Oracle® WebLogic Server

Programming Advanced Features of WebLogic Web Services Using JAX-WS

10*g* Release 3 (10.3)

July 2008

ORACLE

Contents

CHAPTER 1

Introduction

This document is a resource for software developers who program advanced features for WebLogic Web Services using JAX-WS. The advanced features described are summarized in the following table.

Table 1-1 Programming Advanced Features Using JAX-WS

Advanced Features	Description
Invoking a Web Service Using Asynchronous Request-Response	Invoke a Web Service asynchronously.
Publishing a Web Service Endpoint	Publish a Web Service endpoint at runtime, without deploying the Web Service.
Using Callbacks	Notify a client of a Web Service that an event has happened by programming a callback.
Optimizing Binary Data Transmission Using MTOM/XOP	Send binary data using MTOM/XOP and/or streaming SOAP attachments to optimize transmission of binary data.
Creating Dynamic Proxy Clients	Invoke a Web Service based on a service endpoint interface (SEI) dynamically at run-time without using `clientgen`.
Using XML Catalogs	Use XML catalogs to resolve network resources to versions that are stored locally.
Creating and Using SOAP Message Handlers	Create and configure SOAP message handlers for a Web Service.

Table 1-1 Programming Advanced Features Using JAX-WS (Continued)

Advanced Features	Description
Programming RESTful Web Services	Create a Web Service that follows the RESTful design paradigm.
Publishing and Finding Web Services Using UDDI	Use the UDDI features of WebLogic Web Service.

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

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

CHAPTER 2

Invoking a Web Service Using Asynchronous Request-Response

The following sections describe how to invoke a Web Service using asynchronous request-response:

Overview of the Asynchronous Request-Response Feature

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

When implementing asynchronous request-response in your client, rather than invoking the operation directly, you invoke an asynchronous flavor of the same operation. (This asynchronous flavor of the operation is automatically generated by the `clientgen` Ant task.) For example, rather than invoking an operation called `addNumbers` directly, you would invoke `addNumbersAsync` instead. The asynchronous flavor of the operation always returns `void`, even

if the original operation returns a value. You then include methods in your client that handle the asynchronous response or failures when it returns later on. You put any business logic that processes the return value of the Web Service operation invoke or a potential failure in these methods.

Using Asynchronous Request-Response: Main Steps

The following procedure describes how to create a client that asynchronously invokes an operation in a Web Service. For clarity, it is assumed in the procedure that:

- The client Web Service is called `AsyncClient`.
- The `AsyncClientService` service is going to invoke the `testEcho()` operation of the already deployed `AddNumbersService` service whose WSDL is found at the following URL:

  ```
  http://localhost:7001/async/AddNumbers?WSDL
  ```

It is further assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the generated service. For more information, see the following sections in *Getting Started With WebLogic Web Services Using JAX-WS*

- Use Cases and Examples
- Developing WebLogic Web Services
- Programming the JWS File
- Invoking Web Services

Table 2-1 Steps to Use Asynchronous Request-Response

#	Step	Description
1	Create the asynchronous client.	Within the client, you define an asynchronous callback handler to receive the callback notification and invoke the asynchronous flavor of the Web Service method passing a handle to the asynchronous callback handler. Use your favorite IDE or text editor. See "Creating the Asynchronous Client" on page 2-3.
2	Create an external binding declaration file to enable the creation of the asynchronous methods.	See "Applying Asynchronous Binding Declaration to WSDL" on page 2-6.
2	Update your `build.xml` file to compile the asynchronous client.	You pass the external binding declaration file to the `clientgen` task to automatically generate the asynchronous flavor of the Web Service operations. See "Updating the build.xml File When Using Asynchronous Request-Response" on page 2-6.
3	Run the Ant target to build the `AsyncClient`.	For example: `prompt> ant build-client`

Creating the Asynchronous Client

The following example shows a simple client file, `AsyncClient`, that has a single method, `AddNumbersTestDrive`, that asynchronously invokes the `AddNumbersAsync` method of the `AddNumbersService` service. The Java code in **bold** is described following the code sample.

```
package examples.webservices.async.client;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;

import javax.xml.ws.BindingProvider;

import java.util.concurrent.Future;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

public class AsyncClient  {

   private AddNumbersPortType port = null;
   protected void setUp() throws Exception {
```

```
      AddNumbersService service = new AddNumbersService();
      port = service.getAddNumbersPort();
      String serverURI = System.getProperty("wls-server");
      ((BindingProvider) port).getRequestContext().put(
            BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
             "http://" + serverURI + "/JAXWS_ASYNC/AddNumbersService");
   }

/**
*
* Asynchronous callback handler
*/
   class AddNumbersCallbackHandler implements AsyncHandler<AddNumbersResponse> {
      private AddNumbersResponse output;
      public void handleResponse(Response<AddNumbersResponse> response) {
         try {
            output = response.get();
         } catch (ExecutionException e) {
             e.printStackTrace();
         } catch (InterruptedException e) {
              e.printStackTrace();
          }
      }
      AddNumbersResponse getResponse() {
         return output;
      }
   }

   public void AddNumbersTestDrive() throws Exception {
      int number1 = 10;
      int number2 = 20;
      AddNumbersCallbackHandler callbackHandler =
         new AddNumbersCallbackHandler();
      Future<?> resp = port.addNumbersAsync(number1, number2,
         callbackHandler);

      // For the purposes of a test, block until the async call completes
      resp.get(5L, TimeUnit.MINUTES);
      int result = callbackHandler.getResponse().getReturn();
   }
}
```

When creating the asynchronous client file, you need to perform the following tasks:

1. Create an asynchronous handler that implements the `javax.xml.ws.AsyncHandler<T>` interface. The asynchronous handler defines one method, `handleResponse`, that enables clients to receive callback notifications at the completion of service endpoint operations that are invoked asynchronously. The type should be set to `AddNumberResponse`.

```
class AddNumbersCallbackHandler implements
AsyncHandler<AddNumbersResponse> {
   private AddNumbersResponse output;

   public void handleResponse(Response<AddNumbersResponse> response) {
      try {
         output = response.get();
         } catch (ExecutionException e) {
           e.printStackTrace();
         } catch (InterruptedException e) {
           e.printStackTrace();

         }
      }

      AddNumbersResponse getResponse() {
         return output;
   }
}
```

2. Instantiate the asynchronous callback handler.

   ```
   AddNumbersCallbackHandler callbackHandler =
      new AddNumbersCallbackHandler();
   ```

3. Instantiate the `AddNumbersService` Web Service and call the asynchronous version of the Web Service method, `addNumbersAsync`, passing a handle to the asynchronous callback handler.

   ```
   AddNumbersService service = new AddNumbersService();
   port = service.getAddNumbersPort();
   ...

   Future<?> resp = port.addNumbersAsync(number1, number2,
      callbackHandler);
   ```

   `java.util.concurrent.Future` represents the result of an asynchronous computation and provides methods for checking the status of the asynchronous task, getting the result, or canceling the task execution.

4. Get the result of the asynchronous computation. In this example, a timeout value is specified to wait for the computation to complete.

   ```
   resp.get(5L, TimeUnit.MINUTES);
   ```

5. Use the callback handler to access the response message.

   ```
   int result = callbackHandler.getResponse().getReturn();
   ```

Applying Asynchronous Binding Declaration to WSDL

To generate asynchronous polling and callback methods in the service endpoint interface when the WSDL is compiled, enable the `jaxws:enableAsyncMapping` binding declaration in the WSDL file.

You can create an external binding declarations file that contains all binding declarations for a specific WSDL or XML Schema document. Then, pass the binding declarations file to the `<binding>` child element of the `wsdlc`, `jwsc`, or `clientgen` Ant task.

The following provides an example of a binding declarations file that enables the `jaxws:enableAsyncMapping` binding declaration:

```
<bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="AddNumbers.wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <package name="examples.webservices.async"/>
        <enableAsyncMapping>true</enableAsyncMapping>
    </bindings>
</bindings>
```

For more information, see "Creating an External Binding Declarations File Using JAX-WS Binding Declarations" in *Getting Started With WebLogic Web Services Using JAX-WS*.

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

To update a `build.xml` file to generate client artifacts and compile the client that invokes a Web Service operation asynchronously, add `taskdefs` and a `build-client` target that includes a reference to the external binding declarations file containing the asynchronous binding declaration. See the description following the example for details.

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

<target name="build_client">

<clientgen
      type="JAXWS"
      wsdl="AddNumbers.wsdl"
      destDir="${clientclasses.dir}"
```

```
      packageName="examples.webservices.async.client">
      <binding file="jaxws-binding.xml" />
    </clientgen>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/hello_world/client/**/*.java"/>
</target>
```

Use the `taskdef` Ant task to define the full classname of the `clientgen` Ant tasks. Apply the asynchronous binding declaration by specifying an external binding declarations file, as described in "Applying Asynchronous Binding Declaration to WSDL" on page 2-6. In this case, the `clientgen` Ant task generates both synchronous and asynchronous flavors of the Web Service operations in the JAX-WS stubs.

CHAPTER 3

Publishing a Web Service Endpoint

The `javax.xml.ws.Endpoint` API enables you to create a Web Service endpoint at runtime *without deploying* the Web Service to a WebLogic Server instance.

The following table summarizes the steps to publish a Web Service endpoint.

Table 3-1 Steps to Publish a Web Service Endpoint

#	Step	Description
1	Create a Web Service endpoint.	Use the `javax.xml.ws.Endpoint` `create()` method to create the endpoint, specify the *implementor* (that is, the Web Service implementation) to which the endpoint is associated, and optionally specify the binding type. If not specified, the binding type defaults to `SOAP1.1/HTTP`. The endpoint is associated with only one implementation object and one `javax.xml.ws.Binding`, as defined at runtime; these values cannot be changed. For example, the following example creates a Web Service endpoint for the `CallbackWS()` implementation. `Endpoint callbackImpl = Endpoint.create(new CallbackWS());`

Table 3-1 Steps to Publish a Web Service Endpoint (Continued)

#	Step	Description
2	Publish the Web Service endpoint to accept incoming requests.	Use the `javax.xml.ws.Endpoint` `publish()` method to specify the server context, or the address and optionally the implementor of the Web Service endpoint. **Note:** If you wish to update the metadata documents (WSDL or XML schema) associated with the endpoint, you must do so before publishing the endpoint. For example, the following example publishes the Web Service endpoint created in Step 1 using the server context. `Object sc context.getMessageContext().get(MessageContext.SERVLET_CONTEXT); callbackImpl.publish(sc);`
3	Stop the Web Service endpoint to shut it down and prevent additional requests after processing is complete.	Use the `javax.xml.ws.Endpoint` `stop()` method to shut down the endpoint and stop accepting incoming requests. Once stopped, an endpoint cannot be republished. For example: `callbackImpl.stop()`

For an example of publishing a Web Service endpoint within the context of a callback example, see "Programming Guidelines for the Callback Client Web Service" on page 4-6.

In addition to the steps described in the previous table, you can defined the following using the `javax.xml.ws.Endpoint` API methods:

- Endpoint metadata documents (WSDL or XML schema) associated with the endpoint. You must define metadata before publishing the Web Service endpoint.
- Endpoint properties.
- `java.util.concurrent.Executor` that will be used to dispatch incoming requests to the application.

For more information, see the `javax.xml.ws.Endpoint` Javadoc.

CHAPTER 4

Using Callbacks

The following sections describe how to use callbacks to notify clients of events:

Overview of Callbacks

A callback is a contract between a client and service that allows the service to invoke operations on a client-provided endpoint during the invocation of a service method for the purpose of querying the client for additional data, allowing the client to inject behavior, or notifying the client of progress. The service advertises the requirements for the callback using a WSDL that defines the callback port type and the client informs the service of the callback endpoint address using WS-Addressing.

Example Callback Implementation

The example callback implementation described in this section consists of the following three Java files:

- **JWS file that implements the *callback Web Service***: The callback Web Service defines the callback methods. The implementation simply passes information back to the target Web Service that, in turn, passes the information back to the client Web Service.

  In the example in this section, the callback Web Service is called `CallbackService`. The Web Service defines a single callback method called `callback()`.

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

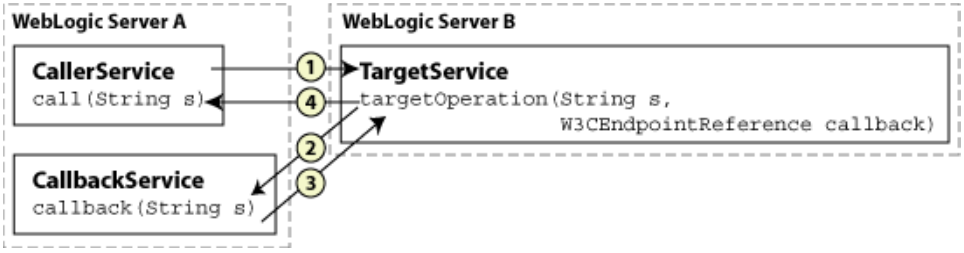
  In the example, this Web Service is called `TargetService` and it defines a single standard method called `targetOperation()`.

- **JWS file that implements the *client Web Service***: The client Web Service invokes an operation of the target Web Service. Often, this Web Service will include one or more methods that specify what the client should do when it receives a callback message back from the target Web Service via a callback method.

  In the example, this Web Service is called `CallerService`. The method that invokes `TargetService` in the standard way is called `call()`.

The following shows the flow of messages for the example callback implementation.

Figure 4-1 Example Callback Implementation



1. The `call()` method of the `CallerService` Web Service, running in one WebLogic Server instance, explicitly invokes the `targetOperation()` method of the `TargetService` and passes a Web Service endpoint to the `CallbackService`. Typically, the `TargetService` service is running in a separate WebLogic Server instance.

2. The implementation of the `TargetService.targetOperation()` method explicitly invokes the `callback()` method of the `CallbackService`, which implements the callback

service, using the Web Service endpoint that is passed in from `CallerService` when the method is called.

3. The `CallbackService.callback()` method sends information back to the `TargetService` Web Service.
4. The `TargetService.targetOperation()` method, in turn, sends the information back to the `CallerService` service, completing the callback sequence.

Programming Callbacks: Main Steps

The procedure in this section describes how to program and compile the three JWS files that are required to implement callbacks: the target Web Service, the client Web Service, and the callback Web Service. The procedure shows how to create the JWS files from scratch; if you want to update existing JWS files, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web Services. For more information, see *Getting Started With WebLogic Web Services Using JAX-WS*.

Table 4-1 Steps to Program Callbacks

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the target Web Service.	Use your favorite IDE or text editor. See "Programming Guidelines for Target Web Service" on page 4-4. **Note:** The JWS file that implements the target Web Service invokes one or more callback methods of the callback Web Service. However, the step that describes how to program the callback Web Service comes later in this procedure. For this reason, programmers typically program the three JWS files at the same time, rather than linearly as implied by this procedure. The steps are listed in this order for clarity only.
2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the target JWS file into a Web Service.	See "Updating the build.xml File for the Target Web Service" on page 4-9.
3	Run the Ant target to build the target Web Service.	For example: `prompt> ant build-target`

Table 4-1 Steps to Program Callbacks (Continued)

#	Step	Description
4	Deploy the target Web Service as usual.	See "Deploying and Undeploying WebLogic Web Services" in *Programming WebLogic Web Services Using JAX-WS*.
5	Create a new JWS file, or update an existing one, that implements the client Web Service.	It is assumed that the client Web Service is deployed to a different WebLogic Server instance from the one that hosts the target Web Service. See "Programming Guidelines for the Callback Client Web Service" on page 4-6.
6	Create the JWS file that implements the callback Web Service.	See "Programming Guidelines for the Callback Web Service" on page 4-8.

7	Update the `build.xml` file that builds the client Web Service.	The `jwsc` Ant task that builds the client Web Service also compiles `CallbackWS.java` and includes the class file in the WAR file using the `Fileset` Ant task element. For example:
		`<clientgen`
		`type="JAXWS"`
		`wsdl="${awsdl}"`
		`packageName="jaxws.callback.client.add"/>`
		`<clientgen`
		`type="JAXWS"`
		`wsdl="${twsdl}"`
		`packageName="jaxws.callback.client.target"/>`
		`<FileSet dir="." >`
		`<include name="CallbackWS.java" />`
		`</FileSet>`

8	Run the Ant target to build the client and callback Web Services.	For example: `prompt> ant build-caller`
9	Deploy the client Web Service as usual.	See "Deploying and Undeploying WebLogic Web Services" in *Programming WebLogic Web Services Using JAX-WS*.

Programming Guidelines for Target Web Service

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

