specification at `http://www.w3.org/TR/2005/REC-soap12-mtom-20050125` for additional

information about the MTOM/XOP feature itself as well as the version of the specification supported by WebLogic JAX-RPC Web services.

4.9.2 Streaming SOAP Attachments

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

See "weblogic.jws.StreamAttachments" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for an example of specifying that attachments should be streamed.

4.9.3 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 `@weblogic.jws.Binding` annotation in your JWS file and set its single attribute to the value `Binding.Type.SOAP12`, as shown in the following example (relevant code shown in **bold**):

```
package examples.webservices.soap12;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Binding;
@WebService(name="SOAP12PortType",
             serviceName="SOAP12Service",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="soap12",
                  serviceUri="SOAP12Service",
                  portName="SOAP12ServicePort")
@Binding(Binding.Type.SOAP12)
/**
  * This JWS file forms the basis of simple Java-class implemented WebLogic
  * Web Service with a single operation: sayHello.  The class uses SOAP 1.2
  * as its binding.
  *
  */
public class SOAP12Impl {
  @WebMethod()
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 }
```

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

See "weblogic.jws.Binding" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for additional information about this annotation.

4.9.4 Specifying that Operations Run Inside of a Transaction

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

```
package examples.webservices.transactional;
import javax.jws.WebMethod;
 import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
 import weblogic.jws.Transactional;
@WebService(name="TransactionPojoPortType",
             serviceName="TransactionPojoService",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="transactionsPojo",
                  serviceUri="TransactionPojoService",
                  portName="TransactionPojoPort")
/**
  * This JWS file forms the basis of simple WebLogic
  * Web Service with a single operation: sayHello.  The operation executes
  * as part of a transaction.
  *
  */
public class TransactionPojoImpl {
  @WebMethod()
   @Transactional(value=true)
   public String sayHello(String message) {
     System.out.println("sayHello:" + message);
     return "Here is the message: '" + message + "'";
   }
 }
```

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

See "weblogic.jws.Transactional" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for information about additional attributes.

4.9.5 Getting the HttpServletRequest/Response Object

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

```
package examples.webservices.http_transport_utils;
import javax.xml.rpc.server.ServiceLifecycle;
 import javax.xml.rpc.server.ServletEndpointContext;
 import javax.xml.rpc.ServiceException;
import javax.servlet.http.HttpServletRequest;
 import javax.servlet.http.HttpServletResponse;
import javax.jws.WebMethod;
 import javax.jws.WebService;
```

```
import weblogic.jws.WLHttpTransport;
import weblogic.wsee.connection.transport.servlet.HttpTransportUtils;
@WebService(name="HttpTransportUtilsPortType",
            serviceName="HttpTransportUtilsService",
            targetNamespace="http://example.org")
@WLHttpTransport(contextPath="servlet", serviceUri="HttpTransportUtils",
                 portName="HttpTransportUtilsPort")
public class HttpTransportUtilsImpl implements ServiceLifecycle {
  private ServletEndpointContext wsctx = null;
  public void init(Object context) throws ServiceException {
     System.out.println("ServletEndpointContext inited...");
     wsctx = (ServletEndpointContext)context;
   }
  public void destroy() {
     System.out.println("ServletEndpointContext destroyed...");
     wsctx = null;
   }
  @WebMethod()
   public String getServletRequestAndResponse() {
    HttpServletRequest request =
        HttpTransportUtils.getHttpServletRequest(wsctx.getMessageContext());
    HttpServletResponse response =
        HttpTransportUtils.getHttpServletResponse(wsctx.getMessageContext());
    System.out.println("HttpTransportUtils API used successfully.");
     return "HttpTransportUtils API used successfully";
  }
}
```

The important parts of the preceding example are as follows:

- Import the required JAX-RPC and Servlet classes:

  ```
  import javax.xml.rpc.server.ServiceLifecycle;
   import javax.xml.rpc.server.ServletEndpointContext;
   import javax.xml.rpc.ServiceException;
  import javax.servlet.http.HttpServletRequest;
   import javax.servlet.http.HttpServletResponse;
  ```

- Import the WebLogic `HttpTransportUtils` class:

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

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

  ```
  public class HttpTransportUtilsImpl implements ServiceLifecycle
  ```

- Create a variable of data type `ServletEndpointContext`:

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

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

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

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

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

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.
- Use `SOAPFaultExceptions` in your JWS file if you want to control the exception information that is passed back to a client application when an error is encountered while invoking a the Web service.
- Even though it is not required, Oracle recommends you always specify the `portName` attribute of the WebLogic-specific `@WLHttpTransport` annotation in your JWS file. If you do not specify this attribute, the `jwsc` Ant task will generate a port name for you when generating the WSDL file, but this name might not be very user-friendly. A consequence of this is that the `getXXX()` method you use in your client applications to invoke the Web service will not be very well-named. To ensure that your client applications use the most user-friendly methods possible when invoking the Web service, specify a relevant name of the Web service port by using the `portName` attribute.

5 Understanding Data Binding

This chapter describes the data binding and the data types (both built-in and user-defined) that are supported for WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

5.1 Overview of Data Binding

With the emergence of XML as the standard for exchanging data across disparate systems, Web service applications need a way to access documents 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.

As in previous releases, WebLogic Web services support a full set of built-in XML Schema, Java, and SOAP types, as specified by the JAX-RPC specification, described at `http://java.net/projects/jax-rpc/`, that you can use in your Web service operations without performing any additional programming steps. Built-in data types are those such as `integer`, `string`, and `time`.

