Oracle® Fusion Middleware

Developing JAX-WS Web Services for Oracle WebLogic Server

14c (14.1.1.0.0)
F18294-03
January 2023

ORACLE

Oracle Fusion Middleware Developing JAX-WS Web Services for Oracle WebLogic Server, 14c (14.1.1.0.0)

F18294-03

Contents

4 Programming the JWS File

5 Using JAXB Data Binding

13 Roadmap for Developing Reliable Web Services and Clients

14 Using Web Services Reliable Messaging

15 Using Web Services Atomic Transactions

16 Optimizing XML Transmission Using Fast Infoset

17 Using SOAP Over JMS Transport

18 Creating and Using SOAP Message Handlers

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

Preface

This documentation describes how to develop Java EE web services for Oracle WebLogic Server 14c using the Java API for XML-based Web services (JAX-WS).

Audience

This document is a resource for software developers who develop Java EE web services for Oracle WebLogic Server 14c using the Java API for XML-based Web services (JAX-WS). It is assumed that the reader is familiar with Java EE and JAX-WS concepts.

Documentation Accessibility

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

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Resources

For an overview of WebLogic web services, standards, samples, and related documentation, see *Understanding WebLogic Web Services for Oracle WebLogic Server*.

For information about WebLogic web service security, see:

- *Securing Web Services and Managing Policies with Oracle Web Services Manager*
- *Securing WebLogic Web Services for Oracle WebLogic Server*

New and Changed WebLogic Server Features

For a comprehensive listing of the new and changed WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I
Introduction

Part I introduces developing WebLogic (Java EE) web services using the Java API for XML-based Web services (JAX-WS).

Sections include:

- Introduction to JAX-WS Web Services
- Examples for JAX-WS Web Service Developers

1

Introduction to JAX-WS Web Services

This chapter provides an overview of developing WebLogic (Java EE) web services using the Java API for XML-based Web Services (JAX-WS). JAX-WS is a standards-based API for coding, assembling, and deploying Java web services.
This chapter includes the following sections:

- Overview of JAX-WS Web Service Development
- Roadmap for Implementing JAX-WS Web Services

For definitions of unfamiliar terms found in this and other books, see the *Glossary*.

Overview of JAX-WS Web Service Development

WebLogic web services are implemented according to the *JSR 109: Implementing Enterprise Web Services* specification (http://www.jcp.org/en/jsr/detail?id=109), which defines the standard Java EE runtime architecture for implementing web services in Java. The specification also describes a standard Java EE web service packaging format, deployment model, and runtime services, all of which are implemented by WebLogic web services.

The following sections describe:

- The Programming Model—Metadata Annotations
- The Development Model—Bottom-up and Top-down

The Programming Model—Metadata Annotations

The *JSR 109: Implementing Enterprise Web Services* specification (http://www.jcp.org/en/jsr/detail?id=109) describes that a Java EE web service is implemented by one of the following components:

- A Java class running in the Web container.
- A stateless or singleton session EJB running in the EJB container.

The code in the Java class or EJB implements the business logic of your web service. Oracle recommends that, instead of coding the raw Java class or EJB directly, you use the JWS annotations programming model, which makes programming a WebLogic web service much easier.

This programing model takes advantage of the JDK metadata annotations feature in which you create an annotated Java file and then use Ant tasks to compile the file into a Java class and generate all the associated artifacts. The Java Web Service (JWS) annotated file is the core of your web service. It contains the Java code that determines how your web service behaves. A JWS file is an ordinary Java class file that uses annotations to specify the shape and characteristics of the web service. The JWS annotations you can use in a JWS file include the standard ones defined by the *Web Services Metadata for the Java Platform* specification (http://www.jcp.org/en/jsr/detail?id=181) as well as a set of other standard or WebLogic-specific annotations, depending on the type of web service you are creating.

Once you have coded the basic WebLogic web service, you can program and configure additional advanced features. For example, you can specify that the SOAP messages be digitally signed and encrypted (as specified by the WS-Security specification at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss). You configure these more advanced features of WebLogic web services using WS-Policy files, which is an XML file that adheres to the WS-Policy specification and contains security-specific or web service reliable messaging-specific XML elements that describe the security and reliable-messaging configuration, respectively.

The Development Model—Bottom-up and Top-down

There are two approaches to web service development: bottom-up and top-down. Each approach is described in the following sections.

Bottom-up Approach: Starting from Java

In the bottom-up approach, you develop your the JWS file from scratch. After you create the JWS file, you use the `jwsc` WebLogic web service Ant task to compile the JWS file, as described by the *JSR 109: Implementing Enterprise Web Services* specification at http://www.jcp.org/en/jsr/detail?id=109.

The `jwsc` Ant task always compiles the JWS file into a plain Java class; the only time it implements a stateless or singleton session EJB is if you implement a stateless or singleton session EJB in your JWS file. The `jwsc` Ant task also generates all the supporting artifacts for the web service, packages everything into an archive file, and creates an Enterprise Application that you can then deploy to WebLogic Server.

By default, the `jwsc` Ant task packages the web service in a standard Web application WAR file with all the standard WAR artifacts. The WAR file, however, contains additional artifacts to indicate that it is also a web service; these additional artifacts include deployment descriptor files, the WSDL file that describes the public contract of the web service, and so on. If you execute `jwsc` against more than one JWS file, you can choose whether `jwsc` packages the web services in a single WAR file or each web service in a separate WAR file. In either case, `jwsc` generates a single Enterprise Application.

If you implement a stateless or singleton session EJB in your JWS file, then the `jwsc` Ant task packages the web service in a standard EJB JAR with all the usual artifacts, such as the `ejb-jar.xml` and `weblogic-ejb.jar.xml` deployment descriptor files. The EJB JAR file also contains additional web service-specific artifacts, as described in the preceding paragraph, to indicate that it is a web service. Similarly, you can choose whether multiple JWS files are packaged in a single or multiple EJB JAR files.

Alternatively, you can specify that your session EJB be packaged as a Web application WAR file by updating the `jwsc` Ant task in your `build.xml` file to enable the `ejbWsInWar` attribute in the `module` child element. For more information, see jwsc in *WebLogic Web Services Reference for Oracle WebLogic Server*.

For more information about the bottom-up approach, see Developing WebLogic Web Services Starting From Java: Main Steps.

Top-down Approach: Starting from WSDL

In the top-down approach, you create the web service from a WSDL file. You can use the `wsdlc` Ant task to generate a partial implementation of the web service described by the WSDL file. The `wsdlc` Ant task generates the JWS service endpoint interface (SEI), the stubbed-out JWS class file, JavaBeans that represent the XML Schema data types, and so on, into output directories.

After running the `wsdlc` Ant task, (which typically you only do once) you update the generated JWS implementation file, for example, to add Java code to the methods so that they function as defined by your business requirements. The generated JWS implementation file does not initially contain any business logic because the `wsdlc` Ant task does not know how you want your web service to function, although it does know the shape of the web service, based on the WSDL file.

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

For more information about the top-down approach, see Developing WebLogic Web Services Starting From a WSDL File: Main Steps.

Roadmap for Implementing JAX-WS Web Services

The following table provides a roadmap of common tasks for developing, packaging and deploying, invoking, and administering JAX-WS web services and clients using WebLogic Server.

Table 1-1 Roadmap for Implementing JAX-WS Web Services

This chapter . . .	Describes how to . . .
Developing Basic JAX-WS Web Services	Develop basic JAX-WS web services using the WebLogic development environment. Program the JWS file that implements your web service and use the Java Architecture for XML Binding (JAXB) data binding.
Developing Basic JAX-WS Web Service Clients	Develop WebLogic web service clients using JAX-WS and apply best practices.
Developing Advanced Features of JAX-WS Web Services	Develop advanced features of WebLogic web services using JAX-WS. Advanced features include asynchronous clients, reliable messaging, atomic transactions, and so on. Test and monitor web services.
Reference	Use pre-packaged WS-Policy files for web services reliable messaging and Make Connection, use batch reliable messaging, and migrate JAX-RPC web services and clients to JAX-WS.

> **Note:**
>
> The JAX-WS implementation in Oracle WebLogic Server is extended from the JAX-WS Reference Implementation (RI) developed by the Glassfish Community (see https://github.com/eclipse-ee4j/metro-jax-ws). All features defined in the JAX-WS specification (JSR-224) are fully supported by Oracle WebLogic Server.
>
> The JAX-WS RI also contains a variety of extensions, provided by Glassfish contributors. Unless specifically documented, JAX-WS RI extensions are not supported for use in Oracle WebLogic Server.

For an overview of WebLogic web services, standards, samples, and related documentation, see *Understanding WebLogic Web Services for Oracle WebLogic Server*.

For information about WebLogic web service security, see:

- *Securing WebLogic Web Services for Oracle WebLogic Server*

2

Examples for JAX-WS Web Service Developers

This chapter summarizes the examples for developing WebLogic web services using the Java API for XML-based Web services (JAX-WS).

Table 2-1 Examples for JAX-WS Web Service Developers

Example	For More Information
Web service sample applications	Samples for Java EE Web Service Developers in *Understanding WebLogic Services for Oracle WebLogic Server*
Common web service code examples	Examples of Developing JAX-WS Web Services
Common web service client examples	Examples of Developing JAX-WS Web Service Clients
Advanced web service client example	Example 7-1
Asynchronous web service client example	Example 11-1
Reliable web service client example	Example 13-1

Part II
Developing Basic JAX-WS Web Services

Part II describes how to develop basic WebLogic web services using Java API for XML-based Web Services (JAX-WS).

Sections include:

3

Developing JAX-WS Web Services

This chapter describes the iterative development process for WebLogic web services using Java API for XML-based Web Services (JAX-WS).
This chapter includes the following topics:

Overview of the WebLogic Web Service Programming Model

The WebLogic web services programming model centers around *JWS files*—Java files that use *JWS annotations* to specify the shape and behavior of the web service—and Ant tasks that execute on the JWS file. JWS annotations are based on the metadata feature, introduced in Version 5.0 of the JDK (specified by JSR-175 at `http://www.jcp.org/en/jsr/detail?id=175`) and include standard annotations defined by *Web Services Metadata for the Java Platform* specification (JSR-181), described at `http://www.jcp.org/en/jsr/detail?id=181`, the JAX-WS specification (JSR-224), described at `https://www.jcp.org/en/jsr/detail?id=224`, as well as additional ones. For a complete list of JWS annotations that are supported, see Web Service Annotation Support in *WebLogic Web Services Reference for Oracle WebLogic Server*. For additional detailed information about this programming model, see The Programming Model—Metadata Annotations.

Web services can be created using two development methods: bottom-up or top-down. Bottom-up development refers to the process of developing a web service from the underlying Java implementation using SOAP. Top-development describes the development of a web service from the WSDL source.

The following sections describe the high-level steps for iteratively developing a web service, either starting from Java (bottom-up) or starting from an existing WSDL file (top-down):

- Developing WebLogic Web Services Starting From Java: Main Steps

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

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

Configuring Your Domain For Advanced Web Services Features

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

- Asynchronous messaging, as described in Developing Asynchronous Clients.
- Web services reliable messaging, as described in Using Web Services Reliable Messaging.
- Message buffering, as described in Configuring Message Buffering for Web Services.
- Security using WS-SecureConversation, as described in Configuring Message-level Security in *Securing WebLogic Web Services for Oracle WebLogic Server*.