```
package examples.webservices.callback;
```

```
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.wsaddressing.W3CEndpointReference;
import examples.webservices.callback.callbackservice.*;

@WebService(
                portName="TargetPort",
                serviceName="TargetService",
                targetNamespace="http://example.bea.com",
                endpointInterface=
                   "examples.webservices.callback.target.TargetPortType",
                wsdlLocation="/wsdls/Target.wsdl")
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")

public class TargetImpl {
  public String targetOperation(String s, W3CEndpointReference callback)
  {
      CallbackService aservice = new CallbackService();
      CallbackPortType aport =
         aservice.getPort(callback, CallbackPortType.class);
      String result = aport.callback(s);
      return result + " processed by target";
  }
}
```

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

- Import the packages required to pass the callback service endpoint and access the `CallbackService` stub implementation.

  ```
  import javax.xml.ws.wsaddressing.W3CEndpointReference;
  import examples.webservices.callback.callbackservice.*;
  ```

- Create an instance of the `CallbackService` implementation using the stub implementation and get a port by passing the `CallbackService` service endpoint, which is passed by the calling application (`CallerService`).

  ```
  CallbackService aservice = new CallbackService();
  CallbackPortType aport =
       aservice.getPort(callback, CallbackPortType.class);
  ```

- Invoke the callback operation of `CallbackService` using the port you instantiated:

```
String result = aport.callback(s);
```

- Return the result to the `CallerService` service.

```
return result + " processed by target";
```

Programming Guidelines for the Callback Client Web Service

The following example shows a simple JWS file for a client Web Service that invokes the target Web Service described in "Programming Guidelines for Target Web Service" on page 4-4; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.callback;

import javax.annotation.Resource;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.Endpoint;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.WebServiceRef;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.wsaddressing.W3CEndpointReference;

import examples.webservices.callback.target.*;

@WebService(
                portName="CallerPort",
                serviceName="CallerService",
  targetNamespace="http://example.bea.com")
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")

public class CallerImpl
{
   @Resource
   private WebServiceContext context;

   @WebServiceRef()
   private TargetService target;
```

```
   @WebMethod()
   public String call(String s) {
      Object sc =
          context.getMessageContext().get(MessageContext.SERVLET_CONTEXT);
      Endpoint callbackImpl = Endpoint.create(new CallbackWS());
      callbackImpl.publish(sc);
      TargetPortType tPort = target.getTargetPort();
      String result = tPort.targetOperation(s,
           callbackImpl.getEndpointReference(W3CEndpointReference.class));
       callbackImpl.stop();
      return result;
  }
}
```

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

- Import the packages required to access the servlet context, publish the Web Service endpoint, and access the `TargetService` stub implementation.

  ```
  import javax.xml.ws.Endpoint;
  import javax.xml.ws.WebServiceContext;
  import javax.xml.ws.handler.MessageContext;
  import javax.xml.ws.wsaddressing.W3CEndpointReference;
  import examples.webservices.callback.target.*;
  ```

- Get the servlet context using the `WebServiceContext` and `MessageContext`. You will use the servlet context when publishing the Web Service endpoint, later.

  ```
  @Resource
  private WebServiceContext context;
  .
  .
  .
  Object sc
         context.getMessageContext().get(MessageContext.SERVLET_CONTEXT);
  ```

  For more information about accessing runtime information using `WebServiceContext` and `MessageContext`, see "Accessing Runtime Information About a Web Service" in *Getting Started With WebLogic Web Services Using JAX-WS*.

- Create a Web Service endpoint to the `CallbackService` implementation and publish that endpoint to accept incoming requests.

```
Endpoint callbackImpl = Endpoint.create(new CallbackWS());
callbackImpl.publish(sc);
```

For more information about Web Service publishing, see "Publishing a Web Service Endpoint" on page 3-1.

- Access an instance of the `TargetService` stub implementation and invoke the `targetOperation` operation of `TargetService` using the port you instantiated. You pass the `CallbackService` service endpoint as a `javax.xml.ws.wsaddressing.W3CEndpointReference` data type:

```
@WebServiceRef()
private TargetService target;
.
.
.
TargetPortType tPort = target.getTargetPort();
String result = tPort.targetOperation(s,
    callbackImpl.getEndpointReference(W3CEndpointReference.class));
```

- Stop publishing the endpoint:

```
callbackImpl.stop();
```

Programming Guidelines for the Callback Web Service

The following example shows a simple JWS file for a callback Web Service. The `callback` operation is shown in **bold**.

```
package examples.webservices.callback;

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

@WebService(
     portName="CallbackPort",
     serviceName="CallbackService",
     targetNamespace="http://example.bea.com",
     endpointInterface=
         "examples.webservices.callback.callbackservice.CallbackPortType",
     wsdlLocation="/wsdls/Callback.wsdl")

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

public class CallbackWS implements
     examples.webservices.callback.callbackservice.CallbackPortType {

     public CallbackWS() {
     }
```

```
    public java.lang.String callback(java.lang.String arg0) {
        return arg0.toUpperCase();
    }
}
```

Updating the build.xml File for the Target Web Service

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

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

<target name="build-target">
  <jwsc srcdir="src" destdir="${ear-dir}" listfiles="true">
       <jws file="TargetImpl.java"
         compiledWsdl="${cowDir}/target/Target_wsdl.jar" type="JAXWS">
       <WLHttpTransport contextPath="target" serviceUri="TargetService"/>
       </jws>
       <clientgen
         type="JAXWS"
         wsdl="Callback.wsdl"
         packageName="examples.webservices.callback.callbackservice"/>
  </jwsc>
   <zip destfile="${ear-dir}/jws.war" update="true">
     <zipfileset dir="src/examples/webservices/callback" prefix="wsdls">
        <include name="Callback*.wsdl"/>
      </zipfileset>
   </zip>
 </target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks. Update the `jwsc` Ant task that compiles the client Web Service to include:

- `<clientgen>` child element of the `<jws>` element to generate and compile the `Service` interface stubs for the deployed `CallbackService` Web Service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web Service can immediately access the stubs. You do this because the `TartgetImpl` JWS file imports and uses one of the generated classes.

- `<zip>` element to include the WSDL for the `CallbackService` service in the WAR file so that other Web Services can access the WSDL from the following URL: `http://${wls.hostname}:${wls.port}/callback/wsdls/Callback.wsdl`

For more information about `jwsc`, see "Running the jwsc WebLogic Web Services Ant Task" in *Programming WebLogic Web Services Using JAX-RPC*.

CHAPTER 5

Optimizing Binary Data Transmission Using MTOM/XOP

The following sections describe how to use MTOM/XOP to send binary data:

Sending Binary Data Using MTOM/XOP

SOAP Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM/XOP) defines a method for optimizing the transmission of XML data of type `xs:base64Binary` or `xs:hexBinary` 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 `base64Binary` or `hexBinary` data. The binary data optimization process involves the following steps: 1) encode the binary data, 2) remove the binary data from the SOAP envelope, 3) compress the binary data, 4) attach the binary data to the MIME package, and 5) add references to the MIME package in the SOAP envelope.

MTOM/XOP support is standard in JAX-WS via the use of JWS annotations. The MTOM specification does not require that, when MTOM is enabled, the Web Service runtime use XOP binary optimization when transmitting `base64binary` or `hexBinary` data. Rather, the specification allows the runtime to choose to do so. This is because in certain cases the runtime may decide that it is more efficient to send the binary 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 following Java types are mapped to the `base64Binary` XML data type, by default: `javax.activation.DataHandler`, `java.awt.Image`, and `javax.xml.transform.Source`. The elements of type `base64Binary` or `hexBinary` are mapped to `byte[]`, by default.

The following table summarizes the steps required to use MTOM/XOP to send `base64Binary` or `hexBinary` attachments.

Table 5-1 Steps to Use MTOM/XOP to Send Binary Data

#	Step	Description
1	Annotate the data types that you are going to use as an MTOM attachment. (Optional)	Depending on your programming model, you can annotate your Java class or WSDL to define the content types that are used for sending binary data. This step is optional. By default, XML binary types are mapped to Java `byte[]`. For more information, see "Annotating the Data Types" on page 5-2.
2	Enable MTOM on the Web Service.	See "Enabling MTOM on the Web Service" on page 5-4.
3	Enable MTOM on the client of the Web Service.	See "Enabling MTOM on the Client" on page 5-4.
4	Set the attachment threshold.	Set the attachment threshold to specify when the `xs:binary64` data is sent inline or as an attachment. See "Setting the Attachment Threshold" on page 5-5.

Annotating the Data Types

Depending on your programming model, you can annotate your Java class or WSDL to define the MIME content types that are used for sending binary data. This step is optional.

The following table defines the mapping of MIME content types to Java types. In some cases, a default MIME type-to-Java type mapping exists. If no default exists, the MIME content types are mapped to `DataHandler`.

Table 5-2 Mapping of MIME Content Types to Java Types

MIME Content Type	Java Type
`image/gif`	`java.awt.Image`
`image/jpeg`	`java.awt.Image`

Table 5-2 Mapping of MIME Content Types to Java Types (Continued)

MIME Content Type	Java Type
`text/plain`	`java.lang.String`
`text/xml` or `application/xml`	`javax.xml.transform.Source`
`*/*`	`javax.activation.DataHandler`

The following sections describe how to annotate the data types based on whether you are starting from Java or WSDL.

- "Annotating the Data Types: Start From Java" on page 5-3
- "Annotating the Data Types: Start From WSDL" on page 5-3

Annotating the Data Types: Start From Java

When starting from Java, to define the content types that are used for sending binary data, annotate the field that holds the binary data using the `@XmlMimeType` annotation.

The field that contains the binary data must be of type `DataHandler`.

The following example shows how to annotate a field in the Java class that holds the binary data.

```
@WebMethod
@Oneway
public void dataUpload(
    @XmlMimeType("application/octet-stream") DataHandler data)
{
}
```

Annotating the Data Types: Start From WSDL

When starting from WSDL, to define the content types that are used for sending binary data, annotate the WSDL element of type `xs:base64Binary` or `xs:hexBinary` using one of the following attributes:

- `xmime:contentType`—Defines the content type of the element.
- `xmime:expectedContentType`—Defines the range of media types that are acceptable for the binary data.

