



BEA WebLogic Server®

WebLogic Web Services: Advanced Programming

Version 10.0
Revised: April 28, 2008

Contents

1. Introduction and Roadmap

Document Scope and Audience	1-1
WebLogic Web Services Documentation Set	1-2
Guide to This Document	1-2
Related Documentation	1-3
Samples for the Web Services Developer	1-4
Release-Specific WebLogic Web Services Information	1-4
Summary of WebLogic Web Services Features	1-5

2. Using Web Services Reliable Messaging

Overview of Web Service Reliable Messaging	2-2
Use of WS-Policy Files for Web Service Reliable Messaging Configuration	2-3
DefaultReliability.xml WS-Policy File	2-3
LongRunningReliability.xml WS-Policy File	2-4
Using Web Service Reliable Messaging: Main Steps	2-5
Configuring the Destination WebLogic Server Instance	2-7
Cluster Considerations	2-8
Configuring the Source WebLogic Server Instance	2-8
Creating the Web Service Reliable Messaging WS-Policy File	2-9
Programming Guidelines for the Reliable JWS File	2-11
Using the @Policy Annotation	2-13
Using the @Oneway Annotation	2-14

Using the @BufferQueue Annotation	2-14
Using the @ReliabilityBuffer Annotation	2-15
Programming Guidelines for the JWS File That Invokes a Reliable Web Service	2-15
WsrnUtils Utility Class	2-18
Updating the build.xml File for a Client of a Reliable Web Service	2-18
Client Considerations When Redeploying a Reliable Web Service	2-19
Using Reliable Messaging With a Proxy Server	2-20
3. Using Callbacks to Notify Clients of Events	
Overview of Callbacks	3-1
Callback Implementation Overview and Terminology	3-2
Programming Callbacks: Main Steps	3-3
Programming Guidelines for Target Web Service	3-5
Programming Guidelines for the Callback Client Web Service	3-6
Programming Guidelines for the Callback Interface	3-9
Updating the build.xml File for the Client Web Service	3-10
4. Creating Conversational Web Services	
Overview of Conversational Web Services	4-1
Creating a Conversational Web Service: Main Steps	4-3
Programming Guidelines for the Conversational JWS File	4-5
Programming Guidelines for the JWS File That Invokes a Conversational Web Service	4-8
ConversationUtils Utility Class	4-11
Updating the build.xml File for a Client of a Conversational Web Service	4-11
Updating a Stand-Alone Java Client to Invoke a Conversational Web Service	4-12
Client Considerations When Redeploying a Conversational Web Service	4-14
5. Creating Buffered Web Services	
Overview of Buffered Web Services	5-1

Creating a Buffered Web Service: Main Steps	5-2
Configuring the Host WebLogic Server Instance for the Buffered Web Service	5-3
Programming Guidelines for the Buffered JWS File	5-4
Programming the JWS File That Invokes the Buffered Web Service	5-6
Updating the build.xml File for a Client of the Buffered Web Service	5-7

6. Invoking a Web Service Using Asynchronous Request-Response

Overview of the Asynchronous Request-Response Feature	6-1
Using Asynchronous Request-Response: Main Steps	6-2
Writing the Asynchronous JWS File	6-4
Coding Guidelines for Invoking a Web Service Asynchronously	6-5
Example of a Synchronous Invoke	6-8
Updating the build.xml File When Using Asynchronous Request-Response	6-9
Disabling The Internal Asynchronous Service	6-10
Using Asynchronous Request Response With a Proxy Server	6-10

7. Using the Asynchronous Features Together

Using the Asynchronous Features Together	7-1
Example of a JWS File That Implements a Reliable Conversational Web Service	7-2
Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service	7-4

8. Using JMS Transport as the Connection Protocol

Overview of Using JMS Transport	8-1
Using JMS Transport Starting From Java: Main Steps	8-2
Using JMS Transport Starting From WSDL: Main Steps	8-4
Using the @WLJmsTransport JWS Annotation	8-6
Using the <WLJmsTransport> Child Element of the jwsc Ant Task	8-8

Updating the WSDL to Use JMS Transport	8-9
Invoking a WebLogic Web Service Using JMS Transport	8-9
Overriding the Default Service Address URL	8-10
Using JMS BytesMessage Rather Than the Default TextMessage	8-11
Disabling HTTP Access to the WSDL File	8-12

9. Creating and Using SOAP Message Handlers

Overview of SOAP Message Handlers	9-1
Adding SOAP Message Handlers to a Web Service: Main Steps	9-4
Designing the SOAP Message Handlers and Handler Chains	9-5
Creating the GenericHandler Class	9-7
Implementing the Handler.init() Method	9-10
Implementing the Handler.destroy() Method	9-10
Implementing the Handler.getHeaders() Method	9-10
Implementing the Handler.handleRequest() Method	9-11
Implementing the Handler.handleResponse() Method	9-12
Implementing the Handler.handleFault() Method	9-13
Directly Manipulating the SOAP Request and Response Message Using SAAJ . .	9-14
Configuring Handlers in the JWS File	9-16
@javax.jws.HandlerChain	9-16
@javax.jws.soap.SOAPMessageHandlers	9-18
Creating the Handler Chain Configuration File	9-20
Compiling and Rebuilding the Web Service	9-22
Creating and Using Client-Side SOAP Message Handlers	9-22
Using Client-Side SOAP Message Handlers: Main Steps	9-23
Example of a Client-Side Handler Class	9-24
Creating the Client-Side SOAP Handler Configuration File	9-25
XML Schema for the Client-Side Handler Configuration File	9-26

Specifying the Client-Side SOAP Handler Configuration File to clientgen 9-27

10. Publishing and Finding Web Services Using UDDI

Overview of UDDI.	10-1
UDDI and Web Services.	10-2
UDDI and Business Registry	10-2
UDDI Data Structure	10-3
WebLogic Server UDDI Features	10-4
UDDI 2.0 Server	10-5
Configuring the UDDI 2.0 Server.	10-5
Configuring an External LDAP Server.	10-6
Description of Properties in the uddi.properties File	10-12
UDDI Directory Explorer	10-20
UDDI Client API	10-20
Pluggable tModel	10-21
XML Elements and Permissible Values	10-21
XML Schema for Pluggable tModels.	10-23
Sample XML for a Pluggable tModel.	10-24

Introduction and Roadmap

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

- [“Document Scope and Audience” on page 1-1](#)
- [“WebLogic Web Services Documentation Set” on page 1-2](#)
- [“Guide to This Document” on page 1-2](#)
- [“Related Documentation” on page 1-3](#)
- [“Samples for the Web Services Developer” on page 1-4](#)
- [“Release-Specific WebLogic Web Services Information” on page 1-4](#)
- [“Summary of WebLogic Web Services Features” on page 1-5](#)

Document Scope and Audience

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

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

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

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

WebLogic Web Services Documentation Set

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

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

Guide to This Document

This document is organized as follows:

- This chapter, [Chapter 1, “Introduction and Roadmap,”](#) introduces the organization of this guide.

- [Chapter 2, “Using Web Services Reliable Messaging,”](#) describes how to create a reliable Web Service, as specified by the WS-ReliableMessaging specification, and then how to create a client Web Services that invokes the reliable Web Service.
- [Chapter 3, “Using Callbacks to Notify Clients of Events,”](#) describes how to notify a client of a Web Service that an event has happened by programming a callback.
- [Chapter 4, “Creating Conversational Web Services,”](#) describes how to create a conversational Web Service which communicates back and forth with a client.
- [Chapter 5, “Creating Buffered Web Services,”](#) describes how to create a buffered Web Service, which is a simpler type of reliable Web Service that one specified by the WS-ReliableMessaging specification.
- [Chapter 6, “Invoking a Web Service Using Asynchronous Request-Response,”](#) describes how to invoke a Web Service asynchronously.
- [Chapter 7, “Using the Asynchronous Features Together,”](#) describes how to use the asynchronous features, such as reliable messaging, asynchronous request-response, and conversations, together in a single Web Service.
- [Chapter 8, “Using JMS Transport as the Connection Protocol,”](#) describes how to specify that JMS, rather than the default HTTP/S, is the connection protocol when invoking a Web Service.
- [Chapter 9, “Creating and Using SOAP Message Handlers,”](#) describes how to create and configure SOAP message handlers for a Web Service.
- [Chapter 10, “Publishing and Finding Web Services Using UDDI,”](#) describes the UDDI features of WebLogic Web Service.

Related Documentation

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

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- [Developing WebLogic Server Applications](#) is a guide to developing WebLogic Server components (such as Web applications and EJBs) and applications.

- [Developing Web Applications, Servlets, and JSPs for WebLogic Server](#) is a guide to developing Web applications, including servlets and JSPs, that are deployed and run on WebLogic Server.
- [Programming WebLogic Enterprise Java Beans](#) is a guide to developing EJBs that are deployed and run on WebLogic Server.
- [Programming WebLogic XML](#) is a guide to designing and developing applications that include XML processing.
- [Deploying Applications to WebLogic Server](#) is the primary source of information about deploying WebLogic Server applications. Use this guide for both development and production deployment of your applications.
- [Configuring Applications for Production Deployment](#) describes how to configure your applications for deployment to a production WebLogic Server environment.
- [WebLogic Server Performance and Tuning](#) contains information on monitoring and improving the performance of WebLogic Server applications.
- [Overview of WebLogic Server System Administration](#) is an overview of administering WebLogic Server and its deployed applications.

Samples for the Web Services Developer

In addition to this document, BEA Systems provides a variety of code samples for Web Services developers. The examples and tutorials illustrate WebLogic Web Services in action, and provide practical instructions on how to perform key Web Service development tasks.

BEA recommends that you run some or all of the Web Service examples before programming your own application that use Web Services.

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

Release-Specific WebLogic Web Services Information

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

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

Summary of WebLogic Web Services Features

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

Using Web Services Reliable Messaging

The following sections describe how to use Web Services Reliable Messaging:

- [“Overview of Web Service Reliable Messaging” on page 2-2](#)
- [“Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 2-3](#)
- [“Using Web Service Reliable Messaging: Main Steps” on page 2-5](#)
- [“Configuring the Destination WebLogic Server Instance” on page 2-7](#)
- [“Configuring the Source WebLogic Server Instance” on page 2-8](#)
- [“Creating the Web Service Reliable Messaging WS-Policy File” on page 2-9](#)
- [“Programming Guidelines for the Reliable JWS File” on page 2-11](#)
- [“Programming Guidelines for the JWS File That Invokes a Reliable Web Service” on page 2-15](#)
- [“WsrnUtils Utility Class” on page 2-18](#)
- [“Updating the build.xml File for a Client of a Reliable Web Service” on page 2-18](#)
- [“Client Considerations When Redeploying a Reliable Web Service” on page 2-19](#)
- [“Using Reliable Messaging With a Proxy Server” on page 2-20](#)

WARNING: This feature can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

Overview of Web Service Reliable Messaging

Web Service reliable messaging is a framework whereby an application running in one application server can reliably invoke a Web Service running on another application server, assuming that both servers implement the WS-ReliableMessaging specification. *Reliable* is defined as the ability to guarantee message delivery between the two Web Services.

Note: Web Services reliable messaging works between *any* two application servers that implement the [WS-ReliableMessaging](#) specification. In this document, however, it is assumed that the two application servers are WebLogic Server instances.

WebLogic Web Services conform to the [WS-ReliableMessaging](#) specification (February 2005), which describes how two Web Services running on different application servers can communicate reliably in the presence of failures in software components, systems, or networks. In particular, the specification describes an interoperable protocol in which a message sent from a *source endpoint* (or client Web Service) to a *destination endpoint* (or Web Service whose operations can be invoked reliably) is guaranteed either to be delivered, according to one or more delivery assurances, or to raise an error.

A reliable WebLogic Web Service provides the following delivery assurances:

- **AtMostOnce**—Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
- **AtLeastOnce**—Every message is delivered at least once. It is possible that some messages are delivered more than once.
- **ExactlyOnce**—Every message is delivered exactly once, without duplication.
- **InOrder**—Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

See the [WS-ReliableMessaging](#) specification for detailed documentation about the architecture of Web Service reliable messaging. “[Using Web Service Reliable Messaging: Main Steps](#)” on [page 2-5](#) describes how to create the reliable and client Web Services and how to configure the two WebLogic Server instances to which the Web Services are deployed.

Note: Web Services reliable messaging is not supported with the JMS transport feature.

Use of WS-Policy Files for Web Service Reliable Messaging Configuration

WebLogic Web Services use WS-Policy files to enable a destination endpoint to describe and advertise its Web Service reliable messaging capabilities and requirements. The [WS-Policy specification](#) provides a general purpose model and syntax to describe and communicate the policies of a Web service.

These WS-Policy files are XML files that describe features such as the version of the supported WS-ReliableMessaging specification, the source endpoint's retransmission interval, the destination endpoint's acknowledgment interval, and so on.

You specify the names of the WS-Policy files that are attached to your Web Service using the `@Policy` JWS annotation in your JWS file. Use the `@Policies` annotation to group together multiple `@Policy` annotations. For reliable messaging, you specify these annotations only at the class level.

WebLogic Server includes two simple WS-Policy files that you can specify in your JWS file if you do not want to create your own WS-Policy files:

- `DefaultReliability.xml`—Specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmission interval of 3 seconds. See [“DefaultReliability.xml WS-Policy File” on page 2-3](#) for the actual WS-Policy file.
- `LongRunningReliability.xml`—Similar to the preceding default reliable messaging WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours.) See [“LongRunningReliability.xml WS-Policy File” on page 2-4](#) for the actual WS-Policy file.

You cannot change these pre-packaged files, so if their values do not suit your needs, you must create your own WS-Policy file.

See [“Creating the Web Service Reliable Messaging WS-Policy File” on page 2-9](#) for details about creating your own WS-Policy file if you do not want to one included with WebLogic Server. See [Web Service Reliable Messaging Policy Assertion Reference](#) for reference information about the reliable messaging policy assertions.

DefaultReliability.xml WS-Policy File

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

```

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

  <wsrm:RMAssertion >

    <wsrm:InactivityTimeout
      Milliseconds="600000" />
    <wsrm:BaseRetransmissionInterval
      Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
    <wsrm:AcknowledgementInterval
      Milliseconds="200" />
    <beapolicy:Expires Expires="P1D" optional="true"/>
  </wsrm:RMAssertion>
</wsp:Policy>

```

LongRunningReliability.xml WS-Policy File

```

<?xml version="1.0"?>

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

  <wsrm:RMAssertion >

    <wsrm:InactivityTimeout
      Milliseconds="86400000" />
    <wsrm:BaseRetransmissionInterval
      Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
    <wsrm:AcknowledgementInterval
      Milliseconds="200" />
    <beapolicy:Expires Expires="P1M" optional="true"/>
  </wsrm:RMAssertion>

```

```
</wsp:Policy>
```

Using Web Service Reliable Messaging: Main Steps

Configuring reliable messaging for a WebLogic Web Service requires standard JMS tasks such as creating JMS servers and Store and Forward (SAF) agents, as well as Web Service-specific tasks, such as adding additional JWS annotations to your JWS file. Optionally, you create WS-Policy files that describe the reliable messaging capabilities of the reliable Web Service if you do not use the pre-packaged ones.

If you are using the WebLogic client APIs to invoke a reliable Web Service, the client application must run on WebLogic Server. Thus, configuration tasks must be performed on both the *source* WebLogic Server instance on which the Web Service that includes client code to invoke the reliable Web Service reliably is deployed, as well as the *destination* WebLogic Server instance on which the reliable Web Service itself is deployed.

The following procedure describes how to create a reliable Web Service, as well as a client Web Service that in turn invokes an operation of the reliable Web Service reliably. The procedure shows how to create the JWS files that implement the two Web Services from scratch; if you want to update existing JWS files, use this procedure as a guide. The procedure also shows how to configure the source and destination WebLogic Server instances.

It is assumed that you have created a WebLogic Server instance where 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 reliable Web Service. It is further assumed that you have a similar setup for another WebLogic Server instance that hosts the client Web Service that invokes the Web Service reliably. For more information, see:

- [Common Web Services Use Cases and Examples](#)
- [Iterative Development of WebLogic Web Services](#)
- [Programming the JWS File](#)
- [Invoking Web Services](#)

1. Configure the destination WebLogic Server instance for Web Service reliable messaging.
This is the WebLogic Server instance to which the reliable Web Service is deployed.
See [“Configuring the Destination WebLogic Server Instance” on page 2-7](#).
2. Configure the source WebLogic Server instance for Web Service reliable messaging.

This is the WebLogic Server instance to which the client Web Service that invokes the reliable Web Service is deployed.

See [“Configuring the Source WebLogic Server Instance” on page 2-8.](#)

3. Using your favorite XML or plain text editor, optionally create a WS-Policy file that describes the reliable messaging capabilities of the Web Service running on the destination WebLogic Server. This step is not required if you plan to use one of the two WS-Policy files that are included in WebLogic Server; see [“Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 2-3](#) for more information.

See [“Creating the Web Service Reliable Messaging WS-Policy File” on page 2-9](#) for details about creating your own WS-Policy file.

4. Create a new JWS file, or update an existing one, which implements the reliable Web Service that will run on the destination WebLogic Server.

See [“Programming Guidelines for the Reliable JWS File” on page 2-11.](#)

5. Update your `build.xml` file to include a call to the `jwsc` Ant task which will compile the reliable JWS file into a Web Service.

See [Running the jwsc WebLogic Web Services Ant Task](#) for general information about using the `jwsc` task.

6. Compile your destination JWS file by calling the appropriate target and deploy to the destination WebLogic Server. For example:

```
prompt> ant build-mainService deploy-mainService
```

7. Create a new JWS file, or update an existing one, that implements the client Web Service that invokes the reliable Web Service. This service will be deployed to the source WebLogic Server.

See [“Programming Guidelines for the JWS File That Invokes a Reliable Web Service” on page 2-15.](#)

8. Update the `build.xml` file that builds the client Web Service.

See [“Updating the build.xml File for a Client of a Reliable Web Service” on page 2-18.](#)

9. Compile your client JWS file by calling the appropriate target and deploy to the source WebLogic Server. For example:

```
prompt> ant build-clientService deploy-clientService
```

Configuring the Destination WebLogic Server Instance

Configuring the WebLogic Server instance on which the reliable Web Service is deployed involves configuring JMS and store and forward (SAF) resources.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see [Configuring Your Domain For Web Services Features](#).

If, however, you prefer to configure the resources yourself, use the following high-level procedure which lists the tasks and then points to the Administration Console Online Help for details on performing the tasks.