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

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

If a Web service uses XMLBeans that are compiled with the `-noupa` option, then `-Dweblogic.wsee.bind.setCompileNoUpaRule=true` flag is required to be set in the WebLogic server startup script to ensure the Web service deploys successfully.Otherwise, deployment will fail with the following error: `cos-nonambig: Content model violates the unique particle attribution rule`.

5.2 Supported Built-In Data Types

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

If, however, you use user-defined data types, then you must create the data binding artifacts that convert the data between XML and Java.WebLogic Server includes the `jwsc` and `wsdlc` Ant tasks that can automatically generate the data binding artifacts for most user-defined data types. See Section 5.3, "Supported User-Defined Data Types" for a list of supported XML and Java data types.

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

The following table lists the supported XML Schema data types (target namespace `http://www.w3.org/2001/XMLSchema`) and their corresponding Java data types.

For a list of the supported user-defined XML data types, see Section 5.2.2, "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	Equivalent Java Data Type (lower case indicates a primitive data type)
`boolean`	`boolean`
`byte`	`byte`
`short`	`short`
`int`	`int`
`long`	`long`
`float`	`float`
`double`	`double`
`integer`	`java.math.BigInteger`
`decimal`	`java.math.BigDecimal`
`string`	`java.lang.String`
`dateTime`	`java.util.Calendar`

Table 5–1 (Cont.) Mapping XML Schema Built-in Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
base64Binary	byte[]
hexBinary	byte[]
duration	java.lang.String
time	java.util.Calendar
date	java.util.Calendar
gYearMonth	java.util.Calendar
gYear	java.util.Calendar
gMonthDay	java.util.Calendar
gDay	java.util.Calendar
gMonth	java.util.Calendar
anyURI	java.net.URI
NOTATION	java.lang.String
token	java.lang.String
normalizedString	java.lang.String
language	java.lang.String
Name	java.lang.String
NMTOKEN	java.lang.String
NCName	java.lang.String
NMTOKENS	java.lang.String[]
ID	java.lang.String
IDREF	java.lang.String
ENTITY	java.lang.String
IDREFS	java.lang.String[]
ENTITIES	java.lang.String[]
nonPositiveInteger	java.math.BigInteger
nonNegativeInteger	java.math.BigInteger
negativeInteger	java.math.BigInteger
unsignedLong	java.math.BigInteger
positiveInteger	java.math.BigInteger
unsignedInt	long
unsignedShort	int
unsignedByte	short
Qname	javax.xml.namespace.QName

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

For a list of the supported user-defined Java data types, see Section 5.3.2, "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)	Equivalent XML Schema Data Type
int	int
short	short
long	long
float	float
double	double
byte	byte
boolean	boolean
char	string (with facet of length=1)
java.lang.Integer	int
java.lang.Short	short
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double
java.lang.Byte	byte
java.lang.Boolean	boolean
java.lang.Character	string (with facet of length=1)
java.lang.String	string
java.math.BigInteger	integer
java.math.BigDecimal	decimal
java.util.Calendar	dateTime
java.util.Date	dateTime
byte[]	base64Binary
javax.xml.namespace.QName	Qname
java.net.URI	anyURI
javax.xml.datatype.XMLGregorianCalendar	anySimpleType
javax.xml.datatype.Duration	duration
java.lang.Object	anyType
java.awt.Image	base64Binary
javax.activation.DataHandler	base64Binary
javax.xml.transform.Source	base64Binary
java.util.UUID	string

5.3 Supported User-Defined Data Types

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

If your XML or Java data type is not listed in these tables, and it is not one of the built-in data types listed in Section 5.2, "Supported Built-In Data Types," then you must create the user-defined data type artifacts manually.

5.3.1 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.

For details and examples of the data types, see the JAX-RPC specification at `http://java.net/projects/jax-rpc/`.

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

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

5.3.2 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 User-Defined Java Data Types

Java Data Type	Equivalent XML Schema Data Type
JavaBean whose properties are any supported data type.	`<xsd:complexType>` whose content model is a `<xsd:sequence>` of elements corresponding to JavaBean properties.
Array and multidimensional array of any supported data type (when used as a JavaBean property)	An element in a `<xsd:complexType>` with the `maxOccurs` attribute set to `unbounded`.
`java.lang.Object` **Note:** The data type of the run-time object must be a known type.	`<xsd:anyType>`
Apache XMLBeans (that are inherited from `org.apache.xmlbeans.XmlObject` only) **Note**: The Web service that uses an Apache XMLBeans data type as a return type or parameter must be defined as `document-literal-wrapped` or `document-literal-bare`.	See Apache XMLBeans at `http://xmlbeans.apache.org/index.html`.
`java.util.Collection`	Literal Array
`java.util.List`	Literal Array
`java.util.ArrayList`	Literal Array
`java.util.LinkedList`	Literal Array
`java.util.Vector`	Literal Array
`java.util.Stack`	Literal Array
`java.util.Set`	Literal Array
`java.util.TreeSet`	Literal Array
`java.utils.SortedSet`	Literal Array
`java.utils.HashSet`	Literal Array

Note: The following user-defined Java data type, used as a parameter or return value of a WebLogic Web service in Version 8.1, is no longer supported: JAX-RPC-style enumeration class.

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

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

---

6 Invoking Web Services

This chapter describes how to invoke WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

> **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:

- Java SE client—In its simplest form, a Java SE client is a Java program that has the `Main` public class that you invoke with the `java` command.
- 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.

You can invoke a Web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). For information about support for *stand-alone* Java applications that are running in an environment

where WebLogic Server libraries are not available, see Section 6.4, "Using a Stand-Alone Client JAR File When Invoking Web Services".