The following example maps the `image` element of type `base64binary` to `image/gif` MIME type (which maps to the `java.awt.Image` Java type).

```
<element name="image" type="base64Binary"
xmime:expectedContentTypes="image/gif"
xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/>
```

Enabling MTOM on the Web Service

To enable MTOM in the Web Service, specify the `@java.xml.ws.soap.MTOM` annotation on the service endpoint implementation class, as illustrated in the following example. Relevant code is shown in **bold**.

```
package examples.webservices.mtom;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.soap.MTOM;

@MTOM
@WebService(name="MtomPortType",
            serviceName="MtomService",
            targetNamespace="http://example.org")
public class MTOMImpl {
  @WebMethod
  public String echoBinaryAsString(byte[] bytes) {
    return new String(bytes);

  }
}
```

Enabling MTOM on the Client

To enable MTOM on the client of the Web Service, pass the `javax.xml.ws.soap.MTOMFeature` as a parameter when creating the Web Service proxy or dispatch, as illustrated in the following example. Relevant code is shown in **bold**.

```
package examples.webservices.mtom.client;

import javax.xml.ws.soap.MTOMFeature;

public class Main {
  public static void main(String[] args) {
```

```
    String FOO = "FOO";
    MtomService service = new MtomService()
    MtomPortType port = service.getMtomPortTypePort(new MTOMFeature());
    String result = null;
    result = port.echoBinaryAsString(FOO.getBytes());
    System.out.println( "Got result: " + result );
  }
}
```

Setting the Attachment Threshold

You can set the attachment threshold to specify when the `xs:binary64` data is sent inline or as an attachment. By default, the attachment threshold is 0 bytes—all `xs:binary64` data is sent as an attachment.

To set the attachment threshold:

- On the Web Service, pass the `threshold` attribute to the `@java.xml.ws.soap.MTOM` annotation. For example:

  ```
  @MTOM(threshold=3072)
  ```

- On the client of the Web Service, pass the threshold value to `javax.xml.ws.soap.MTOMFeature`. For example:

  ```
  MtomPortType port = service.getMtomPortTypePort(new MTOMFeature(3072));
  ```

In each of the examples above, if a message is greater than or equal to 3 KB, it will be sent as an attachment. Otherwise, the content will be sent inline, as part of the SOAP message body.

Streaming SOAP Attachments

Note: The `com.sun.xml.ws.developer.StreamingDataHandler` API is supported as an extension to the JAX-WS RI, provided by Sun Microsystems. Because this API is not provided as part of the WebLogic software, it is subject to change.

Using MTOM and the `javax.activation.DataHandler` and `com.sun.xml.ws.developer.StreamingDataHandler` APIs 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 particulary large.

The following sections describe how to employ streaming SOAP attachments on the client and server sides.

Client Side Example

The following provides an example that employs streaming SOAP attachments on the client side.

```
package examples.webservices.mtomstreaming.client;

import java.util.Map;
import java.io.InputStream;
import javax.xml.ws.soap.MTOMFeature;
import javax.activation.DataHandler;
import javax.xml.ws.BindingProvider;
import com.sun.xml.ws.developer.JAXWSProperties;
import com.sun.xml.ws.developer.StreamingDataHandler;

public class Main {
   public static void main(String[] args) {
      MtomStreamingService service = new MtomStreamingService();
      MTOMFeature feature = new MTOMFeature();
      MtomStreamingPortType port = service.getMtomStreamingPortTypePort(
         feature);
     Map<String, Object> ctxt=((BindingProvider)port).getRequestContext();
      ctxt.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192);
      DataHandler dh = port.fileUpload(...);
      StreamingDataHandler sdh = {StreamingDataHandler)dh;
      InputStream in = sdh.readOnce();
      ...
      in.close();
      sdh.close();
  }
}
```

The preceding example demonstrates the following:

- To enable MTOM on the client of the Web Service, pass the `javax.xml.ws.soap.MTOMFeature` as a parameter when creating the Web Service proxy or dispatch.

- Configure HTTP streaming support by enabling HTTP chunking on the MTOM streaming client.

  ```
  Map<String, Object> ctxt = ((BindingProvider)port).getRequestContext();
     ctxt.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192);
  ```

- Call the `port.fileUpload` method.
- Cast the `DataHandler` to `StreamingDataHandler` and use the `StreamingDataHandler.readOnce()` method to read the attachment.

Server Side Example

The following provides an example that employs streaming SOAP attachments on the server side.

```
package examples.webservices.mtomstreaming;

import java.io.File;
import java.io.InputStream;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.soap.MTOM;
import javax.activation.DataHandler;
import com.sun.xml.ws.developer.StreamingDataHandler;
...

@MTOM
@WebService(name="MtomStreaming",
            serviceName="MtomStreamingService",
            targetNamespace="http://example.org",
            wsdlLocation="StreamingImplService.wsdl")
public class StreamingImpl {

   // Use @XmlMimeType to map to DataHandler on the client side
   public void fileUpload(String fileName,
                          @XmlMimeType("application/octet-stream")
                          DataHandler data) {
      try {
            StreamingDataHandler dh = (StreamingDataHandler) data;
            File file = new File(fileName);
            dh.moveTo(file);
```

```
            dh.close();
        } catch (Exception e) {
            throw new WebServiceException(e);
    }
}
```

The preceding example demonstrates the following:

- The `@XmlMimeType` annotation is used to map the `DataHandler`, as follows:
  - If starting from WSDL, it is used to map the `xmime:expectedContentTypes="application/octet-stream"` to `DataHandler` in the generated SEI.
  - If starting from Java, it is used to generate an appropriate schema type in the generated WSDL.
- Cast the `DataHandler` to `StreamingDataHandler` and use the `StreamingDataHandler.moveTo(File)` method to store the contents of the attachment to a file.

Configuring Streaming SOAP Attachments

You can configure streaming SOAP attachments on the client and server sides to specify the following:

- Directory in which large attachments are stored.
- Whether to parse eagerly the streaming attachments.
- Maximum attachment size (bytes) that can be stored in memory. Attachments that exceed the specified number of bytes are written to a file.

Configuring Streaming SOAP Attachments on the Server

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

To configure streaming SOAP attachments on the server, add the `@StreamingAttachment` annotation on the endpoint implementation. The following example specifies that streaming attachments are to be parsed eagerly and sets the memory threshold to 4MB. Attachments under 4MB are stored in memory.

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

@StreamingAttachment(parseEagerly=true, memoryThreshold=4000000L)
@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
public class StreamingImpl {
}
```

Configuring Streaming SOAP Attachments on the Client

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

To configure streaming SOAP attachments on the client, create a `StreamingAttachmentFeature` object and pass this as an argument when creating the `PortType` stub implementation. The following example sets the directory in which large attachments are stored to `/tmp`, specifies that streaming attachments are to be parsed eagerly and sets the memory threshold to 4MB. Attachments under 4MB are stored in memory.

```
...
import com.sun.xml.ws.developer.StreamingAttachmentFeature;
...
MTOMFeature mtom = new MTOMFeature();
StreamingAttachmentFeature stf = new StreamingAttachmentFeature("/tmp",
true, 4000000L);
MtomStreamingService service = new MtomStreamingService();
MtomStreamingPortType port = service.getMtomStreamingPortTypePort(
        feature, stf);
...
```

Creating Dynamic Proxy Clients

A *dynamic proxy client* enables a Web Service client to invoke a Web Service based on a service endpoint interface (SEI) dynamically at run-time without using `clientgen`. The steps to create a dynamic proxy client are outlined in the following table. For more information, see the `javax.xml.ws.Service` Javadoc.

Table 6-1 Steps to Create a Dynamic Proxy Client

#	Step	Description

1	Create the `javax.xml.ws.Service` instance.	Create the `Service` instance using the `Service.create` method.
		You must pass the service name and optionally the location of the WSDL document. The method details are as follows:
		`public static Service create (QName serviceName) throws javax.xml.ws.WebServiceException {}`
		`public static Service create (URL wsdlDocumentLocation, QName serviceName) throws javax.xml.ws.WebServiceException {}`
		For example:
		`URL wsdlLocation = new URL("http://example.org/my.wsdl");`
		`QName serviceName = new QName("http://example.org/sample", "MyService");`
		`Service s = Service.create(wsdlLocation, serviceName);`

Table 6-1 Steps to Create a Dynamic Proxy Client (Continued)

#	Step	Description

2	Create the proxy stub.	Use the `Service.getPort` method to create the proxy stub. You can use this stub to invoke operations on the target service endpoint.
		You must pass the service endpoint interface (SEI) and optionally the name of the port in the WSDL service description. The method details are as follows:
		`public <T> T getPort(QName portName, Class<T> serviceEndpointInterface) throws javax.xml.ws.WebServiceException {}`
		`public <T> T getPort(Class<T> serviceEndpointInterface) throws javax.xml.ws.WebServiceException {}`
		For example:
		`MyPort port = s.getPort(MyPort.class);`

CHAPTER 7

Using XML Catalogs

The following sections describe how to use XML catalogs:

Overview of XML Catalogs

An XML catalog enables your application to reference imported XML resources, such as WSDLs and XSDs, from a source that is different from that which is part of the description of the Web Service. Redirecting the XML resources in this way may be required to improve performance or to ensure your application runs properly in your local environment.

For example, a WSDL may be accessible during client generation, but may no longer be accessible when the client is run. You may need to reference a resource that is local to or bundled with your application rather than a resource that is available over the network. Using an XML catalog file, you can specify the location of the WSDL that will be used by the Web Service at runtime.

The following table summarizes how XML catalogs are supported in the WebLogic Server Ant tasks.

Table 7-1 Support for XML Catalogs in WebLogic Server Ant Tasks

Ant Task	Description
`clientgen`	Define and reference XML catalogs in one of the following ways: • Use the `catalog` attribute to specify the name of the external XML catalog file. For more information, see "Defining an External XML Catalog" on page 7-4. • Use the `<xmlcatalog>` child element to reference an embedded XML catalog file. For more information, see "Embedding an XML Catalog" on page 7-5. When you execute the `clientgen` Ant task to build the client (or the `jwsc` Ant task if the `clientgen` task is embedded), the `jax-ws-catalog.xml` file is generated and copied to the client runtime environment. The `jax-ws-catalog.xml` file contains the XML catalog(s) that are defined in the external XML catalog file(s) and/or embedded in the `build.xml` file. This file is copied, along with the referenced XML targets, to the `META-INF` or `WEB-INF` folder for Enterprise or Web applications, respectively. **Note:** The contents of the XML resources are not impacted during this process. You can disable the `jax-ws-catalog.xml` file from being copied to the client runtime environment, as described in "Disabling XML Catalogs in the Client Runtime" on page 7-7.

Table 7-1 Support for XML Catalogs in WebLogic Server Ant Tasks (Continued)

Ant Task	Description
`wsdlc`	Define and reference XML catalogs in one of the following ways: • Use the `catalog` attribute to specify the name of the external XML catalog file. For more information, see "Defining an External XML Catalog" on page 7-4. • Use the `<xmlcatalog>` child element to reference an embedded XML catalog file. For more information, see "Embedding an XML Catalog" on page 7-5. When you execute the `wsdlc` Ant task, the XML resources are copied to the compiled WSDL JAR file or exploded directory.
`wsdlget`	Define and reference XML catalogs in one of the following ways: • Use the `catalog` attribute to specify the name of the external XML catalog file. For more information, see "Defining an External XML Catalog" on page 7-4. • Use the `<xmlcatalog>` child element to reference an embedded XML catalog file. For more information, see "Embedding an XML Catalog" on page 7-5. When you execute the `wsdlget` Ant task, the WSDL and imported resources are downloaded to the specified directory. **Note:** The contents of the XML resources are updated to reference the resources defined in the XML catalog(s).

The following sections describe how to:

- Define and reference an XML catalog to specify the XML resources that you want to redirect. See "Defining and Referencing XML Catalogs" on page 7-4,
- Disable XML catalogs in the client runtime. See "Disabling XML Catalogs in the Client Runtime" on page 7-7.
- Get a local copy of the WSDL and its imported XML resources using `wsdlget`. These files can be packaged with your application and referenced from within an XML catalog. See "Getting a Local Copy of XML Resources" on page 7-7.

For more information about XML catalogs, see the *Oasis XML Catalogs* specification.

Defining and Referencing XML Catalogs

You define an XML catalog and then reference it from the `clientgen` or `wsdlc` Ant task in your `build.xml` file in one of the following ways:

- **Define an external XML catalog**—Define an external XML catalog file and reference that file from the `clientgen` or `wsdlc` Ant tasks in your `build.xml` file using the `catalogs` attribute. For more information, see "Defining an External XML Catalog" on page 7-4.
- **Embed an XML catalog**—Embed the XML catalog directly in the `build.xml` file using the `<xmlcatalog>` element and reference it from the `clientgen` or `wsdlc` Ant tasks in your `build.xml` file using the `<xmlcatalog>` child element. For more information, see "Embedding an XML Catalog" on page 7-5.

In the event of a conflict, entries defined in an embedded XML catalog take precedence over those defined in an external XML catalog.

Note: You can use the `wsdlget` Ant task to get a local copy of the XML resources, as described in "Disabling XML Catalogs in the Client Runtime" on page 7-7.

Defining an External XML Catalog

To define an external XML catalog:

1. Create an external XML catalog file that defines the XML resources that you want to be redirected. See "Creating an External XML Catalog File" on page 7-4.
2. Reference the XML catalog file from the `clientgen` or `wsdlc` Ant task in your `build.xml` file using the `catalogs` attribute. See "Referencing the External XML Catalog File" on page 7-5.

Each step is described in more detail in the following sections.

Creating an External XML Catalog File

The `<catalog>` element is the root element of the XML catalog file and serves as the container for the XML catalog entities. To specify XML catalog entities, you can use the `system` or `public` elements, for example.

The following provides a sample XML catalog file:

```
<catalog xmln="urn:oasis:names:tc:entity:xmlns:xml:catalog"
    prefer="system">
```

```
    <system systemId="http://foo.org/hello?wsdl"
            uri="HelloService.wsdl" />
    <public publicId="ISO 8879:1986//ENTITIES Added Latin 1//EN"
            uri="wsdl/myApp/myApp.wsdl"/>
</catalog>
```

In the above example:

- The `<catalog>` root element defines the XML catalog namespace and sets the `prefer` attribute to `system` to specify that system matches are preferred.
- The `<system>` element associates a URI reference with a system identifier.
- The `<public>` element associates a URI reference with a public identifier.

For a complete description of the XML catalog file syntax, see the *Oasis XML Catalogs* specification.

Referencing the External XML Catalog File

To reference the XML catalog file from the `clientgen` or `wsdlc` Ant task in your `build.xml` file, use the `catalogs` attribute.

The following example shows how to reference an XML catalog file using `clientgen`. Relevant code lines are shown in **bold**.