1. Invoke the Administration Console for the domain that contains the destination WebLogic Server in your browser.

See [Invoking the Administration Console](#) for instructions on the URL that invokes the Administration Console.
2. Optionally create a persistent store (either file or JDBC) that will be used by the destination WebLogic Server to store internal Web Service reliable messaging information. You can use an existing one, or the default store that always exists, if you do not want to create a new one.

See [Create file stores](#).
3. Create a JMS Server. If a JMS server already exists, you can use it if you do not want to create a new one.

See [Create JMS servers](#).
4. Create a JMS module, and then define a JMS queue in the module. If a JMS module already exists, you can use it if you do not want to create a new one. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name.

Take note of the JNDI name you define for the JMS queue because you will later use it when you program the JWS file that implements your reliable Web Service.

See [Create JMS modules](#) and [Create queues](#).
5. Create a store and forward (SAF) agent. You can use an existing one if you do not want to create a new one.

When you create the SAF agent:

- Set the **Agent Type** field to `Both` to enable both sending and receiving agents.
- Be sure to target the SAF agent by clicking **Next** on the first assistant page to view the Select targets page (rather than clicking **Finish**).
- If you are using reliable messaging within a cluster, you must target the SAF agent to the cluster.

See [Create Store and Forward agents](#).

Cluster Considerations

If you are using the Web Service reliable messaging feature in a cluster, you must:

- Still create a *local* JMS queue, rather than a distributed queue, when creating the JMS queue in [step 4 in “LongRunningReliability.xml WS-Policy File”](#).
- Explicitly target this JMS queue to each server in the cluster.
- Explicitly target the SAF agent to the cluster.

Configuring the Source WebLogic Server Instance

Configuring the WebLogic Server instance on which the client Web Service is deployed involves configuring JMS and store and forward (SAF) resources.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see [Configuring Your Domain For Web Services Features](#).

If, however, you prefer to configure the resources yourself, use the following high-level procedure which lists the tasks and then points to the Administration Console Online Help for details on performing the tasks.

1. Invoke the Administration Console for the domain that contains the source WebLogic Server in your browser.

See [Invoking the Administration Console](#) for instructions on the URL that invokes the Administration Console.

2. Create a persistent store (file or JDBC) that will be used by the source WebLogic Server to store internal Web Service reliable messaging information. You can use an existing one if you do not want to create a new one.

See [Create file stores](#).

3. Create a JMS Server. You can use an existing one if you do not want to create a new one.

See [Create JMS servers](#).

4. Create a store and forward (SAF) agent. You can use an existing one if you do not want to create a new one.

Be sure when you create the SAF agent that you set the **Agent Type** field to **Both** to enable both sending and receiving agents.

See [Create Store and Forward agents](#).

Creating the Web Service Reliable Messaging WS-Policy File

A WS-Policy file is an XML file that contains policy assertions that comply with the WS-Policy specification. In this case, the WS-Policy file contains Web Service reliable messaging policy assertions.

You can use one of the two default reliable messaging WS-Policy files included in WebLogic Server; these files are adequate for most use cases. However, because these files cannot be changed, if they do not suit your needs, you must create your own. See [“Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 2-3](#) for a description of the included WS-Policy files. The remainder of this section describes how to create your own WS-Policy file.

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declarations for using Web Service reliable messaging policy assertions:

```
<wsp:Policy
  xmlns:wsm="http://schemas.xmlsoap.org/ws/2005/02/rm"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:beapolicy="http://www.bea.com/wsm/policy">
```

You wrap all Web Service reliable messaging policy assertions inside of a `<wsm:RMAssertion>` element. The assertions that use the `wsm:` namespace are standard ones defined by the [WS-ReliableMessaging](#) specification. The assertions that use the `beapolicy:` namespace are WebLogic-specific. See [Web Service Reliable Messaging Policy Assertion Reference](#) for details.

All Web Service reliable messaging assertions are optional, so only set those whose default values are not adequate. The order in which the assertions appear is important. You can specify

the following assertions; the order they appear in the following list is the order in which they should appear in your WS-Policy file:

- `<wsrm:InactivityTimeout>`—Number of milliseconds, specified with the `Milliseconds` attribute, which defines an inactivity interval. After this amount of time, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may consider the sequence to have terminated due to inactivity. The same is true for the source endpoint. By default, sequences never timeout.
- `<wsrm:BaseRetransmissionInterval>`—Interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message. Default value is set by the SAF agent on the source endpoint's WebLogic Server instance.
- `<wsrm:ExponentialBackoff>`—Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm. This element has no attributes.
- `<wsrm:AcknowledgmentInterval>`—Maximum interval, in milliseconds, in which the destination endpoint must transmit a stand-alone acknowledgement. The default value is set by the SAF agent on the destination endpoint's WebLogic Server instance.
- `<beapolicy:Expires>`—Amount of time after which the reliable Web Service expires and does not accept any new sequence messages. The default value is to never expire. This element has a single attribute, `Expires`, whose data type is an [XML Schema duration type](#). For example, if you want to set the expiration time to one day, use the following:
`<beapolicy:Expires Expires="P1D" />`.
- `<beapolicy:QOS>`—Delivery assurance level, as described in [“Overview of Web Service Reliable Messaging” on page 2-2](#). The element has one attribute, `QOS`, which you set to one of the following values: `AtMostOnce`, `AtLeastOnce`, or `ExactlyOnce`. You can also include the `InOrder` string to specify that the messages be in order. The default value is `ExactlyOnce InOrder`. This element is typically not set.

The following example shows a simple Web Service reliable messaging WS-Policy file:

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

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

  <wsrm:RMAssertion>
```



```

<wsrm:InactivityTimeout
  Milliseconds="600000" />
<wsrm:BaseRetransmissionInterval
  Milliseconds="500" />
<wsrm:ExponentialBackoff />
<wsrm:AcknowledgementInterval
  Milliseconds="2000" />

</wsrm:RMAssertion>

</wsp:Policy>

```

Programming Guidelines for the Reliable JWS File

This section describes how to create the JWS file that implements the reliable Web Service.

The following JWS annotations are used in the JWS file that implements a reliable Web Service:

- `@weblogic.jws.Policy`—Required. See [“Using the @Policy Annotation” on page 2-13](#).
- `@javax.jws.Oneway`—Required only if you are using Web Service reliable messaging on its own, without also using the asynchronous request-response feature. See [“Using the @Oneway Annotation” on page 2-14](#) and [Chapter 7, “Using the Asynchronous Features Together.”](#)
- `@weblogic.jws.BufferQueue`—Optional. See [“Using the @BufferQueue Annotation” on page 2-14](#).
- `@weblogic.jws.ReliabilityBuffer`—Optional. See [“Using the @ReliabilityBuffer Annotation” on page 2-15](#)

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

```

package examples.webservices.reliable;

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

import weblogic.jws.WLHttpTransport;

import weblogic.jws.ReliabilityBuffer;
import weblogic.jws.BufferQueue;
import weblogic.jws.Policy;

```

```

/**
 * Simple reliable Web Service.
 */

@WebService(name="ReliableHelloWorldPortType",
            serviceName="ReliableHelloWorldService")

@WLHttpTransport(contextPath="ReliableHelloWorld",
                 serviceUri="ReliableHelloWorld",
                 portName="ReliableHelloWorldServicePort")

@Policy(uri="ReliableHelloWorldPolicy.xml",
        direction=Policy.Direction.both,
        attachToWsdl=true)

@BufferQueue(name="webservices.reliable.queue")

public class ReliableHelloWorldImpl {

    @WebMethod()
    @Oneway()
    @ReliabilityBuffer(retryCount=10, retryDelay="10 seconds")

    public void helloWorld(String input) {
        System.out.println(" Hello World " + input);
    }
}

```

In the example, the `ReliableHelloWorldPolicy.xml` file is attached to the Web Service at the class level, which means that the policy file is applied to all public operations of the Web Service. The policy file is applied only to the request Web Service message (as required by the reliable messaging feature) and it is attached to the WSDL file.

The JMS queue that WebLogic Server uses internally to enable the Web Service reliable messaging has a JNDI name of `webservices.reliable.queue`, as specified by the `@BufferQueue` annotation.

The `helloWorld()` method has been marked with both the `@WebMethod` and `@Oneway` JWS annotations, which means it is a public operation called `helloWorld`. Because of the `@Policy` annotation, the operation can be invoked reliably. The Web Services runtime attempts to deliver reliable messages to the service a maximum of 10 times, at 10-second intervals, as described by the `@ReliabilityBuffer` annotation. The message may require re-delivery if, for example, the transaction is rolled back or otherwise does not commit.

Using the @Policy Annotation

Use the `@Policy` annotation in your JWS file to specify that the Web Service has a WS-Policy file attached to it that contains reliable messaging assertions.

See [“Use of WS-Policy Files for Web Service Reliable Messaging Configuration” on page 2-3](#) for descriptions of the two WS-Policy files (`DefaultReliability.xml` and `LongRunningReliability.xml`) included in WebLogic Server that you can use instead of writing your own.

You must follow these requirements when using the `@Policy` annotation for Web Service reliable messaging:

- Specify the `@Policy` annotation *only* at the class-level.
- Because Web Service reliable messaging is applied to both the request and response SOAP message, set the `direction` attribute of the `@Policy` annotation only to its default value: `Policy.Direction.both`.

Use the `uri` attribute to specify the build-time location of the policy file, as follows:

- If you have created your own WS-Policy file, specify its location relative to the JWS file. For example:

```
@Policy(uri="ReliableHelloWorldPolicy.xml",
        direction=Policy.Direction.both,
        attachToWSDL=true)
```

The example shows that the `ReliableHelloWorldPolicy.xml` file is located in the same directory as the JWS file.

- To specify that the Web Service is going to use a WS-Policy file that is part of WebLogic Server, use the `policy:` prefix along with the name and path of the policy file. This syntax tells the `jwsc` Ant task at build-time *not* to look for an actual file on the file system, but rather, that the Web Service will retrieve the WS-Policy file from WebLogic Server at the time the service is deployed. Use this syntax when specifying one of the pre-packaged WS-Policy files or when specifying a WS-Policy file that is packaged in a shared Java EE library.

Note: Shared Java EE libraries are useful when you want to share a WS-Policy file with multiple Web Services that are packaged in different Enterprise applications. As long as the WS-Policy file is located in the `META-INF/policies` or `WEB-INF/policies` directory of the shared Java EE library, you can specify the policy file in the same way as if it were packaged in the same archive at the Web Service. See [Creating](#)

[Shared Java EE Libraries and Optional Packages](#) for information on creating libraries and setting up your environment so the Web Service can find the policy files.

- To specify that the policy file is published somewhere on the Web, use the `http:` prefix along with the URL, as shown in the following example:

```
@Policy(uri="http://someSite.com/policies/mypolicy.xml"  
        direction=Policy.Direction.both,  
        attachToWsd=true)
```

You can also set the `attachToWsd` attribute of the `@Policy` annotation to specify whether the policy file should be attached to the WSDL file that describes the public contract of the Web Service. Typically you want to publicly publish the policy so that client applications know the reliable messaging capabilities of the Web Service. For this reason, the default value of this attribute is `true`.

Using the `@Oneway` Annotation

If you plan on invoking the reliable Web Service operation synchronously (or in other words, *not* using the asynchronous request-response feature), then the implementing method is required to be annotated with the `@Oneway` annotation to specify that the method is one-way. This means that the method cannot return a value, but rather, must explicitly return `void`.

Conversely, if the method is *not* annotated with the `@Oneway` annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See [Chapter 6, “Invoking a Web Service Using Asynchronous Request-Response,”](#) and [Chapter 7, “Using the Asynchronous Features Together.”](#)

Using the `@BufferQueue` Annotation

Use the `@BufferQueue` annotation to specify the JNDI name of the JMS queue which WebLogic Server uses to store reliable messages internally. The JNDI name is the one you configured when creating a JMS queue in [step 4 in “LongRunningReliability.xml WS-Policy File”](#).

The `@BufferQueue` annotation is optional; if you do not specify it in your JWS file then WebLogic Server uses a queue with a JNDI name of `weblogic.wsee.DefaultQueue`. You must, however, still explicitly create a JMS queue with this JNDI name using the Administration Console.

Using the @ReliabilityBuffer Annotation

Use this annotation to specify the number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web Service implementation (default 3) and the amount of time that the server should wait in between retries (default 5 seconds).

Use the `retryCount` attribute to specify the number of retries and the `retryDelay` attribute to specify the wait time. The format of the `retryDelay` attribute is a number and then one of the following strings:

- seconds
- minutes
- hours
- days
- years

For example, to specify a retry count of 20 and a retry delay of two days, use the following syntax:

```
@ReliabilityBuffer(retryCount=20, retryDelay="2 days")
```

Note: For the `@ReliabilityBuffer` annotation, the retry of a request is only triggered by a system level failure (for example, a JMS resource issue). Exceptions raised from the user code (JWS or handler chain) does not trigger a retry.

Programming Guidelines for the JWS File That Invokes a Reliable Web Service

If you are using the WebLogic client APIs, you must invoke a reliable Web Service from within a Web Service; you cannot invoke a reliable Web Service from a stand-alone client application.

The following example shows a simple JWS file for a Web Service that invokes a reliable operation from the service described in [“Programming Guidelines for the Reliable JWS File” on page 2-11](#); see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.reliable;

import java.rmi.RemoteException;

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

```

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.ReliabilityErrorHandler;

import examples.webservices.reliable.ReliableHelloWorldPortType;

import weblogic.wsee.reliability.ReliabilityErrorContext;
import weblogic.wsee.reliability.ReliableDeliveryException;

@WebService(name="ReliableClientPortType",
            serviceName="ReliableClientService")

@WLHttpTransport(contextPath="ReliableClient",
                 serviceUri="ReliableClient",
                 portName="ReliableClientServicePort")

public class ReliableClientImpl
{
    @ServiceClient(
        serviceName="ReliableHelloWorldService",
        portName="ReliableHelloWorldServicePort")

    private ReliableHelloWorldPortType port;

    @WebMethod
    public void callHelloWorld(String input, String serviceUrl)
        throws RemoteException {

        port.helloWorld(input);

        System.out.println(" Invoked the ReliableHelloWorld.helloWorld operation
reliably." );

    }

    @ReliabilityErrorHandler(target="port")
    public void onReliableMessageDeliveryError(ReliabilityErrorContext ctx) {

        ReliableDeliveryException fault = ctx.getFault();
        String message = null;
        if (fault != null) {
            message = ctx.getFault().getMessage();
        }
        String operation = ctx.getOperationName();
        System.out.println("Reliable operation " + operation + " may have not invoked.
The error message is " + message);
    }
}

```

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

- Import the `@ServiceClient` and `@ReliabilityErrorHandler` JWS annotations:

```
import weblogic.jws.ServiceClient;
import weblogic.jws.ReliabilityErrorHandler;
```

- Import the JAX-RPC stub, created later by the `<clientgen>` child element of the `jws` Ant task, of the port type of the reliable Web Service you want to invoke. The stub package is specified by the `packageName` attribute of `<clientgen>`, and the name of the stub is determined by the WSDL of the invoked Web Service.

```
import examples.webservices.reliable.ReliableHelloWorldPortType;
```

- Import the WebLogic APIs that you will use in the method that handles the error that results when the client Web Service does not receive an acknowledgement of message receipt from the reliable Web Service:

```
import weblogic.wsee.reliability.ReliabilityErrorContext;
import weblogic.wsee.reliability.ReliableDeliveryException;
```

- In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the name and port of the reliable Web Service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.

```
@ServiceClient(
    serviceName="ReliableHelloWorldService",
    portName="ReliableHelloWorldServicePort")
private ReliableHelloWorldPortType port;
```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the reliable operation:

```
port.helloWorld(input);
```

Because the operation has been marked one-way, it does not return a value.

- Create a method that handles the error when the client Web Service does not receive an acknowledgement from the reliable Web Service that the latter has received a message and annotate this method with the `@weblogic.jws.ReliabilityErrorHandler` annotation:

```
@ReliabilityErrorHandler(target="port")
public void onReliableMessageDeliveryError(ReliabilityErrorContext
ctx) {
    ReliableDeliveryException fault = ctx.getFault();
    String message = null;
```

```

        if (fault != null) {
            message = ctx.getFault().getMessage();
        }
        String operation = ctx.getOperationName();
        System.out.println("Reliable operation " + operation + " may have not
invoked. The error message is " + message);
    }

```

This method takes `ReliabilityErrorContext` as its single parameter and returns `void`.

See [weblogic.jws.ReliabilityErrorHandler](#) for details about programming this error-handling method.

When programming the client Web Service, be sure you do *not*:

- Specify any reliable messaging annotations (other than `@ReliabilityErrorHandler`) or use any reliable messaging assertions in the associated WS-Policy files.
- Specify the `wsdlLocation` attribute of the `@ServiceClient` annotation. This is because the runtime retrieval of the specified WSDL might not succeed, thus it is better to let WebLogic Server use a local WSDL file instead.

WsrUtils Utility Class

WebLogic Server provides a utility class for use with the Web Service Reliable Messaging feature. Use this class to perform common tasks such as set configuration options, get the sequence id, and terminate a reliable sequence. Some of these tasks are performed in the reliable Web Service, some are performed in the Web Service that invokes the reliable Web Service.

See [weblogic.wsee.reliability.WsrUtils](#) for details.

Updating the build.xml File for a Client of a Reliable Web Service

To update a `build.xml` file to generate the JWS file that invokes the operation of a reliable Web Service, add `taskdefs` and a `build-reliable-client` targets that look something like the following; see the description after the example for details:

```

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

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

```



```

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

  <jws file="examples/webservices/reliable/ReliableClientImpl.java">

    <clientgen

wsdl="http://${wls.destination.host}:${wls.destination.port}/ReliableHello
World/ReliableHelloWorld?WSDL"
      packageName="examples.webservices.reliable"/>

    </jws>
  </jwsc>
</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 a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `ReliableHelloWorld` 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 `ReliableClientImpl` JWS file imports and uses one of the generated classes.

Client Considerations When Redeploying a Reliable Web Service

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

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

For additional information about production redployment and Web Service clients, see [Client Considerations When Redeploying a Web Service](#).

Using Reliable Messaging With a Proxy Server

Client applications that invoke reliable Web Services might not invoke the operation directly, but rather, use a proxy server. Reasons for using a proxy include the presence of a firewall or the deployment of the invoked Web Service to a cluster.

In this case, the WebLogic Server instance that hosts the invoked Web Service must be configured with the address and port of the proxy server. If your Web Service is deployed to a cluster, you must configure every server in the cluster.

For each server instance:

1. Create a network channel for the protocol you use to invoke the Web Service. You must name the network channel `weblogic-wsee-proxy-channel-xxx`, where `xxx` refers to the protocol. For example, to create a network channel for HTTPS, call it `weblogic-wsee-proxy-channel-https`.