The sections that follow describe how to use Oracle's implementation of the JAX-RPC 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 client that runs as part of a WebLogic Server, or a stand-alone client that runs in an environment where WebLogic Server libraries are not available.

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

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

6.1.1 Invoking Web Services Using JAX-RPC

The Java API for XML based RPC (JAX-RPC) is a specification that defines the APIs used to invoke a Web service. WebLogic Server implements the JAX-RPC specification.

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

Table 6–1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
`Service`	Main client interface.
`ServiceFactory`	Factory class for creating `Service` instances.
`Stub`	Base class of the client proxy used to invoke the operations of a Web service.
`Call`	Used to dynamically invoke a Web service.
`JAXRPCException`	Exception thrown if an error occurs while invoking a Web service.

6.1.2 Examples of Clients That Invoke Web Services

WebLogic Server includes examples of creating and invoking WebLogic Web services in the *WL_HOME*`/samples/server/examples/src/examples/webservices` directory, where *WL_HOME* refers to the main WebLogic Server directory. For detailed instructions on how to build and run the examples, open the *WL_HOME*`/samples/server/docs/index.html` Web page in your browser and expand the **WebLogic Server Examples->Examples->API->Web Services** node.

6.2 Invoking a Web Service from a Java SE Client

> **Note:** As described in this section, you can invoke a Web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). For information about support for *stand-alone* Java applications that are running in an environment where WebLogic Server libraries are not available, see Section 6.4, "Using a Stand-Alone Client JAR File When Invoking Web Services".

The following table summarizes the main steps to create a Java SE 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 Java Client."

Table 6–2 Steps to Invoke a Web Service from a Java SE 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 JAX-RPC `Stub` and `Service` interface implementations for the particular Web service you want to invoke.
- The Java class for any user-defined XML Schema data types included in the WSDL file.
- The JAX-RPC mapping deployment descriptor file which contains information about the mapping between the Java user-defined data types and their corresponding XML Schema types in the WSDL file.
- A client-side copy of the WSDL file.

For additional information about the `clientgen` Ant task, such as all the available attributes, see "Ant Task Reference" 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="JAXRPC"/>
 </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 also optional; if not specified, 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.

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

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 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 JAX-RPC stubs and look at the generated `*.java` files. These files are generated into the directory specified by the `destDir` attribute, with subdirectories corresponding to either the value of the `packageName` attribute, or, if this attribute is not specified, to a package based on the `targetNamespace` of the WSDL.

- The *`ServiceName`*`.java` source file contains the `getPortName()` methods for getting the Web service port, where *`ServiceName`* refers to the name of the Web service and *`PortName`* refers to the name of the port. If the Web service was implemented with a JWS file, the name of the Web service is the value of the `serviceName` attribute of the `@WebService` JWS annotation and the name of the port is the value of the `portName` attribute of the `@WLHttpTransport` annotation.
- The *`PortType`*`.java` file contains the method signatures that correspond to the public operations of the Web service, where *`PortType`* refers to the port type of the Web service. If the Web service was implemented with a JWS file, the port type is the value of the `name` attribute of the `@WebService` JWS annotation.

You can also examine the actual WSDL of the Web service; see 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 Java application invokes a Web service operation. The client application takes a single argument: the WSDL of the Web service.The application then uses standard JAX-RPC API code and the Web service-specific implementation of the `Service` interface, generated by `clientgen`, to invoke an operation of the Web service.

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

```
package examples.webservices.simple_client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
// import the BasicStruct class, used as a param and return value of the
 // echoComplexType operation.  The class is generated automatically by
 // the clientgen Ant task.
import examples.webservices.complex.BasicStruct;
/**
  * This is a simple Java client application that invokes the
  * the echoComplexType operation of the ComplexService Web service.
  */
public class Main {
  public static void main(String[] args)
       throws ServiceException, RemoteException {
    ComplexService service = new ComplexService_Impl (args[0] + "?WSDL" );
     ComplexPortType port = service.getComplexServicePort();
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
     in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
     System.out.println("echoComplexType called. Result: " + result.getIntValue()
```

```
+ ", " + result.getStringValue());
   }
 }
```

In the preceding example:

- The following code shows how to create a `ComplexPortType` stub:

  ```
  ComplexService service = new ComplexService_Impl (args[0] + "?WSDL");
   ComplexPortType port = service.getComplexServicePort();
  ```

  The `ComplexService_Impl` stub factory implements the JAX-RPC `Service` interface. The constructor of `ComplexService_Impl` creates a stub based on the provided WSDL URI (`args[0] + "?WSDL"`). The `getComplexServicePort()` method is used to return an instance of the `ComplexPortType` stub implementation.

- The following code shows how to invoke the `echoComplexType` operation of the `ComplexService` Web service:

  ```
  BasicStruct result = port.echoComplexType(in);
  ```

  The `echoComplexType` operation returns the user-defined data type called `BasicStruct`.

The method of your application that invokes the Web service operation must throw or catch `java.rmi.RemoteException` and `javax.xml.rpc.ServiceException`, both of which are thrown from the generated JAX-RPC stubs.

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="JAXRPC"/>
  <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.

You must also copy the following `clientgen`-generated files from `clientgen`'s destination directory to `javac`'s destination directory, keeping the same subdirectory hierarchy in the destination:

packageName/*ServiceName*_internaldd.xml
packageName/*ServiceName*_java_wsdl_mapping.xml
packageName/*ServiceName*_saved_wsdl.wsdl

where *packageName* refers to the subdirectory hierarchy that corresponds to the package of the generated JAX-RPC stubs and *ServiceName* refers to the name of the Web service.