```
<target name="clientgen">
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"/>
</clientgen>
</target>
```

Embedding an XML Catalog

To embed an XML catalog:

1. Create an embedded XML catalog in the `build.xml` file. See "Creating an Embedded XML Catalog" on page 7-6.

2. Reference the embedded XML catalog from the `clientgen` or `wsdlc` Ant task using the `xmlcatalog` child element. See "Referencing an Embedded XML Catalog" on page 7-6.

Each step is described in more detail in the following sections.

Note: In the event of a conflict, entries defined in an embedded XML catalog take precedence over those defined in an external XML catalog.

Creating an Embedded XML Catalog

The `<xmlcatalog>` element enables you to embed an XML catalog directly in the `build.xml` file. The following shows a sample of an embedded XML catalog in the `build.xml` file.

```
<xmlcatalog id="wsimportcatalog">
     <entity publicid="http://helloservice.org/types/HelloTypes.xsd"
             location="${basedir}/HelloTypes.xsd"/>
</xmlcatalog>
```

For a complete description of the embedded XML catalog syntax, see the *Oasis XML Catalogs* specification.

Referencing an Embedded XML Catalog

The `<xmlcatalog>` child element of the `clientgen` or `wsdlc` Ant tasks enables you to reference an embedded XML catalog. To specify the `<xmlcatalog>` element, use the following syntax:

```
<xmlcatalog refid="id"/>
```

The *id* referenced by the `<xmlcatalog>` child element must match the ID of the embedded XML catalog.

The following example shows how to reference an embedded XML catalog using `clientgen`. Relevant code lines are shown in **bold**.

```
<target name="clientgen">
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"/>
     <xmlcatalog refid="wsimportcatalog"/>
</clientgen>
</target>
```

```
<xmlcatalog id="wsimportcatalog">
     <entity publicid="http://helloservice.org/types/HelloTypes.xsd"
             location="${basedir}/HelloTypes.xsd"/>
</xmlcatalog>
```

Disabling XML Catalogs in the Client Runtime

By default, when you define and reference XML catalogs in your `build.xml` file, as described in "Defining and Referencing XML Catalogs" on page 7-4, when you execute the `clientgen` Ant task to build the client, the `jax-ws-catalog.xml` file is generated and copied to the client runtime environment. The `jax-ws-catalog.xml` file contains the XML catalog(s) that are defined in the external XML catalog file(s) and/or embedded in the `build.xml` file. This file is copied, along with the referenced XML targets, to the `META-INF` or `WEB-INF` folder for Enterprise or Web applications, respectively.

You can disable the generation of the XML catalog artifacts in the client runtime environment by setting the `genRuntimeCatalog` attribute of the `clientgen` to `false`. For example:

```
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"
     genRuntimeCatalog="false"/>
```

In this case, the `jax-ws-catalog.xml` file will not be copied to the runtime environment.

If you generated your client with the `genRuntimeCatalog` attribute set to `false`, to subsequently enable the XML catalogs in the client runtime, you will need to create the `jax-ws-catalog.xml` file manually and copy it to the `META-INF` or `WEB-INF` folder for Enterprise or Web applications, respectively. Ensure that the `jax-ws-catalog.xml` file contains all of the entries defined in the external XML catalog file(s) and/or embedded in the `build.xml` file.

Getting a Local Copy of XML Resources

The `wsdlget` Ant task enables you to get a local copy of XML resources, such as WSDL and XSD files. Then, you can refer to the local version of the XML resources using an XML catalog, as described in "Defining and Referencing XML Catalogs" on page 7-4.

The following excerpt from an Ant `build.xml` file shows how to use the `wsdlget` Ant task to download a WSDL and its XML resources. The XML resources will be saved to the `wsdl` folder in the directory from which the Ant task is run.

```
<target name="wsdlget"
     <wsdlget
          wsdl="http://host/service?wsdl"
          destDir="./wsdl/"
     />
</target>
```

CHAPTER 8

Creating and Using SOAP Message Handlers

The following sections provide information about creating and using SOAP message handlers:

Overview of SOAP Message Handlers

Web Services and their clients may need to access the SOAP message for additional processing of the message request or response. You can create SOAP message handlers to enable Web Services and clients to perform this additional processing on the SOAP message. A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web Service.

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

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

JAX-WS supports two types of SOAP message handlers: SOAP handlers and logical handlers. SOAP handlers can access the entire SOAP message, including the message headers and body. Logical handlers can access the payload of the message only, and cannot change any protocol-specific information (like headers) in a message.

Adding Server-side SOAP Message Handlers: Main Steps

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

It is assumed that you have created a basic JWS file that implements a Web Service and that you want to update the Web Service by adding SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `jwsc` Ant task. For more information, see in *Getting Started With WebLogic Web Services Using JAX-WS*:

- Use Cases and Examples
- Developing WebLogic Web Services
- Programming the JWS File
- Invoking Web Services

Table 8-1 Steps to Add SOAP Message Handlers to a Web Service

#	Step	Description
1	Design the handlers and handler chains.	Design SOAP message handlers and group them together in a *handler chain*. See "Designing the SOAP Message Handlers and Handler Chains" on page 8-4.
2	For each handler in the handler chain, create a Java class that implements the SOAP message handler interface.	See "Creating the SOAP Message Handler" on page 8-7.
3	Update your JWS file, adding annotations to configure the SOAP message handlers.	See "Configuring Handler Chains in the JWS File" on page 8-15.
4	Create the handler chain configuration file.	See "Creating the Handler Chain Configuration File" on page 8-16.
5	Compile all handler classes in the handler chain and rebuild your Web Service.	See "Compiling and Rebuilding the Web Service" on page 8-17.

Adding Client-side SOAP Message Handlers: Main Steps

You can configure client-side SOAP message handlers for both stand-alone clients and clients that run inside of WebLogic Server. You create the actual Java client-side handler in the same way you create a server-side handler—by creating a Java class that implements the SOAP message handler interface. In many cases you can use the exact same handler class on both the Web Service running on WebLogic Server *and* the client applications that invoke the Web Service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client.

The following procedure describes the high-level steps to add client-side SOAP message handlers to the client application that invokes a Web Service operation.

It is assumed that you have created the client application that invokes a deployed Web Service, and that you want to update the client application by adding client-side SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the

`clientgen` Ant task. For more information, see "Invoking a Web Service from a Stand-alone Client: Main Steps" in *Getting Started With WebLogic Web Services Using JAX-WS*.

Table 8-2 Steps to Use Client-side SOAP Message Handlers

#	Step	Description
1	Design the handlers and handler chains.	This step is similar to designing the server-side SOAP message handlers, except the perspective is from the client application, rather than a Web Service. See "Designing the SOAP Message Handlers and Handler Chains" on page 8-4.
2	For each handler in the handler chain, create a Java class that implements the SOAP message handler interface.	This step is similar to designing the server-side SOAP message handlers, except the perspective is from the client application, rather than a Web Service. See "Creating the SOAP Message Handler" on page 8-7 for details about programming a handler class.
3	Update your client to programmatically configure the SOAP message handlers.	See "Configuring the Client-side SOAP Message Handlers" on page 8-18.
4	Update the build.xml file that builds your application, specifying to the clientgen Ant task the customization file.	See "Compiling and Rebuilding the Web Service" on page 8-17.
5	Rebuild your client application by running the relevant task.	`prompt> ant build-client`

When you next run the client application, the SOAP messaging handlers listed in the configuration file automatically execute before the SOAP request message is sent and after the response is received.

Note: You do *not* have to update your actual client application to invoke the client-side SOAP message handlers; as long as you specify to the `clientgen` Ant task the handler configuration file, the generated interface automatically takes care of executing the handlers in the correct sequence.

Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers, you must decide:

- The number of handlers needed to perform the work.
- The sequence of execution.

You group SOAP message handlers together in a *handler chain*. Each handler in a handler chain may define methods for both inbound and outbound messages.

Typically, each SOAP message handler defines a separate set of steps to process the request and response SOAP message because the same type of processing typically must happen for the inbound and outbound message. You can, however, design a handler that processes only the SOAP request and does no equivalent processing of the response. You can also choose not to invoke the next handler in the handler chain and send an immediate response to the client application at any point.

Server-side Handler Execution

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

1. The *inbound* methods for handlers in the handler chain are all executed in the order specified by the JWS annotation. Any of these inbound methods might change the SOAP message request.
2. When the last handler in the handler chain executes, WebLogic Server invokes the back-end component that implements the Web Service, passing it the final SOAP message request.
3. When the back-end component has finished executing, the *outbound* methods of the handlers in the handler chain are executed in the *reverse* order specified by the JWS annotation. Any of these outbound methods might change the SOAP message response.
4. When the first handler in the handler chain executes, WebLogic Server returns the final SOAP message response to the client application that invoked the Web Service.

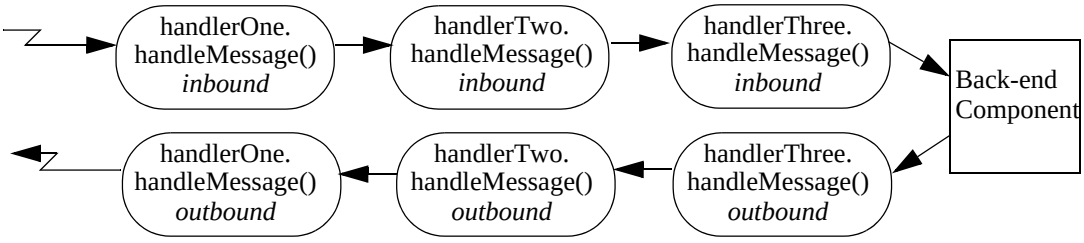
For example, assume that you are going to use the `@HandlerChain` JWS annotation in your JWS file to specify an external configuration file, and the configuration file defines a handler chain called `SimpleChain` that contains three handlers, as shown in the following sample:

```
<?xml version="1.0" encoding="UTF-8" ?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
   <handler-chain>
      <handler>
         <handler-class>
            Handler1
         </handler-class>
```

```
      </handler>
   </handler-chain>
   <handler-chain>
      <handler>
         <handler-class>
            Handler2
         </handler-class>
      </handler>
   </handler-chain>
   <handler-chain>
      <handler>
         <handler-class>
            Handler3
         </handler-class>
      </handler>
   </handler-chain>
</handler-chains>
```

The following graphic shows the order in which WebLogic Server executes the inbound and outbound methods of each handler.

Figure 8-1 Order of Execution of Handler Methods



Client-side Handler Execution

In the case of a client-side handler, the handler executes twice:

- Directly before the client application sends the SOAP request to the Web Service
- Directly after the client application receives the SOAP response from the Web Service

Creating the SOAP Message Handler

There are two types of SOAP message handlers that you can create, as defined in the following table.

Table 8-3 Types of SOAP Message Handlers

Handler Type	Description
SOAP handler	Enables you to access the full SOAP message including headers. SOAP handlers are defined using the `javax.xml.ws.handler.soap.SOAPHandler` interface. They are invoked using the import `javax.xml.ws.handler.soap.SOAPMessageContext` which extends `javax.xml.ws.handler.MessageContext`. The `SOAPMessageContext.getMessage()` method returns a `javax.xml.soap.SOAPMessage`.
Logical handlers	Provides access to the payload of the message. Logical handlers cannot change any protocol-specific information (like headers) in a message. Logical handlers are defined using the `javax.xml.ws.handler.LogicalHandler` interface. They are invoked using the `javax.xml.ws.handler.LogicalMessageContext` which extends `javax.xml.ws.handler.MessageContext`. The `LogicalMessageContext.getMessage()` method returns a `javax.xml.ws.LogicalMessage`. The payload can be accessed either as a JAXB object or as a `javax.xml.transform.Source` object.

Each type of message handler extends the `javax.xml.ws.Handler` interface which defines the methods defined in the following table.

Table 8-4 Handler Interface Methods

Method	Description
`handleMessage()`	Manages normal processing of inbound and outbound messages. A property in the `MessageContext` object is used to determine if the message is inbound or outbound. See "Implementing the Handler.handleMessage() Method" on page 8-10.
`handleFault()`	Manages fault processing of inbound and outbound messages. See "Implementing the Handler.handleFault() Method" on page 8-11.
`close()`	Concludes the message exchange and cleans up resources that were accessed during processing. See "Implementing the Handler.close() Method" on page 8-12.

In addition, you can use the `@javax.annotation.PostConstruct` and `@javax.annotation.PreDestroy` annotations to identify methods that must be executed after the handler is created and before the handler is destroyed, respectively.

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the `javax.xml.soap.SOAPMessage` abstract class, which is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification For details, see "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 8-13.

Example of a SOAP Handler

The following example illustrates a simple SOAP handler that returns whether the message is inbound or outbound along with the message content.

```
package examples.webservices.handler;

import java.util.Set;
import java.util.Collections;
import javax.xml.namespace.QName;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPMessage;
```

```
public class Handler1 implements SOAPHandler<SOAPMessageContext>
{
  public Set<QName> getHeaders()
  {
    return Collections.emptySet();
  }

  public boolean handleMessage(SOAPMessageContext messageContext)
  {
     Boolean outboundProperty = (Boolean)
         messageContext.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);

     if (outboundProperty.booleanValue()) {
         System.out.println("\nOutbound message:");
     } else {
         System.out.println("\nInbound message:");
     }

    System.out.println("** Response: "+messageContext.getMessage().toString());
    return true;
  }

  public boolean handleFault(SOAPMessageContext messageContext)
  {
    return true;
  }

  public void close(MessageContext messageContext)
  {
  }
}
```

Example of a Logical Handler

The following example illustrates a simple logical handler that returns whether the message is inbound or outbound along with the message content.