See [Configure Custom Network Channels](#) for general information about creating a network channel.

2. Configure the network channel, updating the **External Listen Address** and **External Listen Port** fields with the address and port of the proxy server, respectively.

Using Callbacks to Notify Clients of Events

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

- [“Overview of Callbacks” on page 3-1](#)
- [“Callback Implementation Overview and Terminology” on page 3-2](#)
- [“Programming Callbacks: Main Steps” on page 3-3](#)
- [“Programming Guidelines for Target Web Service” on page 3-5](#)
- [“Programming Guidelines for the Callback Client Web Service” on page 3-6](#)
- [“Programming Guidelines for the Callback Interface” on page 3-9](#)
- [“Updating the build.xml File for the Client Web Service” on page 3-10](#)

WARNING: This feature can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

Overview of Callbacks

Callbacks notify a client of your Web Service that some event has occurred. For example, you can notify a client when the results of that client's request are ready, or when the client's request cannot be fulfilled.

When you expose method as a standard public operation in your JWS file (by using the `@WebMethod` annotation), the client sends a SOAP message to the Web Service to invoke the

operation. When you add a callback to a Web Service, however, you define a message that the Web Service sends *back to the client Web Service*, notifying the client of an event that has occurred. So exposing a method as a public operation and defining a callback are completely symmetrical processes, with opposite recipients.

WebLogic Server automatically routes the SOAP message from client invoke to the target Web Service. In order to receive callbacks, however, the client must be operating in an environment that provides the same services. This typically means the client is a Web Service running on a Web server. If the client does not meet these requirements, it is likely not capable of receiving callbacks from your Web Service.

The protocol and message format used for callbacks is always the same as the protocol and message format used by the conversation start method that initiated the current conversation. If you attempt to override the protocol or message format of a callback, an error is thrown.

Callback Implementation Overview and Terminology

To implement callbacks, you must create or update the following three Java files:

- **Callback interface:** Java interface file that defines the callback methods. You do not explicitly implement this file yourself; rather, the `jwsc` Ant task automatically generates an implementation of the interface. The implementation simply passes a message from the target Web Service back to the client Web Service. The generated Web Service is deployed to the same WebLogic Server that hosts the client Web Service.

In the example in this section, the callback interface is called `CallbackInterface`. The interface defines a single callback method called `callbackOperation()`.

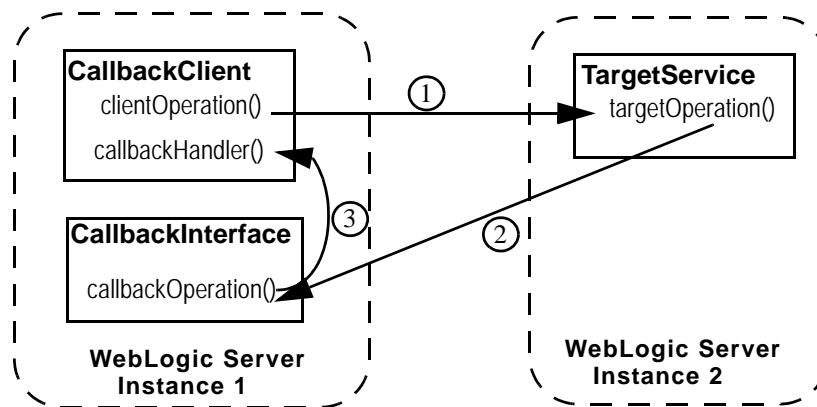
- **JWS file that implements the target Web Service:** The target Web Service includes one or more standard operations that invoke a method defined in the callback interface; this method in turn sends a message back to the client Web Service that originally invoked the operation of the target Web Service.

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. This Web Service includes one or more methods that specify what the client should do when it receives a callback message back from the target Web Service via a callback method.

In the example, this Web Service is called `CallbackClient` and the method that is automatically invoked when it receives a callback is called `callbackHandler()`. The method that invokes `TargetService` in the standard way is called `clientOperation()`.

The following graphic shows the flow of messages:



1. The `clientOperation()` method of the `CallbackClient` Web Service, running in one WebLogic Server instance, explicitly invokes the `targetOperation()` operation of the `TargetService`. The `TargetService` service might be running in a separate WebLogic Server instance.
2. The implementation of the `TargetService.targetOperation()` method explicitly invokes the `callbackOperation()` operation of the `CallbackInterface`, which implements the callback service. The callback service is deployed to the WebLogic Server which hosts the client Web Service.
3. The jws-c-generated implementation of the `CallbackInterface.callbackOperation()` method simply sends a message back to the `CallbackClient` Web Service. The client Web Service includes a method `callbackHandler()` that handles this message.

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 interface. 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:

- [Common Web Services Use Cases and Examples](#)
 - [Iterative Development of WebLogic Web Services](#)
 - [Programming the JWS File](#)
 - [Invoking Web Services](#)
1. Using your favorite IDE, create a new JWS file, or update an existing one, that implements the target Web Service.
See [“Programming Guidelines for Target Web Service” on page 3-5](#).
Note: The JWS file that implements the target Web Service invokes one or more callback methods of the callback interface. However, the step that describes how to program the callback interface 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 [Running the jwsc WebLogic Web Services Ant Task](#).
 3. Run the Ant target to build the target Web Service. For example:

```
prompt> ant build-mainService
```
 4. Deploy the target Web Service as usual.
See [Deploying and Undeploying WebLogic Web Services](#).
 5. Using your favorite IDE or text editor, 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 3-6](#).
 6. Create the callback JWS interface that implements the callback Web Service.
See [“Programming Guidelines for the Callback Interface” on page 3-9](#).

7. Update the `build.xml` file that builds the client Web Service. The `jwsC` Ant task that builds the client Web Service also implicitly generates the callback Web Service from the callback interface file.

See [“Updating the build.xml File for the Client Web Service” on page 3-10](#).

8. Run the Ant target to build the client and callback Web Services.

```
prompt> ant build-clientService
```

9. Deploy the client Web Service as usual. Because the callback service is packaged in the same EAR file as the client Web Service, it will also be deployed at the same time.

See [Deploying and Undeploying WebLogic Web Services](#).

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 weblogic.jws.WLHttpTransport;
import weblogic.jws.Callback;

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

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

@WLHttpTransport(contextPath="callback",
                 serviceUri="TargetService",
                 portName="TargetServicePort")

/**
 * callback service
 */

public class TargetServiceImpl {

    @Callback
    CallbackInterface callback;

    @WebMethod
    public void targetOperation (String message) {
```

```

        callback.callbackOperation (message);
    }
}

```

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 required JWS annotations:

```
import weblogic.jws.Callback;
```

- Use the `@weblogic.jws.Callback` JWS annotation to specify that a variable is a callback, which means that you can use the annotated variable to send callback events back to a client Web Service that invokes an operation of the `TargetService` Web Service. The data type of the variable is the callback interface, which in this case is called `CallbackInterface`.

```
@Callback
CallbackInterface callback;
```

- In a method that implements an operation of the `TargetService`, use the annotated variable to invoke one of the callback methods of the callback interface, which in this case is called `callbackOperation()`:

```
callback.callbackOperation (message);
```

See [JWS Annotation Reference](#) for additional information about the WebLogic-specific JWS annotations discussed in this section.

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 3-5](#); see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.callback;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.CallbackMethod;
import weblogic.jws.security.CallbackRolesAllowed;
import weblogic.jws.security.SecurityRole;

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



```

import examples.webservices.callback.CallbackPortType;

import java.rmi.RemoteException;

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

@WLHttpTransport(contextPath="callbackClient",
                 serviceUri="CallbackClient",
                 portName="CallbackClientPort")

public class CallbackClientImpl {

    @ServiceClient(
        wsdlLocation="http://localhost:7001/callback/TargetService?WSDL",
        serviceName="TargetService",
        portName="TargetServicePort")
    @CallbackRolesAllowed(@SecurityRole(role="mgr", mapToPrincipals="joe"))
    private CallbackPortType port;

    @WebMethod
    public void clientOperation (String message) {

        try {

            port.targetOperation(message);
        }
        catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @CallbackMethod(target="port", operation="callbackOperation")
    @CallbackRolesAllowed(@SecurityRole(role="engineer",
mapToPrincipals="shackell"))
    public void callbackHandler(String msg) {

        System.out.println (msg);
    }
}

```

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 required JWS annotations:

```

import weblogic.jws.ServiceClient;
import weblogic.jws.CallbackMethod;

```

- Optionally import the security-related annotations if you want to specify the roles that are allowed to invoke the callback methods:

```
import weblogic.jws.security.CallbackRolesAllowed;
import weblogic.jws.security.SecurityRole;
```

- Import the JAX-RPC stub of the port type of the target Web Service you want to invoke. The actual stub itself will be created later by the `jwsc` Ant task. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of `<jws>`, and the name of the stub is determined by the WSDL of the invoked Web Service.

```
import examples.webservices.callback.CallbackPortType;
```

- In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the target Web Service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.

```
@ServiceClient(
    wsdlLocation="http://localhost:7001/callback/TargetService?WSDL",
    serviceName="TargetService",
    portName="TargetServicePort")
@CallbackRolesAllowed(@SecurityRole(role="mgr",
    mapToPrincipals="joe"))
private CallbackPortType port;
```

The preceding code also shows how to use the optional `@CallbackRolesAllowed` annotation to specify the list of `@SecurityRoles` that are allowed to invoke the callback methods.

- Using the variable you annotated with the `@ServiceClient` annotation, invoke an operation of the target Web Service. This operation in turn will invoke a callback method of the callback interface:

```
port.targetOperation(message);
```

- Create a method that will handle the callback message received from the callback service. You can name this method anything you want. However, its signature should exactly match the signature of the corresponding method in the callback interface.

Annotate the method with the `@CallbackMethod` annotation to specify that this method handles callback messages. Use the `target` attribute to specify the name of the JAX-RPC port for which you want to receive callbacks (in other words, the variable you previously annotated with `@ServiceClient`). Use the `operation` attribute to specify the name of the callback method in the callback interface from which this method will handle callback messages.

```

    @CallbackMethod(target="port", operation="callbackOperation")
    @CallbackRolesAllowed(@SecurityRole(role="engineer",
mapToPrincipals="shackell"))
    public void callbackHandler(String msg) {
        System.out.println (msg);
    }

```

The preceding code also shows how to use the optional `@CallbackRolesAllowed` annotation to further restrict the security roles that are allowed to invoke this particular callback method.

See [JWS Annotation Reference](#) for additional information about the WebLogic-specific JWS annotations discussed in this section.

Programming Guidelines for the Callback Interface

The callback interface is also a JWS file that implements a Web Service, except for one big difference: instead of using the standard `@javax.jws.WebService` annotation to specify that it is a standard Web Service, you use the WebLogic-specific `@weblogic.jws.CallbackService` to specify that it is a callback service. The attributes of `@CallbackService` are a restricted subset of the attributes of `@WebService`.

Follow these restrictions on the allowed data types and JWS annotations when programming the JWS file that implements a callback service:

- You cannot use any WebLogic-specific JWS annotations other than `@weblogic.jws.CallbackService`.
- You can use all standard JWS annotations except for the following:
 - `javax.jws.HandlerChain`
 - `javax.jws.soap.SOAPMessageHandler`
 - `javax.jws.soap.SOAPMessageHandlers`
- You can use all supported data types as parameters or return values except `Holder` classes (user-defined data types that implement the `javax.xml.rpc.holders.Holder` interface).

The following example shows a simple callback interface file that implements a callback Web Service. The target Web Service, described in [“Programming Guidelines for Target Web Service” on page 3-5](#), explicitly invokes a method in this interface. The `jws-c`-generated implementation of the callback interface then automatically sends a message back to the client Web Service that originally invoked the target Web Service; the client service is described in [“Programming Guidelines for the Callback Client Web Service” on page 3-6](#). See the explanation after the example for coding guidelines that correspond to the Java code in bold.

```

package examples.webservices.callback;

import weblogic.jws.CallbackService;

import javax.jws.Oneway;
import javax.jws.WebMethod;

@CallbackService
public interface CallbackInterface {

    @WebMethod
    @Oneway
    public void callbackOperation (String msg);
}

```

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

- Import the required JWS annotation:

```
import weblogic.jws.CallbackService;
```

- Annotate the interface declaration with the `@CallbackService` annotation to specify that the JWS file implements a callback service:

```
@CallbackService
public interface CallbackInterface {
```

- Create a method that the target Web Service explicitly invokes; this is the method that automatically sends a message back to the client service that originally invoked the target Web Service. Because this is a Java interface file, you do not provide an implementation of this method. Rather, the WebLogic Web Services runtime generates an implementation of the method via the `jwsc` Ant task.

```
public void callbackOperation (String msg);
```

Note: Although the example shows the callback method returning void and annotated with the `@Oneway` annotation, this is not a requirement.

See [JWS Annotation Reference](#) for additional information about the WebLogic-specific JWS annotations discussed in this section.

Updating the build.xml File for the Client Web Service

When you run the `jwsc` Ant task against the JWS file that implements the client Web Service, the task implicitly also generates the callback Web Service, as described in this section.

You update a `build.xml` file to generate a client Web Service that invokes the target Web Service by adding `taskdefs` and a `build-clientService` 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-clientService">
  <jwsc
    srcdir="src"
    destdir="${clientService-ear-dir}" >
    <jws file="examples/webservices/callback/CallbackClientImpl.java" >
      <clientgen
        wsdl="http://${wls.hostname}:${wls.port}/callback/TargetService?WSDL"
        packageName="examples.webservices.callback"
        serviceName="TargetService" />
      </jws>
    </jwsc>
  </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 a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `TargetService` 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 `CallbackClientImpl` JWS file imports and uses one of the generated classes.

Because the WSDL of the target Web Service includes an additional `<service>` element that describes the callback Web Service (which the target Web Service invokes), the `<clientgen>` child element of the `jwsc` Ant task also generates and compiles the callback Web Service and packages it in the same EAR file as the client Web Service.

Creating Conversational Web Services

The following sections describe how to create a conversational Web Service:

- [“Overview of Conversational Web Services” on page 4-1](#)
- [“Creating a Conversational Web Service: Main Steps” on page 4-3](#)
- [“Programming Guidelines for the Conversational JWS File” on page 4-5](#)
- [“Programming Guidelines for the JWS File That Invokes a Conversational Web Service” on page 4-8](#)
- [“ConversationUtils Utility Class” on page 4-11](#)
- [“Updating the build.xml File for a Client of a Conversational Web Service” on page 4-11](#)
- [“Updating a Stand-Alone Java Client to Invoke a Conversational Web Service” on page 4-12](#)
- [“Client Considerations When Redeploying a Conversational Web Service” on page 4-14](#)

WARNING: This feature can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

Overview of Conversational Web Services

A Web Service and the client application that invokes it may communicate multiple times to complete a single task. Also, multiple client applications might communicate with the same Web

Service at the same time. *Conversations* provide a straightforward way to keep track of data between calls and to ensure that the Web Service always responds to the correct client.

Conversations meet two challenges inherent in persisting data across multiple communications:

- Conversations uniquely identify a two-way communication between one client application and one Web Service so that messages are always returned to the correct client. For example, in a shopping cart application, a conversational Web Service keeps track of which shopping cart belongs to which customer. A conversational Web Service implements this by creating a unique conversation ID each time a new conversation is started with a client application.
- Conversations maintain state between calls to the Web Service; that is, they keep track of the data associated with a particular client application between its calls to the service. Conversations ensure that the data associated with a particular client is saved until it is no longer needed or the operation is complete. For example, in a shopping cart application, a conversational Web Service remembers which items are in the shopping cart while the customer continues shopping. Maintaining state is also needed to handle failure of the computer hosting the Web Service in the middle of a conversation; all state-related data is persisted to disk so that when the computer comes up it can continue the conversation with the client application.

WebLogic Server manages this unique ID and state by creating a conversation context each time a client application initiates a new conversation. The Web Service then uses the context to correlate calls to and from the service and to persist its state-related data.

Conversations between a client application and a Web Service have three distinct phases:

- **Start**—A client application initiates a conversation by invoking the start operation of the conversational Web Service. The Web Service in turn creates a new conversation context and an accompanying unique ID, and starts an internal timer to measure the idle time and the age of the conversation.
- **Continue**—After the client application has started the conversation, it invokes one or more continue operations to continue the conversation. The conversational Web Service uses the ID associated with the invoke to determine which client application it is conversing with, what state to persist, and which idle timer to reset. A typical continue operation would be one that requests more information from the client application, requests status, and so on.
- **Finish**—A client application explicitly invokes the finish operation when it has finished its conversation; the Web Service then marks any data or resources associated with the conversation as deleted.

Conversations typically occur between two WebLogic Web Services: one is marked conversational and defines the start, continue, and finish operations and the other Web Service uses the `@ServiceClient` annotation to specify that it is a client of the conversational Web Service. You can also invoke a conversational Web Service from a stand-alone Java client, although there are restrictions.

As with other WebLogic Web Service features, you use JWS annotations to specify that a Web Service is conversational.

WARNING: The client Web Service that invokes a conversational Web Service is not required to also be conversational. However, if the client is *not* conversational, there is a danger of multiple instances of this client accessing the same conversational Web Service stub and possibly corrupting the saved conversational state. If you believe this might true in your case, then specify that the client Web Service also be conversational. In this case you cannot use a stand-alone Java client, because there is no way to mark it as conversational using the WebLogic APIs.

Caution: A conversational Web Service on its own does not guarantee message delivery or that the messages are delivered in order, exactly once. If you require this kind of message delivery guarantee, you must also specify that the Web Service be reliable. See [Chapter 2, “Using Web Services Reliable Messaging,”](#) and [Chapter 7, “Using the Asynchronous Features Together.”](#)

Creating a Conversational Web Service: Main Steps

The following procedure describes how to create a conversational Web Service, as well as a client Web Service and stand-alone Java client application, both of which initiate and conduct a conversation. The procedure shows how to create the JWS files that implement the two Web Services 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 generated conversational Web Service. It is further assumed that you have a similar setup for the WebLogic Server instance that hosts the client Web Service that initiates the conversation. For more information, see:

- [Common Web Services Use Cases and Examples](#)
- [Iterative Development of WebLogic Web Services](#)
- [Programming the JWS File](#)

- [Invoking Web Services](#)

1. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that implements the conversational Web Service.

See [“Programming Guidelines for the Conversational JWS File”](#) on page 4-5.

2. Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the conversational JWS file into a Web Service.

See [Running the jwsc WebLogic Web Services Ant Task](#).

3. Run the Ant target to build the conversational Web Service. For example:

```
prompt> ant build-mainService
```

4. Deploy the Web Service as usual.

See [Deploying and Undeploying WebLogic Web Services](#).

5. If the client application is a stand-alone Java client, see [“Updating a Stand-Alone Java Client to Invoke a Conversational Web Service”](#) on page 4-12. If the client application is itself a Web Service, follow these steps:

- a. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that implements the client Web Service that initiates and conducts the conversation with the conversational Web Service. It is assumed that the client Web Service is deployed to a different WebLogic Server instance from the one that hosts the conversational Web Service.

See [“Programming Guidelines for the JWS File That Invokes a Conversational Web Service”](#) on page 4-8.

- b. Update the `build.xml` file that builds the client Web Service.

See [“Updating the build.xml File for a Client of a Conversational Web Service”](#) on page 4-11.

- c. Run the Ant target to build the client Web Service:

```
prompt> ant build-clientService
```

- d. Deploy the Web Service as usual.

See [Deploying and Undeploying WebLogic Web Services](#).

Programming Guidelines for the Conversational JWS File

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

```
package examples.webservices.conversation;

import java.io.Serializable;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Conversation;
import weblogic.jws.Conversational;
import weblogic.jws.Context;

import weblogic.wsee.jws.JwsContext;
import weblogic.wsee.jws.ServiceHandle;

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

@Conversational(maxIdleTime="10 minutes",
               maxAge="1 day",
               runAsStartUser=false,
               singlePrincipal=false )

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

@WLHttpTransport(contextPath="conv",
                 serviceUri="ConversationalService",
                 portName="ConversationalServicePort")

/**
 * Conversational Web Service.
 */

public class ConversationalServiceImpl implements Serializable {

    @Context
    private JwsContext ctx;
    public String status = "undefined";

    @WebMethod
    @Conversation (Conversation.Phase.START)
    public String start() {

        ServiceHandle handle = ctx.getService();
        String convID = handle.getConversationID();
```

```

        status = "start";
        return "Starting conversation, with ID " + convID + " and status equal to "
+ status;
    }

    @WebMethod
    @Conversation (Conversation.Phase.CONTINUE)
    public String middle(String message) {

        status = "middle";
        return "Middle of conversation; the message is: " + message + " and status
is " + status;
    }

    @WebMethod
    @Conversation (Conversation.Phase.FINISH)
    public String finish(String message ) {

        status = "finish";
        return "End of conversation; the message is: " + message + " and status is
" + status;
    }
}

```

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

- Conversational Web Services must implement `java.io.Serializable`, so you must first import the class into your JWS file:

```
import java.io.Serializable;
```

- Import the conversational JWS annotations:

```
import weblogic.jws.Conversation;
import weblogic.jws.Conversational;
```

- If you want to access runtime information about the conversational Web Service, import the `@Context` annotation and context APIs:

```
import weblogic.jws.Context;

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

See [Accessing Runtime Information about a Web Service Using the JwsContext](#) for more information about the runtime Web Service context.