To run the client application, add a `run` target to the `build.xml` that includes a call to the `java` task, as shown below:

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

The following example shows a complete `build.xml` file for generating and compiling a 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="JAXRPC"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/simple_client/*.java"/>
   </target>
  <target name="run" >
     <java fork="true"
           classname="examples.webservices.simple_client.Main"
           failonerror="true" >
       <classpath refid="client.class.path"/>
       <arg  line="http://${wls.hostname}:${wls.port}/complex/ComplexService"
       />
     </java>
   </target>
</project>
```

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 another Java application, as described in Section 6.2, "Invoking a Web Service from a Java SE Client." However, instead of using the `clientgen` Ant task to generate the JAX-RPC stubs of the Web service to be invoked, you use the `<clientgen>` child element of the `<jws>` element, inside the `jwsc` Ant task that compiles the invoking Web service. In the JWS file that invokes the other Web service, however, you still use the same standard JAX-RPC APIs to get `Service` and `PortType` instances to invoke the Web service operations.

It is assumed that you have read and understood Section 6.2, "Invoking a Web Service from a Java SE Client." 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-RPC 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-RPC stubs of the invoked Web service, as well as the Java representation of any user-defined data types used as parameters or return values in the operations of the invoked Web service.

  Note: The user-defined data types are generated into a package based on the XML Schema of the data type in the WSDL, *not* in the package specified by `clientgen`. The JAX-RPC stubs, however, use the package name specified by the `packageName` attribute of the `<clientgen>` element.

- Update the method that contains the invoke of the Web service to either throw or catch both `java.rmi.RemoteException` and `javax.xml.rpc.ServiceException`.
- Get the `Service` and `PortType` JAX-RPC stubs 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="JAXRPC">
           <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="JAXRPC"/>
    <javac
       srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
       includes="**/*.java"/>
    <javac
       srcdir="src" destdir="${clientclass-dir}"
       includes="examples/webservices/service_to_service/client/**/*.java"/>
  </target>
  <target name="run">
     <java classname="examples.webservices.service_to_service.client.Main"
           fork="true"
           failonerror="true" >
           <classpath refid="client.class.path"/>
           <arg

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

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 java.rmi.RemoteException;
 import javax.xml.rpc.ServiceException;
import javax.jws.WebService;
 import javax.jws.WebMethod;
import weblogic.jws.WLHttpTransport;
// Import the BasicStruct data type, generated by clientgen and used
 // by the ComplexService Web Service
import examples.webservices.complex.BasicStruct;
// Import the JAX-RPC Stubs for invoking the ComplexService Web Service.
 // Stubs generated by clientgen
import examples.webservices.service_to_service.ComplexPortType;
 import examples.webservices.service_to_service.ComplexService_Impl;
 import examples.webservices.service_to_service.ComplexService;
@WebService(name="ClientPortType", serviceName="ClientService",
             targetNamespace="http://examples.org")
@WLHttpTransport(contextPath="ClientService", serviceUri="ClientService",
                  portName="ClientServicePort")
public class ClientServiceImpl {
  @WebMethod()
   public String callComplexService(BasicStruct input, String serviceUrl)
       throws ServiceException, RemoteException
   {
    // Create service and port stubs to invoke ComplexService
     ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
     ComplexPortType port = service.getComplexServicePort();
    // Create service and port stubs to invoke ComplexService
     ComplexService service = new ComplexService_Impl(serviceUrl + "?WSDL");
     ComplexPortType port = service.getComplexServicePortTypePort();
     // Invoke the echoComplexType operation of ComplexService
     BasicStruct result = port.echoComplexType(input);
     System.out.println("Invoked ComplexPortType.echoComplexType." );
    return "Invoke went okay!  Here's the result: '" + result.getIntValue() + ",
" + result.getStringValue() + "'";
  }
 }
```

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

- Import any user-defined data types that are used by the invoked Web service. In this example, the `ComplexService` uses the `BasicStruct` JavaBean:

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

- Import the JAX-RPC stubs of the `ComplexService` Web service; the stubs are generated by the `<cliengen>` child element of `<jws>`:

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

- Ensure that your client Web service throws or catches `ServiceException` and `RemoteException`:

```
throws ServiceException, RemoteException
```

- Create the JAX-RPC `Service` and `PortType` instances for the `ComplexService`:

```
ComplexService service = new
      ComplexService_Impl(serviceUrl + "?WSDL");
 ComplexPortType port = service.getComplexServicePortTypePort();
```

- Invoke the `echoComplexType` operation of `ComplexService` using the port you just instantiated:

```
BasicStruct result = port.echoComplexType(input);
```

6.4 Using a Stand-Alone Client JAR File When Invoking Web Services

It is assumed in this document that, when you invoke a Web service using the client-side artifacts generated by the `clientgen` or `wsdlc` Ant tasks, you have the entire set of WebLogic Server classes in your CLASSPATH. If, however, your computer does *not* have WebLogic Server installed, you can still invoke a Web service by using the stand-alone WebLogic Web services client JAR file, as described in this section.

The standalone client JAR file supports basic client-side functionality, such as:

- Use with client-side artifacts created by both the `clientgen` Ant tasks
- Processing SOAP messages
- Using client-side SOAP message handlers
- Using MTOM
- Invoking JAX-RPC Web services
- Using SSL

The stand-alone client JAR file does *not*, however, support invoking Web services that use the following advanced features:

- Web services reliable SOAP messaging
- Message-level security (WS-Security)
- Conversations
- Asynchronous request-response
- Buffering
- JMS transport

To use the stand-alone WebLogic Web services client JAR file with your client application, follow these steps:

1. Copy the file *`WL_HOME`*`/server/lib/wseeclient.zip` from the computer hosting WebLogic Server to the client computer, where *`WL_HOME`* refers to the WebLogic Server installation directory, such as `/Oracle/Middleware/wlserver_12.1`.
2. Unzip the `wseeclient.zip` file into the appropriate directory. For example, you might unzip the file into a directory that contains other classes used by your client application.