```
package examples.webservices.handler;

import java.util.Set;
import java.util.Collections;
import javax.xml.namespace.QName;
import javax.xml.ws.handler.LogicalHandler;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.LogicalMessageContext;
import javax.xml.ws.LogicalMessage;
import javax.xml.transform.Source;
```

```
public class Handler2 implements LogicalHandler<LogicalMessageContext>
{
  public Set<QName> getHeaders()
  {
    return Collections.emptySet();
  }

  public boolean handleMessage(LogicalMessageContext messageContext)
  {
     Boolean outboundProperty = (Boolean)
         messageContext.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);
     if (outboundProperty.booleanValue()) {
            System.out.println("\nOutbound message:");
     } else {
            System.out.println("\nInbound message:");
     }

    System.out.println("** Response: "+messageContext.getMessage().toString());
    return true;
  }

  public boolean handleFault(LogicalMessageContext messageContext)
  {
    return true;
  }

  public void close(MessageContext messageContext)
  {
  }
}
```

Implementing the Handler.handleMessage() Method

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

```
public boolean handleMessage(C context)
  throws java.lang.RuntimeException, java.xml.ws.ProtocolException {}
```

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

`C` extends `javax.xml.ws.handler.MessageContext`. The `MessageContext` properties allow the handlers in a handler chain to determine if a message is inbound or outbound and to share processing state. Use the `SOAPMessageContext` or `LogicalMessageContext` sub-interface of `MessageContext` to get or set the contents of the SOAP or logical message, respectively. For more information, see "Using the Message Context Property Values and Methods" on page 8-12.

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

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

  The next handler on the request chain is specified as the next `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation.

- Block processing of the handler request chain by returning `false`.

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

  Although the handler request chain does not continue processing, WebLogic Server does invoke the handler *response* chain, starting at the current handler.

- Throw the `java.lang.RuntimeException` or `java.xml.ws.ProtocolException` for any handler-specific runtime errors.

  WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server log file, and invokes the `handleFault()` method of this handler.

Implementing the Handler.handleFault() Method

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

```
public boolean handleFault(C context)
  throws java.lang.RuntimeException, java.xml.ws.ProtocolException{}
```

Implement this method to handle processing of any SOAP faults generated by the `handleMessage()` method, as well as faults generated by the back-end component.

`C` extends `javax.xml.ws.handler.MessageContext`. The `MessageContext` properties allow the handlers in a handler chain to determine if a message is inbound or outbound and to share processing state.Use the `LogicalMessageContext` or `SOAPMessageContext` sub-interface of `MessageContext` to get or set the contents of the logical or SOAP message, respectively. For more information, see "Using the Message Context Property Values and Methods" on page 8-12.

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

- Invoke the `handleFault()` method on the next handler in the handler chain by returning `true`.
- Block processing of the handler fault chain by returning `false`.

Implementing the Handler.close() Method

The `Handler.close()` method concludes the message exchange and cleans up resources that were accessed during processing. Its signature is:

```
public boolean close(MessageContext context) {}
```

Using the Message Context Property Values and Methods

The following context objects are passed to the SOAP message handlers.

Table 8-5 Message Context Property Values

Message Context Property Values	Description
`javax.xml.ws.handler.LogicalMessageContext`	Context object for logical handlers.
`javax.xml.ws.handler.soap.SOAPMessageContext`	Context object for SOAP handlers.

Each context object extends `javax.xml.ws.handler.MessageContext` which enables you to access a set of runtime properties of a SOAP message handler from the client application or Web Service, or directly from the `javax.xml.ws.WebServiceContext` from a Web Service.

For example, the `MessageContext.MESSAGE_OUTBOUND_PROPERTY` holds a `Boolean` value that is used to determine the direction of a message. During a request, you can check the value of this property to determine if the message is an inbound or outbound request. The property would be `true` when accessed by a client-side handler or `false` when accessed by a server-side handler.

For more information about the `MessageContext` property values that are available, see "Using the MessageContext Property Values" in *Getting Started With WebLogic Web Services Using JAX-WS*.

The `LogicalMessageContext` class defines the following method for processing the Logical message. For more information, see the `javax.xml.ws.handler.LogicalMessageContext` Javadoc.

Table 8-6 LogicalMessageContext Class Method

Method	Description
`getMessage()`	Gets a `javax.xml.ws.LogicalMessage` object that contains the SOAP message.

The `SOAPMessageContext` class defines the following methods for processing the SOAP message. For more information, see the `javax.xml.ws.handler.soap.SOAPMessageContext` Javadoc.

Note: The SOAP message itself is stored in a `javax.xml.soap.SOAPMessage` object. For detailed information on this object, see "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 8-13.

Table 8-7 SOAPMessageContext Class Methods

Method	Description
`getHeaders()`	Gets headers that have a particular qualified name from the message in the message context.
`getMessage()`	Gets a `javax.xml.soap.SOAPMessage` object that contains the SOAP message.
`getRoles()`	Gets the SOAP actor roles associated with an execution of the handler chain.
`setMessage()`	Sets the SOAP message.

Directly Manipulating the SOAP Request and Response Message Using SAAJ

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

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

Refer to the SAAJ Javadocs for the full description of the `SOAPMessage` class.

The SOAPPart Object

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

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

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

The AttachmentPart Object

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

Note: If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see "Manipulating Image Attachments in a SOAP Message Handler" on page 8-15 for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments. For more information, see the `javax.xml.soap.SOAPMessage` Javadoc.

Table 8-8 SOAPMessage Class Methods to Manipulate Attachments

Method	Description
`addAttachmentPart()`	Adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.
`countAttachments()`	Returns the number of attachments in this SOAP message.
`createAttachmentPart()`	Create an `AttachmentPart` object from another type of `Object`.
`getAttachments()`	Gets all the attachments (as `AttachmentPart` objects) into an `Iterator` object.

Manipulating Image Attachments in a SOAP Message Handler

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

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

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

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

Configuring Handler Chains in the JWS File

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

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

```
package examples.webservices.handler;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.HandlerChain;
import javax.annotation.Resource;
import javax.xml.ws.WebServiceContext;
@WebService(name = "Handler", targetNamespace = "http://example.org")
@HandlerChain(file="handler-chain.xml")
public class HandlerWS
{
  @Resource
  WebServiceContext ctx;
```

```
  @WebMethod()
  public String getProperty(String propertyName)
  {
    return (String) ctx.getMessageContext().get(propertyName);
  }
}
```

Before you use the `@HandlerChain` annotation, you must import it into your JWS file, as shown above.

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

Note: It is an error to specify more than one `@HandlerChain` annotation in a single JWS file.

For details about creating the external configuration file, see "Creating the Handler Chain Configuration File" on page 8-16.

For additional detailed information about the standard JWS annotations discussed in this section, see the Web Services Metadata for the Java Platform specification.

Creating the Handler Chain Configuration File

As described in the previous section, you use the `@HandlerChain` annotation in your JWS file to associate a handler chain with a Web Service. You must create the handler chain file that consists of an external configuration file that specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on.

Because this file is external to the JWS file, you can configure multiple Web Services to use this single configuration file to standardize the handler configuration file for all Web Services in your enterprise. Additionally, you can change the configuration of the handler chains without needing to recompile all your Web Services.

The configuration file uses XML to list one or more handler chains, as shown in the following simple example:

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-class>examples.webservices.handler.Handler1</handler-class>
    </handler>
```

```
  </handler-chain>
  <handler-chain>
    <handler>
      <handler-class>examples.webservices.handler.Handler2</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

In the example, the handler chain contains two handlers implemented with the class names specified with the `<handler-class>` element. The two handlers execute in forward order before the relevant Web Service operation executes, and in reverse order after the operation executes.

Use the `<init-param>` and `<soap-role>`child elements of the `<handler>` element to specify the handler initialization parameters and SOAP roles implemented by the handler, respectively.

You can include logical and SOAP handlers in the same handler chain. At runtime, the handler chain is re-ordered so that all logical handlers are executed before SOAP handlers for an outbound message, and vice versa for an inbound message.

For the XML Schema that defines the external configuration file, additional information about creating it, and additional examples, see the Web Services Metadata for the Java Platform specification.

Compiling and Rebuilding the Web Service

It is assumed in this section that you have a working `build.xml` Ant file that compiles and builds your Web Service, and you want to update the build file to include handler chain. See "Developing WebLogic Web Services" in *Getting Started With WebLogic Web Services Using JAX-WS* for information on creating this `build.xml` file.

Follow these guidelines to update your development environment to include message handler compilation and building:

- After you have updated the JWS file with the `@HandlerChain` annotation, you must rerun the `jwsc` Ant task to recompile the JWS file and generate a new Web Service. This is true anytime you make a change to an annotation in the JWS file.

  If you used the `@HandlerChain` annotation in your JWS file, reran the `jwsc` Ant task to regenerate the Web Service, and subsequently changed only the external configuration file, you do not need to rerun `jwsc` for the second change to take affect.

- The `jwsc` Ant task compiles SOAP message handler Java files into handler classes (and then packages them into the generated application) if all the following conditions are true:
  - The handler classes are referenced in the `@HandlerChain` annotation of the JWS file.

– The Java files are located in the directory specified by the `sourcepath` attribute.

– The classes are not currently in your CLASSPATH.

If you want to compile the handler classes yourself, rather than let `jwsc` compile them automatically, ensure that the compiled classes are in your CLASSPATH before you run the `jwsc` Ant task.

- You deploy and invoke a Web Service that has a handler chain associated with it in the same way you deploy and invoke one that has no handler chain. The only difference is that when you invoke any operation of the Web Service, the WebLogic Web Services runtime executes the handlers in the handler chain both before and after the operation invoke.

Configuring the Client-side SOAP Message Handlers

You configure client-side SOAP message handlers in one of the following ways:

- Set a handler chain directly on the `javax.xml.ws.BindingProvider`, such as a port proxy or `javax.xml.ws.Dispatch` object. For example:

```
package examples.webservices.handler.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;

import javax.xml.ws.handler.Handler;
import javax.xml.ws.Binding;
import javax.xml.ws.BindingProvider;
import java.util.List;

import examples.webservices.handler.Handler1;
import examples.webservices.handler.Handler2;

public class Main {
  public static void main(String[] args) {
     HandlerWS test;
     try {
          test = new HandlerWS(new URL(args[0] + "?WSDL"), new
                  QName("http://example.org", "HandlerWS") );
      } catch (MalformedURLException murl) { throw new RuntimeException(murl); }
      HandlerWSPortType port = test.getHandlerWSPortTypePort();

      Binding binding = ((BindingProvider)port).getBinding();
      List<Handler> handlerList = binding.getHandlerChain();
      handlerList.add(new Handler1());
      handlerList.add(new Handler2());
      binding.setHandlerChain(handlerList);
```

```
      String result = null;
      result = port.sayHello("foo bar");
      System.out.println( "Got result: " + result );
  }
}
```

- Implement a `javax.xml.ws.handler.HandlerResolver` on a `Service` instance. For example:

```
public static class MyHandlerResolver implements HandlerResolver {
   public List<Handler> getHandlerChain(PortInfo portInfo) {
     List<Handler> handlers = new ArrayList<Handler>();
     // add handlers to list based on PortInfo information
     return handlers;
   }
}
```

  Add a handler resolver to the `Service` instance using the `setHandlerResolver()` method. In this case, the port proxy or `Dispatch` object created from the `Service` instance uses the `HandlerResolver` to determine the handler chain. For example:

```
test.setHandlerResolver(new MyHandlerResolver());
```

- Create a customization file that includes a `<binding>` element that contains a handler chain description. The schema for the `<handler-chains>` element is the same for both handler chain files (on the server) and customization files. For example:

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:7001/handler/HandlerWS?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions"
        xmlns:jws="http://java.sun.com/xml/ns/javaee">
    <handler-chains>
       <handler-chain>
          <handler>
             <handler-class>examples.webservices.handler.Handler1
             </handler-class>
          </handler>
       </handler-chain>
       <handler-chain>
          <handler>
             <handler-class>examples.webservices.handler.Handler2
             </handler-class>
          </handler>
       </handler-chain>
    </handler-chains>
</bindings>
```

Use the `<binding>` child element of the `clientgen` command to pass the customization file.

CHAPTER 9

Programming RESTful Web Services

The following sections describe how to use callbacks to notify clients of events:

Overview of RESTful Web Services

Representational State Transfer (REST) describes any simple interface that transmits data over a standardized interface (such as HTTP) without an additional messaging layer, such as SOAP. REST provides a set of design rules for creating stateless services that are viewed as *resources*, or sources of specific information, and can be identified by their unique URIs. A client accesses the resource using the URI and a *representation* of the resource is returned. The client is said to *transfer* state with each new resource representation.

You build RESTful endpoints using the `invoke()` method of the `javax.xml.ws.Provider<T>` interface. The `Provider` interface provides a dynamic alternative to building an service endpoint interface (SEI).

Programming RESTful Web Services: Main Steps

The procedure in this section describes how to program and compile the JWS file required to implement the RESTful Web SErvice. The procedure shows how to create the JWS file from scratch; if you want to update an existing JWS file, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web Services. For more information, see *Getting Started With WebLogic Web Services Using JAX-WS*.

Table 9-1 Steps to Program RESTful Web Services

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the RESTful Web Service.	Use your favorite IDE or text editor. See "Programming Guidelines for the RESTful Web Service" on page 9-2.