> **Note:**
>
> To configure your domain for SOAP over JMS transport, see Configuring the WebLogic Server Domain for JMS Transport.

Use of the WebLogic Advanced Web Services for JAX-WS Extension template is only required when you need to ensure recoverability of advanced web services. The extension template configures the following resources that support recoverability by enabling WebLogic Server to retain critical state information in the event of a server failure:

- JMS queues for storing reliable messaging requests.
- Default web service persistence configuration that provides a built-in, high-performance storage solution for web services.

The benefits of using the WebLogic Advanced Web Services for JAX-WS Extension template include:

- Web services and clients, by default, use the reliable and high performance WebLogic storage solution.
- Web services that use reliable messaging can buffer incoming asynchronous requests to increase fault tolerance and better absorb load. (This feature is enabled by default.)

- Messages between web services and clients can be configured such that they are fault-tolerant and recoverable in the event of a client failure, service failure, or both.

If you do not use the WebLogic Advanced Web Services for JAX-WS Extension template, you can still develop using the advanced features, but with a reduced quality of service, as describe below:

- By default, the state of an advanced web service is stored in memory; in the event of server failure, the data will be lost.
- Web services will not buffer incoming asynchronous requests when using reliable messaging.
- Web services and clients will not be recoverable in the event of a failure; any in-flight requests between them will be lost.

For more information, see Resources Required by Advanced Web Service Features.

> **Note:**
>
> If you do not apply the WebLogic Advanced Web Services for JAX-WS Extension template to support recoverability:
>
> - You must ensure that buffering is disabled for web services reliable messaging on the destination server. For more information, see Configuring a Non-buffered Destination for a Web Service.
> - Quality of service features that you have configured for your web service may not be in effect. In this case, a message will be logged to the sever log to indicate the feature has been disabled.

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

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

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

Resources Required by Advanced Web Service Features

Table 3-1 lists the resources that are defined automatically when using the WebLogic Advanced Web Services for JAX-WS Extension template.

If you do not apply the extension template, you need to configure the resources manually using the Oracle WebLogic Server Administration Console or WLST. Be sure to configure

JMS targeting according to best practices defined in Best Practices for JMS Beginners and Advanced Users in *Administering JMS Resources for Oracle WebLogic Server*. Specifically:

- Configure a JMS server, Store-and-forward (SAF) service agent, and persistent store on each WebLogic Server. In a cluster, target each to a local migratable target (not the server). The host server's "default migratable target" is sufficient in most cases.
- Target JMS modules to a cluster (or single server if not using a clustered environment).
- Create exactly one subdeployment per module, and populate the subdeployment with the applicable JMS servers or SAF agents only, not the servers.
- Target JMS destinations to the subdeployment (referred to as Advanced Targeting in the WebLogic Server Administration Console). JMS destinations must never use the default targeting option.

The following variables are used in the table:

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

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

Table 3-1 Resources Required by Advanced Web Services Features

Resource Name	Resource Type	Description
`WseeJaxwsJmsModule`	JMS Module	Defines a JMS module that defines the JMS resources needed for advanced web services. All associated targets (JMS servers targeted to a server) on this JMS module will be used to support JAX-WS web services. All servers to which this module is targeted must have the proper web services resources configured. Oracle recommends that you target this module to *all* servers in the domain. **Note**: You must configure the JMS module as a Uniform Distributed Destination (UDD). Any queues that are used by web services on JAX-WS must be Uniform Distributed Queues. Otherwise, an exception is thrown. To configure distributed destinations manually and for more information, see Using Distributed Destination in *Developing JMS Applications for Oracle WebLogic Server*.

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

Resource Name	Resource Type	Description
`WseeJaxwsFileStore_server_designator`	File store	Specifies the file store, or physical store, used by the WebLogic Server to handle the I/O operations to save and retrieve data from the physical storage (such as file, DBMS, and so on). A separate file store is configured on each Managed Server targeted by the `WseeJaxwsJmsModule`, as specified by *`server_designator`*. In a single server domain, the file store is named `WseeJaxwsFileStore`. **Note**: Oracle recommends targeting the file store to a migratable target. To configure the file stores manually, see Using Custom File Stores in *Administering the WebLogic Persistent Store*.
`WseeJaxwsJmsServer_server_designator`	JMS server	Specifies the JMS server management container. A separate JMS Server is configured on each Managed Server targeted by `WseeJaxwsJmsModule`, as specified by *`server_designator`*. The JMS server uses `WseeFileStore_server_designator` as the file store. When configuring the JMS server, Oracle recommends the following: • Target the JMS server to a migratable target. • Set realistic quotas on each JMS server. For more information, see Tuning WebLogic JMS in *Tuning Performance of Oracle WebLogic Server*. To configure the JMS server manually, see JMS Configuration in *Administering JMS Resources for Oracle WebLogic Server*.
`WseeJaxwsJmsServeruniqueID`	JMS subdeployment	Specifies the JMS subdeployment targeting the JMS servers defined on all Managed Servers in the cluster. To configure the JMS subdeployment manually, see Configure subdeployments in JMS system modules in *Oracle WebLogic Server Administration Console Online Help*.
`weblogic.wsee.jaxws.mdb.DispatchPolicy`	Work Manager	Enables an application to execute multiple work items concurrently within a container. One Work Manager is generated for the domain and targeted to all servers to which the `WseeJaxwsJmsModule` is targeted. **Note**: You should not change the name of the Work Manager resource. To configure Work Managers manually, see Description of the Work Manager API in *Developing CommonJ Applications for Oracle WebLogic Server*.

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

Resource Name	Resource Type	Description
`ReliableWseeJaxwsSAFAgent_server_name`	SAF service agent	Provides highly available JMS message production. A separate SAF agent is configured on each Managed Server, as specified by *`server_name`*. The SAF agent uses `WseeFileStore_server_name` as the file store. In a single server domain, the SAF agent is named `ReliableWseeJaxwsSAFAgent`. When configuring the SAF agent, Oracle recommends that you set realistic quotas on each JMS server. For more information, see Tuning WebLogic JMS in *Tuning Performance of Oracle WebLogic Server*. To configure SAF service agents, see Understanding the Store-and-Forward Service in *Administering the Store-and-Forward Service for Oracle WebLogic Server*.
`WseeBufferedRequestQueue_server_designator`	JMS queue	Specifies the queue used for buffered requests. A separate queue is configured on each Managed Server, as specified by *`server_name`*. In a single server domain, the queue is named `WseeBufferedRequestQueue`. In a clustered domain, each JMS queue is prefixed by `dist_`. To configure the queues manually, see Configure queues in *Oracle WebLogic Server Administration Console Online Help*.
`WseeBufferedRequestErrorQueue_server_designator`	JMS queue	Specifies the error queue used for `WseeBufferedRequestQueue` for buffered requests that cannot be processed within the maximum number of retries. A separate queue is configured on each Managed Server, as specified by *`server_name`*. In a single server domain, the queue is named `WseeBufferedRequestErrorQueue`. In a clustered domain, each JMS queue is prefixed by `dist_`. To configure the queues manually, see Configure queues in *Oracle WebLogic Server Administration Console Online Help*.
`WseeBufferedResponseQueue_server_designator`	JMS queue	Specifies the queue used for buffered responses. A separate queue is configured on each Managed Server, as specified by *`server_designator`*. In a single server domain, the queue is named `WseeBufferedResponseQueue`. In a clustered domain, each JMS queue is prefixed by `dist_`. To configure the queues manually, see Configure queues in *Oracle WebLogic Server Administration Console Online Help*.

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

Resource Name	Resource Type	Description
`WseeBufferedResponseErrorQueue_server_designator`	JMS queue	Specifies the error queue used for `WseeBufferedResponseQueue` for buffered responses that cannot be delivered within the maximum number of retries. A separate queue is configured on each Managed Server, as specified by *`server_designator`*. In a single server domain, the queue is named `WseeBufferedResponseErrorQueue`. In a clustered domain, each JMS queue is prefixed by `dist_`. To configure the queues manually, see Configure queues in *Oracle WebLogic Server Administration Console Online Help*.
`WseeStore`	Logical store	Defines the logical store. A separate logical store is configured on each Managed Server targeted by `WseeJaxwsJmsModule`. The logical store points to the `WseeBufferedRequestQueue` queue for its configuration and file store. To configure the logical store manually, see Configuring the Logical Store.

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

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

- Creating a Domain With the Web Services Extension Template
- Extending a Domain With the Web Services Extension Template

Creating a Domain With the Web Services Extension Template

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

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

a. On the Select Optional Configuration screen, at a minimum select **Managed Servers, Clusters, and Machines** to define the Managed Servers and clusters. Select any other items, as desired, and click **Next**.

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

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

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

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

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

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

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

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

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

Extending a Domain With the Web Services Extension Template

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

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

   a. On the Select Optional Configuration screen, at a minimum select **Managed Servers, Clusters, and Machines** to define the Managed Servers and clusters. Select any other items, as desired, and click **Next**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Apply the template to the domain to configure the servers for advanced web service features.
installDir = install_directory/wlserver_10.3
templateLocation = install_directory + '/oracle_common/common/templates/wls/oracle.wls-webservice-jaxws-
template.jar'
addTemplate(templateLocation)

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

# Read the domain
readDomain(domain_dir)

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

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

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

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

sys.path.append(domain_dir)

applyJAXWS(globals())

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

Updating Resources Added After Extending Your Domain

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

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

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

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

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

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

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

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

sys.path.append(domain_dir)

applyJAXWS(globals())

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

Developing WebLogic Web Services Starting From Java: Main Steps

This section describes the general procedure for developing WebLogic web services starting from Java—in effect, coding the JWS file from scratch and later generating the WSDL file that describes the service. See Examples of Developing JAX-WS Web Services for specific examples of this process.

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

> **Note:**
>
> This procedure does not use the WebLogic web services split development directory environment. If you are using this development environment, and would like to integrate web services development into it, see Integrating Web Services Into the WebLogic Split Development Directory Environment for details.

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

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

See Developing Web Service Clients for information on writing client applications that invoke a web service.

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

This section describes the general procedure for developing WebLogic web services based on an existing WSDL file. See Examples of Developing JAX-WS Web Services, for a specific example of this process.

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

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

> **Note:**
>
> This procedure does not use the WebLogic web services split development directory environment. If you are using this development environment, and would like to integrate web services development into it, see Integrating Web Services Into the WebLogic Split Development Directory Environment for details.

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

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.
2	Create a project directory.	The project directory will contain the generated artifacts and the Ant `build.xml` file.
3	Create a basic Ant build file, `build.xml`.	See Creating the Basic Ant build.xml File.
4	Put your WSDL file in a directory that the `build.xml` Ant build file is able to read.	For example, you can put the WSDL file in a `wsdl_files` child directory of the project directory.
5	Run the `wsdlc` Ant task against the WSDL file.	The `wsdlc` Ant task generates the JWS service endpoint interface (SEI), the stubbed-out JWS class file, JavaBeans that represent the XML Schema data types, and so on, into output directories. See Running the wsdlc WebLogic Web Services Ant Task.
6	Update the stubbed-out JWS file generated by the `wsdlc` Ant task.	The `wsdlc` Ant task generates a stubbed-out JWS file. You need to add your business code to the web service so it behaves as you want. See Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc.