3. Add the `wseeclient.jar` file (unzipped from the `wseeclient.zip` file) to your CLASSPATH.

> **Note:** Also be sure that your CLASSPATH includes the JAR file that contains the Ant classes (`ant.jar`). This JAR file is typically located in the `lib` directory of the Ant distribution.

6.5 Using a Proxy Server When Invoking a Web Service

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

6.5.1 Using the HttpTransportInfo API to Specify the Proxy Server

You can programmatically specify within the Java client application itself the details of the proxy server that will proxy the Web service invoke by using the standard `java.net.*` classes and the WebLogic-specific `HttpTransportInfo` API. You use the `java.net` classes to create a `Proxy` object that represents the proxy server, and then use the WebLogic API and properties to set the proxy server on the JAX-RPC stub, as shown in the following sample client that invokes the `echo` operation of the `HttpProxySampleService` Web service. The code in **bold** is described after the example:

```
package dev2dev.proxy.client;
import java.net.Proxy;
 import java.net.InetSocketAddress;
import weblogic.wsee.connection.transport.http.HttpTransportInfo;
/**
  * Sample client to invoke a service through a proxy server via
  * programmatic API
  */
public class HttpProxySampleClient {
   public static void main(String[] args) throws Throwable{
     assert args.length == 5;
     String endpoint = args[0];
     String proxyHost = args[1];
     String proxyPort = args[2];
     String user = args[3];
     String pass = args[4];
    //create service and port
     HttpProxySampleService service = new HttpProxySampleService_Impl();
     HttpProxySamplePortType port = service.getHttpProxySamplePortTypeSoapPort();
    //set endpoint address
     ((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, endpoint);
    //set proxy server info
    Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
Integer.parseInt(proxyPort)));
     HttpTransportInfo info = new HttpTransportInfo();
     info.setProxy(p);
     ((Stub)port)._setProperty("weblogic.wsee.connection.transportinfo",info);
    //set proxy-authentication info
     ((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",user);
     ((Stub)port)._setProperty("weblogic.webservice.client.proxypassword",pass);
```

```
    //invoke
     String s = port.echo("Hello World!");
     System.out.println("echo: " + s);
   }
 }
```

The sections of the preceding example to note are as follows:

- Import the required `java.net.*` classes:

  ```
  import java.net.Proxy;
   import java.net.InetSocketAddress;
  ```

- Import the WebLogic `HttpTransportInfo` API:

  ```
  import weblogic.wsee.connection.transport.http.HttpTransportInfo;
  ```

- Create a `Proxy` object that represents the proxy server:

  ```
  Proxy p = new Proxy(Proxy.Type.HTTP, new  InetSocketAddress(proxyHost,
  Integer.parseInt(proxyPort)));
  ```

  The `proxyHost` and `proxyPort` arguments refer to the host computer and port of the proxy server.

- Create an `HttpTransportInfo` object and use the setProxy() method to set the proxy server information:

  ```
  HttpTransportInfo info = new HttpTransportInfo();
   info.setProxy(p);
  ```

- Use the `weblogic.wsee.connection.transportinfo` WebLogic stub property to set the `HttpTransportInfo` object on the JAX-RPC stub:

  ```
  ((Stub)port)._setProperty("weblogic.wsee.connection.transportinfo",info);
  ```

- Use `weblogic.webservice.client.proxyusername` and `weblogic.webservice.client.proxypassword` WebLogic-specific stub properties to specify the username and password of a user who is authenticated to access the proxy server:

  ```
  ((Stub)port)._setProperty("weblogic.webservice.client.proxyusername",user);
       ((Stub)port)._
  setProperty("weblogic.webservice.client.proxypassword",pass);
  ```

  Alternatively, you can use the `setProxyUsername()` and `setProxyPassword()` methods of the `HttpTransportInfo` API to set the proxy username and password, as shown in the following example:

  ```
      info.setProxyUsername("juliet".getBytes());
       info.setProxyPassword("secret".getBytes());
  ```

6.5.2 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 system properties when you execute the client application.

You have a choice of using standard Java system properties or historical WebLogic properties. If the proxySet system property is set to `false` (proxySet=false), proxy properties will be ignored and no proxy will be used.

The following table summarizes the Java system properties. In this case, the proxySet system property must not be set.

Table 6–3 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>
```

The following table summarizes the WebLogic system properties. In this case, the proxySet system property must be set to `true`.

Table 6–4 WebLogic System Properties Used to Specify the Proxy Server

Property	Description
proxySet=true	Flag that specifies that the historical WebLogic proxy properties should be used.
proxyHost=*proxyHost*	Name of the host computer on which the proxy server is running.
proxyPort=*proxyPort*	Port to which the proxy server is listening.
weblogic.webservice.client. proxyusername=*username*	Username used to access the proxy server.
weblogic.webservice.client. proxypassword=*password*	Password used to access the proxy server.

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

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