2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the RESTful JWS file into a Web Service.	For example:
		`<jwsc srcdir="." destdir="output/restEar">`
		`<jws file="NearbyCity.java" type="JAXWS"/>`
		`</jwsc>`
		For more information, see "Running the jwsc WebLogic Web Services Ant Task" in *Getting Started With the WebLogic Web Services Using JAX-WS*.

3	Run the Ant target to build the RESTful Web Service.	For example: `prompt> ant build-rest`
4	Deploy the RESTful Web Service as usual.	See "Deploying and Undeploying WebLogic Web Services" in *Programming WebLogic Web Services Using JAX-WS*.
5	Access the RESTful Web Service from your Web Service client.	See "Accessing the RESTful Web Service from a Client" on page 9-6.

Programming Guidelines for the RESTful Web Service

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

```
package examples.webservices.jaxws.rest;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.BindingType;
import javax.xml.ws.Provider;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.ws.http.HTTPException;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import java.io.ByteArrayInputStream;
import java.util.StringTokenizer;

@WebServiceProvider(
   targetNamespace="http://example.org",
   serviceName = "NearbyCityService")
@BindingType(value = HTTPBinding.HTTP_BINDING)

public class NearbyCity implements Provider<Source> {
  @Resource(type=Object.class)
  protected WebServiceContext wsContext;

  public Source invoke(Source source)  {
    try {
        MessageContext messageContext = wsContext.getMessageContext();
        String query =
             (String)messageContext.get(MessageContext.QUERY_STRING);
        if (query != null && query.contains("lat=") &&
            query.contains("long=")) {
            return createSource(query);
        } else {

            System.err.println("Query String = "+query);
            throw new HTTPException(404);
        }
    } catch(Exception e) {
        e.printStackTrace();
        throw new HTTPException(500);
```

```
  }
}
  private Source createSource(String str) throws Exception {
      StringTokenizer st = new StringTokenizer(str, "=&/");
      String latLong = st.nextToken();
      double latitude = Double.parseDouble(st.nextToken());
      latLong = st.nextToken();
      double longitude = Double.parseDouble(st.nextToken());
      City nearby = City.findNearBy(latitude, longitude);
      String body = nearby.toXML();
     return new StreamSource(new ByteArrayInputStream(body.getBytes()));
  }

  static class City {
       String city;
       String state;
       double latitude;
       double longitude;

       City(String city, double lati, double longi, String st) {
           this.city = city;
           this.state = st;
           this.latitude = lati;
           this.longitude = longi;
       }

       double distance(double lati, double longi) {
          return Math.sqrt((lati-this.latitude)*(lati-this.latitude) +
              (longi-this.longitude)*(longi-this.longitude)) ;
       }

       static final City[] cities = {
          new City("San Francisco",37.7749295,-122.4194155,"CA"),
          new City("Columbus",39.9611755,-82.9987942,"OH"),
          new City("Indianapolis",39.7683765,-86.1580423,"IN"),
          new City("Jacksonville",30.3321838,-81.655651,"FL"),
          new City("San Jose",37.3393857,-121.8949555,"CA"),
          new City("Detroit",42.331427,-83.0457538,"MI"),
          new City("Dallas",32.7830556,-96.8066667,"TX"),
```

```
            new City("San Diego",32.7153292,-117.1572551,"CA"),
            new City("San Antonio",29.4241219,-98.4936282,"TX"),
            new City("Phoenix",33.4483771,-112.0740373,"AZ"),
            new City("Philadelphia",39.952335,-75.163789,"PA"),
            new City("Houston",29.7632836,-95.3632715,"TX"),
            new City("Chicago",41.850033,-87.6500523,"IL"),
            new City("Los Angeles",34.0522342,-118.2436849,"CA"),
            new City("New York",40.7142691,-74.0059729,"NY")};
        static City findNearBy(double lati, double longi) {
            int n = 0;
            for (int i = 1; i < cities.length; i++)  {
                if (cities[i].distance(lati, longi) <
                     cities[n].distance(lati, longi)) {
                    n = i;
                }
            }
            return cities[n];
        }

        public String toXML() {
            return "<ns:NearbyCity xmlns:ns=\"http://example.org\"><City>"
                  +this.city+"</City><State>"+ this.state+"</State><Lat>"
                  +this.latitude +
                  "</Lat><Lng>"+this.longitude+"</Lng></ns:NearbyCity>";
        }
    }
}
```

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

- Import the packages required to implement the RESTful Web Service.

  ```
  import javax.xml.ws.WebServiceProvider;
  import javax.xml.ws.BindingType;
  import javax.xml.ws.Provider;
  ```

- Annotate the `Provider` implementation class and set the binding type to HTTP.

  ```
  @WebServiceProvider(
     targetNamespace="http://example.org",
  ```

```
   serviceName = "NearbyCityService")
@BindingType(value = HTTPBinding.HTTP_BINDING)
```

- Implement the `invoke()` method of the `Provider` interface.

```
public class NearbyCity implements Provider<Source> {
  @Resource(type=Object.class)
  protected WebServiceContext wsContext;

  public Source invoke(Source source)  {
  ...
  }
```

- Get the request string using the `QUERY_STRING` field in the `javax.xml.ws.handler.MessageContext` for processing. The query string is then passed to the `createSource()` method that returns the city, state, longitude, and latitude that is closest to the specified values.

```
String query =
             (String)messageContext.get(MessageContext.QUERY_STRING);
.
.
.
return createSource(query);
```

Accessing the RESTful Web Service from a Client

To access a RESTful Web Service from a Web Service client, use the resource URI. For example:

```
http://localhost:7001/NearbyCity/NearbyCityService?lat=35&long=-120
```

In this example, you set the latitude (`lat`) and longitude (`long`) values, as required, to access the required resource.

CHAPTER 10

Publishing and Finding Web Services Using UDDI

The following sections provide information about publishing and finding Web Services through the UDDI registry:

Overview of UDDI

UDDI stands for Universal Description, Discovery, and Integration. The UDDI Project is an industry initiative aims to enable businesses to quickly, easily, and dynamically find and carry out transactions with one another.

A populated UDDI registry contains cataloged information about businesses; the services that they offer; and communication standards and interfaces they use to conduct transactions.

Built on the Simple Object Access Protocol (SOAP) data communication standard, UDDI creates a global, platform-independent, open architecture space that will benefit businesses.

The UDDI registry can be broadly divided into two categories:

- UDDI and Web Services
- UDDI and Business Registry

For details about the UDDI data structure, see "UDDI Data Structure" on page 10-3.

UDDI and Web Services

The owners of Web Services publish them to the UDDI registry. Once published, the UDDI registry maintains pointers to the Web Service description and to the service.

The UDDI allows clients to search this registry, find the intended service, and retrieve its details. These details include the service invocation point as well as other information to help identify the service and its functionality.

Web Service capabilities are exposed through a programming interface, and usually explained through Web Services Description Language (WSDL). In a typical publish-and-inquire scenario, the provider publishes its business; registers a service under it; and defines a binding template with technical information on its Web Service. The binding template also holds reference to one or several *tModels*, which represent abstract interfaces implemented by the Web Service. The tModels might have been uniquely published by the provider, with information on the interfaces and URL references to the WSDL document.

A typical client inquiry may have one of two objectives:

- To find an implementation of a known interface. In other words, the client has a tModel ID and seeks binding templates referencing that tModel.
- To find the updated value of the invocation point (that is., access point) of a known binding template ID.

UDDI and Business Registry

As a Business Registry solution, UDDI enables companies to advertise the business products and services they provide, as well as how they conduct business transactions on the Web. This use of UDDI complements business-to-business (B2B) electronic commerce.

The minimum required information to publish a business is a single business name. Once completed, a full description of a business entity may contain a wealth of information, all of which helps to advertise the business entity and its products and services in a precise and accessible manner.

A Business Registry can contain:

- **Business Identification**—Multiple names and descriptions of the business, comprehensive contact information, and standard business identifiers such as a tax identifier.
- **Categories**—Standard categorization information (for example a D-U-N-S business category number).
- **Service Description**—Multiple names and descriptions of a service. As a container for service information, companies can advertise numerous services, while clearly displaying the ownership of services. The `bindingTemplate` information describes how to access the service.
- **Standards Compliance**—In some cases it is important to specify compliance with standards. These standards might display detailed technical requirements on how to use the service.
- **Custom Categories**—It is possible to publish proprietary specifications (tModels) that identify or categorize businesses or services.

UDDI Data Structure

The data structure within UDDI consists of four constructions: a `businessEntity` structure, a `businessService` structure, a `bindingTemplate` structure and a `tModel` structure.

The following table outlines the difference between these constructions when used for Web Service or Business Registry applications.

Table 10-1 UDDI Data Structure

Data Structure	Web Service	Business Registry
businessEntity	Represents a Web Service provider: • Company name • Contact detail • Other business information	Represents a company, a division or a department within a company: • Company name(s) • Contact details • Identifiers and Categories
businessService	A logical group of one or several Web Services. API(s) with a single name stored as a child element, contained by the business entity named above.	A group of services may reside in a single `businessEntity`. • Multiple names and descriptions • Categories • Indicators of compliancy with standards

Table 10-1 UDDI Data Structure

Data Structure	Web Service	Business Registry
bindingTemplate	A single Web Service. Technical information needed by client applications to bind and interact with the target Web Service. Contains access point (that is, the URI to invoke a Web Service).	Further instances of standards conformity. Access points for the service in form of URLs, phone numbers, email addresses, fax numbers or other similar address types.
tModel	Represents a technical specification; typically a specifications pointer, or metadata about a specification document, including a name and a URL pointing to the actual specifications. In the context of Web Services, the actual specifications document is presented in the form of a WSDL file.	Represents a standard or technical specification, either well established or registered by a user for specific use.

WebLogic Server UDDI Features

WebLogic Server provides the following UDDI features:

- UDDI 2.0 Server
- UDDI Directory Explorer
- UDDI Client API
- Pluggable tModel

UDDI 2.0 Server

The UDDI 2.0 Server is part of WebLogic Server and is started automatically when WebLogic Server is started. The UDDI Server implements the UDDI 2.0 server specification.

Configuring the UDDI 2.0 Server

To configure the UDDI 2.0 Server:

1. Stop WebLogic Server.

2. Update the `uddi.properties` file, located in the *`WL_HOME`*`/server/lib` directory, where *`WL_HOME`* refers to the main WebLogic Server installation directory.

   Note: If your WebLogic Server domain was created by a user different from the user that installed WebLogic Server, the WebLogic Server administrator must change the permissions on the `uddi.properties` file to give access to all users.

3. Restart WebLogic Server.

Never edit the `uddi.properties` file while WebLogic Server is running. Should you modify this file in a way that prevents the successful startup of the UDDI Server, refer to the *`WL_HOME`*`/server/lib/uddi.properties.booted` file for the last known good configuration.

To restore your configuration to its default, remove the `uddi.properties` file from the *`WL_HOME`*`/server/lib` directory. Oracle strongly recommends that you move this file to a backup location, because a new `uddi.properties` file will be created and with its successful startup, the `uddi.properties.booted` file will also be overwritten. After removing the properties file, start the server. Minimal default properties will be loaded and written to a newly created `uddi.properties` file.

The following section describes the UDDI Server properties that you can include in the `uddi.properites` file. The list of properties has been divided according to component, usage, and functionality. At any given time, you do not need all these properties to be present.

Configuring an External LDAP Server

The UDDI 2.0 Server is automatically configured with an embedded LDAP server. You can, however, also configure an external LDAP Server by following the procedure in this section.

Note: Currently, WebLogic Server supports only the SunOne Directory Server for use with the UDDI 2.0 Server.

To configure the SunOne Directory Server to be used with UDDI, follow these steps:

1. Create a file called `51acumen.ldif` in the *`LDAP_DIR`*`/Sun/MPS/slapd-`*`LDAP_INSTANCE_NAME`*`/config/schema` directory, where *`LDAP_DIR`* refers to the root installation directory of your SunOne Directory Server and *`LDAP_INSTANCE_NAME`* refers to the instance name.

2. Update the `51acumen.ldif` file with the content described in "51acumen.ldif File Contents" on page 10-6.

3. Restart the SunOne Directory Server.

4. Update the `uddi.properties` file of the WebLogic UDDI 2.0 Server, adding the following properties:

   ```
   datasource.ldap.manager.password
   datasource.ldap.manager.uid
   datasource.ldap.server.root
   datasource.ldap.server.url
   ```

   The value of the properties depends on the configuration of your SunOne Directory Server. The following example shows a possible configuration that uses default values:

   ```
   datasource.ldap.manager.password=password
   datasource.ldap.manager.uid=cn=Directory Manager
   datasource.ldap.server.root=dc=beasys,dc=com
   datasource.ldap.server.url=ldap://host:port
   ```

   See Table 10-11 for information about these properties.