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

#	Step	Description
7	Run the `jwsc` Ant task against the JWS file.	Specify the artifacts generated by the `wsdlc` Ant task as well as your updated JWS implementation file, to generate an Enterprise Application that implements the web service. See Running the jwsc WebLogic Web Services Ant Task.
8	Deploy the web service to WebLogic Server.	See Deploying and Undeploying WebLogic Web Services.
9	Browse to the WSDL of the web service.	Browse to the WSDL of the web service to ensure that it was deployed correctly. See Browsing to the WSDL of the Web Service.
10	Test the web service.	See Testing the Web Service.
11	Edit the web service. (Optional)	To make changes to the web service, update the JWS file, undeploy the web service as described in Deploying and Undeploying WebLogic Web Services, then repeat the steps starting from running the `jwsc` Ant task (Step 6).

See Developing Web Service Clients for information on writing client applications that invoke a web service.

Creating the Basic Ant build.xml File

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

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

```
<project default="all">
  <target name="all"
          depends="clean,build-service,deploy" />
  <target name="clean">
    <delete dir="output" />
  </target>
  <target name="build-service">
     <!--add jwsc and related tasks here -->
  </target>
  <target name="deploy">
      <!--add wldeploy task here -->
 </dftarget>
</project>
```

For detailed information about how to integrate and use Ant tasks in your development environment to program a web service and a client application that invokes the web service, see:

- Using Oracle WebLogic Server Ant Tasks in *Understanding WebLogic Web Services for Oracle WebLogic Server*
- Ant Task Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*

- The following sections in *Developing Applications for Oracle WebLogic Server*:
  - Using Ant Tasks to Configure and Use a WebLogic Server Domain
  - wldeploy Ant Task Reference

Running the jwsc WebLogic Web Services Ant Task

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

The `jwsc`-generated artifacts include:

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

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

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

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

The `jwsc` Ant task includes attributes and child elements that enable you to:

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

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

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

```
<target name="build-service">
    <jwsc
      srcdir="src_directory"
      destdir="ear_directory"
      >
      <jws file="JWS_file"
           compiledWsdl="WSDLC_Generated_JAR"
            type="WebService_type"/>
    </jwsc>
  </target>
```

where:

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

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

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

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

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

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

Specifying the Transport Used to Invoke the Web Service

The `<jws>` child element of `jwsc` includes the following optional child elements for specifying the transports (HTTP/S or JMS) that are used to invoke the web service:

- `WLHttpTransport`—Specifies the context path and service URI sections of the URL used to invoke the web service over the HTTP/S transport, as well as the name of the port in the generated WSDL. For more information, see WLHttpTransport in *WebLogic Web Services Reference for Oracle WebLogic Server*.
- `JmsTransportService`—Enables and configures SOAP over JMS transport. Optionally, you can configure the destination name, destination type, delivery mode, request and

response queues, and other JMS transport properties. For more information, see Developing JAX-WS Web Services.

The following guidelines describe the usage of the transport elements for the `jwsc` Ant task:

- The transports you specify to `jwsc` *always override any corresponding transport annotations in the JWS file. In addition, all* attributes of the transport annotation are ignored, even if you have not explicitly specified the corresponding attribute for the transport element, in which case the default value of the transport element attribute is used.
- You can specify both transport elements for a particular JWS file. However, you can specify only *one* instance of a particular transport element. For example, although you cannot specify two different `<WLHttpTransport>` elements for a given JWS file, you can specify one `<WLHttpTransport>` and one `<WLJmsTransport>` element.
- The value of the `serviceURI` attribute can be the same when you specify both `<WLJMSTransport>` and `<WLHttpTransport>`.
- All transports associated with a particular JWS file must specify the *same* `contextPath` attribute value.
- If you specify more than one transport element for a particular JWS file, the value of the `portName` attribute for each element must be unique among all elements. This means that you must explicitly specify this attribute if you add more than one transport child element to `<jws>`, because the default value of the element will always be the same and thus cause an error when running the `jwsc` Ant task.
- If you do not specify any transport as either one of the transport elements to the `jwsc` Ant task or a transport annotation in the JWS file, then the web service's default URL corresponds to the default value of the `WLHttpTransport` element.

Defining the Context Path of a WebLogic Web Service

There are a variety of places where the context path (also called context root) of a WebLogic web service can be specified. This section describes how to determine which is the true context path of the service based on its configuration, even if it is has been set in multiple places.

In the context of this discussion, a web service context path is the string that comes after the `host:port` portion of the web service URL. For example, if the deployed WSDL of a WebLogic web service is as follows:

`http://`*`hostname`*`:7001/financial/GetQuote?WSDL`

The context path for this web service is `financial`.

The following list describes the order of precedence, from most to least important, of all possible context path specifications:

1. The `contextPath` attribute of the `<module>` element and `<jws>` element (when used as a direct child of the `jwsc` Ant task.)
2. The `contextPath` attribute of the `<WLHttpTransport>` child elements of `<jws>`.
3. The default value of the context path, which is the name of the JWS file without any extension.

Assume that you update the `build.xml` file and add a `<WLHttpTransport>` child element to the `<jws>` element that specifies the JWS file and set its `contextPath` attribute to `finance`. The context path of the web service would now be `finance`. If, however, you then group the `<jws>` element (including its child `<WLHttpTransport>` element) under a `<module>` element, and set its `contextPath` attribute to `money`, then the context path of the web service would now be `money`.

If you do not specify *any* `contextPath` attribute in either the JWS file or the `jwsc` Ant task, then the context path of the web service is the default value: the name of the JWS file without its `*.java` extension.

If you group two or more `<jws>` elements under a `<module>` element and do not set the context path using any of the other options listed above, then you *must* specify the `contextPath` attribute of `<module>` to specify the common context path used by all the web services in the module. Otherwise, the default context paths for all the web services in the module are going to be different (due to different names of the implementing JWS files), which is not allowed in a single WAR file.

Examples of Using jwsc

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

```
<taskdef name="jwsc"
         classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
    <jwsc
      srcdir="src"
      destdir="output/helloWorldEar">
      <jws
         file="examples/webservices/hello_world/HelloWorldImpl.java"
         type="JAXWS"/>
    </jwsc>
</target>
```

In the example:

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

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

```
<taskdef name="jwsc"
         classname="weblogic.wsee.tools.anttasks.JwscTask" />
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="output/wsdlcEar">
    <jws
        file="examples/webservices/wsdlc/TemperaturePortTypeImpl.java"
        compiledWsdl="output/compiledWsdl/TemperatureService_wsdl.jar"
```

```
        type="JAXWS"/>
  </jwsc>
</target>
```

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

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

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

Running the wsdlc WebLogic Web Services Ant Task

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

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

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

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

```
<taskdef name="wsdlc"
        classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
<target name="generate-from-wsdl">
  <wsdlc
      srcWsdl="WSDLFile"
      destJwsDir="JWS_interface_directory"
      destImplDir="JWS_implementation_directory"
      packageName="Package_name"
      type="WebService_type"/>
</target>
```

where:

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

  The name of the generated JAR file is *`WSDLFile`*`_wsdl.jar`, where *`WSDLFile`* refers to the root name of the WSDL file. For example, if the name of the WSDL file you specify to the file attribute is `MyService.wsdl`, then the generated JAR file is `MyService_wsdl.jar`.

- *JWS_implementation_directory* refers to the top directory into which the stubbed-out JWS implementation file is generated. The file is generated into a subdirectory hierarchy corresponding to its package name.

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

- *Package_name* refers to the package into which the generated JWS SEI and implementation files should be generated. If you do not specify this attribute, the wsdlc Ant task generates a package name based on the targetNamespace of the WSDL.

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

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

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

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

```
<taskdef name="wsdlc"
        classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
<target name="generate-from-wsdl">
    <wsdlc
        srcWsdl="wsdl_files/TemperatureService.wsdl"
        destJwsDir="output/compiledWsdl"
        destImplDir="impl_output"
        packageName="examples.webservices.wsdlc"
        type="JAXWS" />
</target>
```

In the example:

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

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

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

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

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

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

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

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

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

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

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

- You can include the following annotations from the standard (JSR-181) `javax.jws` package in the JWS implementation file: `@WebService`, `@HandlerChain`, `@SOAPMessageHandler`, and `@SOAPMessageHandlers`. If you specify any other JWS annotation from the `javax.jws` package, the `jwsc` Ant task returns error when you try to compile the JWS file into a web service. For example, if you specify the `@Policy` annotation in a your JWS implementation file, the `jwsc` Ant task throws a compilation error.
- You can specify *only* the `serviceName`, `endpointInterface`, and `targetNamespace` attributes of the `@WebService` annotation. Use the `serviceName` attribute to specify a different `<service>` WSDL element from the one that the `wsdlc` Ant task used, in the rare case that the WSDL file contains more than one `<service>` element. Use the `endpointInterface` attribute to specify the JWS SEI generated by the `wsdlc` Ant task. Use the `targetNamespace` attribute to specify the namespace of a WSDL service, which can be different from the on in JWS SEI.
- You can specify JAX-WS—JSR 224, JAXB (JSR 222)—or Common (JSR 250) annotations, as required. For more information about the annotations that are supported, see JWS Annotation Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*.

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

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

```
package examples.webservices.wsdlc;
import javax.jws.WebService;
/**
 * TemperaturePortTypeImpl class implements web service endpoint interface
 * TemperaturePortType */
@WebService(
  serviceName="TemperatureService",
  endpointInterface="examples.webservices.wsdlc.TemperaturePortType")
public class TemperaturePortTypeImpl implements TemperaturePortType {
  public TemperaturePortTypeImpl() {
  }
  public float getTemp(java.lang.String zipcode)
  {
    //replace with your impl here
     return 0;
  }
}
```

Deploying and Undeploying WebLogic Web Services

Because web services are packaged as Enterprise Applications, deploying a web service simply means deploying the corresponding EAR file or exploded directory.

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

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

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

Using the wldeploy Ant Task to Deploy Web Services

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

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

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

where:

- *DeploymentName* refers to the deployment name of the Enterprise Application, or the name that appears in the WebLogic Server Administration Console under the list of deployments.
- *Source* refers to the name of the Enterprise Application EAR file or exploded directory that is being deployed. By default, the `jwsc` Ant task generates an exploded Enterprise Application directory.
- *AdminUser* refers to administrative username.
- *AdminPassword* refers to the administrative password.
- *AdminServerURL* refers to the URL of the Administration Server, typically `t3://localhost:7001`.
- *ServerName* refers to the name of the WebLogic Server instance to which you are deploying the web service.

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

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

To actually deploy the web service, execute the `deploy` target at the command-line:

```
prompt> ant deploy
```

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

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

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

Using the Administration Console to Deploy Web Services

To use the WebLogic Server Administration Console to deploy the web service, first invoke it in your browser using the following URL:

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

where:

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

Then use the deployment assistants to help you deploy the Enterprise application. See Enterprise Applications in the *Oracle WebLogic Server Administration Console Online Help*.

Browsing to the WSDL of the Web Service

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

The following URL shows how to display the web service WSDL in your browser:

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

where:

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

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

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

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

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

Configuring the Server Address Specified in the Dynamic WSDL

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

```
<definitions name="ComplexServiceDefinitions"
             targetNamespace="http://example.org">
...
  <service name="ComplexService">
    <port binding="s0:ComplexServiceSoapBinding" name="ComplexServicePort">
      <s1:address location="http://myhost:7101/complex/ComplexService"/>
    </port>
  </service>
</definitions>
```

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

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

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

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

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

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

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

- If the web service is deployed to a cluster, the following values are used in the server address of the dynamic WSDL, in order of precedence:
  - Configured network channel, as described in Configuring the Identity of the Front-end SOAP Router Using Network Channels.