- Use the class-level `@Conversational` annotation to specify that the Web Service is conversational. Although this annotation is optional (assuming you *are* specifying the `@Conversation` method-level annotation), it is a best practice to always use it in your JWS file to clearly specify that your Web Service is conversational.

Specify any of the following optional attributes: `maxIdleTime` is the maximum amount of time that the Web Service can be idle before WebLogic Server finishes the conversation; `maxAge` is the maximum age of the conversation; `runAsStartUser` indicates whether the continue and finish phases of an existing conversation are run as the user who started the conversation; and `singlePrincipal` indicates whether users other than the one who started a conversation are allowed to execute the continue and finish phases of the conversation.

```
@Conversational(maxIdleTime="10 minutes",
                maxAge="1 day",
                runAsStartUser=false,
                singlePrincipal=false )
```

If a JWS file includes the `@Conversational` annotation, all operations of the Web Service are conversational. The default phase of an operation, if it does not have an explicit `@Conversation` annotation, is `continue`. However, because a conversational Web Service is required to include at least one start and one finish operation, you *must* use the method-level `@Conversation` annotation to specify which methods implement these operations.

See [weblogic.jws.Conversational](#) for additional information and default values for the attributes.

- Your JWS file must implement `java.io.Serializable`:

```
public class ConversationalServiceImpl implements Serializable {
```

- To access runtime information about the Web Service, annotate a private class variable, of data type `weblogic.wsee.jws.JwsContext`, with the field-level `@Context` JWS annotation:

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

- Use the `@Conversation` annotation to specify the methods that implement the start, continue, and finish phases of your conversation. A conversation is required to have at least one start and one finish operation; the continue operation is optional. Use the following parameters to the annotation to specify the phase:
`Conversation.Phase.START`, `Conversation.Phase.CONTINUE`, or `Conversation.Phase.FINISH`. The following example shows how to specify the start operation:

```
@WebMethod
@Conversation (Conversation.Phase.START)
public String start() {...
```

If you mark just one method of the JWS file with the `@Conversation` annotation, then the entire Web Service becomes conversational and each operation is considered part of the conversation; this is true even if you have not used the optional class-level `@Conversational` annotation in your JWS file. Any methods not explicitly annotated with `@Conversation` are, by default, continue operations. This means that, for example, if a client application invokes one of these continue methods without having previously invoked a start operation, the Web Service returns a runtime error.

Finally, if you plan to invoke the conversational Web Service from a stand-alone Java client, the start operation is required to be request-response, or in other words, it *cannot* be annotated with the `@Oneway` JWS annotation. The operation can return `void`. If you are going to invoke the Web Service only from client applications that run in WebLogic Server, then this requirement does not apply.

See [weblogic.jws.Conversation](#) for additional information.

- Use the `JwsContext` instance to get runtime information about the Web Service.

For example, the following code in the start operation gets the ID that WebLogic Server assigns to the new conversation:

```
ServiceHandle handle = ctx.getService();
String convID = handle.getConversationID();
```

See [Accessing Runtime Information about a Web Service Using the JwsContext](#) for detailed information on using the context-related APIs.

Programming Guidelines for the JWS File That Invokes a Conversational Web Service

The following example shows a simple JWS file for a Web Service that invokes the conversational Web Service described in “[Programming Guidelines for the Conversational JWS File](#)” on page 4-5; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package examples.webservices.conversation;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;

import weblogic.wsee.conversation.ConversationUtils;
```

Programming Guidelines for the JWS File That Invokes a Conversational Web Service

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

import javax.xml.rpc.Stub;

import examples.webservices.conversation.ConversationalPortType;

import java.rmi.RemoteException;

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

@WLHttpTransport(contextPath="convClient",
                 serviceUri="ConversationalClient",
                 portName="ConversationalClientPort")

/**
 * client that has a conversation with the ConversationalService.
 */

public class ConversationalClientImpl {

    @ServiceClient(
        wsdlLocation="http://localhost:7001/conv/ConversationalService?WSDL",
        serviceName="ConversationalService",
        portName="ConversationalServicePort")

    private ConversationalPortType port;

    @WebMethod
    public void runConversation(String message) {

        try {

            // Invoke start operation
            String result = port.start();
            System.out.println("start method executed.");
            System.out.println("The message is: " + result);

            // Invoke continue operation
            result = port.middle(message );
            System.out.println("middle method executed.");
            System.out.println("The message is: " + result);

            // Invoke finish operation
            result = port.finish(message );
            System.out.println("finish method executed.");
            System.out.println("The message is: " + result);
            ConversationUtils.renewStub((Stub)port);
        }
    }
}
```

```

    }
    catch (RemoteException e) {
        e.printStackTrace();
    }
}
}

```

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

- Import the `@ServiceClient` JWS annotation:

```
import weblogic.jws.ServiceClient;
```

- Optionally import the WebLogic utility class for further configuring a conversation:

```
import weblogic.wsee.conversation.ConversationUtils;
```

- Import the JAX-RPC stub of the port type of the conversational Web Service you want to invoke. The actual stub itself will be created later by the `jwsc` Ant task. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of `<jws>`, and the name of the stub is determined by the WSDL of the invoked Web Service.

```
import examples.webservices.conversation.ConversationalPortType;
```

- In the body of the JWS file, use the `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the conversational Web Service you want to invoke. You specify this annotation at the field-level on a private variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.

```

@ServiceClient(

    wsdlLocation="http://localhost:7001/conv/ConversationalService?WSDL",
    serviceName="ConversationalService",
    portName="ConversationalServicePort")

    private ConversationalPortType port;

```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the start operation of the conversational Web Service to start the conversation. You can invoke the start method from any location in the JWS file (constructor, method, and so on):

```
String result = port.start();
```

- Optionally invoke the continue methods to continue the conversation. Be sure you use the same stub instance so that you continue the same conversation you started:

```
result = port.middle(message );
```


- Once the conversation is completed, invoke the finish operation so that the conversational Web Service can free up the resources it used for the current conversation:

```
result = port.finish(message );
```

- If you want to reuse the Web Service conversation stub to start a new conversation, you must explicitly renew the stub using the `renewStub()` method of the `weblogic.wsee.conversation.ConversationUtils` utility class:

```
ConversationUtils.renewStub((Stub)port);
```

WARNING: The client Web Service that invokes a conversational Web Service is not required to also be conversational. However, if the client is *not* conversational, there is a danger of multiple instances of this client accessing the same conversational Web Service stub and possibly corrupting the saved conversational state. If you believe this might true in your case, then specify that the client Web Service also be conversational.

ConversationUtils Utility Class

WebLogic Server provides a utility class for use with the conversation feature. Use this class to perform common tasks such as getting and setting the conversation ID and setting configuration options. Some of these tasks are performed in the conversational Web Service, some are performed in the client that invokes the conversational Web Service. See [“Programming Guidelines for the JWS File That Invokes a Conversational Web Service” on page 4-8](#) for an example of using this class.

See `weblogic.wsee.conversation.ConversationUtils` for details.

Updating the build.xml File for a Client of a Conversational Web Service

You update a `build.xml` file to generate the JWS file that invokes a conversational Web Service by adding `taskdefs` and a `build-clientService` 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-clientService">

    <jwsc
        enableAsyncService="true"
```

```

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

        <jws
file="examples/webservices/conversation/ConversationalClientImpl.java" >
        <clientgen

wsdl="http://${wls.hostname}:${wls.port}/conv/ConversationalService?WSDL"
        packageName="examples.webservices.conversation"/>

        </jws>

    </jws>

</target>

```

Use the `taskdef` Ant task to define the full classname of the `jws` Ant tasks.

Update the `jws` Ant task that compiles the client Web Service to include a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `ConversationalService` Web Service. The `jws` 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 `ConversationalClientImpl` JWS file imports and uses one of the generated classes.

Updating a Stand-Alone Java Client to Invoke a Conversational Web Service

The following example shows a simple stand-alone Java client that invokes the conversational Web Service described in [“Programming Guidelines for the Conversational JWS File” on page 4-5](#). See the explanation after the example for coding guidelines that correspond to the Java code in bold.

```

package examples.webservices.conv_standalone.client;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

import weblogic.wsee.jaxrpc.WLStub;

```

```

/**
 * stand-alone client that invokes and converses with ConversationlService.
 */

public class Main {

    public static void main(String[] args)
        throws ServiceException, RemoteException{

        ConversationalService service = new ConversationalService_Impl(args[0] +
"?WSDL");
        ConversationalPortType port = service.getConversationalServicePort();

        // Set property on stub to specify that client is invoking a Web Service
        // that uses advanced features; this property is automatically set if
        // the client runs in a WebLogic Server instance.

        Stub stub = (Stub)port;
        stub._setProperty(WLStub.COMPLEX, "true");

        // Invoke start operation to begin the conversation
        String result = port.start();
        System.out.println("start method executed.");
        System.out.println("The message is: " + result);

        // Invoke continue operation
        result = port.middle("middle" );
        System.out.println("middle method executed.");
        System.out.println("The message is: " + result);

        // Invoke finish operation
        result = port.finish("finish" );
        System.out.println("finish method executed.");
        System.out.println("The message is: " + result);

    }
}

```

Follow these guidelines when programming the stand-alone Java client that invokes a conversational Web Service. Code snippets of the guidelines are shown in bold in the preceding example.

- Import the `weblogic.wsee.jaxrpc.WLStub` class:

```
import weblogic.wsee.jaxrpc.WLStub;
```
- Set the `WLStub.Complex` property on the JAX-RPC stub of the `ConversationalService` using the `_setProperty` method:

```
Stub stub = (Stub)port;  
stub._setProperty(WLStub.COMPLEX, "true");
```

This property specifies to the Web Services runtime that the client is going to invoke an advanced Web Service, in this case a conversational one. This property is automatically set when invoking a conversational Web Service from another WebLogic Web Service.

- Invoke the start operation of the conversational Web Service to start the conversation:

```
String result = port.start();
```

- Optionally invoke the continue methods to continue the conversation:

```
result = port.middle(message );
```

- Once the conversation is completed, invoke the finish operation so that the conversational Web Service can free up the resources it used for the current conversation:

```
result = port.finish(message );
```

Client Considerations When Redeploying a Conversational Web Service

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

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

For additional information about production redployment and Web Service clients, see [Client Considerations When Redeploying a Web Service](#).

Creating Buffered Web Services

The following sections describe how to create a buffered Web Service:

- [“Overview of Buffered Web Services” on page 5-1](#)
- [“Creating a Buffered Web Service: Main Steps” on page 5-2](#)
- [“Configuring the Host WebLogic Server Instance for the Buffered Web Service” on page 5-3](#)
- [“Programming Guidelines for the Buffered JWS File” on page 5-4](#)
- [“Programming the JWS File That Invokes the Buffered Web Service” on page 5-6](#)
- [“Updating the build.xml File for a Client of the Buffered Web Service” on page 5-7](#)

WARNING: This feature can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

Overview of Buffered Web Services

When a buffered operation is invoked by a client, the method operation goes on a JMS queue and WebLogic Server deals with it asynchronously. As with Web Service reliable messaging, if WebLogic Server goes down while the method invocation is still in the queue, it will be dealt with as soon as WebLogic Server is restarted. When a client invokes the buffered Web Service, the client does not wait for a response from the invoke, and the execution of the client can continue.

Creating a Buffered Web Service: Main Steps

The following procedure describes how to create a buffered Web Service and a client Web Service that invokes an operation of the buffered Web Service. The procedure shows how to create the JWS files that implement the two Web Services from scratch. If you want to update existing JWS files, use this procedure as a guide. The procedure also shows how to configure the WebLogic Server instance that hosts the buffered Web Service.

Note: Unless you are also using the asynchronous request-response feature, you do not need to invoke a buffered Web Service from another Web Service, you can also invoke it from a stand-alone Java application.

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 generated buffered Web Service. It is further assumed that you have a similar setup for the WebLogic Server instance that hosts the client Web Service that invokes the buffered Web Service. For more information, see:

- [Common Web Services Use Cases and Examples](#)
- [Iterative Development of WebLogic Web Services](#)
- [Programming the JWS File](#)
- [Invoking Web Services](#)

1. Configure the WebLogic Server instance that hosts the buffered Web Service.

See “[Configuring the Host WebLogic Server Instance for the Buffered Web Service](#)” on [page 5-3](#).

2. Create a new JWS file, or update an existing one, that will implement the buffered Web Service.

See “[Programming Guidelines for the Buffered JWS File](#)” on [page 5-4](#).

3. Update the `build.xml` file to include a call to the `jwsc` Ant task to compile the JWS file into a buffered Web Service; for example:

```
<jwsc
  srcdir="src"
  destdir="${service-ear-dir}" >
  <jws
    file="examples/webservices/async_buffered/AsyncBufferedImpl.java"
  />
</jwsc>
```

See [Running the jwsc WebLogic Web Services Ant Task](#) for general information about using the `jwsc` task.

4. Recompile your destination JWS file by calling the appropriate target and deploying the Web Service to WebLogic Server. For example:

```
prompt> ant build-mainService deploy-mainService
```

5. Create a new JWS file, or update an existing one, that implements the client Web Service that invokes the buffered Web Service.

See [“Programming the JWS File That Invokes the Buffered Web Service”](#) on page 5-6.

6. Update the `build.xml` file that builds the client Web Service.

See [“Updating the build.xml File for a Client of the Buffered Web Service”](#) on page 5-7.

7. Recompile your client JWS file by calling the appropriate target, then redeploy the Web Service to the client WebLogic Server. For example:

```
prompt> ant build-clientService deploy-clientService
```

Configuring the Host WebLogic Server Instance for the Buffered Web Service

Configuring the WebLogic Server instance on which the buffered Web Service is deployed involves configuring JMS resources, such as JMS servers and modules, that are used internally by the Web Services runtime.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see [Configuring Your Domain For Web Services Features](#).

If, however, you prefer to configure the resources yourself, use the following high-level procedure which lists the tasks and then points to the Administration Console Online Help for details on performing the tasks.

1. Invoke the Administration Console for the domain that contains the WebLogic Server instance that hosts the buffered Web Service in your browser.

See [Invoking the Administration Console](#) for instructions on the URL that invokes the Administration Console.

2. Create a JMS Server. You can use an existing one if you do not want to create a new one.

See [Create JMS servers](#).

3. Create a JMS module that contains a JMS queue. Target the JMS queue to the JMS server you created in the preceding step. Be sure you specify that this JMS queue is local, typically by setting the local JNDI name.

If you want the buffered Web Service to use the default Web Services queue, set the JNDI name of the JMS queue to `weblogic.wsee.DefaultQueue`. Otherwise, if you use a different JNDI name, be sure to use the `@BufferQueue` annotation in the JWS file to specify this JNDI name to the reliable Web Service. See [“Programming Guidelines for the Buffered JWS File” on page 5-4](#).

If you are using the buffered Web Service feature in a cluster, you must still create a local queue rather than a distributed queue. In addition, you must explicitly target this queue to each server in the cluster.

See [Create JMS modules](#) and [Create queues](#).

Programming Guidelines for the Buffered JWS File

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

```
package examples.webservices.buffered;

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

import weblogic.jws.WLHttpTransport;
import weblogic.jws.MessageBuffer;
import weblogic.jws.BufferQueue;

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

@WLHttpTransport(contextPath="buffered",
                 serviceUri="BufferedService",
                 portName="BufferedPort")

// Annotation to specify a specific JMS queue rather than the default
@BufferQueue(name="my.jms.queue")
```



```
/**
 * Simple buffered Web Service.
 */

public class BufferedImpl {

    @WebMethod()
    @MessageBuffer(retryCount=10, retryDelay="10 seconds")
    @Oneway()

    public void sayHelloNoReturn(String message) {
        System.out.println("sayHelloNoReturn: " + message);
    }
}
```

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

- Import the JWS annotations used for buffered Web Services:

```
import javax.jws.Oneway;

import weblogic.jws.MessageBuffer;
import weblogic.jws.BufferQueue;
```

See the following bullets for guidelines on which JWS annotations are required.

- Optionally use the class-level `@BufferQueue` JWS annotation to specify the JNDI name of the JMS queue used internally by WebLogic Server when it processes a buffered invoke; for example:

```
@BufferQueue(name="my.jms.queue")
```

If you do not specify this JWS annotation, then WebLogic Server uses the default Web Services JMS queue (`weblogic.wsee.DefaultQueue`).

You must create both the default JMS queue and any queues specified with this annotation before you can successfully invoke a buffered operation. See [“Configuring the Host WebLogic Server Instance for the Buffered Web Service” on page 5-3](#) for details.

- Use the `@MessageBuffer` JWS annotation to specify the operations of the Web Service that are buffered. The annotation has two optional attributes:
 - `retryCount`: The number of times WebLogic Server should attempt to deliver the message from the JMS queue to the Web Service implementation (default 3).

- `retryDelay`: The amount of time that the server should wait in between retries (default 5 minutes).

For example:

```
@MessageBuffer(retryCount=10, retryDelay="10 seconds")
```

You can use this annotation at the class-level to specify that all operations are buffered, or at the method-level to choose which operations are buffered.

- If you plan on invoking the buffered Web Service operation synchronously (or in other words, *not* using the asynchronous request-response feature), then the implementing method is required to be annotated with the `@Oneway` annotation to specify that the method is one-way. This means that the method cannot return a value, but rather, must explicitly return `void`. For example:

```
@Oneway()  
public void sayHelloNoReturn(String message) {
```

Conversely, if the method is *not* annotated with the `@Oneway` annotation, then you must invoke it using the asynchronous request-response feature. If you are unsure how the operation is going to be invoked, consider creating two flavors of the operation: synchronous and asynchronous.

See [Chapter 6, “Invoking a Web Service Using Asynchronous Request-Response,”](#) and [Chapter 7, “Using the Asynchronous Features Together.”](#)

Programming the JWS File That Invokes the Buffered Web Service

You can invoke a buffered Web Service from both a stand-alone Java application (if not using asynchronous request-response) and from another Web Service. Unlike other WebLogic Web Services asynchronous features, however, you do not use the `@ServiceClient` JWS annotation in the client Web Service, but rather, you invoke the service as you would any other. For details, see [Invoking a Web Service from Another Web Service](#).

The following sample JWS file shows how to invoke the `sayHelloNoReturn` operation of the `BufferedService` Web Service:

```
package examples.webservices.buffered;  
  
import java.rmi.RemoteException;  
import javax.xml.rpc.ServiceException;
```

Updating the build.xml File for a Client of the Buffered Web Service

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

import weblogic.jws.WLHttpTransport;

import examples.webservices.buffered.BufferedPortType;
import examples.webservices.buffered.BufferedService_Impl;
import examples.webservices.buffered.BufferedService;

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

@WLHttpTransport(contextPath="bufferedClient",
                 serviceUri="BufferedClientService",
                 portName="BufferedClientPort")

public class BufferedClientImpl {

    @WebMethod()
    public String callBufferedService(String input, String serviceUrl)
        throws RemoteException {

        try {

            BufferedService service = new BufferedService_Impl(serviceUrl + "?WSDL");
            BufferedPortType port = service.getBufferedPort();

            // Invoke the sayHelloNoReturn() operation of BufferedService
            port.sayHelloNoReturn(input);

            return "Invoke went okay!";

        } catch (ServiceException se) {

            System.out.println("ServiceException thrown");
            throw new RuntimeException(se);

        }
    }
}
```

Updating the build.xml File for a Client of the Buffered Web Service

To update a build.xml file to generate the JWS file that invokes a buffered Web Service operation, add taskdefs and a build-clientService targets that look 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-clientService">
  <jwsc
    enableAsyncService="true"
    srcdir="src"
    destdir="${clientService-ear-dir}" >
    <jws file="examples/webservices/buffered/BufferedClientImpl.java">
      <clientgen
        wsdl="http://${wls.hostname}:${wls.port}/buffered/BufferedService?WSDL"
        packageName="examples.webservices.buffered"/>
      </jws>
    </jwsc>
  </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 a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `BufferedService` 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 `BufferedClientImpl` JWS file imports and uses one of the generated classes.

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” on page 6-1](#)
- [“Using Asynchronous Request-Response: Main Steps” on page 6-2](#)
- [“Writing the Asynchronous JWS File” on page 6-4](#)
- [“Updating the build.xml File When Using Asynchronous Request-Response” on page 6-9](#)
- [“Disabling The Internal Asynchronous Service” on page 6-10](#)
- [“Using Asynchronous Request Response With a Proxy Server” on page 6-10](#)

WARNING: This feature can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

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 might be adequate. 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.

You invoke a Web Service asynchronously only from a client running in a WebLogic Web Service, never from a stand alone client application. The invoked Web Service does not change in any way, thus you can invoke any deployed Web Service (both WebLogic and non-WebLogic) asynchronously as long as the application server that hosts the Web Service supports the [WS-Addressing](#) specification.

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 `jwsc` Ant task.) For example, rather than invoking an operation called `getQuote` directly, you would invoke `getQuoteAsync` 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. You use both naming conventions and JWS annotations to specify these methods to the JWS compiler. For example, if the asynchronous operation is called `getQuoteAsync`, then these methods might be called `onGetQuoteAsyncResponse` and `onGetQuoteAsyncFailure`.

Note: For information about using asynchronous request-response with other asynchronous features, such as Web Service reliable messaging or buffering, see [Chapter 7, “Using the Asynchronous Features Together.”](#) This section describes how to use the asynchronous request-response feature on its own.

Note: The asynchronous request-response feature works only with HTTP; you cannot use it with the HTTPS or JMS transport.

Using Asynchronous Request-Response: Main Steps

The following procedure describes how to create a client Web Service that asynchronously invokes an operation in a different Web Service. The procedure shows how to create the JWS file that implements the client Web Service from scratch; if you want to update an existing JWS file, use this procedure as a guide.

For clarity, it is assumed in the procedure that:

- The client Web Service is called `StockQuoteClientService`.

- The `StockQuoteClientService` service is going to invoke the `getQuote(String)` operation of the already-deployed `StockQuoteService` service whose WSDL is found at the following URL:

`http://localhost:7001/async/StockQuote?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. See:

- [Common Web Services Use Cases and Examples](#)
- [Iterative Development of WebLogic Web Services](#)
- [Programming the JWS File](#)
- [Invoking Web Services](#)

1. Using your favorite IDE or text editor, create a new JWS file, or update an existing one, that implements the `StockQuoteClientService` Web Service.

See [“Writing the Asynchronous JWS File” on page 6-4](#).

2. Update your `build.xml` file to compile the JWS file that implements the `StockQuoteClientService`. You will add a `<clientgen>` child element to the `jwsc` Ant task so as to automatically generate the asynchronous flavor of the Web Service operations you are invoking.

See [“Updating the build.xml File When Using Asynchronous Request-Response” on page 6-9](#).

3. Run the Ant target to build the `StockQuoteClientService`:

```
prompt> ant build-clientService
```

4. Deploy the `StockQuoteClientService` Web Service as usual.

See [Deploying and Undeploying WebLogic Web Services](#).

When you invoke the `StockQuoteClientService` Web Service, which in turn invokes the `StockQuoteService` Web Service, the second invoke will be asynchronous rather than synchronous.

Writing the Asynchronous JWS File

The following example shows a simple JWS file that implements a Web Service called `StockQuoteClient` that has a single method, `asyncOperation`, that in turn asynchronously invokes the `getQuote` method of the `StockQuote` service. The Java code in bold is described [“Coding Guidelines for Invoking a Web Service Asynchronously” on page 6-5](#). See [“Example of a Synchronous Invoke” on page 6-8](#) to see how the asynchronous invoke differs from a synchronous invoke of the same operation.

```
package examples.webservices.async_req_res;

import weblogic.jws.WLHttpTransport;

import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;

import weblogic.wsee.async.AsyncPreCallContext;
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPostCallContext;

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

import examples.webservices.async_req_res.StockQuotePortType;

import java.rmi.RemoteException;

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

@WLHttpTransport(contextPath="asyncClient",
                 serviceUri="StockQuoteClient",
                 portName="StockQuoteClientServicePort")

/**
 * Client Web Service that invokes the StockQuote Service asynchronously.
 */

public class StockQuoteClientImpl {

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

    private StockQuotePortType port;

    @WebMethod
    public void asyncOperation (String symbol) throws RemoteException {
```



```

AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
apc.setProperty("symbol", symbol);

try {
    port.getQuoteAsync(apc, symbol );
    System.out.println("in getQuote method of StockQuoteClient WS");

} catch (RemoteException re) {

    System.out.println("RemoteException thrown");
    throw new RuntimeException(re);
}

}

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

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

}

```

Coding Guidelines for Invoking a Web Service Asynchronously

The following guidelines for invoking an operation asynchronously correspond to the Java code shown in bold in the example described in [“Writing the Asynchronous JWS File” on page 6-4](#). These guidelines are in addition to the standard ones for creating JWS files. See [“Example of a Synchronous Invoke” on page 6-8](#) to see how the asynchronous invoke differs from a synchronous invoke of the same operation.

To invoke an operation asynchronously in your JWS file:

- Import the following WebLogic-specific JWS annotations related to the asynchronous request-response feature:

```

import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;

```

- Import the JAX-RPC stub, created later by the `jwsc` Ant task, of the port type of the Web Service you want to invoke. The stub package is specified by the `packageName` attribute of the `<clientgen>` child element of `jwsc`, and the name of the stub is determined by the WSDL of the invoked Web Service.

```
import examples.webservices.async_req_res.StockQuotePortType;
```

- Import the asynchronous pre- and post-call context WebLogic APIs:

```
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPreCallContext;
import weblogic.wsee.async.AsyncPostCallContext;
```

The `AsyncPreCallContext` and `AsyncPostCallContext` APIs describe asynchronous contexts that you can use for a variety of reasons in your Web Service: to set a property in the pre-context so that the method that handles the asynchronous response can distinguish between different asynchronous calls; to set and get contextual variables, such as the name of the user invoking the operation, their password, and so on; to get the name of the JAX-RPC stub that invoked a method asynchronously; and to set a time-out interval on the context.

See [Javadocs](#) for additional reference information about these APIs.

- In the body of the JWS file, use the required `@ServiceClient` JWS annotation to specify the WSDL, name, and port of the Web Service you will be invoking asynchronously. You specify this annotation at the field-level on a variable, whose data type is the JAX-RPC port type of the Web Service you are invoking.

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

private StockQuotePortType port;
```

When you annotate a variable (in this case, `port`) with the `@ServiceClient` annotation, the Web Services runtime automatically initializes and instantiates the variable, preparing it so that it can be used to invoke another Web Service asynchronously.

- In the method of the JWS file which is going to invoke the `getQuote` operation asynchronously, get a pre-call asynchronous context using the context factory:

```
AsyncPreCallContext apc =
    AsyncCallContextFactory.getAsyncPreCallContext();
```

- Use the `setProperty` method of the pre-call context to create a property whose name and value is the same as the parameter to the `getQuote` method:

```
apc.setProperty("symbol", symbol);
```

- Using the stub you annotated with the `@ServiceClient` annotation, invoke the operation (in this case, `getQuote`). Instead of invoking it directly, however, invoke the asynchronous flavor of the operation, which has `Async` added on to the end of its name. The asynchronous flavor always returns `void`. Pass the asynchronous context as the first parameter:

```
port.getQuoteAsync(apc, symbol);
```

- For each operation you will be invoking asynchronously, create a method called `onOperationnameAsyncResponse`, where *Operationname* refers to the name of the operation, with initial letter always capitalized. The method must return `void`, and have two parameters: the post-call asynchronous context and the return value of the operation you are invoking. Annotate the method with the `@AsyncResponse` JWS annotation; use the `target` attribute to specify the variable whose datatype is the JAX-RPC stub and the `operation` attribute to specify the name of the operation you are invoking asynchronously. Inside the body of the method, put the business logic that processes the value returned by the operation.

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

- For each operation you will be invoking asynchronously, create a method called `onOperationnameAsyncFailure`, where *Operationname* refers to the name of the operation, with initial letter capitalized. The method must return `void`, and have two parameters: the post-call asynchronous context and a `Throwable` object, the superclass of all exceptions to handle any type of exception thrown by the invoked operation. Annotate the method with the `@AsyncFailure` JWS annotation; use the `target` attribute to specify the variable whose datatype is the JAX-RPC stub and the `operation` attribute to specify the name of the operation you are invoking asynchronously. Inside the method, you can determine the exact nature of the exception and write appropriate Java code.

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

Note: You are not required to use the `@AsyncResponse` and `@AsyncFailure` annotations, although it is a good practice because it clears up any ambiguity and makes your JWS file clean and understandable. However, in the rare use case where you want one of the `onXXX` methods to handle the asynchronous response or failure from two (or more) stubs that are invoking operations from two different Web Services that have the same name, then you should explicitly NOT use these annotations. Be sure that the name of the `onXXX` methods follow the correct naming conventions exactly, as described above.

Example of a Synchronous Invoke

The following example shows a JWS file that invokes the `getQuote` operation of the `StockQuote` Web Service synchronously. The example is shown only so you can compare it with the corresponding asynchronous invoke shown in [“Writing the Asynchronous JWS File” on page 6-4](#).

```
package examples.webservices.async_req_res;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;

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

import java.rmi.RemoteException;

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

@WLHttpTransport(contextPath="syncClient",
                 serviceUri="SyncClient",
                 portName="SyncClientPort")

/**
 * Normal service-to-service client that invokes StockQuote service
 * synchronously.
 */

public class SyncClientImpl {

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

    @WebMethod
    public void nonAsyncOperation(String symbol) throws RemoteException {

        int quote = port.getQuote(symbol);
    }
}
```

```

        System.out.println("-----");
        System.out.println("Got quote " + quote );
        System.out.println("-----");
    }
}

```

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

To update a `build.xml` file to generate the JWS file that invokes a Web Service operation asynchronously, add `taskdefs` and a `build-clientService` target that looks something like the following; see the description after the example for details:

```

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

<target name="build-clientService">

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

        <jws
            file="examples/webservices/async_req_res/StockQuoteClientImpl.java" >

                <clientgen
                    wsdl="http://${wls.hostname}:${wls.port}/async/StockQuote?WSDL"
                    packageName="examples.webservices.async_req_res"/>

            </jws>

        </jwsc>

    </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 a `<clientgen>` child element of the `<jws>` element so as to generate and compile the JAX-RPC stubs for the deployed `StockQuote` 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. By default, the `jwsc` Ant task in this case generates both synchronous and asynchronous flavors of the Web Service

operations in the JAX-RPC stubs. You do this because the `StockQuoteClientImpl` JWS file imports and uses one of the generated classes.

Disabling The Internal Asynchronous Service

By default, every WebLogic Server instance deploys an internal asynchronous Web Service that handles the asynchronous request-response feature. To specify that you do *not* want to deploy this internal service, start the WebLogic Server instance using the

`-Dweblogic.wsee.skip.async.response=true` Java system property.

One reason for disabling the asynchronous service is if you use a WebLogic Server instance as a Web proxy to a WebLogic cluster. In this case, asynchronous messages will never get to the cluster, as required, because the asynchronous service on the proxy server consumes them instead. For this reason, you must disable the asynchronous service on the proxy server using the system property.

For details on specifying Java system properties to configure WebLogic Server, see [Specifying Java Options for a WebLogic Server Instance](#).

Using Asynchronous Request Response With a Proxy Server

Client applications that use the asynchronous request-response feature might not invoke the operation directly, but rather, use a proxy server. Reasons for using a proxy include the presence of a firewall or the deployment of the invoked Web Service to a cluster.

In this case, the WebLogic Server instance that hosts the invoked Web Service must be configured with the address and port of the proxy server. If your Web Service is deployed to a cluster, you must configure every server in the cluster.

For each server instance:

1. Create a network channel for the protocol you use to invoke the Web Service. You must name the network channel `weblogic-wsee-proxy-channel-xxx`, where `xxx` refers to the protocol. For example, to create a network channel for HTTPS, call it `weblogic-wsee-proxy-channel-https`.

See [Configure Custom Network Channels](#) for general information about creating a network channel.

2. Configure the network channel, updating the **External Listen Address** and **External Listen Port** fields with the address and port of the proxy server, respectively.

Using the Asynchronous Features Together

The following sections describe how to use the asynchronous features together:

- [“Using the Asynchronous Features Together” on page 7-1](#)
- [“Example of a JWS File That Implements a Reliable Conversational Web Service” on page 7-2](#)
- [“Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service” on page 7-4](#)

WARNING: These features can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

Using the Asynchronous Features Together

The preceding sections describe how to use the WebLogic Web Service asynchronous features (Web Service reliable messaging, conversations, asynchronous request-response, and buffering) on their own. Typically, however, Web Services use the features together; see [“Example of a JWS File That Implements a Reliable Conversational Web Service” on page 7-2](#) and [“Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service” on page 7-4](#) for examples.

When used together, some restrictions described in the individual feature sections do not apply, and sometimes additional restrictions apply.

- **Asynchronous request-response with Web Service reliable messaging or buffering—**
The asynchronous response from the reliable Web Service is also reliable. This means that you must also configure a JMS server, module, and queue on the *source* WebLogic Server instance, in a similar way you configured the destination WebLogic Server instance, to handle the response.

When you create the JMS queue on the source WebLogic Server instance, you are required to specify a JNDI name of `weblogic.wsee.DefaultQueue`; you can name the queue anything you want. You must also ensure that you specify that this JMS queue is *local*, typically by setting the local JNDI name.
- **Asynchronous request-response with Web Service reliable messaging or buffering—**
The reliable or buffered operation *cannot* be one-way; in other words, you cannot annotate the implementing method with the `@Oneway` annotation.
- **Asynchronous request-response with Web Service reliable messaging—**If you set a property in one of the asynchronous contexts (`AsyncPreCallContext` or `AsyncPostCallContext`), then the property must implement `java.io.Serializable`.
- **Asynchronous request-response with buffering—**You must use the `@ServiceClient` JWS annotation in the client Web Service that invokes the buffered Web Service operation.
- **Conversations with Web Service reliable messaging—**If you set the property `WLStub.CONVERSATIONAL_METHOD_BLOCK_TIMEOUT` on the stub of the client Web Service, the property is ignored because the client does not block.
- **Conversations with Web Service reliable messaging—**At least one method of the reliable conversational Web Service must *not* be marked with the `@Oneway` annotation.
- **Conversations with asynchronous request-response—**Asynchronous responses between a client conversational Web Service and any other Web Service also participate in the conversation. For example, assume `WebServiceA` is conversational, and it invokes `WebServiceB` using asynchronous request-response. Because `WebServiceA` is conversational the asynchronous responses from `WebServiceB` also participates in the same conversation.