5. Restart WebLogic Server.

51acumen.ldif File Contents

Use the following content to create the `51acumen.ldif` file:

```
dn: cn=schema
#
# attribute types:
#
attributeTypes: ( 11827.0001.1.0 NAME 'uddi-Business-Key'                DESC
'Business Key' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.1 NAME 'uddi-Authorized-Name'             DESC
'Authorized Name for publisher of data' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.2 NAME 'uddi-Operator'                    DESC
'Name of UDDI Registry Operator' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255}
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.3 NAME 'uddi-Name'                        DESC
'Business Entity Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{258} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.4 NAME 'uddi-Description'                 DESC
'Description of Business Entity' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255}
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.7 NAME 'uddi-Use-Type'                    DESC
'Name of convention that the referenced document follows' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.8 NAME 'uddi-URL'                         DESC
'URL' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.9 NAME 'uddi-Person-Name'                 DESC
```

```
'Name of Contact Person' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.10 NAME 'uddi-Phone'                      DESC
'Telephone Number' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{50} X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.11 NAME 'uddi-Email'                      DESC
'Email address' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.12 NAME 'uddi-Sort-Code'                  DESC
'Code to sort addresses' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{10} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.13 NAME 'uddi-tModel-Key'                 DESC
'Key to reference a tModel entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255}
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.14 NAME 'uddi-Address-Line'               DESC
'Actual address lines in free form text' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{80} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.15 NAME 'uddi-Service-Key'                DESC
'Service Key' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.16 NAME 'uddi-Service-Name'               DESC
'Service Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.17 NAME 'uddi-Binding-Key'                DESC
'Binding Key' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.18 NAME 'uddi-Access-Point'             DESC 'A
text field to convey the entry point address for calling a web service' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.19 NAME 'uddi-Hosting-Redirector'         DESC
'Provides a Binding Key attribute to redirect reference to a different binding
template' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.20 NAME 'uddi-Instance-Parms'             DESC
'Parameters to use a specific facet of a bindingTemplate description' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.21 NAME 'uddi-Overview-URL'               DESC
'URL reference to a long form of an overview document' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.22 NAME 'uddi-From-Key'                   DESC
'Unique key reference to first businessEntity assertion is made for' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.23 NAME 'uddi-To-Key'                     DESC
'Unique key reference to second businessEntity assertion is made for' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{41} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.24 NAME 'uddi-Key-Name'                   DESC
'An attribute of the KeyedReference structure' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.25 NAME 'uddi-Key-Value'                  DESC
```

```
'An attribute of the KeyedReference structure' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.26 NAME 'uddi-Auth-Info'                    DESC
'Authorization information' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{4096} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.27 NAME 'uddi-Key-Type'                     DESC
'The key for all UDDI entries' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.28 NAME 'uddi-Upload-Register'              DESC
'The upload register' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.29 NAME 'uddi-URL-Type'                     DESC
'The type for the URL' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{16} X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.30 NAME 'uddi-Ref-Keyed-Reference'          DESC
'reference to a keyedReference entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12{255}
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.31 NAME 'uddi-Ref-Category-Bag'             DESC
'reference to a categoryBag entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12{255}
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.32 NAME 'uddi-Ref-Identifier-Bag'           DESC
'reference to a identifierBag entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12{255}
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.33 NAME 'uddi-Ref-TModel'                   DESC
'reference to a TModel entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12{255}
SINGLE-VALUE X-ORIGIN 'acumen defined' )
# id names for each entry
attributeTypes: ( 11827.0001.1.34 NAME 'uddi-Contact-ID'                   DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.35 NAME 'uddi-Discovery-URL-ID'             DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.36 NAME 'uddi-Address-ID'                   DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.37 NAME 'uddi-Overview-Doc-ID'              DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.38 NAME 'uddi-Instance-Details-ID'          DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.39 NAME 'uddi-tModel-Instance-Info-ID'      DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.40 NAME 'uddi-Publisher-Assertions-ID'      DESC
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.41 NAME 'uddi-Keyed-Reference-ID'           DESC
```

```
'Unique ID which will serve as the Distinguished Name of each entry' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{16} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.42 NAME 'uddi-Ref-Attribute'             DESC 'a
reference to another entry' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12{255} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.43 NAME 'uddi-Entity-Name'                DESC
'Business entity Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{258} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.44 NAME 'uddi-tModel-Name'                DESC
'tModel Name' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255} X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.45 NAME 'uddi-tMII-TModel-Key'            DESC
'tModel key referneced in tModelInstanceInfo' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.46 NAME 'uddi-Keyed-Reference-TModel-Key' DESC
'tModel key referneced in KeyedReference' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.47 NAME 'uddi-Address-tModel-Key'         DESC
'tModel key referneced in Address' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255}
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.48 NAME 'uddi-isHidden'                  DESC 'a
flag to indicate whether an entry is hidden' SYNTAX
1.3.6.1.4.1.1466.115.121.1.15{255} SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.49 NAME 'uddi-Time-Stamp'                 DESC
'modification time satmp' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{255}
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.50 NAME 'uddi-next-id'                    DESC
'generic counter' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.51 NAME 'uddi-tModel-origin'              DESC
'tModel origin' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.52 NAME 'uddi-tModel-type'                DESC
'tModel type' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN 'acumen
defined' )
attributeTypes: ( 11827.0001.1.53 NAME 'uddi-tModel-checked'             DESC
'tModel field to check or not' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.54 NAME 'uddi-user-quota-entity'          DESC
'quota for business entity' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE
X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.55 NAME 'uddi-user-quota-service'         DESC
'quota for business services per entity' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.56 NAME 'uddi-user-quota-binding'         DESC
'quota for binding templates per service' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.57 NAME 'uddi-user-quota-tmodel'          DESC
'quota for tmodels' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 SINGLE-VALUE X-ORIGIN
```

```
'acumen defined' )
attributeTypes: ( 11827.0001.1.58 NAME 'uddi-user-quota-assertion'       DESC
'quota for publisher assertions' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.59 NAME 'uddi-user-quota-messagesize'     DESC
'quota for maximum message size' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
SINGLE-VALUE X-ORIGIN 'acumen defined' )
attributeTypes: ( 11827.0001.1.60 NAME 'uddi-user-language'              DESC
'user language' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.61 NAME 'uddi-Name-Soundex'               DESC
'name in soundex format' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{258} X-ORIGIN
'acumen defined' )
attributeTypes: ( 11827.0001.1.62 NAME 'uddi-var'                        DESC
'generic variable' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'acumen
defined' )
#
# objectclasses:
#
objectClasses: ( 11827.0001.2.0 NAME 'uddi-Business-Entity'              DESC
'Business Entity object' SUP top STRUCTURAL MUST (uddi-Business-Key $
uddi-Entity-Name $ uddi-isHidden $ uddi-Authorized-Name ) MAY (
uddi-Name-Soundex $ uddi-Operator $ uddi-Description $ uddi-Ref-Identifier-Bag
$ uddi-Ref-Category-Bag ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.1 NAME 'uddi-Business-Service'             DESC
'Business Service object' SUP top STRUCTURAL MUST ( uddi-Service-Key $
uddi-Service-Name $ uddi-isHidden ) MAY ( uddi-Name-Soundex $ uddi-Description
$ uddi-Ref-Category-Bag ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.2 NAME 'uddi-Binding-Template'             DESC
'Binding Template object' SUP TOP STRUCTURAL   MUST ( uddi-Binding-Key $
uddi-isHidden ) MAY ( uddi-Description $ uddi-Access-Point $
uddi-Hosting-Redirector ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.3 NAME 'uddi-tModel'                       DESC
'tModel object' SUP top STRUCTURAL   MUST (uddi-tModel-Key $ uddi-tModel-Name $
uddi-isHidden $ uddi-Authorized-Name ) MAY ( uddi-Name-Soundex $ uddi-Operator
$ uddi-Description $ uddi-Ref-Identifier-Bag $ uddi-Ref-Category-Bag $
uddi-tModel-origin $ uddi-tModel-checked $ uddi-tModel-type ) X-ORIGIN 'acumen
defined' )
objectClasses: ( 11827.0001.2.4 NAME 'uddi-Publisher-Assertion'          DESC
'Publisher Assertion object' SUP TOP STRUCTURAL   MUST (
uddi-Publisher-Assertions-ID $ uddi-From-Key $ uddi-To-Key $
uddi-Ref-Keyed-Reference ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.5 NAME 'uddi-Discovery-URL'                DESC
'Discovery URL' SUP TOP STRUCTURAL  MUST ( uddi-Discovery-URL-ID $ uddi-Use-Type
$ uddi-URL ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.6 NAME 'uddi-Contact'                      DESC
'Contact Information' SUP TOP STRUCTURAL   MUST ( uddi-Contact-ID $
uddi-Person-Name ) MAY ( uddi-Use-Type $ uddi-Description $ uddi-Phone $
uddi-Email $ uddi-tModel-Key ) X-ORIGIN 'acumen defined' )
```

```
objectClasses: ( 11827.0001.2.7 NAME 'uddi-Address'                        DESC
'Address information for a contact entry' SUP TOP STRUCTURAL   MUST (
uddi-Address-ID ) MAY ( uddi-Use-Type $ uddi-Sort-Code $ uddi-Address-tModel-Key
$ uddi-Address-Line ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.8 NAME 'uddi-Keyed-Reference'                DESC
'KeyedReference' SUP TOP STRUCTURAL   MUST ( uddi-Keyed-Reference-ID $
uddi-Key-Value ) MAY ( uddi-Key-Name $ uddi-Keyed-Reference-TModel-Key )
X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.9 NAME 'uddi-tModel-Instance-Info'           DESC
'tModelInstanceInfo' SUP TOP STRUCTURAL   MUST ( uddi-tModel-Instance-Info-ID $
uddi-tMII-TModel-Key ) MAY ( uddi-Description ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.10 NAME 'uddi-Instance-Details'              DESC
'instanceDetails' SUP TOP STRUCTURAL   MUST ( uddi-Instance-Details-ID ) MAY (
uddi-Description $ uddi-Instance-Parms ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.11 NAME 'uddi-Overview-Doc'                  DESC
'overviewDoc' SUP TOP STRUCTURAL   MUST ( uddi-Overview-Doc-ID ) MAY (
uddi-Description $ uddi-Overview-URL ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.12 NAME 'uddi-Ref-Object'                    DESC
'an object class conatins a reference to another entry' SUP TOP STRUCTURAL MUST
( uddi-Ref-Attribute ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.13 NAME 'uddi-Ref-Auxiliary-Object'          DESC
'an auxiliary type object used in another structural class to hold a reference
to a third entry' SUP TOP AUXILIARY MUST ( uddi-Ref-Attribute ) X-ORIGIN 'acumen
defined' )
objectClasses: ( 11827.0001.2.14 NAME 'uddi-ou-container'                  DESC
'an organizational unit with uddi attributes' SUP organizationalunit STRUCTURAL
MAY ( uddi-next-id $ uddi-var ) X-ORIGIN 'acumen defined' )
objectClasses: ( 11827.0001.2.15 NAME 'uddi-User'                        DESC 'a
User with uddi attributes' SUP inetOrgPerson STRUCTURAL MUST ( uid $
uddi-user-language $ uddi-user-quota-entity $ uddi-user-quota-service $
uddi-user-quota-tmodel $ uddi-user-quota-binding $ uddi-user-quota-assertion $
uddi-user-quota-messagesize ) X-ORIGIN 'acumen defined' )
```

Description of Properties in the uddi.properties File

The following tables describe properties of the `uddi.properties` file, categorized by the type of UDDI feature they describe:

- Basic UDDI Configuration
- UDDI User Defaults
- General Server Configuration
- Logger Configuration
- Connection Pools

- LDAP Datastore Configuration
- Replicated LDAP Datastore Configuration
- File Datastore Configuration
- General Security Configuration
- LDAP Security Configuration
- File Security Configuration

Table 10-2 Basic UDDI Configuration

UDDI Property Key	Description
`auddi.discoveryurl`	DiscoveryURL prefix that is set for each saved business entity. Typically this is the full URL to the uddilistener servlet, so that the full DiscoveryURL results in the display of the stored BusinessEntity data.
`auddi.inquiry.secure`	Permissible values are `true` and `false`. When set to `true`, inquiry calls to UDDI Server are limited to secure https connections only. Any UDDI inquiry calls through a regular http URL are rejected.
`auddi.publish.secure`	Permissible values are `true` and `false`. When set to `true`, publish calls to UDDI Server are limited to secure https connections only. Any UDDI publish calls through a regular http URL are rejected.
`auddi.search.maxrows`	Maximum number of returned rows for search operations. When the search results in a higher number of rows then the limit set by this property, the result is truncated.
`auddi.search.timeout`	Timeout value for search operations. The value is indicated in milliseconds.
`auddi.siteoperator`	Name of the UDDI registry site operator. The specified value will be used as the operator attribute, saved in all future BusinessEntity registrations. This attribute will later be returned in responses, and indicates which UDDI registry has generated the response.

Table 10-2 Basic UDDI Configuration

UDDI Property Key	Description
`security.cred.life`	Credential life, specified in seconds, for authentication. Upon authentication of a user, an AuthToken is assigned which will be valid for the duration specified by this property.
`pluggableTModel.file.list`	UDDI Server is pre-populated with a set of Standard TModels. You can further customize the UDDI server by providing your own taxonomies, in the form of TModels. Taxonomies must be defined in XML files, following the provided XML schema. The value of this property a comma-separated list of URIs to such XML files. Values that refer to these TModels are checked and validated against the specified taxonomy.

Table 10-3 UDDI User Defaults

UDDI Property Key	Description
`auddi.default.lang`	User's initial language, assigned to user profile by default at the time of creation. User profile settings can be changed at sign-up or later.
`auddi.default.quota.assertion`	User's initial assertion quota, assigned to user profile by default at the time of creation. The assertion quota is the maximum number of publisher assertions that the user is allowed to publish. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.
`auddi.default.quota.binding`	User's initial binding quota, assigned to user profile by default at the time of creation. The binding quota is the maximum number of binding templates that the user is allowed to publish, per each business service. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.
`auddi.default.quota.entity`	User's initial business entity quota, assigned to user profile by default at the time of creation. The entity quota is the maximum number of business entities that the user is allowed to publish. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.

Table 10-3 UDDI User Defaults

UDDI Property Key	Description
`auddi.default.quota.messageSize`	User's initial message size limit, assigned to his user profile by default at the time of creation. The message size limit is the maximum size of a SOAP call that the user may send to UDDI Server. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.
`auddi.default.quota.service`	User's initial service quota, assigned to user profile by default at the time of creation. The service quota is the maximum number of business services that the user is allowed to publish, per each business entity. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.
`auddi.default.quota.tmodel`	User's initial TModel quota, assigned to user profile by default at the time of creation. The TModel quota is the maximum number of TModels that the user is allowed to publish. To impose no limits, set a value of -1. A user's profile settings can be changed at sign-up or later.