  - `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` configured for the cluster, as described in Configure HTTP Settings for a Cluster in *Oracle WebLogic Server Administration Console Online Help*.
  - `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` configured for the local server, as described in Configure HTTP Protocol in *Oracle WebLogic Server Administration Console Online Help*.
  - If none of the above items are set, the `Cluster Address` **must** be set for the cluster, as described in Configure Clusters in *Oracle WebLogic Server Administration Console Online Help*. The server channel for the specified protocol from the request URL (for example, `http`) will be used to generate the cluster address that is displayed in the WSDL.
- If the web service is deployed to an individual server, the `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` configured for the local server are used in the server address of the dynamic WSDL, as described in Configure HTTP Protocol in *Oracle WebLogic Server Administration Console Online Help*.

Web service is a callback service

1. If the callback service is deployed to a cluster, the following values are used in the server address of the dynamic WSDL, in order of precedence:
   - Configured network channel, as described in Configuring the Identity of the Front-end SOAP Router Using Network Channels.
   - `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` configured for the cluster, as described in Configure HTTP Settings for a Cluster in *Oracle WebLogic Server Administration Console Online Help*.
   - `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` configured for the local server, as described in Configure HTTP Protocol in *Oracle WebLogic Server Administration Console Online Help*.
   - `Cluster Address` for the cluster, as described in Configure Clusters in *Oracle WebLogic Server Administration Console Online Help*. The `Cluster Address` is required if no other values are set.
2. If the callback service is deployed to an individual server, the `Frontend Host`, `Frontend HTTP Port`, and `Frontend HTTPS Port` configured for the local server are used in the server address of the dynamic WSDL, as described in Configure HTTP Protocol in *Oracle WebLogic Server Administration Console Online Help*.
3. If none of the preceding values are set, but the `Listen Address` of the server to which the callback service is deployed is set, then WebLogic Server uses this value in the server address.

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

Web service is invoked using a proxy server

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

See Configure HTTP Settings for a Cluster or Configure HTTP Protocol in *Oracle WebLogic Server Administration Console Online Help*.

Testing the Web Service

After you have deployed a WebLogic web service, you can test basic and advanced features of your web service, such as security, quality of service (QoS), HTTP headers, and so on. You can also perform stress testing of the security features. For information about testing web services using the Web Services Test Client or Fusion Middleware Control Test Web Service page, see Testing Web Services in *Administering Web Services*.

Integrating Web Services Into the WebLogic Split Development Directory Environment

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

For detailed information about the WebLogic split development directory environment, see Creating a Split Development Directory Environment in *Developing Applications for Oracle WebLogic Server* and the `splitdir/helloWorldEar` example installed with WebLogic Server, located in the *`ORACLE_HOME`*`/wlserver/samples/server/examples/src/examples` directory, where *`ORACLE_HOME`* represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

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

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

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

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

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

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

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

4. In the `build.xml` file that builds the Enterprise application, create a new target to build the web service, adding a call to the `jwsc` WebLogic web service Ant task, as described in Running the jwsc WebLogic Web Services Ant Task.

   The `jwsc srcdir` attribute should point to the top-level directory that contains the JWS file (`helloWebService` in this example). The `jwsc destdir` attribute should

point to the same destination directory you specify for `wlcompile`, as shown in the following example:

```
<target name="build.helloWebService">
  <jwsc
      srcdir="helloWebService"
      destdir="destination_dir"
      keepGenerated="yes" >
      <jws file="examples/splitdir/hello/HelloWorldImpl.java"
       type="JAXWS" />
  </jwsc>
</target>
```

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

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

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

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

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

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

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

   For example, add the following to the `application.xml` file for the helloWorld web service:

```
<application>
...
  <module>
    <web>
```

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

> **Note:**
>
> The `jwsc` Ant task always generates a Web Application WAR file from the JWS file that implements your web service, unless you JWS file defines an EJB via the `@Stateless` annotation. In that case you must add an `<ejb>` module element to the `application.xml` file instead.

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

4

Programming the JWS File

This chapter describes how to program the JWS file that implements your WebLogic web service using Java API for XML-based web services (JAX-WS).
This chapter includes the following sections:

Overview of JWS Files and JWS Annotations

There are two ways to program a WebLogic web service from scratch:

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

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

The Java web service (JWS) annotated file is the core of your web service. It contains the Java code that determines how your web service behaves. A JWS file is an ordinary Java class file that uses Java metadata annotations to specify the shape and characteristics of the web service. The JWS annotations you can use in a JWS file include the standard ones defined by the web services Metadata for the Java Platform specification (JSR-181), described at `http://www.jcp.org/en/jsr/detail?id=181`, plus a set of additional annotations based on the type of web service you are building—JAX-WS or JAX-RPC. For a complete list of JWS annotations that are supported for JAX-WS and JAX-RPC web services, see Web Service Annotation Support in *WebLogic Web Services Reference for Oracle WebLogic Server*.

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

Java Requirements for a JWS File

When you program your JWS file, you must follow a set of requirements, as specified by the *Web Services Metadata for the Java Platform* specification (JSR-181) at `http://www.jcp.org/en/jsr/detail?id=181`. In particular, the Java class that implements the web service:

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

Programming the JWS File: Typical Steps

The following procedure describes the typical steps for programming a JWS file that implements a web service.

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

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

Table 4-1 Steps to Program the JWS File

#	Step	Description

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

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

Example of a JWS File

The following sample JWS file shows how to implement a simple web service.

```
package examples.webservices.simple;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
// Standard JWS annotation that specifies that the porType name of the Web
// Service is "SimplePortType", the service name is "SimpleService", and the
// targetNamespace used in the generated WSDL is "http://example.org"
@WebService(name="SimplePortType", serviceName="SimpleService",
            targetNamespace="http://example.org")
// Standard JWS annotation that specifies the mapping of the service onto the
// SOAP message protocol.  In particular, it specifies that the SOAP messages
// are document-literal-wrapped.
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello
 *
 */
public class SimpleImpl {
  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
 // same as the method name: sayHello.
  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

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

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

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

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

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

- `endpointInterface`—Fully qualified name of an existing service endpoint interface file. This annotation allows the separation of interface definition from the implementation. If you specify this attribute, the `jwsc` Ant task does not generate

the interface for you, but assumes you have created it and it is in your CLASSPATH.

- `portname`—Name that is used in the `wsdl:port`.

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

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

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

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

In the example, the web service uses document-wrapped-style encodings and literal message formats, which are also the default formats if you do not specify the `@SOAPBinding` annotation. In general, document-literal-wrapped web services are the most interoperable type of web service.

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

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

Table 4-2 Attributes of the @SOAPBinding Annotation

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

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

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

```
public class SimpleImpl {
  @WebMethod(operationName="sayHelloOperation")
  public String sayHello(String message) {
```

```
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
...
```

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

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

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

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

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

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

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

None of the attributes of the `@WebMethod` annotation is required. See the *Web Services Metadata for the Java Platform* (JSR 181) at `http://www.jcp.org/en/jsr/detail?id=181` for the default values of each attribute, as well as additional information about the `@WebMethod` and `@Oneway` annotations.

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

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

```
public class SimpleImpl {
  @WebMethod()
  @WebResult(name="IntegerOutput",
             targetNamespace="http://example.org/docLiteralBare")
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/docLiteralBare")
      int input)
  {
      System.out.println("echoInt '" + input + "' to you too!");
      return input;
  }
...
```

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

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

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

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

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

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

```
public class Simple {
  @WebMethod()
  @WebResult(name="IntegerOutput",
             targetNamespace="http://example.org/docLiteralBare")
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/docLiteralBare")
      int input)
  {
      System.out.println("echoInt '" + input + "' to you too!");
      return input;
  }
...
```

In the example, the name of the return value of the `echoInt` operation in the generated WSDL is `IntegerOutput`; if the `@WebResult` annotation were not present in the JWS file, the name of the return value in the generated WSDL file would be the hard-coded name `return`.

The `targetNamespace` attribute specifies that the XML namespace for the return value is `http://example.org/docLiteralBare`; this attribute is relevant only when using document-style SOAP bindings where the return value maps to an XML element.

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

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

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

```
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;
 public class Simple {
  @WebService()
  @BindingType(value=SOAPBinding.SOAP12HTTP_BINDING)
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/docLiteralBare")
      int input)
  {
      System.out.println("echoInt '" + input + "' to you too!");
      return input;
  }
...
```

In the example, the deployed endpoint would use the SOAP 1.2 over HTTP binding. If not specified, the binding defaults to SOAP 1.1 over HTTP.

Table 4-3 lists the bindings that are supported for JAX-WS web services.

Table 4-3 Bindings Supported for JAX-WS Web Services

Binding	Description
`javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING`	SOAP 1.2 over HTTP binding.
`javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`	SOAP 1.1 over HTTP binding. This is the default for SOAP over HTTP transport connection protocol.
`javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING`	SOAP 1.2 over HTTP and Message Transmission Optimized Mechanism (MTOM) binding.
`javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING`	SOAP 1.1 over HTTP and Message Transmission Optimized Mechanism (MTOM) binding.

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

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

For more information about the `@BindingType` annotation, see Annotations in the JAX-WS specification at `https://jcp.org/en/jsr/detail?id=224`.

Accessing Runtime Information About a Web Service

When a client application invokes a WebLogic web service that was implemented with a JWS file, WebLogic Server automatically creates a *context* that the web service or client can use to access, and sometimes change, runtime information about the service.

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

- `javax.xml.ws.BindingProvider` (`http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/BindingProvider.html`)—From the client application, access the request and response context of the protocol binding. See Accessing the Protocol Binding Context.
- `javax.xml.ws.WebServiceContext` (`http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/WebServiceContext.html`)—From the web service, access runtime message context and security information relative to a request being served. Typically, a `WebServiceContext` is injected into an endpoint using the `@Resource` annotation. See Accessing the Web Service Context.
- `javax.xml.ws.handler.MessageContext` (`http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/MessageContext.html`)—Access a set of runtime properties from a message handler—from the client application or web service—or directly from the `WebServiceContext` from a web service. See Using the MessageContext Property Values.

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

Accessing the Protocol Binding Context

> **Note:**
>
> The `com.sun.xml.ws.developer.JAXWSProperties` and `com.sun.xml.ws.client.BindingProviderProperties` APIs are supported as an extension to JDK 8.0. Because the APIs are not provided as part of the JDK 8.0 kit, they are subject to change.

The `javax.xml.ws.BindingProvider` interface enables you to access from the client application the request and response context of the protocol binding. For more information, see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/BindingProvider.html`. For more information about developing web service client files, see Developing Web Service Clients.

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

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

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

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

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

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

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

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

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

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

  ```
  Map requestContext = ((BindingProvider)port).getRequestContext();
  requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                      "http://examples.com/HelloWorldImpl/HelloWorldService");
  requestContext.put(JAXWSProperties.CONNECT_TIMEOUT, 300);
  requestContext.put(BindingProviderProperties.REQUEST_TIMEOUT, 300);
  ../.
  ```