```
  <java fork="true"
        classname="clients.InvokeMyService"
        failonerror="true">
    <classpath refid="client.class.path"/>
    <arg line="${http-endpoint}"/>
    <jvmarg line=
     "-DproxySet=true
      -DproxyHost=${proxy-host}
      -DproxyPort=${proxy-port}
      -Dweblogic.webservice.client.proxyusername=${proxy-username}
      -Dweblogic.webservice.client.proxypassword=${proxy-passwd}"
    />
  </java>
</target>
```

6.6 Client Considerations When Redeploying a Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated WebLogic Web service alongside an older version of the same Web service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web service. If the client is connected to a conversational or reliable Web service, its work is considered complete when the existing conversation or reliable messaging sequence is explicitly ended by the client or because of a timeout.

You can continue using the old client application with the new version of the Web service, as long as the following Web service artifacts have not changed in the new version:

- 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-RPC stubs used by the client application by re-running the `clientgen` Ant task.

For example, if you change the signature of an operation in the new version of the Web service, then the WSDL file that describes the new version of the Web service will also change. In this case, you must regenerate the JAX-RPC stubs. If, however, you simply change the implementation of an operation, but do not change its public contract, then you can continue using the existing client application.

6.7 WebLogic Web Services Stub Properties

WebLogic Server provides a set of stub properties that you can set in the JAX-RPC Stub used to invoke a WebLogic Web service. Use the `Stub._setProperty()` method to set the properties, as shown in the following example:

```
((Stub)port)._setProperty(WLStub.MARSHAL_FORCE_INCLUDE_XSI_TYPE,"true");
```

Most of the stub properties are defined in the `WLStub` class. See `"weblogic.wsee.jaxrpc.WLStub"` in the *Oracle WebLogic Server API Reference*for details.

The following table describes additional stub properties not defined in the `WLStub` class.

Table 6–5 Additional Stub Properties

Stub Property	Description
`weblogic.wsee.transport.connection.timeout`	Specifies, in seconds, how long a client application that is attempting to invoke a Web service waits to make a connection. After the specified time elapses, if a connection hasn't been made, the attempt times out.
`weblogic.wsee.transport.read.timeout`	Specifies, in seconds, how long a client application waits for a response from a Web service it is invoking. After the specified time elapses, if a response hasn't arrived, the client times out.
`weblogic.wsee.security.bst.serverVerifyCert`	Specifies the certificate that the client application uses to validate the signed response from WebLogic Server. By default, WebLogic Server includes the certification used to validate in the response SOAP message itself; if this is not possible, then use this stub property to specify a different one. This stub property applies *only* to client applications that run inside of a WebLogic Server container, and not to stand-alone client applications. The value of the property is an object of data type `java.security.cert.X509Certificate`.
`weblogic.wsee.security.bst.serverEncryptCert`	Specifies the certificate that the client application uses to encrypt the request SOAP message sent to WebLogic Server. By default, the client application uses the public certificate published in the Web service's WSDL; if this is not possible, then use this stub property to specify a different one. This stub property applies *only* to client applications that run inside of a WebLogic Server container, and not to stand-alone client applications. The value of the property is an object of data type `java.security.cert.X509Certificate`.

`weblogic.wsee.marshal.forceIncludeXsiType`	Specifies that the SOAP messages for a Web service operation invoke should include the XML Schema data type of each parameter. By default, the SOAP messages do not include the data type of each parameter.
	If you set this property to `True`, the elements in the SOAP messages that describe operation parameters will include an `xsi:type` attribute to specify the data type of the parameter, as shown in the following example:
	`<soapenv:Envelope>`
	`...`
	`<maxResults xsi:type="xs:int">10</maxResults>`
	`...`
	By default (or if you set this property to `False`), the parameter element would look like the following example:
	`<soapenv:Envelope>`
	`...`
	`<maxResults>10</maxResults>`
	`...`
	Valid values for this property are `True` and `False`; default value is `False`.

6.8 Setting the Character Encoding For the Response SOAP Message

Use the `weblogic.wsee.jaxrpc.WLStub.CHARACTER_SET_ENCODING` WLStub property to set the character encoding of the response (outbound) SOAP message. You can set it to the following two values:

- UTF-8
- UTF-16

The following code snippet from a client application shows how to set the character encoding to UTF-16:

```
  Simple port = service.getSimpleSoapPort();
   ((Stub) port)._setProperty(weblogic.wsee.jaxrpc.WLStub.CHARACTER_SET_ENCODING,
"UTF-16");
   port.invokeMethod();
```

See `"weblogic.wsee.jaxrpc.WLStub"` in the *Oracle WebLogic Server API Reference* for additional WLStub properties you can set.

7 Administering Web Services

This chapter describes how to administer WebLogic Web services using Java API for XML-based RPC (JAX-RPC).

This chapter includes the following topics:

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 JMS resources used by Web service reliable messaging and JMS transport
- Configuring the WS-Policy files associated with a Web service endpoint or its operations.

> **Note:** If you used the `@Policy` annotation in your Web service to specify an associated WS-Policy file at the time you programmed the JWS file, you cannot change this association at run-time using the Administration Console or other administrative tools. You can only associate a *new* WS-Policy file, or disassociate one you added at run-time.

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

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 Administration Console"
- 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 Administration Console

The WebLogic Server Administration Console is a Web browser-based, graphical user interface you use to manage a WebLogic Server domain, one or more WebLogic Server instances, clusters, and applications, including Web services, that are deployed to the server or cluster.

One instance of WebLogic Server in each domain is configured as an Administration Server. The Administration Server provides a central point for managing a WebLogic Server domain. All other WebLogic Server instances in a domain are called Managed Servers. In a domain with only a single WebLogic Server instance, that server functions both as Administration Server and Managed Server. The Administration Server hosts the Administration Console, which is a Web Application accessible from any supported Web browser with network access to the Administration Server.

You can use the System Administration Console to:

- "Install an Enterprise application"
- "Start and stop a deployed Enterprise application"