Example of a JWS File That Implements a Reliable Conversational Web Service

The following sample JWS file implements a Web Service that is both reliable and conversational:

```
package examples.webservices.async_mega;
```


Example of a JWS File That Implements a Reliable Conversational Web Service

```
import java.io.Serializable;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.Conversation;
import weblogic.jws.Policy;

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

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

@Policy(uri="AsyncReliableConversationPolicy.xml",
        attachToWSDL=true)

@WLHttpTransport(contextPath="asyncMega",
                 serviceUri="AsyncMegaService",
                 portName="AsyncMegaServicePort")

/**
 * Web Service that is both reliable and conversational.
 */

public class AsyncMegaServiceImpl implements Serializable {

    @WebMethod
    @Conversation (Conversation.Phase.START)
    public String start() {
        return "Starting conversation";
    }

    @WebMethod
    @Conversation (Conversation.Phase.CONTINUE)
    public String middle(String message) {
        return "Middle of conversation; the message is: " + message;
    }

    @WebMethod
    @Conversation (Conversation.Phase.FINISH)
    public String finish(String message ) {
        return "End of conversation; the message is: " + message;
    }
}
```

Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service

The following JWS file shows how to implement a client Web Service that reliably invokes the various conversational methods of the Web Service described in [“Example of a JWS File That Implements a Reliable Conversational Web Service” on page 7-2](#); the client JWS file uses the asynchronous request-response feature as well.

```
package examples.webservices.async_mega;

import weblogic.jws.WLHttpTransport;
import weblogic.jws.ServiceClient;
import weblogic.jws.AsyncResponse;
import weblogic.jws.AsyncFailure;

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

import weblogic.wsee.async.AsyncPreCallContext;
import weblogic.wsee.async.AsyncCallContextFactory;
import weblogic.wsee.async.AsyncPostCallContext;

import examples.webservices.async_mega.AsyncMegaPortType;
import examples.webservices.async_mega.AsyncMegaService;
import examples.webservices.async_mega.AsyncMegaServiceImpl;

import java.rmi.RemoteException;

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

@WLHttpTransport(contextPath="asyncMegaClient",
                 serviceUri="AsyncMegaClient",
                 portName="AsyncMegaClientServicePort")

/**
 * Client Web Service that has a conversation with the AsyncMegaService
 * reliably and asynchronously.
 */

public class AsyncMegaClientImpl {

    @ServiceClient(
        wsdlLocation="http://localhost:7001/asyncMega/AsyncMegaService?WSDL",
        serviceName="AsyncMegaService",
        portName="AsyncMegaServicePort")

    private AsyncMegaPortType port;
```

Example of Client Web Service That Asynchronously Invokes a Reliable Conversational Web Service

```
@WebMethod
public void runAsyncReliableConversation(String message) {

    AsyncPreCallContext apc = AsyncCallContextFactory.getAsyncPreCallContext();
    apc.setProperty("message", message);

    try {
        port.startAsync(apc);
        System.out.println("start method executed.");

        port.middleAsync(apc, message );
        System.out.println("middle method executed.");

        port.finishAsync(apc, message );
        System.out.println("finish method executed.");
    }
    catch (RemoteException e) {
        e.printStackTrace();
    }
}

@AsyncResponse(target="port", operation="start")
public void onStartAsyncResponse(AsyncPostCallContext apc, String message) {
    System.out.println("-----");
    System.out.println("Got message " + message );
    System.out.println("-----");
}

@AsyncResponse(target="port", operation="middle")
public void onMiddleAsyncResponse(AsyncPostCallContext apc, String message) {
    System.out.println("-----");
    System.out.println("Got message " + message );
    System.out.println("-----");
}

@AsyncResponse(target="port", operation="finish")
public void onFinishAsyncResponse(AsyncPostCallContext apc, String message) {
    System.out.println("-----");
    System.out.println("Got message " + message );
    System.out.println("-----");
}

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

```

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

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

```

Using JMS Transport as the Connection Protocol

The following sections provide information about using JMS transport as the connection protocol:

- [“Overview of Using JMS Transport” on page 8-1](#)
- [“Using JMS Transport Starting From Java: Main Steps” on page 8-2](#)
- [“Using JMS Transport Starting From WSDL: Main Steps” on page 8-4](#)
- [“Using the @WLJmsTransport JWS Annotation” on page 8-6](#)
- [“Using the <WLJmsTransport> Child Element of the jwsc Ant Task” on page 8-8](#)
- [“Updating the WSDL to Use JMS Transport” on page 8-9](#)
- [“Invoking a WebLogic Web Service Using JMS Transport” on page 8-9](#)

WARNING: This feature can be implemented *only* for a JAX-RPC 1.1-based Web Service; you cannot implement it for a JAX-WS 2.0 Web Service.

Overview of Using JMS Transport

Typically, client applications use HTTP/S as the connection protocol when invoking a WebLogic Web Service. You can, however, configure a WebLogic Web Service so that client applications use JMS as the transport instead. You configure transports using either JWS annotations or child elements of the `jwsc` Ant task, as described in later sections.

When a WebLogic Web Service is configured to use JMS as the connection transport, the endpoint address specified for the corresponding port in the generated WSDL of the Web Service uses `jms://` in its URL rather than `http://`. An example of a JMS endpoint address is as follows:

```
jms://myHost:7001/transport/JMSTransport?URI=JMSTransportQueue
```

The `URI=JMSTransportQueue` section of the URL specifies the JMS queue that has been configured for the JMS transport feature. Although you cannot invoke the Web Service using HTTP, you can view its WSDL using HTTP, which is how the `clientgen` is still able to generate JAX-RPC stubs for the Web Service.

For each transport that you specify, WebLogic Server generates an additional port in the WSDL. For this reason, if you want to give client applications a choice of transports they can use when they invoke the Web Service (JMS, HTTP, or HTTPS), you should explicitly add the transports using the appropriate JWS annotations or child elements of `jwsc`.

Caution: Using JMS transport is an added-value WebLogic feature; non-WebLogic client applications, such as a .NET client, may not be able to invoke the Web Service using the JMS port.

Using JMS Transport Starting From Java: Main Steps

To use JMS transport when starting from Java, you must perform at least one of the following tasks:

- Add the `@WLJmsTransport` annotation to your JWS file.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task. This setting overrides the transports defined in the JWS file.

Note: Because you might not know at the time that you are coding the JWS file which transport best suits your needs, it is often better to specify the transport at build-time using the `<WLJmsTransport>` child element.

The following procedure describes the complete set of steps required so that your Web Service can be invoked using the JMS transport when starting from Java.

Note: It is assumed that you have already created a basic JWS file that implements a Web Service and that you want to configure the Web Service to be invoked using JMS. 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 targets for running the `jwsc` Ant task and deploying the service. For more information, see:

- [Common Web Services Use Cases and Examples](#)
- [Iterative Development of WebLogic Web Services](#)
- [Programming the JWS File](#)
- [Invoking Web Services](#)

1. Configure the WebLogic Server domain for the required JMS components.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see [Configuring Your Domain For Web Services Features](#).

If, however, you prefer to configure the resources yourself, follow these steps:

- a. Invoke the Administration Console in your browser, as described in [Invoking the Administration Console](#).
- b. Using the Administration Console, create and configure the following JMS components, if they do not already exist:
 - JMS Server. See [Create JMS servers](#).
 - JMS Module, targeted to the preceding JMS server. See [Create JMS system modules](#).
 - JMS Queue, contained within the preceding JMS module. You can either specify the JNDI name of the JMS queue that WebLogic Web Services listen to by default (`weblogic.wsee.DefaultQueue`) or specify a different name. If you specify a different JNDI name, you later pass this name to the Web Service itself. When you configure the queue, be sure you specify that this JMS queue is *local*, typically by setting the local JNDI name. See [Create queues in a system module](#).

Except for the JNDI name of the JMS queue, you can name the other components anything you want.

2. Add the `@WLJmsTransport` annotation to your JWS file.

This step is optional. If you do not add the `@WLJmsTransport` annotation to your JWS file, then you must add a `<WLJmsTransport>` child element to the `jwsc` Ant task, as described in Step 3. See [“Using the @WLJmsTransport JWS Annotation” on page 8-6](#).

3. Add a `<WLJmsTransport>` child element to the `jwsc` Ant task.

Use the `<WLJmsTransport>` child element to override the transports defined in the JWS file. This step is required if you did not add the `@WLJmsTransport` annotation to your JWS file in Step 2. Otherwise, this step is optional.

See [“Using the `<WLJmsTransport>` Child Element of the `jwsc` Ant Task” on page 8-8](#) for details.

4. Rebuild your Web Service by re-running the target in the `build.xml` Ant file that calls the `jwsc` task.

For example, if the target that calls the `jwsc` Ant task is called `build-service`, then you would run:

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

5. Redeploy your Web Service to WebLogic Server.

See [“Invoking a WebLogic Web Service Using JMS Transport” on page 8-9](#) for information about updating your client application to invoke the Web Service using JMS transport.

Using JMS Transport Starting From WSDL: Main Steps

To use JMS transport when starting from WSDL, you must perform at least one of the following tasks:

- Update the WSDL to use JMS transport before running the `wsdlc` Ant task.
- Update the stubbed-out JWS implementation file generated by the `wsdlc` Ant task to add the `@WLJmsTransport` annotation.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task used to build the JWS implementation file. This setting overrides the transports defined in the JWS file.

Note: Because you might not know at the time that you are coding the JWS file which transport best suits your needs, it is often better to specify the transport at build-time using the `<WLJmsTransport>` child element.

The following procedure describes the complete set of steps required so that your Web Service can be invoked using the JMS transport when starting from WSDL.

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

1. Configure the WebLogic Server domain for the required JMS components.

You can either configure these resources yourself, or you can use the Configuration Wizard to extend the WebLogic Server domain using a Web Services-specific extension

template. Using the Configuration Wizard greatly simplifies the required configuration steps; for details, see [Configuring Your Domain For Web Services Features](#).

If, however, you prefer to configure the resources yourself, follow these steps:

- a. Invoke the Administration Console in your browser, as described in [Invoking the Administration Console](#).
- b. Using the Administration Console, create and configure the following JMS components, if they do not already exist:
 - JMS Server. See [Create JMS servers](#).
 - JMS Module, targeted to the preceding JMS server. See [Create JMS system modules](#).
 - JMS Queue, contained within the preceding JMS module. You can either specify the JNDI name of the JMS queue that WebLogic Web Services listen to by default (`weblogic.wsee.DefaultQueue`) or specify a different name. If you specify a different JNDI name, you later pass this name to the Web Service itself. When you configure the queue, be sure you specify that this JMS queue is *local*, typically by setting the local JNDI name. See [Create queues in a system module](#).

Except for the JNDI name of the JMS queue, you can name the other components anything you want.

2. Update the WSDL to use JMS transport.

This step is optional. If you do not update the WSDL to use JMS transport, then you must do at least one of the following:

- Edit the stubbed out JWS file to add the `@WLJmsTransport` annotation to your JWS file, as described in Step 4.
- Add a `<WLJmsTransport>` child element to the `jwsc` Ant task, as described in Step 5.

See [“Updating the WSDL to Use JMS Transport” on page 8-9](#).

3. Run the `wsdlc` Ant task against the WSDL file.

For example, if the target that calls the `wsdlc` Ant task is called `generate-from-wsdl`, then you would run:

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

4. Update the stubbed-out JWS file.

The `wsdlc` Ant task generates a stubbed-out JWS file. You need to add your business code to the Web Service so it behaves as you want.

If you updated the WSDL to use the JMS transport in Step 2, the JWS file includes the `@WLJmsTransport` annotation that defines the JMS transport. If the `@WLJmsTransport` annotation is not included in the JWS file, you must do at least one of the following:

- Edit the JWS file to add the `@WLJmsTransport` annotation to your JWS file, as described in [“Using the @WLJmsTransport JWS Annotation” on page 8-6](#).
 - Add a `<WLJmsTransport>` child element to the `jwsc` Ant task, as described in Step 5.
5. Add a `<WLJmsTransport>` child element to the `jwsc` Ant task.

Use the `<WLJmsTransport>` child element to override the transports defined in the JWS file. This step is required if the JWS file does not include the `@WLJmsTransport` annotation, as noted in Step 4. Otherwise, this step is optional. See [“Using the <WLJmsTransport> Child Element of the jwsc Ant Task” on page 8-8](#).

6. Run the `jwsc` Ant task against the JWS file to build the Web Service.

Specify the artifacts generated by the `wsdlc` Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the Web Service. See [Running the jwsc WebLogic Web Services Ant Task](#).

7. Deploy your Web Service to WebLogic Server.

See [Deploying and Undeploying WebLogic Web Services](#).

See [“Invoking a WebLogic Web Service Using JMS Transport” on page 8-9](#) for information about updating your client application to invoke the Web Service using JMS transport.

Using the @WLJmsTransport JWS Annotation

If you know at the time that you program the JWS file that you want client applications to use JMS transport (instead of HTTP/S) to invoke the Web Service, you can use the `@WLJmsTransport` to specify the details of the invoke. Later, at build-time, you can override the one in the JWS file and add additional JMS transport specifications, by specifying the `<WLJmsTransport>` child element of the `jwsc` Ant task, as described in [“Using the <WLJmsTransport> Child Element of the jwsc Ant Task” on page 8-8](#).

Follow these guidelines when using the `@WLJmsTransport` annotation:

- You can include only *one* `@WLJmsTransport` annotation in a JWS file.
- Use the `queue` attribute to specify the JNDI name of the JMS queue you configured earlier in the section. If you want to use the default Web Services queue (`weblogic.wsee.DefaultQueue`) then you do not have to specify the `queue` attribute.

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

```
package examples.webservices.jmstransport;

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

import weblogic.jws.WLJmsTransport;

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

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

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

@WLJmsTransport(contextPath="transports", serviceUri="JMSTransport",
                queue="JMSTransportQueue", portName="JMSTransportServicePort")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 * @author Copyright (c) 2005 by BEA Systems. All rights reserved.
 */

public class JMSTransportImpl {

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

Using the <WLJmsTransport> Child Element of the jwsc Ant Task

You can also specify the JMS transport at build-time by using the <WLJmsTransport> child element of the <jws> element of the jwsc Ant task. Reasons for specifying the transport at build-time include:

- You need to override the attribute values specified in the JWS file.
- The JWS file specifies a different transport, and at build-time you decide that JMS should be the transport.
- The JWS file does not include a @WLXXXTransport annotation; thus by default the HTTP transport is used, but at build-time you decide you want to clients to use the JMS transport to invoke the Web Service.

If you specify a transport to the jwsc Ant task, it takes precedence over any transport annotation in the JWS file.

The following example shows how to specify a transport to the jwsc Ant task:

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

  <jwsc
    srcdir="src"
    destdir="${ear-dir}">
    <jws file="examples/webservices/jmstransport/JMSTransportImpl.java">

      <WLJmsTransport
        contextPath="transports"
        serviceUri="JMSTransport"
        portName="JMSTransportServicePort"
        queue="JMSTransportQueue" />

    </jws>
  </jwsc>
</target>
```

The preceding example shows how to specify the same values for the URL and JMS queue as were specified in the JWS file shown in [“Using the @WLJmsTransport JWS Annotation” on page 8-6](#).

For more information about using the jwsc Ant task, see [jwsc](#).

Updating the WSDL to Use JMS Transport

To update the WSDL to use JMS transport, you need to add `<wsdl:binding>` and `<wsdl:service>` definitions that define JMS transport information. You can add the definitions in one of the following ways:

- Edit the existing HTTP `<wsdl:binding>` and `<wsdl:service>` definitions.
- To specify multiple transport options in the WSDL, copy the existing HTTP `<wsdl:binding>` and `<wsdl:service>` definitions and edit them to use JMS transport.

In either case, you must modify the `<wsdl:binding>` and `<wsdl:service>` definitions to use JMS transport as follows:

- Set the `transport` attribute of the `<soapwsdl:binding>` child element of the `<wsdl:binding>` element to `http://www.openuri.org/2002/04/soap/jms`. For example:

```
<binding name="JmsTransportServiceSoapBindingjms"
type="tns:JmsTransportPortType">
  <soap:binding style="document"
transport="http://www.openuri.org/2002/04/soap/jms"/>
```

- Specify a JMS-style endpoint URL for the `location` attribute of the `<soapwsdl:address>` child element of the `<wsdl:service>`. For example:

```
<s0:service name="JmsTransportService">
  <s0:port binding="s1:JmsTransportServiceSoapBindingjms"
name="JmsTransportServicePort">
    <s2:address
location="jms://localhost:7001/transport/JmsTransport?URI=JMSTransport
Queue"/>
  </s0:port>
</s0:service>
```

Invoking a WebLogic Web Service Using JMS Transport

You write a client application to invoke a Web Service using JMS transport in the same way as you write one using the HTTP transport; the only difference is that you must ensure that the JMS queue (specified by the `@WLJmsTransport` annotation or `<WLJmsTransport>` child element of the `jwsc` Ant task) and other JMS objects have already been created. See [“Using JMS Transport Starting From Java: Main Steps” on page 8-2](#) or [“Using JMS Transport Starting From WSDL: Main Steps” on page 8-4](#) for more information.

Although you cannot *invoke* a JMS-transport-configured Web Service using HTTP, you can view its WSDL using HTTP, which is how the `clientgen` Ant task is still able to create the JAX-RPC stubs for the Web Service. For example, the URL for the WSDL of the Web Service shown in this section would be:

```
http://host:port/transport/JMSTransport?WSDL
```

However, because the endpoint address in the WSDL of the deployed Web Service uses `jms://` instead of `http://`, and the address includes the qualifier `?URI=JMS_QUEUE`, the `clientgen` Ant task automatically creates the stubs needed to use the JMS transport when invoking the Web Service, and your client application need not do anything different than normal. An example of a JMS endpoint address is as follows:

```
jms://host:port/transport/JMSTransport?URI=JMSTransportQueue
```

WARNING: If you have specified that the Web Service you invoke using JMS transport also runs within the context of a transaction (in other words, the JWS file includes the `@weblogic.jws.Transactional` annotation), you must use asynchronous request-response when invoking the service. If you do not, a deadlock will occur and the invocation will fail.

For general information about invoking a Web Service, see [Invoking Web Services](#).

Overriding the Default Service Address URL

When you write a client application that uses the `clientgen`-generated JAX-RPC stubs to invoke a Web Service, the default service address URL of the Web Service is the one specified in the `<address>` element of the WSDL file argument of the `Service` constructor.

Sometimes, however, you might need to override this address, in particular when invoking a WebLogic Web Service that is deployed to a cluster and you want to specify the cluster address or a list of addresses of the managed servers in the cluster. You might also want to use the `t3` protocol to invoke the Web Service. To override this service endpoint URL when using JMS transport, use the `weblogic.wsee.jaxrpc.WLStub.JMS_TRANSPORT_JNDI_URL` stub property as shown in the following example:

```
package examples.webservices.jmstransport.client;

import weblogic.wsee.jaxrpc.WLStub;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
```

```

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the JMSTransport Web service.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class Main {

    public static void main(String[] args)
        throws ServiceException, RemoteException{

        JMSTransportService service = new JMSTransportService_Impl(args[0] +
"?WSDL" );
        JMSTransportPortType port = service.getJMSTransportServicePort();

        Stub stub = (Stub) port;

        stub._setProperty(WLStub.JMS_TRANSPORT_JNDI_URL,
            "t3://shackell101.amer.bea.com:7001");

        try {
            String result = null;

            result = port.sayHello("Hi there! ");

            System.out.println( "Got JMS result: " + result );

        } catch (RemoteException e) {
            throw e;
        }
    }
}

```

See [WLStub](#) reference documentation for additional stub properties.

Using JMS BytesMessage Rather Than the Default TextMessage

When you use JMS transport, the Web Services runtime uses, by default, the `javax.jms.TextMessage` object to send the message. This is usually adequate for most client applications, but sometimes you might need to send binary data rather than ordinary text; in this case you must request that the Web Services runtime use `javax.jms.BytesMessage` instead. To do this, use the `WLStub.JMS_TRANSPORT_MESSAGE_TYPE` stub property in your client

application and set it to the value `WLStub.JMS_BYTESMESSAGE`, as shown in the following example:

```
stub._setProperty(WLStub.JMS_TRANSPORT_MESSAGE_TYPE,  
                  WLStub.JMS_BYTESMESSAGE);
```

The Web Services runtime sends back the response using the same message data type as the request.

See [“Overriding the Default Service Address URL” on page 1-7](#) for a full example of a client application in which you can set this property. See [WLStub](#) reference documentation for additional stub properties.

Disabling HTTP Access to the WSDL File

As described in [“Invoking a WebLogic Web Service Using JMS Transport” on page 1-6](#), the WSDL of the deployed Web Service is, by default, still accessible using HTTP. If you want to disable access to the WSDL file, in particular if your Web Service can be accessed outside of a firewall, then you can do one of the following:

- Use the `weblogic.jws.WSDL` annotation in your JWS file to programmatically disable access. For details, see [weblogic.jws.WSDL](#).
- Use the Administration Console to disable access to the WSDL file *after* the Web Service has been deployed. In this case, the configuration information will be stored in the deployment plan rather than through the annotation.

To use the Administration Console to perform this task, go to the Configuration -> General page of the deployed Web Service and uncheck the **View Dynamic WSDL Enabled** checkbox. After saving the configuration to the deployment plan, you must redeploy (update) the Web Service, or Enterprise Application which contains it, for the change to take effect.

Creating and Using SOAP Message Handlers

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

- [“Overview of SOAP Message Handlers” on page 9-1](#)
- [“Adding SOAP Message Handlers to a Web Service: Main Steps” on page 9-4](#)
- [“Designing the SOAP Message Handlers and Handler Chains” on page 9-5](#)
- [“Creating the GenericHandler Class” on page 9-7](#)
- [“Configuring Handlers in the JWS File” on page 9-16](#)
- [“Creating the Handler Chain Configuration File” on page 9-20](#)
- [“Compiling and Rebuilding the Web Service” on page 9-22](#)
- [“Creating and Using Client-Side SOAP Message Handlers” on page 9-22](#)

Overview of SOAP Message Handlers

Some Web Services need access to the SOAP message, for which you can create SOAP message handlers.

A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web Service. You can create handlers in both the Web Service itself and the client applications that invoke 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.

The following table lists the standard JWS annotations that you can use in your JWS file to specify that a Web Service has a handler chain configured; later sections discuss how to use the annotations in more detail. For additional information, see the [Web Services Metadata for the Java Platform \(JSR-181\) specification at http://www.jcp.org/en/jsr/detail?id=181](http://www.jcp.org/en/jsr/detail?id=181).

Table 9-1 JWS Annotations Used To Configure SOAP Message Handler Chains

JWS Annotation	Description
<code>javax.jws.HandlerChain</code>	Associates the Web Service with an externally defined handler chain. Use this annotation when multiple Web Services need to share the same handler configuration, or if the handler chain consists of handlers for multiple transports.
<code>javax.jws.soap.SOAPMessageHandlers</code>	<p>Specifies a list of SOAP handlers that run before and after the invocation of each Web Service operation. Use this annotation (rather than <code>@HandlerChain</code>) if embedding handler configuration information in the JWS file itself is preferred, rather than having an external configuration file.</p> <p>The <code>@SOAPMessageHandler</code> annotation is an array of <code>@SOAPMessageHandlers</code>. The handlers are executed in the order they are listed in this array.</p> <p>Note: This annotation works <i>only</i> with JAX-RPC 1.1-based Web Services.</p>
<code>javax.jws.soap.SOAPMessageHandler</code>	Specifies a single SOAP message handler in the <code>@SOAPMessageHandlers</code> array.

The following table describes the main classes and interfaces of the `javax.xml.rpc.handler` API, some of which you use when creating the handler itself. These APIs are discussed in detail in a later section. For additional information about these APIs, see the [JAX-RPC 1.1 specification at `http://java.sun.com/xml/jaxrpc/index.jsp`](http://java.sun.com/xml/jaxrpc/index.jsp).

Table 9-2 JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
<code>Handler</code>	Main interface that is implemented when creating a handler. Contains methods to handle the SOAP request, response, and faults.
<code>GenericHandler</code>	<p>Abstract class that implements the <code>Handler</code> interface. User should extend this class when creating a handler, rather than implement <code>Handler</code> directly.</p> <p>The <code>GenericHandler</code> class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the lifecycle methods <code>init</code> and <code>destroy</code> and also different handle methods. A handler developer should only override methods that it needs to specialize as part of the derived handler implementation class.</p>
<code>HandlerChain</code>	Interface that represents a list of handlers. An implementation class for the <code>HandlerChain</code> interface abstracts the policy and mechanism for the invocation of the registered handlers.
<code>HandlerRegistry</code>	Interface that provides support for the programmatic configuration of handlers in a <code>HandlerRegistry</code> .
<code>HandlerInfo</code>	Class that contains information about the handler in a handler chain. A <code>HandlerInfo</code> instance is passed in the <code>Handler.init</code> method to initialize a <code>Handler</code> instance.
<code>MessageContext</code>	Abstracts the message context processed by the handler. The <code>MessageContext</code> properties allow the handlers in a handler chain to share processing state.

Table 9-2 JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
<code>soap.SOAPMessageContext</code>	Sub-interface of the <code>MessageContext</code> interface used to get at or update the SOAP message.
<code>javax.xml.soap.SOAPMessage</code>	Object that contains the actual request or response SOAP message, including its header, body, and attachment.

Adding SOAP Message Handlers to a Web Service: 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 already 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:

- [Common Web Services Use Cases and Examples](#)
- [Iterative Development of WebLogic Web Services](#)
- [Programming the JWS File](#)
- [Invoking Web Services](#)

1. Design the handlers and handler chains.

See [“Designing the SOAP Message Handlers and Handler Chains” on page 9-5](#).

2. For each handler in the handler chain, create a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class.

See [“Creating the GenericHandler Class” on page 9-7](#).

3. Update your JWS file, adding annotations to configure the SOAP message handlers.

See [“Configuring Handlers in the JWS File” on page 9-16](#).

4. If you are using the `@HandlerChain` standard annotation in your JWS file, create the handler chain configuration file.

See [“Creating the Handler Chain Configuration File” on page 9-20](#).

5. Compile all handler classes in the handler chain and rebuild your Web Service.

See [“Compiling and Rebuilding the Web Service” on page 9-22](#).

For information about creating client-side SOAP message handlers and handler chains, see [“Creating and Using Client-Side SOAP Message Handlers” on page 9-22](#).

Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers and handler chains, you must decide:

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

Each handler in a handler chain has one method for handling the request SOAP message and another method for handling the response SOAP message. An ordered group of handlers is referred to as a *handler chain*. You specify that a Web Service has a handler chain attached to it with one of two JWS annotations: `@HandlerChain` or `@SOAPMessageHandler`. When to use which is discussed in a later section.

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

1. The `handleRequest()` methods of the handlers in the handler chain are all executed in the order specified by the JWS annotation. Any of these `handleRequest()` methods might change the SOAP message request.
2. When the `handleRequest()` method of 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 `handleResponse()` methods of the handlers in the handler chain are executed in the *reverse* order specified in by the JWS annotation. Any of these `handleResponse()` methods might change the SOAP message response.

4. When the `handleResponse()` method of 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:

```
<jwshc:handler-config xmlns:jwshc="http://www.bea.com/xml/ns/jws"
  xmlns:soap1="http://HandlerInfo.org/Server1"
  xmlns:soap2="http://HandlerInfo.org/Server2"
  xmlns="http://java.sun.com/xml/ns/j2ee" >

  <jwshc:handler-chain>

    <jwshc:handler-chain-name>SimpleChain</jwshc:handler-chain-name>

    <jwshc:handler>
      <handler-name>handlerOne</handler-name>

      <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler
1</handler-class>
    </jwshc:handler>

    <jwshc:handler>
      <handler-name>handlerTwo</handler-name>

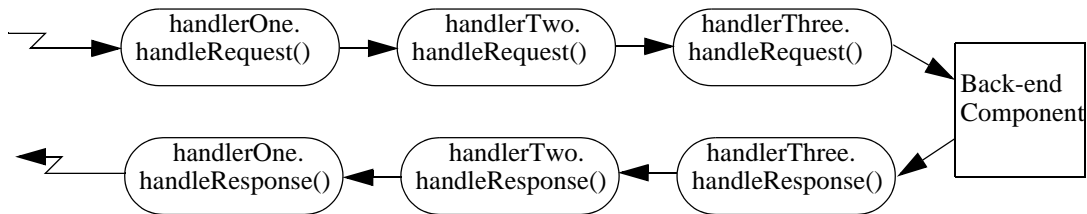
      <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler
2</handler-class>
    </jwshc:handler>

    <jwshc:handler>
      <handler-name>handlerThree</handler-name>

      <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler
3</handler-class>
    </jwshc:handler>

  </jwshc:handler-chain>
</jwshc:handler-config>
```

The following graphic shows the order in which WebLogic Server executes the `handleRequest()` and `handleResponse()` methods of each handler.

Figure 9-1 Order of Execution of Handler Methods

Each SOAP message handler has a separate method to process the request and response SOAP message because the same type of processing typically must happen for the inbound and outbound message. For example, you might design an Encryption handler whose `handleRequest()` method decrypts secure data in the SOAP request and `handleResponse()` method encrypts the SOAP response.

You can, however, design a handler that process 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.

Creating the GenericHandler Class

Your SOAP message handler class should extend the `javax.rpc.xml.handler.GenericHandler` abstract class, which itself implements the `javax.rpc.xml.handler.Handler` interface.

The `GenericHandler` class is a convenience abstract class that makes writing handlers easy. This class provides default implementations of the lifecycle methods `init()` and `destroy()` and the various `handleXXX()` methods of the `Handler` interface. When you write your handler class, only override those methods that you need to customize as part of your `Handler` implementation class.

In particular, the `Handler` interface contains the following methods that you can implement in your handler class that extends `GenericHandler`:

- `init()`
See [“Implementing the Handler.init\(\) Method” on page 9-10.](#)
- `destroy()`
See [“Implementing the Handler.destroy\(\) Method” on page 9-10.](#)

- `getHeaders()`

See [“Implementing the Handler.getHeaders\(\) Method”](#) on page 9-10.

- `handleRequest()`

See [“Implementing the Handler.handleRequest\(\) Method”](#) on page 9-11.

- `handleResponse()`

See [“Implementing the Handler.handleResponse\(\) Method”](#) on page 9-12.

- `handleFault()`

See [“Implementing the Handler.handleFault\(\) Method”](#) on page 9-13.

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 9-14.

The following example demonstrates a simple SOAP message handler that prints out the SOAP request and response messages to the WebLogic Server log file:

```
package examples.webservices.soap_handlers.global_handler;

import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;

import weblogic.logging.NonCatalogLogger;

/**
 * This class implements a handler in the handler chain, used to access the SOAP
 * request and response message.
 * <p>
 * This class extends the <code>javax.xml.rpc.handler.GenericHandler</code>
 * abstract class and simply prints the SOAP request and response messages to
 * the server log file before the messages are processed by the backend
 * Java class that implements the Web Service itself.
 */

public class ServerHandler1 extends GenericHandler {

    private NonCatalogLogger log;

    private HandlerInfo handlerInfo;
```



```

/**
 * Initializes the instance of the handler. Creates a nonCatalogLogger to
 * log messages to.
 */

public void init(HandlerInfo hi) {

    log = new NonCatalogLogger("WebService-LogHandler");
    handlerInfo = hi;

}

/**
 * Specifies that the SOAP request message be logged to a log file before the
 * message is sent to the Java class that implements the Web Service.
 */

public boolean handleRequest(MessageContext context) {

    SOAPMessageContext messageContext = (SOAPMessageContext) context;

    System.out.println("*** Request: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;

}

/**
 * Specifies that the SOAP response message be logged to a log file before the
 * message is sent back to the client application that invoked the Web
 * service.
 */

public boolean handleResponse(MessageContext context) {

    SOAPMessageContext messageContext = (SOAPMessageContext) context;

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

}

/**
 * Specifies that a message be logged to the log file if a SOAP fault is
 * thrown by the Handler instance.
 */

public boolean handleFault(MessageContext context) {

    SOAPMessageContext messageContext = (SOAPMessageContext) context;

```

```

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

    public QName[] getHeaders() {
        return handlerInfo.getHeaders();
    }
}

```

Implementing the Handler.init() Method

The `Handler.init()` method is called to create an instance of a `Handler` object and to enable the instance to initialize itself. Its signature is:

```
public void init(HandlerInfo config) throws JAXRPCException {}
```

The `HandlerInfo` object contains information about the SOAP message handler, in particular the initialization parameters. Use the `HandlerInfo.getHandlerConfig()` method to get the parameters; the method returns a `java.util.Map` object that contains name-value pairs.

Implement the `init()` method if you need to process the initialization parameters or if you have other initialization tasks to perform.

Sample uses of initialization parameters are to turn debugging on or off, specify the name of a log file to which to write messages or errors, and so on.

Implementing the Handler.destroy() Method

The `Handler.destroy()` method is called to destroy an instance of a `Handler` object. Its signature is:

```
public void destroy() throws JAXRPCException {}
```

Implement the `destroy()` method to release any resources acquired throughout the handler's lifecycle.

Implementing the Handler.getHeaders() Method

The `Handler.getHeaders()` method gets the header blocks that can be processed by this `Handler` instance. Its signature is:

```
public QName[] getHeaders() {}
```

Implementing the Handler.handleRequest() Method

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

```
public boolean handleRequest(MessageContext mc)
    throws JAXRPCException, SOAPFaultException {}
```

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

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message request. The SOAP message request 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 9-14](#).

The `SOAPMessageContext` class defines two methods for processing the SOAP request:

- `SOAPMessageContext.getMessage()` returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message request.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)` updates the SOAP message request after you have made changes to it.

After you code all the processing of the SOAP request, 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 either the next `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the next `@SOAPMessageHandler` in the array specified by the `@SOAPMessageHandlers` annotation. If there are no more handlers in the chain, the method either invokes the back-end component, passing it the final SOAP message request, or invokes the `handleResponse()` method of the last handler, depending on how you have configured your Web Service.

- 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. For example, assume that a handler chain consists of two handlers: handlerA and handlerB, where the `handleRequest()` method of handlerA is invoked before that of handlerB. If processing is blocked in handlerA (and thus the `handleRequest()` method of handlerB is *not* invoked), the handler response chain starts at handlerA and the `handleRequest()` method of handlerB is not invoked either.

- Throw the `javax.xml.rpc.soap.SOAPFaultException` to indicate a SOAP fault.

If the `handleRequest()` method throws a `SOAPFaultException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, and invokes the `handleFault()` method of this handler.

- Throw a `JAXRPCException` for any handler-specific runtime errors.

If the `handleRequest()` method throws a `JAXRPCException`, 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.handleResponse()` Method

The `Handler.handleResponse()` method is called to intercept a SOAP message response after it has been processed by the back-end component, but before it is sent back to the client application that invoked the Web Service. Its signature is:

```
public boolean handleResponse(MessageContext mc) throws JAXRPCException { }
```

Implement this method to perform such tasks as encrypting data in the SOAP message before it is sent back to the client application, to further process returned values, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message response. The SOAP message response itself is stored in a `javax.xml.soap.SOAPMessage` object. See [“Directly Manipulating the SOAP Request and Response Message Using SAAJ” on page 9-14](#).

The `SOAPMessageContext` class defines two methods for processing the SOAP response:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message response.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message response after you have made changes to it.

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

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

The next response on the handler chain is specified as either the preceding `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation, or the preceding `@SOAPMessageHandler` in the array specified by the `@SOAPMessageHandlers` annotation. (Remember that responses on the handler chain execute in the *reverse* order that they are specified in the JWS file. See [“Designing the SOAP Message Handlers and Handler Chains” on page 9-5](#) for more information.)

If there are no more handlers in the chain, the method sends the final SOAP message response to the client application that invoked the Web Service.

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

Blocking the handler response chain processing implies that the remaining handlers on the response chain do not get executed for this invoke of the Web Service and the current SOAP message is sent back to the client application.

- Throw a `JAXRPCException` for any handler specific runtime errors.

If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, 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(MessageContext mc) throws JAXRPCException {}
```

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

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message. The SOAP message itself is stored in a `javax.xml.soap.SOAPMessage` object. See “[Directly Manipulating the SOAP Request and Response Message Using SAAJ](#)” on page 9-14.

The `SOAPMessageContext` class defines the following two methods for processing the SOAP message:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message after you have made changes to it.

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`.

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. For more information on SAAJ, go to <http://java.sun.com/xml/soap/saaj/index.html>.

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.

Caution: 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 9-15](#) for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments:

- `countAttachments()`: returns the number of attachments in this SOAP message.
- `getAttachments()`: retrieves all the attachments (as `AttachmentPart` objects) into an `Iterator` object.
- `createAttachmentPart()`: create an `AttachmentPart` object from another type of `Object`.
- `addAttachmentPart()`: adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.