```
Map responseContext = ((BindingProvider)port).getResponseContext();
Integer responseCode =
         (Integer)responseContext.get(MessageContext.HTTP_RESPONSE_CODE);
```

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

Table 4-4 Methods of the BindingProvider

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

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

Table 4-5 Properties of BindingProvider

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

In addition, in the previous example:

- The `JAXWSProperties.CONNECT_TIMEOUT` property is used to define the connection timeout. For a complete list of `JAXWSProperties` that you can set, see the `com.sun.xml.ws.developer.JAXWSProperties` Javadoc at https://www.javadoc.io/doc/com.sun.xml.ws/jaxws-rt/2.3.2/com.sun.xml.ws.jaxws/com/sun/xml/ws/developer/JAXWSProperties.html.
- The `BindingProviderProperties.REQUEST_TIMEOUT` property is used to define the request timeout. For a complete list of `BindingProviderProperties` that you can set, see the `com.sun.xml.ws.client.BindingProviderProperties` Javadoc at https://www.javadoc.io/doc/com.sun.xml.ws/jaxws-rt/latest/com.sun.xml.ws/com/sun/xml/ws/client/BindingProviderProperties.html.

Accessing the Web Service Context

The `javax.xml.ws.WebServiceContext` interface enables you to access from the web service runtime message context and security information relative to a request being served. Typically, a `WebServiceContext` is injected into an endpoint using the `@Resource` annotation. For more information, see http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/WebServiceContext.html.

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

```
package examples.webservices.jws_context;
import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.Map;
import javax.xml.ws.WebServiceContext;
import javax.annotation.Resource;
import javax.xml.ws.handler.MessageContext;
@WebService(name="JwsContextPortType", serviceName="JwsContextService",
            targetNamespace="http://example.org")
/**
 * Simple web service to show how to use the @Context annotation.
 */
public class JwsContextImpl {
  @Resource
  private WebServiceContext ctx;
  @WebMethod()
  public String msgContext(String msg) {
    MessageContext context=ctx.getMessageContext();
    Map requestHeaders = (Map)context.get(MessageContext.HTTP_REQUEST_HEADERS);
  }
}
```

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

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

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

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

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

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

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

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

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

For more information about the `MessageContext` property values, see Using the MessageContext Property Values.

The following table summarizes the methods of the `javax.xml.ws.WebServiceContext` that you can use in your JWS file to access runtime information about the web service. For more information, see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/WebServiceContext.html`.

Table 4-6 Methods of the WebServiceContext

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

Using the MessageContext Property Values

The following table defined the `javax.xml.ws.handler.MessageContext` property values that you can access from a message handler—from the client application or web service—or directly from the `WebServiceContext` from the web service. For more information, see the `javax.xml.ws.handler.MessageContext` Javadocs at `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/MessageContext.html`.

Table 4-7 Properties of MessageContext

Property	Type	Description
`HTTP_REQUEST_HEADERS`	`java.util.Map`	Map of HTTP request headers for the request message.
`HTTP_REQUEST_METHOD`	`java.lang.String`	HTTP request method for example GET, POST, or PUT.
`HTTP_RESPONSE_CODE`	`java.lang.Integer`	HTTP response status code for the last invocation.
`HTTP_RESPONSE_HEADERS`	`java.util.Map`	HTTP response headers.
`INBOUND_MESSAGE_ATTACHMENTS`	`java.util.Map`	Map of attachments for the inbound messages.
`MESSAGE_OUTBOUND_PROPERTY`	`java.lang.Boolean`	Message direction. This property is `true` for outbound messages and `false` for inbound messages.
`OUTBOUND_MESSAGE_ATTACHMENTS`	`java.util.Map`	Map of attachments for the outbound messages.

Table 4-7 (Cont.) Properties of MessageContext

Property	Type	Description
PATH_INFO	java.lang.String	Request path information.
QUERY_STRING	java.lang.String	Query string for request.
REFERENCE_PARAMETERS	java.awt.List	WS-Addressing reference parameters. The list must include all SOAP headers marked with the wsa:IsReferenceParameter="true" attribute.
SERVLET_CONTEXT	javax.servlet.ServletContext	Servlet context object associated with request.
SERVLET_REQUEST	javax.servlet.http.HttpServletRequest	Servlet request object associated with request.
SERVLET_RESPONSE	javax.servlet.http.HttpServletResponse	Servlet response object associated with request.
WSDL_DESCRIPTION	org.xml.sax.InputSource	Input source (resolvable URI) for the WSDL document.
WSDL_INTERFACE	javax.xml.namespace.QName	Name of the WSDL interface or port type.
WSDL_OPERATION	javax.xml.namespace.QName	Name of the WSDL operation to which the current message belongs.
WSDL_PORT	javax.xml.namespace.QName	Name of the WSDL port to which the message was received.
WSDL_SERVICE	javax.xml.namespace.QName	Name of the service being invoked.

Should You Implement a Stateless or Singleton Session EJB?

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

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

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

By default, EJB-based web services are packaged as a JAR file. When building the EJB-based web service, you can specify that it be packaged as a WAR file by updating the `jwsc` Ant task in your `build.xml` file to enable the `ejbWsInWar` attribute in the

`module` child element. For more information, see jwsc in *WebLogic Web Services Reference for Oracle WebLogic Server*.

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

- Import the EJB annotations, all of which are in the `javax.ejb` package. At a minimum you need to import the `@Stateless` or `@Singleton` annotation. You can also specify additional EJB annotations in your JWS file to specify the shape and behavior of the EJB. For more information, see the `javax.ejb` Javadoc at https://javaee.github.io/javaee-spec/javadocs/javax/ejb/package-summary.html.

  For example:

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

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

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

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

```
package examples.webservices.jaxws;

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

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

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

//Standard EJB annotation
@Stateless
public class SimpleEjbImpl {

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

   // The WebMethod annotation exposes the subsequent method as a public
   // operation on the Web Service.
   @WebMethod()
```

```
   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public String sayHello(String s) throws SystemException {
      Transaction transaction =
         TransactionHelper.getTransactionHelper().getTransaction();
      int status = transaction.getStatus();
      if (Status.STATUS_ACTIVE != status)
           throw new IllegalStateException("transaction did not start,
           status is: " + status + ", check ejb annotation processing");

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

Programming the User-Defined Java Data Type

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

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

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

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

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

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

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

```
package examples.webservices.complex;
/**
 * Defines a simple JavaBean called BasicStruct that has integer, String,
 * and String[] properties
 */
public class BasicStruct {
  // Properties
```

```
  private int intValue;
  private String stringValue;
  private String[] stringArray;
  // Getter and setter methods
  public int getIntValue() {
    return intValue;
  }
  public void setIntValue(int intValue) {
    this.intValue = intValue;
  }
  public String getStringValue() {
    return stringValue;
  }
  public void setStringValue(String stringValue) {
    this.stringValue = stringValue;
  }
  public String[] getStringArray() {
    return stringArray;
  }
  public void setStringArray(String[] stringArray) {
    this.stringArray = stringArray;
  }
}
```

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

```
package examples.webservices.complex;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
// Import the WebLogic-specific JWS annotation interface
// Import the BasicStruct JavaBean
import examples.webservices.complex.BasicStruct;
@WebService(serviceName="ComplexService", name="ComplexPortType",
            targetNamespace="http://example.org")
...
public class ComplexImpl {
  @WebMethod(operationName="echoComplexType")
  public BasicStruct echoStruct(BasicStruct struct)
  {
    return struct;
  }
}
```

Invoking Another Web Service from the JWS File

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

See Invoking a Web Service from Another WebLogic Web Service for detailed instructions.

Using SOAP 1.2

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

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

```
package examples.webservices.soap12;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.SOAPBinding;
@WebService(name="SOAP12PortType",
            serviceName="SOAP12Service",
            targetNamespace="http://example.org")
@BindingType(value = SOAPBinding.SOAP12HTTP_BINDING)
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHello.  The class uses SOAP 1.2
 * as its binding.
 *
 */
public class SOAP12Impl {
  @WebMethod()
  public String sayHello(String message) {
    System.out.println("sayHello:" + message);
    return "Here is the message: '" + message + "'";
  }
}
```

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

Validating the XML Schema

By default, SOAP messages are not validated against their XML schemas. You can enable XML schema validation for document-literal web services on the server or client, as described in the following sections. In the event a SOAP message is invalid, a SOAP fault is returned.

> **Note:**
>
> This feature adds a small amount of extra processing to a web service request.
>
> By default, the stack trace is included in the details of the SOAP fault. To disable the stack trace, see Disabling the Stack Trace from the SOAP Fault.

Enabling Schema Validation on the Server

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

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

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

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

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

Enabling Schema Validation on the Client

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

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

```
package examples.webservices.hello_world.client;
import com.sun.xml.ws.developer.SchemaValidationFeature;
import javax.xml.namespace.QName;
```

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

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

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

JWS Programming Best Practices

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

- When you create a document-literal-bare web service, use the `@WebParam` JWS annotation to ensure that all input parameters for all operations of a given web service have a unique name. Because of the nature of document-literal-bare web services, if you do not explicitly use the `@WebParam` annotation to specify the name of the input parameters, WebLogic Server creates one for you and run the risk of duplicating the names of the parameters across a web service.
- In general, document-literal-wrapped web services are the most interoperable type of web service.
- Use the `@WebResult` JWS annotation to explicitly set the name of the returned value of an operation, rather than always relying on the hard-coded name `return`, which is the default name of the returned value if you do not use the `@WebResult` annotation in your JWS file.

5

Using JAXB Data Binding

This chapter describes how to use Java Architecture for XML Binding (JAXB) data binding. This chapter includes the following topics:

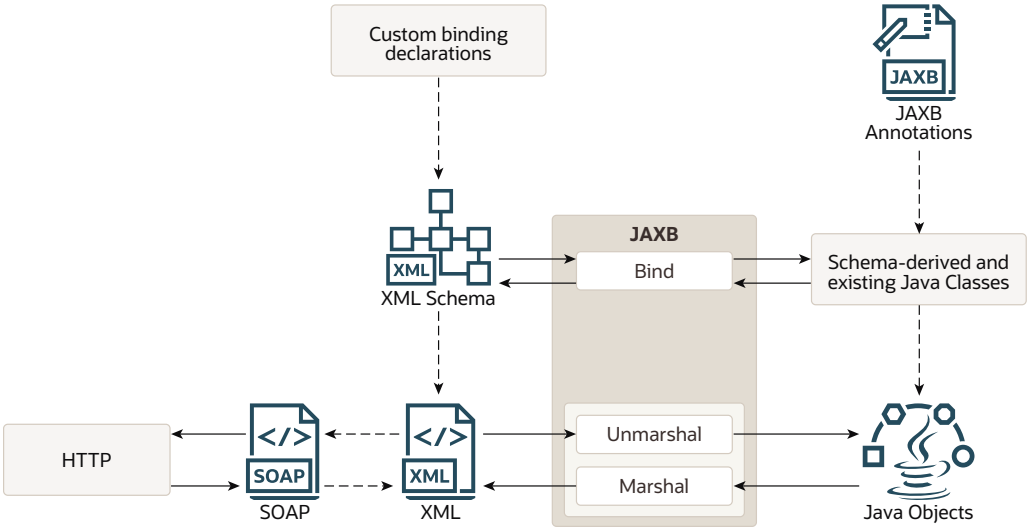
Overview of Data Binding Using JAXB

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

JAX-WS uses JAXB, described at `http://jcp.org/en/jsr/detail?id=222`, to manage all of the data binding tasks. Specifically, JAXB binds Java method signatures and WSDL messages and operations and allows you to customize the mapping while automatically handling the runtime conversion. This makes it easy for you to incorporate XML data and processing functions in applications based on Java technology without having to know much about XML.