Table 10-4 General Server Configuration

UDDI Property Keys	Description
`auddi.datasource.type`	Location of physical storage of UDDI data. This value defaults to `WLS`, which indicates that the internal LDAP directory of WebLogic Server is to be used for data storage. Other permissible values include `LDAP`, `ReplicaLDAP`, and `File`.
`auddi.security.type`	UDDI Server's security module (authentication). This value defaults to `WLS`, which indicates that the default security realm of WebLogic Server is to be used for UDDI authentication. As such, a WebLogic Server user would be an UDDI Server user and any WebLogic Server administrator would also be an UDDI Server administrator, in addition to members of the UDDI Server administrator group, as defined in UDDI Server settings. Other permissible values include `LDAP` and `File`.

Table 10-5 Logger Configuration

UDDI Property Key	Description
`logger.file.maxsize`	Maximum size of logger output files (if output is sent to file), in Kilobytes. Once an output file reaches maximum size, it is closed and a new log file is created.
`logger.indent.enabled`	Permissible values are `true` and `false`. When set to `true`, log messages beginning with "+" and "-", typically TRACE level logs, cause an increase or decrease of indentation in the output.
`logger.indent.size`	Size of each indentation (how many spaces for each indent), specified as an integer.
`logger.log.dir`	Absolute or relative path to a directory where log files are stored.
`logger.log.file.stem`	String that is prefixed to all log file names.
`logger.log.type`	Determines whether log messages are sent to the screen, to a file or to both destinations. Permissible values, respectively, are: `LOG_TYPE_SCREEN`, `LOG_TYPE_FILE`, and `LOG_TYPE_SCREEN_FILE`.
`logger.output.style`	Determines whether logged output will simply contain the message, or thread and timestamp information will be included. Permissible values are `OUTPUT_LONG` and `OUTPUT_SHORT`.
`logger.quiet`	Determines whether the logger itself displays information messages. Permissible values are `true` and `false`.
`logger.verbosity`	Logger's verbosity level. Permissible values (case sensitive) are `TRACE`, `DEBUG`, `INFO`, `WARNING` and `ERROR`, where each severity level includes the following ones accumulatively.

Table 10-6 Connection Pools

UDDI Property Key	Description
`datasource.ldap.pool.increment`	Number of new connections to create and add to the pool when all connections in the pool are busy
`datasource.ldap.pool.initialsize`	Number of connections to be stored at the time of creation and initialization of the pool.
`datasource.ldap.pool.maxsize`	Maximum number of connections that the pool may hold.
`datasource.ldap.pool.systemmaxsize`	Maximum number of connections created, even after the pool has reached its capacity. Once the pool reaches its maximum size, and all connections are busy, connections are temporarily created and returned to the client, but not stored in the pool. However, once the system max size is reached, all requests for new connections are blocked until a previously busy connection becomes available.

Table 10-7 LDAP Datastore Configuration

UDDI Property Key	Description
`datasource.ldap.manager.uid`	Back-end LDAP server administrator or privileged user ID, (for example, cn=Directory Manager) who can save data in LDAP.
`datasource.ldap.manager.password`	Password for the datasource.ldap.manager.uid, establishes connections with the LDAP directory used for data storage.
`datasource.ldap.server.url`	"ldap://" URL to the LDAP directory used for data storage.
`datasource.ldap.server.root`	Root entry of the LDAP directory used for data storage (e.g., dc=acumenat, dc=com).

Note: In a replicated LDAP environment, there are "m" LDAP masters and "n" LDAP replicas, respectively numbered from 0 to (m-1) and from 0 to (n-1). The fifth part of the property keys below, quoted as "i", refers to this number and differs for each LDAP server instance defined.

Table 10-8 Replicated LDAP Datastore Configuration

UDDI Property Key	Description
`datasource.ldap.server.master.i.manager.uid`	Administrator or privileged user ID for this "master" LDAP server node, (for example, cn=Directory Manager) who can save data in LDAP.
`datasource.ldap.server.master.i.manager.password`	Password for the datasource.ldap.server.master.i.manager.uid, establishes connections with the relevant "master" LDAP directory to write data.
`datasource.ldap.server.master.i.url`	"ldap://" URL to the corresponding LDAP directory node.
`datasource.ldap.server.master.i.root`	Root entry of the corresponding LDAP directory node (for example, dc=acumenat, dc=com).
`datasource.ldap.server.replica.i.manager.uid`	User ID for this "replica" LDAP server node (for example, cn=Directory Manager); this person can read the UDDI data from LDAP.
`datasource.ldap.server.replica.i.manager.password`	Password for datasource.ldap.server.replica.i.manager.uid, establishes connections with the relevant "replica" LDAP directory to read data.
`datasource.ldap.server.replica.i.url`	"ldap://" URL to the corresponding LDAP directory node.
`datasource.ldap.server.replica.i.root`	Root entry of the corresponding LDAP directory node (for example, dc=acumenat, dc=com).

Table 10-9 File Datastore Configuration

UDDI Property Key	Description
`datasource.file.directory`	Directory where UDDI data is stored in the file system.

Table 10-10 General Security Configuration

UDDI Property Key	Description
`security.custom.group.operators`	Security group name, where the members of this group are treated as UDDI administrators.

Table 10-11 LDAP Security Configuration

UDDI Property Key	Description
`security.custom.ldap.manager.uid`	Security LDAP server administrator or privileged user ID (for example, cn=Directory Manager); this person can save data in LDAP.
`security.custom.ldap.manager.password`	The value of this property is the password for the above user ID, and is used to establish connections with the LDAP directory used for security.
`security.custom.ldap.url`	The value of this property is an "ldap://" URL to the LDAP directory used for security.
`security.custom.ldap.root`	Root entry of the LDAP directory used for security (for example, dc=acumenat, dc=com).
`security.custom.ldap.userroot`	User's root entry on the security LDAP server. For example, ou=People.
`security.custom.ldap.group.root`	Operator entry on the security LDAP server. For example, "cn=UDDI Administrators, ou=Groups". This entry contains IDs of all UDDI administrators.

Table 10-12 File Security Configuration

UDDI Property Key	Description
`security.custom.file.userdir`	Directory where UDDI security information (users and groups) is stored in the file system.

UDDI Directory Explorer

The UDDI Directory Explorer allows authorized users to publish Web Services in private WebLogic Server UDDI registries and to modify information for previously published Web Services. The Directory Explorer provides access to details about the Web Services and associated WSDL files (if available.)

The UDDI Directory Explorer also enables you to search both public and private UDDI registries for Web Services and information about the companies and departments that provide these Web Services.

To invoke the UDDI Directory Explorer in your browser, enter:

`http://host:port/uddiexplorer`

where

- *host* is the computer on which WebLogic Server is running.
- *port* is the port number where WebLogic Server listens for connection requests. The default port number is 7001.

You can perform the following tasks with the UDDI Directory Explorer:

- Search public registries
- Search private registries
- Publish to a private registry
- Modify private registry details
- Setup UDDI directory explorer

For more information about using the UDDI Directory Explorer, click the **Explorer Help** link on the main page.

UDDI Client API

WebLogic Server includes an implementation of the client-side UDDI API that you can use in your Java client applications to programmatically search for and publish Web Services.

The two main classes of the UDDI client API are `Inquiry` and `Publish`. Use the `Inquiry` class to search for Web Services in a known UDDI registry and the `Publish` class to add your Web Service to a known registry.

WebLogic Server provides an implementation of the following client UDDI API packages:

- `weblogic.uddi.client.service`
- `weblogic.uddi.client.structures.datatypes`
- `weblogic.uddi.client.structures.exception`
- `weblogic.uddi.client.structures.request`
- `weblogic.uddi.client.structures.response`

For detailed information on using these packages, see the UDDI API Javadocs.

Pluggable tModel

A taxonomy is basically a tModel used as reference by a categoryBag or identifierBag. A major distinction is that in contrast to a simple tModel, references to a taxonomy are typically checked and validated. WebLogic Server's UDDI Server takes advantage of this concept and extends this capability by introducing custom taxonomies, called "pluggable tModels". Pluggable tModels allow users (UDDI administrators) to add their own checked taxonomies to the UDDI registry, or overwrite standard taxonomies.

To add a pluggable tModel:

1. Create an XML file conforming to the specified format described in "XML Schema for Pluggable tModels" on page 10-22, for each tModelKey/categorization.
2. Add the comma-delimited, fully qualified file names to the `pluggableTModel.file.list` property in the `uddi.properties` file used to configure UDDI Server. For example:

   ```
   pluggableTModel.file.list=c:/temp/cat1.xml,c:/temp/cat2.xml
   ```

   See "Configuring the UDDI 2.0 Server" on page 10-4 for details about the uddi.properties file.
3. Restart WebLogic Server.

The following sections include a table detailing the XML elements and their permissible values, the XML schema against which pluggable tModels are validated, and a sample XML.

XML Elements and Permissible Values

The following table describes the elements of the XML file that describes your pluggable tModels.

Table 10-13 Description of the XML Elements to Configure Pluggable tModels

Element/Attribute	Required	Role	Values	Comments
Taxonomy	Required	Root Element		
checked	Required	Whether this categorization is checked or not.	true / false	If false, keyValue will not be validated.
type	Required	The type of the tModel.	categorization / identifier / valid values as defined in uddi-org-types	See uddi-org-types tModel for valid values.
applicability	Optional	Constraints on where the tModel may be used.		No constraint is assumed if this element is not provided
scope	Required if the applicability element is included.		businessEntity / businessService / bindingTemplate / tModel	tModel may be used in tModelInstanceInfo if scope "bindingTemplate" is specified.
tModel	Required	The actual tModel, according to the UDDI data structure.	ValidtModelKey must be provided.	
categories	Required if checked is set to true.			
category	Required if element categories is included	Holds actual keyName and keyValue pairs.	keyName / keyValue pairs	category may be nested for grouping or tree structure.

Table 10-13 Description of the XML Elements to Configure Pluggable tModels

Element/Attribute	Required	Role	Values	Comments
keyName	Required			
keyValue	Required			

XML Schema for Pluggable tModels

The XML Schema against which pluggable tModels are validated is as follows:

```
<simpleType name="type">
  <restriction base="string"/>
</simpleType>

<simpleType name="checked">
  <restriction base="NMTOKEN">
    <enumeration value="true"/>
    <enumeration value="false"/>
  </restriction>
</simpleType>

<element name="scope" type="string"/>

<element name = "applicability" type = "uddi:applicability"/>

<complexType name = "applicability">
  <sequence>
    <element ref = "uddi:scope" minOccurs = "1" maxOccurs = "4"/>
  </sequence>
</complexType>

<element name="category" type="uddi:category"/>

<complexType name = "category">
  <sequence>
   <element ref = "uddi:category" minOccurs = "0" maxOccurs = "unbounded"/>
  </sequence>
  <attribute name = "keyName" use = "required" type="string"/>
  <attribute name = "keyValue" use = "required" type="string"/>
</complexType>
```

```
    <element name="categories" type="uddi:categories"/>

    <complexType name = "categories">
      <sequence>
        <element ref = "uddi:category" minOccurs = "1" maxOccurs = "unbounded"/>
      </sequence>
    </complexType>

    <element name="Taxonomy" type="uddi:Taxonomy"/>

    <complexType name="Taxonomy">
      <sequence>
        <element ref = "uddi:applicability" minOccurs = "0" maxOccurs = "1"/>
        <element ref = "uddi:tModel" minOccurs = "1" maxOccurs = "1"/>
        <element ref = "uddi:categories" minOccurs = "0" maxOccurs = "1"/>
      </sequence>
      <attribute name = "type" use = "required" type="uddi:type"/>
      <attribute name = "checked" use = "required" type="uddi:checked"/>
    </complexType>
```

Sample XML for a Pluggable tModel

The following shows a sample XML for a pluggable tModel:

```
<?xml version="1.0" encoding="UTF-8" ?>

  <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

  <SOAP-ENV:Body>

   <Taxonomy checked="true" type="categorization" xmlns="urn:uddi-org:api_v2" >
    <applicability>
        <scope>businessEntity</scope>
        <scope>businessService</scope>
        <scope>bindingTemplate</scope>
    </applicability>
    <tModel  tModelKey="uuid:C0B9FE13-179F-41DF-8A5B-5004DB444tt2" >
        <name> sample pluggable tModel </name>
        <description>used for test purpose only </description>
            <overviewDoc>
                <overviewURL>http://www.abc.com </overviewURL>
            </overviewDoc>
    </tModel>
    <categories>
        <category keyName="name1 " keyValue="1">
```

```
            <category keyName="name11" keyValue="12">
                <category keyName="name111" keyValue="111">
                    <category keyName="name1111" keyValue="1111"/>
                    <category keyName="name1112" keyValue="1112"/>
                </category>
                <category keyName="name112" keyValue="112">
                    <category keyName="name1121" keyValue="1121"/>
                    <category keyName="name1122" keyValue="1122"/>
                </category>
            </category>
        </category>
        <category keyName="name2 " keyValue="2">
            <category keyName="name21" keyValue="22">
                <category keyName="name211" keyValue="211">
                    <category keyName="name2111" keyValue="2111"/>
                    <category keyName="name2112" keyValue="2112"/>
                </category>
                <category keyName="name212" keyValue="212">
                    <category keyName="name2121" keyValue="2121"/>
                    <category keyName="name2122" keyValue="2122"/>
                </category>
            </category>
        </category>
    </categories>
   </Taxonomy>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```