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 Handlers in the JWS File

There are two standard annotations you can use in your JWS file to configure a handler chain for a Web Service: `@javax.jws.HandlerChain` and `@javax.jws.soap.SOAPMessageHandlers`.

`@javax.jws.HandlerChain`

When you use the `@javax.jws.HandlerChain` annotation (also called `@HandlerChain` in this chapter for simplicity) you 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.

Use the `@HandlerChain` annotation, rather than the `@SOAPMessageHandlers` annotation, in your JWS file if one or more of the following conditions apply:

- You want multiple Web Services to share the same configuration.
- Your handler chain includes handlers for multiple transports.
- You want to be able to change the handler chain configuration for a Web Service without recompiling the JWS file that implements it.

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

```
package examples.webservices.soap_handlers.global_handler;

import java.io.Serializable;

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

import weblogic.jws.WLHttpTransport;
```



```

@WebService(serviceName="HandlerChainService",
            name="HandlerChainPortType")

// Standard JWS annotation that specifies that the handler chain called
// "SimpleChain", configured in the HandlerConfig.xml file, should fire
// each time an operation of the Web Service is invoked.

@HandlerChain(file="HandlerConfig.xml", name="SimpleChain")

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

@WLHttpTransport(contextPath="HandlerChain", serviceUri="HandlerChain",
                portName="HandlerChainServicePort")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello. The Web Service also
 * has a handler chain associated with it, as specified by the
 * @HandlerChain annotation.
 * <p>
 * @author Copyright (c) 2005 by BEA Systems, Inc. All Rights Reserved.
 */

public class HandlerChainImpl {

    public String sayHello(String input) {
        weblogic.utils.Debug.say( "in backend component. input:" +input );
        return "'" + input + "' to you too!";
    }

}

```

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

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.

Use the `name` attribute to specify the name of the handler chain in the configuration file that you want to associate with the Web Service. The value of this attribute corresponds to the `name` attribute of the `<handler-chain>` element in the configuration file.

WARNING: It is an error to specify more than one `@HandlerChain` annotation in a single JWS file. It is also an error to combine the `@HandlerChain` annotation with the `@SOAPMessageHandlers` annotation.

For details about creating the external configuration file, see [“Creating the Handler Chain Configuration File” on page 9-20](#).

For additional detailed information about the standard JWS annotations discussed in this section, see the [Web Services Metadata for the Java Platform specification at `http://www.jcp.org/en/jsr/detail?id=181`](http://www.jcp.org/en/jsr/detail?id=181).

@javax.jws.soap.SOAPMessageHandlers

When you use the `@javax.jws.soap.SOAPMessageHandlers` (also called `@SOAPMessageHandlers` in this section for simplicity) annotation, you specify, within the JWS file itself, an array of SOAP message handlers (specified with the `@SOAPMessageHandler` annotation) that execute before and after the operations of a Web Service. The `@SOAPMessageHandler` annotation includes attributes to specify the class name of the handler, the initialization parameters, list of SOAP headers processed by the handler, and so on. Because you specify the list of handlers within the JWS file itself, the configuration of the handler chain is embedded within the Web Service.

Use the `@SOAPMessageHandlers` annotation if one or more of the following conditions apply:

- You prefer to embed the configuration of the handler chain inside the Web Service itself, rather than specify the configuration in an external file.
- Your handler chain includes only SOAP handlers and none for any other transport.
- You prefer to recompile the JWS file each time you change the handler chain configuration.

The following JWS file shows a simple example of using the `@SOAPMessageHandlers` annotation; the relevant Java code is shown in bold:

```
package examples.webservices.soap_handlers.simple;  
  
import java.io.Serializable;
```

```

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

import weblogic.jws.WLHttpTransport;

@WebService(name="SimpleChainPortType",
            serviceName="SimpleChainService")

// Standard JWS annotation that specifies a list of SOAP message handlers
// that execute before and after an invocation of all operations in the
// Web Service.
@SOAPMessageHandlers ( {
    @SOAPMessageHandler (

        className="examples.webservices.soap_handlers.simple.ServerHandler1"),
    @SOAPMessageHandler (

        className="examples.webservices.soap_handlers.simple.ServerHandler2")
    } )

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

@WLHttpTransport(contextPath="SimpleChain", serviceUri="SimpleChain",
                portName="SimpleChainServicePort")

/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello. The Web Service also
 * has a handler chain associated with it, as specified by the
 * @SOAPMessageHandler/s annotations.
 * <p>
 * @author Copyright (c) 2005 by BEA Systems, Inc. All Rights Reserved.
 */

public class SimpleChainImpl {

    // by default all public methods are exposed as operations

```

```

    public String sayHello(String input) {
        weblogic.utils.Debug.say( "in backend component. input:" +input );
        return "'" + input + "' to you too!";
    }
}

```

Before you use the `@SOAPMessageHandlers` and `@SOAPMessageHandler` annotations, you must import them into your JWS file, as shown in the preceding example. Note that these annotations are in the `javax.jws.soap` package.

The order in which you list the handlers (using the `@SOAPMessageHandler` annotation) in the `@SOAPMessageHandlers` array specifies the order in which the handlers execute: in forward order before the operation, and in reverse order after the operation. The preceding example configures two handlers in the handler chain, whose class names are `examples.webservices.soap_handlers.simple.ServerHandler1` and `examples.webservices.soap_handlers.simple.ServerHandler2`.

Use the `initParams` attribute of `@SOAPMessageHandler` to specify an array of initialization parameters expected by a particular handler. Use the `@InitParam` standard JWS annotation to specify the name/value pairs, as shown in the following example:

```

@SOAPMessageHandler(
    className = "examples.webservices.soap_handlers.simple.ServerHandler1",
    initParams = { @InitParam(name="logCategory", value="MyService")}
)

```

The `@SOAPMessageHandler` annotation also includes the `roles` attribute for listing the SOAP roles implemented by the handler, and the `headers` attribute for listing the SOAP headers processed by the handler.

WARNING: It is an error to combine the `@SOAPMessageHandlers` annotation with the `@HandlerChain` annotation.

For additional detailed information about the standard JWS annotations discussed in this section, see the [Web Services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181](http://www.jcp.org/en/jsr/detail?id=181).

Creating the Handler Chain Configuration File

If you decide to use the `@HandlerChain` annotation in your JWS file to associate a handler chain with a Web Service, you must create 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. Finally, if you include handlers in your handler chain that use a non-SOAP transport, then you are required to use the `@HandlerChain` annotation rather than the `@SOAPMessageHandler` annotation.

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

```
<jwshc:handler-config xmlns:jwshc="http://www.bea.com/xml/ns/jws"
  xmlns:soap1="http://HandlerInfo.org/Server1"
  xmlns:soap2="http://HandlerInfo.org/Server2"
  xmlns="http://java.sun.com/xml/ns/j2ee" >
  <jwshc:handler-chain>
    <jwshc:handler-chain-name>SimpleChain</jwshc:handler-chain-name>
    <jwshc:handler>
      <handler-name>handler1</handler-name>

      <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler
1</handler-class>
    </jwshc:handler>
    <jwshc:handler>
      <handler-name>handler2</handler-name>

      <handler-class>examples.webservices.soap_handlers.global_handler.ServerHandler
2</handler-class>
    </jwshc:handler>
  </jwshc:handler-chain>
</jwshc:handler-config>
```

In the example, the handler chain called `SimpleChain` contains two handlers: `handler1` and `handler2`, 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>`, `<soap-role>`, and `<soap-header>` child elements of the `<handler>` element to specify the handler initialization parameters, SOAP roles implemented by the handler, and SOAP headers processed by the handler, respectively.

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 at http://www.jcp.org/en/jsr/detail?id=181](http://www.jcp.org/en/jsr/detail?id=181).

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 [Iterative Development of WebLogic Web Services](#) 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 either the `@HandlerChain` or `@SOAPMessageHandlers` 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` or `@SOAPMessageHandler(s)` annotations 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.

Creating and Using Client-Side SOAP Message Handlers

The preceding sections describe how to create server-side SOAP message handlers that execute as part of the Web Service running on WebLogic Server. You can also create client-side handlers that execute as part of the client application that *invokes* a Web Service operation. 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

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: write a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class. 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.

Similar to the server-side SOAP handler programming, you use an XML file to specify to the `clientgen` Ant task that you want to invoke client-side SOAP message handlers. However, the XML Schema of this XML file is slightly different, as described in the following procedure.

Using Client-Side SOAP Message Handlers: Main Steps

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 already 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](#).

1. Design the client-side SOAP handlers and the handler chain which specifies the order in which they execute. This step is almost exactly the same as that of 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 9-5](#).

2. For each handler in the handler chain, create a Java class that extends the `javax.xml.rpc.handler.GenericHandler` abstract class. This step is very similar to the corresponding server-side step, except that the handler executes in a chain in the client rather than the server.

See [“Creating the GenericHandler Class” on page 9-7](#) for details about programming a handler class. See [“Example of a Client-Side Handler Class” on page 9-24](#) for an example.

3. Create the client-side SOAP handler configuration file. This XML file describes the handlers in the handler chain, the order in which they execute, and any initialization parameters that should be sent.

See [“Creating the Client-Side SOAP Handler Configuration File” on page 9-25.](#)

4. Update the `build.xml` file that builds your client application, specifying to the `clientgen` Ant task the name of the SOAP handler configuration file. Also ensure that the `build.xml` file compiles the handler files into Java classes and makes them available to your client application.

See [“Specifying the Client-Side SOAP Handler Configuration File to `clientgen`” on page 9-27.](#)

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 JAX-RPC stubs automatically take care of executing the handlers in the correct sequence.

Example of a Client-Side Handler Class

The following example shows a simple SOAP message handler class that you can configure for a client application that invokes a Web Service.

```
package examples.webservices.client_handler.client;

import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;

public class ClientHandler1 extends GenericHandler {

    private QName[] headers;

    public void init(HandlerInfo hi) {
        System.out.println("in " + this.getClass() + " init()");
    }
}
```



```

public boolean handleRequest(MessageContext context) {
    System.out.println("in " + this.getClass() + " handleRequest()");
    return true;
}

public boolean handleResponse(MessageContext context) {
    System.out.println("in " + this.getClass() + " handleResponse()");
    return true;
}

public boolean handleFault(MessageContext context) {
    System.out.println("in " + this.getClass() + " handleFault()");
    return true;
}

public QName[] getHeaders() {
    return headers;
}
}

```

Creating the Client-Side SOAP Handler Configuration File

The client-side SOAP handler configuration file specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on. See [“XML Schema for the Client-Side Handler Configuration File” on page 9-26](#) for a full description of this file.

The configuration file uses XML to describe a single handler chain that contains one or more handlers, as shown in the following simple example:

```

<weblogic-wsee-clientHandlerChain
  xmlns="http://www.bea.com/ns/weblogic/90"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee">

  <handler>
    <j2ee:handler-name>clienthandler1</j2ee:handler-name>

    <j2ee:handler-class>examples.webservices.client_handler.client.ClientHandler1<
/j2ee:handler-class>
      <j2ee:init-param>
        <j2ee:param-name>ClientParam1</j2ee:param-name>
        <j2ee:param-value>value1</j2ee:param-value>
      </j2ee:init-param>
    </handler>

    <handler>
      <j2ee:handler-name>clienthandler2</j2ee:handler-name>

      <j2ee:handler-class>examples.webservices.client_handler.client.ClientHandler2<

```

```

/j2ee:handler-class>
    </handler>

</weblogic-wsee-clientHandlerChain>

```

In the example, the handler chain contains two handlers: `clienthandler1` and `clienthandler2`, implemented with the class names specified with the `<j2ee:handler-class>` element. The two handlers execute in forward order directly before the client application sends the SOAP request to the Web Service, and then in reverse order directly after the client application receives the SOAP response from the Web Service.

The example also shows how to use the `<j2ee:init-param>` element to specify one or more initialization parameters to a handler.

Use the `<soap-role>`, `<soap-header>`, and `<port-name>` child elements of the `<handler>` element to specify the SOAP roles implemented by the handler, the SOAP headers processed by the handler, and the port-name element in the WSDL with which the handler is associated with, respectively.

XML Schema for the Client-Side Handler Configuration File

The following XML Schema file defines the structure of the client-side SOAP handler configuration file:

```

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

<schema
    targetNamespace="http://www.bea.com/ns/weblogic/90"
    xmlns:wls="http://www.bea.com/ns/weblogic/90"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
>
    <include schemaLocation="weblogic-j2ee.xsd"/>

    <element name="weblogic-wsee-clientHandlerChain"
        type="wls:weblogic-wsee-clientHandlerChainType">
        <xsd:key name="wsee-clienthandler-name-key">
            <xsd:annotation>
                <xsd:documentation>

                    Defines the name of the handler. The name must be unique within the
                    chain.

```

```

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="j2ee:handler"/>
      <xsd:field xpath="j2ee:handler-name"/>
    </xsd:key>
  </element>

  <complexType name="weblogic-wsee-clientHandlerChainType">
    <sequence>
      <xsd:element name="handler"
        type="j2ee:service-ref_handlerType"
        minOccurs="0" maxOccurs="unbounded">
      </xsd:element>
    </sequence>
  </complexType>
</schema>

```

A single configuration file specifies a single client-side handler chain. The root of the configuration file is `<weblogic-wsee-clientHandlerChain>`, and the file contains zero or more `<handler>` child elements, each of which describes a handler in the chain.

The structure of the `<handler>` element is described by the J2EE `service-ref_handlerType` complex type, specified in the [J2EE 1.4 Web Service client XML Schema](#).

Specifying the Client-Side SOAP Handler Configuration File to `clientgen`

Use the `handlerChainFile` attribute of the `clientgen` Ant task to specify the client-side SOAP handler configuration file, as shown in the following excerpt from a `build.xml` file:

```

<clientgen
  wsdl="http://ariel:7001/handlers/ClientHandlerService?WSDL"
  destDir="${clientclass-dir}"
  handlerChainFile="ClientHandlerChain.xml"
  packageName="examples.webservices.client_handler.client"/>

```

The JAX-RPC stubs generated by `clientgen` automatically ensure that the handlers described by the configuration file execute in the correct order before and after the client application invokes the Web Service operation

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” on page 10-1](#)
- [“WebLogic Server UDDI Features” on page 10-4](#)
- [“UDDI 2.0 Server” on page 10-5](#)
- [“UDDI Directory Explorer” on page 10-20](#)
- [“UDDI Client API” on page 10-20](#)
- [“Pluggable tModel” on page 10-21](#)

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: <ul style="list-style-type: none">• Company name• Contact detail• Other business information	Represents a company, a division or a department within a company: <ul style="list-style-type: none">• 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. <ul style="list-style-type: none">• Multiple names and descriptions• Categories• Indicators of compliancy with standards
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 at `http://www.uddi.org/specification.html`](http://www.uddi.org/specification.html).

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.

WARNING: 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. BEA 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.properties` 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 refernced 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 refernced 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 refernced 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
<code>auddi.discoveryurl</code>	DiscoveryURL prefix that is set for each saved business entity. Typically this is the full URL to the <code>uddilistener</code> servlet, so that the full <code>DiscoveryURL</code> results in the display of the stored <code>BusinessEntity</code> data.
<code>auddi.inquiry.secure</code>	Permissible values are <code>true</code> and <code>false</code> . When set to <code>true</code> , inquiry calls to UDDI Server are limited to secure https connections only. Any UDDI inquiry calls through a regular http URL are rejected.

Table 10-2 Basic UDDI Configuration

UDDI Property Key	Description
audi.publish.secure	Permissible values are <code>true</code> and <code>false</code> . When set to <code>true</code> , publish calls to UDDI Server are limited to secure https connections only. Any UDDI publish calls through a regular http URL are rejected.
audi.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.
audi.search.timeout	Timeout value for search operations. The value is indicated in milliseconds.
audi.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.
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.
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 <code>WLS</code> , which indicates that the internal LDAP directory of WebLogic Server is to be used for data storage. Other permissible values include <code>LDAP</code> , <code>ReplicaLDAP</code> , and <code>File</code> .
auddi.security.type	UDDI Server's security module (authentication). This value defaults to <code>WLS</code> , 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 <code>LDAP</code> and <code>File</code> .
auddi.license.dir	Location of the UDDI Server license file. In the absence of this property, the <code>WL_HOME/server/lib</code> directory is assumed to be the default license directory, where <code>WL_HOME</code> is the main WebLogic Server installation directory. Some WebLogic users are exempt from requiring a UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (for example., UDDI Server Browser).
auddi.license.file	Name of the license file. In the absence of this property, <code>uddilicense.xml</code> is presumed to be the default license filename. Some WebLogic users are exempt from requiring an UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (e.g., UDDI Server Browser).

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 <code>true</code> and <code>false</code> . When set to <code>true</code> , 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: <code>LOG_TYPE_SCREEN</code> , <code>LOG_TYPE_FILE</code> , and <code>LOG_TYPE_SCREEN_FILE</code> .
logger.output.style	Determines whether logged output will simply contain the message, or thread and timestamp information will be included. Permissible values are <code>OUTPUT_LONG</code> and <code>OUTPUT_SHORT</code> .
logger.quiet	Determines whether the logger itself displays information messages. Permissible values are <code>true</code> and <code>false</code> .
logger.verbosity	Logger's verbosity level. Permissible values (case sensitive) are <code>TRACE</code> , <code>DEBUG</code> , <code>INFO</code> , <code>WARNING</code> and <code>ERROR</code> , where each severity level includes the following ones accumulatively.

Table 10-6 Connection Pools

UDDI Property Key	Description
<code>datasource.ldap.pool.increment</code>	Number of new connections to create and add to the pool when all connections in the pool are busy
<code>datasource.ldap.pool.initialsize</code>	Number of connections to be stored at the time of creation and initialization of the pool.
<code>datasource.ldap.pool.maxsize</code>	Maximum number of connections that the pool may hold.
<code>datasource.ldap.pool.systemmaxsize</code>	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
<code>datasource.ldap.manager.uid</code>	Back-end LDAP server administrator or privileged user ID, (for example, <code>cn=Directory Manager</code>) who can save data in LDAP.
<code>datasource.ldap.manager.password</code>	Password for the <code>datasource.ldap.manager.uid</code> , establishes connections with the LDAP directory used for data storage.
<code>datasource.ldap.server.url</code>	"ldap://" URL to the LDAP directory used for data storage.
<code>datasource.ldap.server.root</code>	Root entry of the LDAP directory used for data storage (e.g., <code>dc=acumenat, dc=com</code>).

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

Table 10-9 File Datastore Configuration

UDDI Property Key	Description
<code>datasource.file.directory</code>	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 10-23](#), 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 10-5](#) 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.	Valid tModelKey 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>

```