The following figure shows the JAXB data binding process.

Figure 5-1 Data Binding With JAXB



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

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

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

WebLogic Server provides two data binding and JAXB providers:

- EclipseLink MOXy, the default in this release of WebLogic Server, is a fully compliant JAXB implementation. In addition to offering the standard JAXB features, EclipseLink MOXy provides useful extensions, such as the ability to use an external metadata file to configure the equivalent of JAXB annotations without modifying the Java source it refers to, and XPath based mapping. The JAXB enhancements can be used in the annotations on a service endpoint interface (SEI) or one of the value types used by the SEI. Users of JAXB in standalone mode can also take advantage of these features.

  Some of the additional extensions offered by EclipseLink MOXy include:

  – Extensions for mapping JPA entities to XML

  - Bidirectional mapping
  - Virtual properties
  - Ability to bootstrap from metadata and generate in-memory domain classes (Dynamic MOXy)

  For a web service, the EclipseLink MOXy extensions can be leveraged on the server side only, and only in the Java to WSDL scenario, in which the SEI and value types can use the extended EclipseLink functionality. For more information about these extensions and EclipseLink MOXy, see *The EclipseLink MOXy (JAXB) User's Guide* at `http://wiki.eclipse.org/EclipseLink/UserGuide/MOXy`.

  No configuration is required to use the EclipseLink MOXy providers.

- Glassfish RI JAXB, which is the default Glassfish JAXB implementation, and was the default JAXB offering in WebLogic Server in previous releases. The Glassfish RI JAXB proprietary features will not work with EclipseLink MOXy. If desired, you can enable the Glassfish RI JAXB data binding and JAXB providers at the server or application level. For more information, see Using the Glassfish RI JAXB Data Binding and JAXB Providers.

The following sections describe how to use JAXB data binding with WebLogic Server and how to configure the Glassfish RI JAXB providers if desired:

- Developing the JAXB Data Binding Artifacts—Describes how to develop the JAXB data binding artifacts using WebLogic Server.
- Standard Data Type Mapping—Describes the standard built-in and user-defined data types that are supported.
- Customizing Java-to-XML Schema Mapping Using JAXB Annotations—Describes how you can control and customize the Java-to-XML Schema mapping using JAXB annotations in the JWS file.
- Customizing XML Schema-to-Java Mapping Using Binding Declarations—Describes how you can control and customize the XML Schema-to-Java mapping using binding declarations that are defined in a separate file or embedded inline.
- Using the Glassfish RI JAXB Data Binding and JAXB Providers—Describes the global server-level and application-level procedures required to configure the Glassfish RI JAXB Data Binding and JAXB providers instead of the default EclipseLink MOXy JAXB providers.

Developing the JAXB Data Binding Artifacts

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

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

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

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

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

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

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

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

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

Standard Data Type Mapping

WebLogic web services support a full set of built-in XML Schema, Java, and SOAP types, as specified by the *JSR 222: Java™ Architecture for XML Binding (JAXB) 2.0* specification at `http://jcp.org/en/jsr/detail?id=222`, that you can use in your web service operations without performing any additional programming steps. Built-in data types are those such as `integer`, `string`, and `time`.

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

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

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

Supported Built-In Data Types

The following sections describe the built-in data types supported by WebLogic web services and the mapping between their XML and Java representations. As long as the data types of the parameters and return values of the back-end components that

implement your web service are in the set of built-in data types, WebLogic Server automatically converts the data between XML and Java.

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

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

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

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

XML Schema Data Type	Java Data Type (lower case indicates a primitive data type)
`anySimpleType` (for xsd:element of this type)	`java.lang.Object`
`anySimpleType` (for xsd:attribute of this type)	`java.lang.String`
`base64Binary`	`byte[]`
`boolean`	`boolean`
`byte`	`byte`
`date`	`java.xml.datatype.XMLGregorianCalendar`
`dateTime`	`javax.xml.datatype.XMLGregorianCalendar`
`decimal`	`java.math.BigDecimal`
`double`	`double`
`duration`	`javax.xml.datatype.Duration`
`float`	`float`
`g`	`java.xml.datatype.XMLGregorianCalendar`
`hexBinary`	`byte[]`
`int`	`int`
`integer`	`java.math.BigInteger`
`long`	`long`
`NOTATION`	`javax.xml.namespace.QName`
`QName`	`javax.xml.namespace.QName`
`short`	`short`
`string`	`java.lang.String`
`time`	`java.xml.datatype.XMLGregorianCalendar`
`unsignedByte`	`short`
`unsignedInt`	`long`

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

XML Schema Data Type	Java Data Type (lower case indicates a primitive data type)
unsignedShort	int

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

XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
     <xsd:sequence>
          <xsd:element name="shipTo" type="USAddress"/>
          <xsd:element name="billTo" type="USAddress"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="items" type="Items"/>
     </xsd:sequence>
     <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
<xsd:complexType name="USAddress">
     <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street" type="xsd:string"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zip" type="xsd:decimal"/>
     </xsd:sequence>
<xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
<xsd:complexType name="Items">
     <xsd:sequence>
          <xsd:element name="item" minOccurs="1" maxOccurs="unbounded">
               <xsd:complexType>
                    <xsd:sequence>
                         <xsd:element name="productName" type="xsd:string"/>
                         <xsd:element name="quantity">
                              <xsd:simpleType>
                                   <xsd:restriction base="xsd:positiveInteger">
                                        <xsd:maxExclusive value="100"/>
                                   </xsd:restriction>
                              </xsd:simpleType>
                         </xsd:element>
                         <xsd:element name="USPrice" type="xsd:decimal"/>
                         <xsd:element ref="comment" minOccurs="0"/>
                         <xsd:element name="shipDate" type="xsd:date"
                              minOccurs="0"/>
                    </xsd:sequence>
                    <xsd:attribute name="partNum" type="SKU" use="required"/>
               </xsd:complexType>
          </xsd:element>
     </xsd:sequence>
</xsd:complexType>
<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
     <xsd:restriction base="xsd:string">
```

```
          <xsd:pattern value="\d{3}-[A-Z]{2}"/>
     </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Default Java Binding

```
import javax.xml.datatype.XMLGregorianCalendar; import java.util.List;
public class PurchaseOrderType {
     USAddress getShipTo(){...}
     void setShipTo(USAddress){...}
     USAddress getBillTo(){...}
     void setBillTo(USAddress){...}
     /** Optional to set Comment property. */
     String getComment(){...}
     void setComment(String){...}
     Items getItems(){...}
     void setItems(Items){...}
     XMLGregorianCalendar getOrderDate()
     void setOrderDate(XMLGregorianCalendar)
};
public class USAddress {
     String getName(){...}
     void setName(String){...}
     String getStreet(){...}
     void setStreet(String){...}
     String getCity(){...}
     void setCity(String){...}
     String getState(){...}
     void setState(String){...}
     int getZip(){...}
     void setZip(int){...}
     static final String COUNTRY="USA";
};
public class Items {
     public class ItemType {
          String getProductName(){...}
          void setProductName(String){...}
          /** Type constraint on Quantity setter value 0..99.*/
          int getQuantity(){...}
          void setQuantity(int){...}
          float getUSPrice(){...}
          void setUSPrice(float){...}
          /** Optional to set Comment property. */
          String getComment(){...}
          void setComment(String){...}
          XMLGregorianCalendar getShipDate();
          void setShipDate(XMLGregorianCalendar);
          /** Type constraint on PartNum setter value "\d{3}-[A-Z]{2}".*/
          String getPartNum(){...} void setPartNum(String){...}
     };
     /** Local structural constraint 1 or more instances of Items.ItemType.*/
     List<Items.ItemType> getItem(){...}
}
public class ObjectFactory {
     // type factories
     Object newInstance(Class javaInterface){...}
     PurchaseOrderType createPurchaseOrderType(){...}
     USAddress createUSAddress(){...}
     Items createItems(){...}
     Items.ItemType createItemsItemType(){...}
```

```
    // element factories
JAXBElement<PurchaseOrderType>createPurchaseOrder(PurchaseOrderType){...}
    JAXBElement<String> createComment(String value){...}
}
```

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

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

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

Java Data Type (lower case indicates a primitive data type)	XML Schema Data Type
boolean	boolean
byte	byte
double	double
float	float
long	long
int	int
javax.activation.DataHandler	base64Binary
java.awt.Image	base64Binary
java.lang.Object	anyType
java.lang.String	string
java.math.BigInteger	integer
java.math.BigDecimal	decimal
java.net.URI	string
java.util.Calendar	dateTime
java.util.Date	dateTime
java.util.UUID	string
javax.xml.datatype.XMLGregorianCalendar	anySimpleType
javax.xml.datatype.Duration	duration
javax.xml.namespace.QName	Qname
javax.xml.transform.Source	base64Binary
short	short

Supported User-Defined Data Types

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

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

Supported XML User-Defined Data Types

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

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

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

Supported Java User-Defined Data Types

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

Table 5-4 Supported Java User-defined Data Types

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

Customizing Java-to-XML Schema Mapping Using JAXB Annotations

If required, you can override the default binding rules for Java-to-XML Schema mapping using JAXB annotations. Table 5-5 summarizes the JAXB mapping annotations that you can include in your JWS file to control how the Java objects are mapped to XML. Each of these annotations are available with the `javax.xml.bind.annotation` package, described at `http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/package-summary.html`.

Table 5-5 JAXB Mapping Annotations

Annotation	Description
`@XmlAccessorType`	Specifies whether fields or properties are mapped by default. See Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation).

Table 5-5 (Cont.) JAXB Mapping Annotations

Annotation	Description
`@XmlElement`	Maps a property contained in a class to a local element in the XML Schema complex type to which the containing class is mapped. See Mapping Properties to Local Elements (@XmlElement).
`@XMLMimeType`	Associates the MIME type that controls the XML representation of the property with a textual representation, such as `image/jpeg`. See Specifying the MIME Type (@XmlMimeType Annotation).
`@XmlRootElement`	Maps a top-level class to a global element in the XML Schema that is used by the WSDL of the web service. See Mapping a Top-level Class to a Global Element (@XmlRootElement).
`@XmlSeeAlso`	Binds other classes when binding the current class. See Binding a Set of Classes (@XmlSeeAlso).
`@XmlType`	Maps a class or enum type to an XML Schema type.See Mapping a Value Class to a Schema Type (@XmlType).

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

- Java-to-XML Mapping for Built-In Data Types
- Supported Java User-Defined Data Types

Example of JAXB Annotations

The following provides an example of the JAXB annotations.

```
@XmlRootElement(name = "ComplexService", namespace ="http://examples.org")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "basicStruct", propOrder = {
    "intValue",
    "stringArray",
    "stringValue"
)
public class BasicStruct {
    protected int intValue;
    @XmlElement(nillable = true)
    protected List<String> stringArray;
    protected String stringValue;
    public int getIntValue() {
        return intValue;
    }
    public void setIntValue(int value) {
        this.intValue = value;
    }
    public List<String> getStringArray() {
        if (stringArray == null) {
            stringArray = new ArrayList<String>();
        }
        return this.stringArray;
    }

    public String getStringValue() {
        return stringValue;
    }
```

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

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

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

- Package
- Top-level class

The `@XmlAccessorType` can be specified with the `@XmlType` (see Mapping a Value Class to a Schema Type (@XmlType)) and `@XmlRootElement` (see Mapping a Top-level Class to a Global Element (@XmlRootElement)) annotations.