- "Configure an Enterprise application"
- "Configure Web applications"
- "Configure EJBs"

- "Create a deployment plan"
- "Update a deployment plan"
- "Test the modules in an Enterprise application"
- "Configure JMS resources for Web service reliable messaging"
- "Associate the WS-Policy file with a Web service"
- "View the SOAP message handlers of a Web service"
- "View the WSDL of a Web service"
- "Create a Web service security configuration"

7.3.1 Invoking the Administration Console

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

`http://host:port/console`

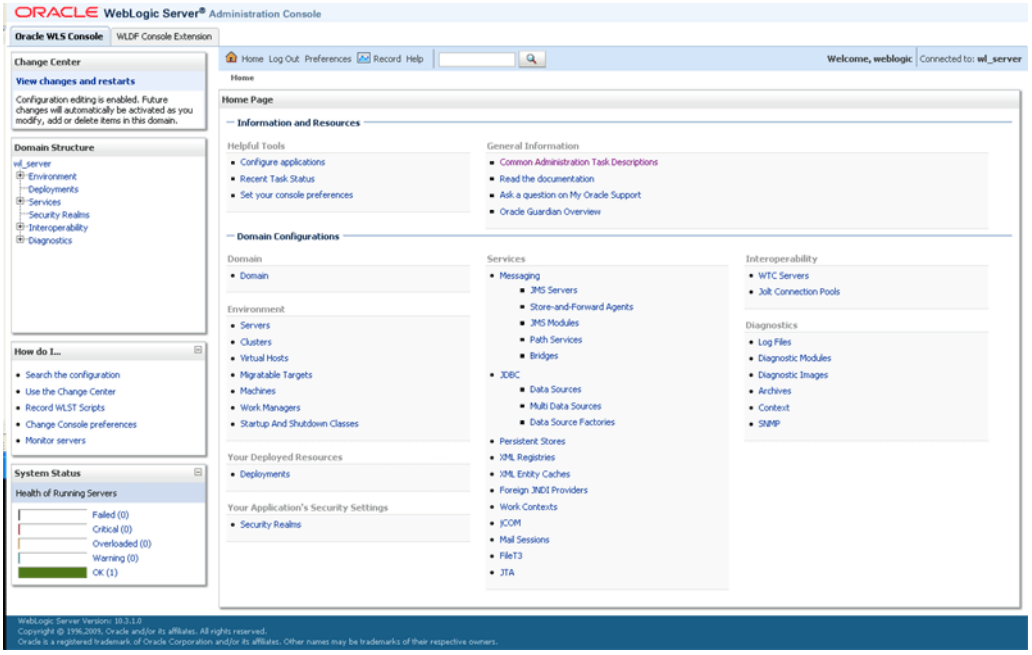
where

- *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

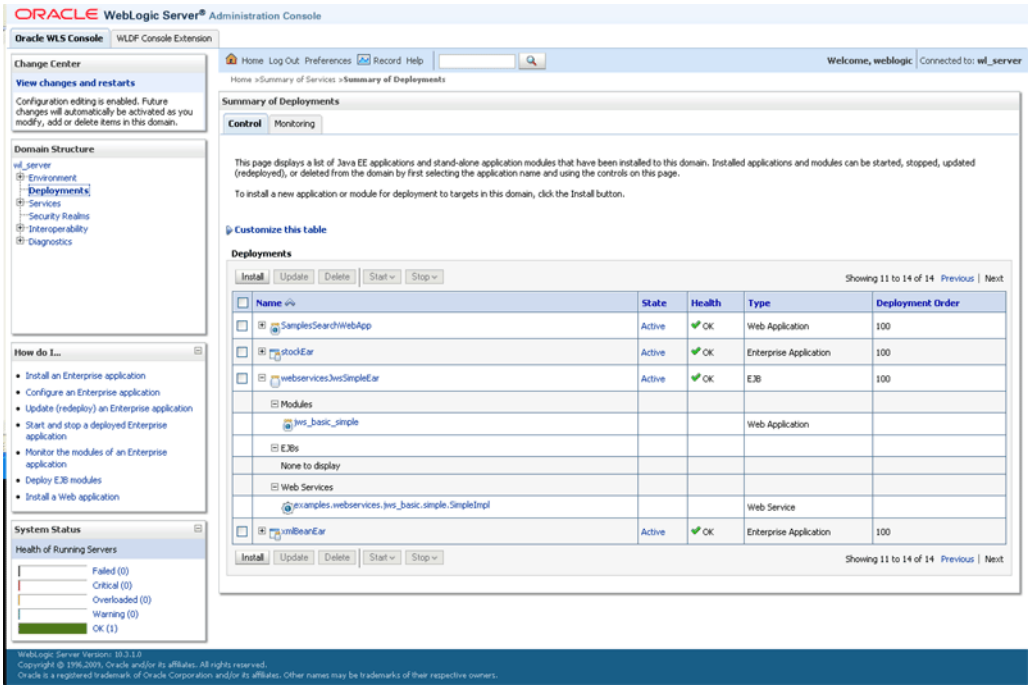
Web services are typically deployed to WebLogic Server as part of an Enterprise Application. The Enterprise Application can be either archived as an EAR, or be in exploded directory format. The Web service itself is almost always packaged as a Web Application; the only exception is if your JWS file implements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



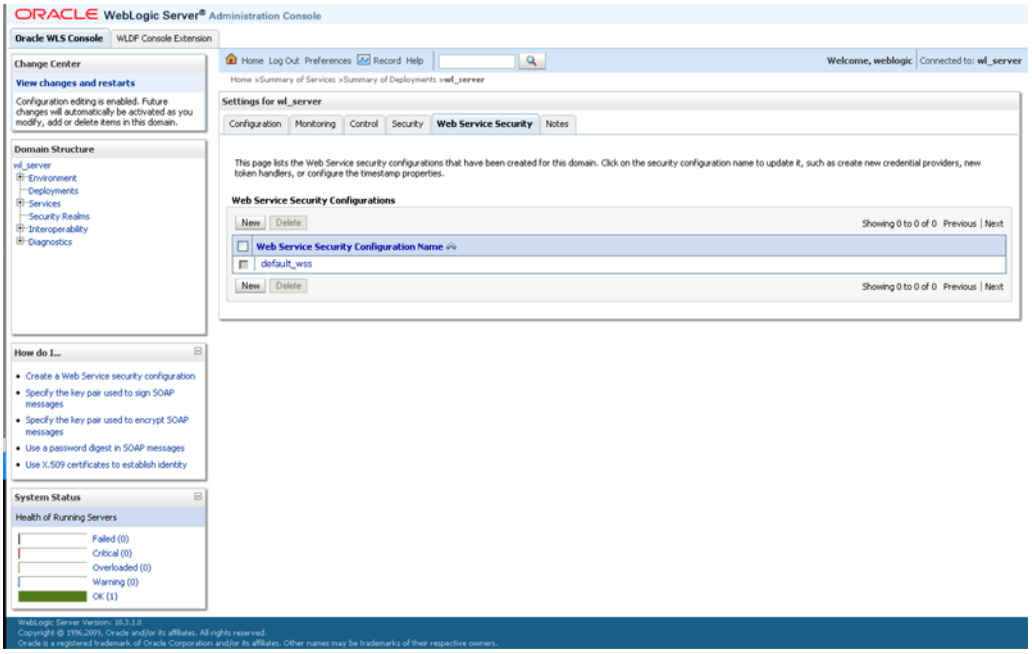
7.3.3 Creating a Web Services Security Configuration

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 run time 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 run-time 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, run-time 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 > 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, run-time 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 run-time 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 > 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, run-time 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**. 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.

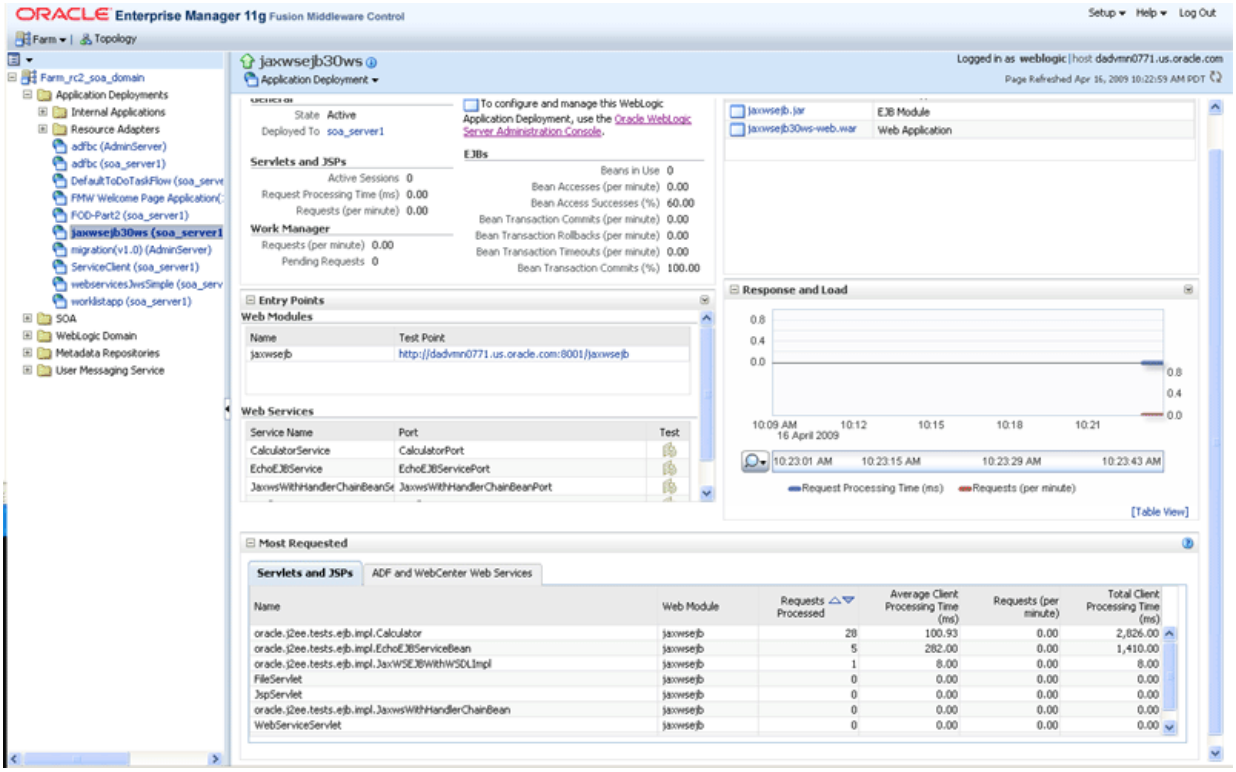
7.4 Using the Oracle Enterprise Manager Fusion Middleware Control

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 Fusion Middleware 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: run-time (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 run-time 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 *Java EE 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 the *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*.