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

Table 5-6 Optional Element for @XMLAccessorType Annotation

Element	Description
`value`	Specifies `XMLAccessType.value`, where `value` can be one of the following values: • `FIELD`—Fields are bound to XML. • `PROPERTY`—JavaBean properties and annotated fields are bound to XML. • `PUBLIC_MEMBER`—Public and annotated fields, and JavaBean properties are bound to XML. This is the default. • `NONE`—Only annotated fields and properties are bound to XML.

For more information, see the `javax.xml.bind.annotation.XmlAccessorType` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlAccessorType.html`. An example is provided in Example of JAXB Annotations.

Mapping Properties to Local Elements (@XmlElement)

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

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

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

Table 5-7 Optional Element Summary for @XMLElement Annotation

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

For more information, see the `javax.xml.bind.annotation.XmlElement` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlElement.html`.

Specifying the MIME Type (@XmlMimeType Annotation)

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

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

Table 5-8 Required Element for @XMLMimeType Annotation

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

For more information, see the `javax.xml.bind.annotation.XmlMimeType` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlMimeType.html`.

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

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

- Top-level class
- Enum type

The `@XmlRootElement` can be specified with the `@XmlType` (see Mapping a Value Class to a Schema Type (@XmlType)) and `@XmlAccessorType` (see Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)) annotations.

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

Table 5-9 Optional Elements for @XmlRootElement Annotation

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

For more information, see the `javax.xml.bind.annotation.XmlRootElement` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlRootElement.html`. An example is provided in Example of JAXB Annotations.

Binding a Set of Classes (@XmlSeeAlso)

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

Table 5-10 Optional Element for @XmlSeeAlso Annotation

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

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

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

- Top-level class
- Enum type

The `@XmlType` can be specified with the `@XmlRootElement` (see Mapping a Top-level Class to a Global Element (@XmlRootElement)) and `@XmlAccessorType` (see Specifying Default Serialization of Fields and Properties (@XmlAccessorType Annotation)) annotations.

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

Table 5-11 Optional Elements for @XmlType Annotation

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

For more information, see the `javax.xml.bind.annotation.XmlType` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlType.html`. An example is provided in Example of JAXB Annotations.

Customizing XML Schema-to-Java Mapping Using Binding Declarations

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

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

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

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

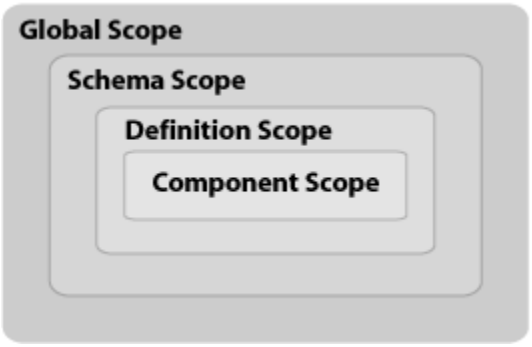
  > **Note:**
  >
  > If customizations are required, Oracle recommends this method to maintain flexibility by keeping the customizations separate from the WSDL or XML Schema document.

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

The binding declarations are semantically equivalent regardless of which method you choose.

Custom binding declarations are associated with a scope, as shown in the following figure.

Figure 5-2 Scopes for Custom Binding Declarations



The following table describes the meaning of each scope.

Table 5-12 Scope for Custom Binding Declarations

Scope	Definition
Global scope	Describes customization values with global scope. Specifically: • For JAX-WS binding declarations, describes customization values that are defined as part of the root element, as described in Specifying the Root Element. • For JAXB annotations, describes customization values that are contained within the `<globalBindings>` binding declaration. Global scope values apply to all of the schema elements in the source schema as well as any schemas that are included or imported.
Schema scope	Describes JAXB customization values that are contained within the `<schemaBindings>` binding declaration. Schema scope values apply to the elements in the target namespace of a schema. **Note**: This scope applies for JAXB binding declarations only.
Definition scope	Describes JAXB customization values that are defined in binding declarations of a type definition or global declaration. Definition scope values apply to elements that reference the type definition or global declaration. **Note**: This scope applies for JAXB binding declarations only.
Component scope	Describes customization values that apply to the WSDL or schema element that was annotated.

Scopes for custom binding declarations adhere to the following inheritance and overriding rules:

- Inheritance—Customization values are inherited from the top down. For example, a WSDL element (JAX-WS) in a component scope inherits a customization value defined in global scope. A schema element (JAXB) in a component scope inherits a customization value defined in global, schema, and definition scopes.
- Overriding—Customization values are overridden from the bottom up. For example, a WSDL element (JAX-WS) in a component scope overrides a customization value defined in global scope. A schema element (JAXB) in a component scope overrides a customization value defined in definition, schema, and global scopes.

The following sections describe how to create custom binding declarations and describe the standard custom binding declarations:

- Creating an External Binding Declarations File
- Embedding Binding Declarations
- JAX-WS Custom Binding Declarations
- JAXB Custom Binding Declarations

For more information about using custom binding declarations, see:

- *WSDL Customization* at `https://javaee.github.io/metro-jax-ws/doc/user-guide/release-documentation.html#users-guide-wsdl-customization`
- "Customizing XML Schema to Java Representation Binding" in the JAXB specification at `http://jcp.org/en/jsr/detail?id=222`.

Creating an External Binding Declarations File

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

The following sections describe:

- Creating an External Binding Declarations File Using JAX-WS Binding Declarations
- Creating an External Binding Declarations File Using JAXB Binding Declarations

Creating an External Binding Declarations File Using JAX-WS Binding Declarations

The following sections describe how to specify the root and child elements of the JAX-WS binding declarations file. For information about the custom binding declarations that you can define, see JAX-WS Custom Binding Declarations.

Specifying the Root Element

The `jaxws:bindings` declaration is the **root** of all other binding declarations and defines the location of the WSDL file and the namespace to which the XML Schema conforms: `http://java.sun.com/xml/ns/jaxws`.

The format of the root declaration is as follows:

```
<jaxws:bindings
    wsdlLocation="uri_of_wsdl"
    jaxws:xmlns="http://java.sun.com/xml/ns/jaxws">
```

uri_of_wsdl specifies the URI of the WSDL file.

The package, wrapper style, and asynchronous mapping customizations, defined in Table 5-5, can be *globally* defined as part of the root binding declaration in the external customization file. Global bindings apply to the entire scope of the `wsdl:definition` in the WSDL referenced by the `wsdlLocation` attribute.

The following provides an example of the root binding element that defines the package name, wrapper style, and asynchronous mapping customizations.

```
<jaxws:bindings
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="http://localhost:7001/simple/SimpleService?WSDL"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
    <package name="example.webservices.simple.simpleservice">
    <enableWrapperStyle>true</enableWrapperStyle>
    <enableAsyncMapping>false</enableAsyncMapping>
</jaxws:bindings>
```

Specifying Child Elements

The root `jaxws:bindings` element can contain **child elements**. You specify the WSDL node that is being customized by passing an XPath expression in the node attribute.

An XML Schema inlined inside a compiled WSDL file can be customized by using standard JAXB bindings. For more information, see "XML Schema Customization" in *WSDL Customization* at `https://javaee.github.io/metro-jax-ws/doc/user-guide/release-`

documentation.html#users-guide-wsdl-customization. For information about the custom JAXB binding declarations that you can define, see JAXB Custom Binding Declarations.

For example, the following example defines the package name as `examples.webservices.complex.complexservice` for the `wsdl:definitions` node of the WSDL document.

```
<jaxws:bindings
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  wsdlLocation="http://localhost:7001/simple/SimpleService?WSDL
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
  <jaxws:bindings node="wsdl:definitions"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <jaxws:package name="examples.webservices.simple.simpleservice"/>
</bindings>
```

Creating an External Binding Declarations File Using JAXB Binding Declarations

The JAXB binding declarations file is an XML document that conforms to the XML Schema for the following namespace: `http://java.sun.com/xml/ns/jaxb`. The following sections describe how to specify the root and child elements of the JAXB binding declarations file. For information about the custom binding declarations that you can define, see JAXB Custom Binding Declarations.

Specifying the Root Element

The `jaxb:bindings` declaration is the **root** of all other binding declarations.The format of the root declaration is as follows:

```
<jaxb:bindings
    schemaLocation="uri_of_schema">
```

uri_of_schema specifies the URI of the XML Schema file.

Specifying Child Elements

The root `jaxb:bindings` element can contain **child elements**. You specify the schema node that is being customized by passing an XPath expression in the node attribute.

For example, the following example defines the package name as `examples.webservices.simple.simpleservice`.

```
<jaxb:bindings
   schemaLocation="simpleservice.xsd">
   <jaxb:bindings node="//xs:simpleType[@name='value1']">
       <jaxb:package name="examples.webservices.simple.simpleservice"/>
   </jaxb:bindings>
</jaxb:bindings>
```

Embedding Binding Declarations

You can embed binding declarations in a WSDL file using one of the following methods:

- Embed a JAX-WS or JAXB binding declaration in the WSDL file using the `jaxws:bindings` element as a WSDL extension. See Embedding JAX-WS or JAXB Binding Declarations in the WSDL File.
- Embed a JAXB binding declaration in the XML Schema as part of an `<appinfo>` element. See Embedding JAXB Binding Declarations in the XML Schema.

Embedding JAX-WS or JAXB Binding Declarations in the WSDL File

You can embed a binding declaration in the WSDL file using the `jaxws:bindings` element as a WSDL extension. For information about the custom binding declarations that you can define, see JAX-WS Custom Binding Declarations.

For example, the following example defines the class name as `SimpleService` for the `SimpleServiceImpl` service endpoint interface (or port).

```
<wsdl:portType name="SimpleServiceImpl">
     <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
           <jaxws:class name="SimpleService"/>
     </jaxws:bindings>
</wsdl:portType>
```

If this binding declaration had not been specified, the class name of the service endpoint interface would be set to the `wsdl:portType` name—`SimpleServiceImpl`—by default.

An XML Schema inlined inside a compiled WSDL file can be customized by using standard JAXB bindings. For more information, see "XML Schema Customization" in *Standard Customizations*, which is available at https://javaee.github.io/metro-jax-ws/doc/user-guide/release-documentation.html#standard-customizations. For information about the custom JAXB binding declarations that you can define, see JAXB Custom Binding Declarations.

Embedding JAXB Binding Declarations in the XML Schema

You can embed a JAXB custom declaration within the `<appinfo>` element of the XML Schema, as illustrated below.

```
<xs:annotation>
    <xs:appinfo>
        <binding declaration>
    </xs:appinfo>
</xs:annotation>
```

For example, the following defines the package name for the schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.w3.org/2001/XMLSchema"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    jaxb:version="2.0">
    <annotation>
        <appinfo>
           <jaxb:schemaBindings>
            <jaxb:package name="example.webservices.simple.simpleservice"/>
           </jaxb:schemaBindings>
        </appinfo>
    </annotation>
</schema>
```

JAX-WS Custom Binding Declarations

The following table summarizes the typical JAX-WS customizations. For a complete list of JAX-WS custom binding declarations, see *WSDL Customization* at https://javaee.github.io/metro-jax-ws/doc/user-guide/release-documentation.html#users-guide-wsdl-customization.

Table 5-13 JAX-WS Custom Binding Declarations

Customization	Description

Package name	Use the `jaxws:package` binding declaration to define the package name.
	If you do not specify this customization, the `wsdlc` Ant task generates a package name based on the `targetNamespace` of the WSDL. This data binding customization is overridden by the `packageName` attribute of the `wsdlc`, `jwsc`, or `clientgen` Ant task. For more information, see wsdlc in the *WebLogic Web Services Reference for Oracle WebLogic Server*.
	This binding declaration can be specified as part of the root binding element, as described in Creating an External Binding Declarations File, or on the `wsdl:definitions` node, as shown in the following example:
	`<bindings`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"`
	`  wsdlLocation=`
	`  "http://localhost:7001/simple/SimpleService?WSDL"`
	`  xmlns="http://java.sun.com/xml/ns/jaxws">`
	`  <bindings node="wsdl:definitions"`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">`
	`  **<package name="example.webservices.simple.simpleService"/>**`
	`</bindings>`

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

Customization	Description

Wrapper-style rules	Use the `jaxws:enablesWrapperStyle` binding declaration to enable or disable the wrapper style rules that control how the parameter types and return types of a WSDL operation are generated.
	This binding declaration can be specified as part of the root binding element, as described in Creating an External Binding Declarations File, or on one of the following nodes:
	• `wsdl:definitions`—Applies to all `wsdl:operations` of all `wsdl:portType` attributes.
	• `wsdl:portType`—Applies to all `wsdl:operations` in the `wsdl:portType`.
	• `wsdl:operation`—Applies to the `wsdl:operation` only.
	The following example disables the wrapper style rules for the `wsdl:definitions` node:
	`<bindings`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"`
	`wsdlLocation="http://localhost:7001/simple/`
	`SimpleService?WSDL"`
	`  xmlns="http://java.sun.com/xml/ns/jaxws">`
	`  <bindings node="wsdl:definitions"`
	`  xmlns:wsdl="http://schemas.xmlsoap.org/`
	`wsdl/">`
	`  **<enableWrapperStyle>**`
	`  **false**`
	`  **</enableWrapperStyle>**`
	`</bindings>`

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

Customization	Description

Asynchrony	Use the `jaxws:enableAsycMapping` binding declaration to instruct the `clientgen` Ant task to generate asynchronous polling and callback operations along with the normal synchronous methods when it compiles a WSDL file.
	This binding declaration can be specified as part of the root binding element, as described in Creating an External Binding Declarations File, or on one of the following nodes:
	• `wsdl:definitions`—Applies to all `wsdl:operations` of all `wsdl:portType` attributes.
	• `wsdl:portType`—Applies to all `wsdl:operations` in the `wsdl:portType`.
	• `wsdl:operation`—Applies to the `wsdl:operation` only.
	The following example disables asynchronous polling and callback operations:
	`<bindings xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" wsdlLocation="http://localhost:7001/simple/SimpleService?WSDL" xmlns="http://java.sun.com/xml/ns/jaxws"> <bindings node="wsdl:definitions" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">`
	`<enableAsyncMapping> false </enableAsyncMapping>`
	`</bindings>`

Provider	Use the `jaxws:provider` binding declaration to mark the part as a provider interface. This binding declaration can be specified as part of the `wsdl:portType`. This binding declaration applies when you are developing a service starting from a WSDL file.

Class name	Use the `jaxws:class` binding declaration to define the class name. This binding declaration can be specified for one of the following nodes:
	• `wsdl:portType`—Defines the interface class name.
	• `wsdl:fault`—Defines fault class names.
	• `soap:headerfault`—Defines exception class names.
	• `wsdl:service`—Defines the implementation class names.
	The following example defines the class name for the implementation class.
	`<bindings node="wsdl:definitions/wsdl:service[@name='SimpleService']">`
	`<class name="myService"></class>`
	`</bindings>`

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

Customization	Description
Method name	Use the `jaxws:method` binding declaration to customize the generated Java method name of a service endpoint interface or the port accessor method in the generated `Service` class. The following example defines the Java method name for the `wsdl:operation EchoHello`. `<bindings node="wsdl:definitions/ wsdl:portType[@name='SimpleServiceImpl']/ wsdl:operation[@name='EchoHello']"> <method name="Greeting"></method> </bindings>`
Java parameter name	Use the `jaxws:parameter` binding declaration to customize the parameter name of generated Java methods. This declaration can be used to change the method parameter of a `wsdl:operation` in a `wsdl:portType`. The following example defines the Java method name for the `wsdl:operation echoHello`. `<bindings node="wsdl:definitions/ wsdl:portType[@name='SimpleServiceImpl']/ wsdl:operation[@name='EchoHello']"> <parameter part="definitions/ message[@name='EchoHello']/ part[@name='parameters']" element="hello" name="greeting"/> </bindings>`
Javadoc	Use the `jaxws:javadoc` binding declaration to specify Javadoc text for a package, class, or method. For example, the following defines Javadoc at the method level. `<bindings node="wsdl:definitions/ wsdl:portType[@name='SimpleServiceImpl']/ wsdl:operation[@name='EchoHello']"> <method name="Hello"> <javadoc>Prints hello.</javadoc> </method> </bindings>`
Handler chain	Use the `javaee:handlerchain` binding declaration to customize or add handlers. The inline handler must conform to the handler chain configuration defined in the *Web Services Metadata for the Java Platform* specification (JSR-181) at `http://www.jcp.org/en/jsr/detail?id=181`.

JAXB Custom Binding Declarations

The following table lists the typical JAXB customizations.

> **Note:**
>
> The following table only summarizes the JAXB custom binding declarations, to help get you started. For a complete list and description of all JAXB custom binding declarations, see the JAXB specification (http://jcp.org/en/jsr/detail?id=222).

Table 5-14 JAXB Custom Binding Declarations

Customization	Description

Global bindings	Use the `<globalBindings>` binding declaration to define binding declarations with global scope (see Figure 5-2).
	You can specify attributes and elements to the `<globalBindings>` binding declaration. For example, the following binding declaration defines:
	• `collectionType` attribute that specifies a type class, `myArray`, that implements the `java.util.List` interface and that is used to represent all lists in the generated implementation.
	• `generateIsSetMethod` attribute to generate the `isSet()` method corresponding to the getter and setter property methods.
	• `javaType` element to customize the binding of an XML Schema atomic datatype to a Java datatype (built-in or application-specific).
	`<jaxb:globalBindings`
	`collectionType ="java.util.myArray"`
	`generateIsSetMethod="false">`
	`<jaxb:javaType name="java.util.Date"`
	`xmlType="xsd:date"`
	`</jaxb:javaType>`
	`</jaxb:globalBindings>`

Schema bindings	Use the `<schemaBindings>` binding declaration to define binding declarations with schema scope (see Figure 5-2). For an example, see the description of "Package name" in this table.

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

Customization	Description

Package name	Use the `<package>` element of the `<schemaBindings>` binding declaration (see Table 5-12) to define the package name for the schema.
	If you do not specify this customization, the `wsdlc` Ant task generates a package name based on the `targetNamespace` of the WSDL. This data binding customization is overridden by the `packageName` attribute of the `wsdlc`, `jwsc`, or `clientgen` Ant task. For more information, see wsdlc in the *WebLogic Web Services Reference for Oracle WebLogic Server*.
	For example, the following defines the package name for all JAXB classes generated from the `simpleservice.xsd` file:
	`<jaxb:bindings`
	`xmlns:xs="http://www.w3.org/2001/XMLSchema"`
	`schemaLocation="simpleservice.xsd"`
	`node="/xs:schema">`
	`<jaxb:schemaBindings>`
	`<jaxb:package name="examples.jaxb"/>`
	`</jaxb:schemaBindings>`
	`</jaxb:bindings>`
	The following shows how to define the package name for an imported XML Schema:
	`<jaxb:bindindgs`
	`xmlns:xs="http://www.w3.org/2001/XMLSchema"`
	`node="//xs:schema/xs:import[@namespace='http://examples.webservices.org/complexservice']">`
	`<jaxb:schemaBindings>`
	`<jaxb:package name="examples.jaxb"/>`
	`</jaxb:schemaBindings>`
	`</jaxb:bindings>`

Class name	Use the `<class>` binding declaration to define the class name for a schema element.
	The following example defines the class name for an `xsd:complexType`:
	`<xs:complexType name="ComplexType">`
	`<xs:annotation><xs:appinfo>`
	`<jaxb:javadoc>This is my class.</jaxb:javadoc>`
	`</jaxb:class>`
	`</xs:appinfo></xs:annotation>`
	`</xs:complexType>`

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

Customization	Description
Java property name	Use the <property> binding declaration to define the property name for a schema element. The following example shows how to define the Java property name: `<jaxb:bindindgs xmlns:xs="http://www.w3.org/2001/XMLSchema" node="//xs:schema/"> <jaxb:schemaBindings> <jaxb:property generateIsSetMethod="true"/> </jaxb:schemaBindings> </jaxb:bindings>`
Java datatype	Use the <javaType> binding declaration to customize the binding of an XML Schema atomic datatype to a Java datatype (built-in or application-specific). For example, see Global bindings (above).
Javadoc	Use the <javadoc> child element of the <class> or <property> binding declaration to specify Javadoc for the element. For example: `<xs:complexType name="ComplexType"> <xs:annotation><xs:appinfo> <jaxb:class name="MyClass"> <jaxb:javadoc>This is my class.</ jaxb:javadoc> </jaxb:class> </xs:appinfo></xs:annotation> </xs:complexType>`

Using the Glassfish RI JAXB Data Binding and JAXB Providers

The Glassfish RI JAXB data binding and JAXB providers provide the standard Glassfish JAXB implementation, and were the default JAXB providers in previous WebLogic Server releases. If desired, you can restore the Glassfish RI providers, either globally on the server, or on a per application basis.

Note that the JAXB data binding provider and the JAXB provider are two distinct entities, although both use EclipseLink MOXy as the default. The JAXB data binding provider is used by the web services tooling and runtime, and performs tasks such as WSDL generation from a Java endpoint, as in the JWS task, and the runtime marshalling and unmarshalling of the contents of the SOAP message. The JAXB provider, on the other hand, specifies which JAXBContext provider to use for all other JAXB-related tasks. Although the JAXB provider configuration does apply to some of the web services tooling, such as Java class generation from WSDL/schema files, it includes all other JAXB usage as well. These two providers can be configured

independently. For example, you could retain EclipseLink MOXy for data binding, but revert to the Glassfish RI JAXB provider for other JAXB tasks.

The data binding and JAXB providers are configured using the following Java Service Provider Interface (SPI) files in *ORACLE_HOME*/oracle_common/modules/com.oracle.webservices.wls.wls-ws-metainf-services-impl.jar:

- META-INF/services/com.sun.xml.ws.spi.db.BindingContextFactory
- META-INF/services/javax.xml.bind.JAXBContext

> **Note:**
>
> In 12.1.2.0, the providers were located in *ORACLE_HOME*/oracle_common/modules/com.oracle.webservices.wls.wls-ws-metainf-services_2.0.0.0.jar. In 12.1.1.0, the providers were located in WL_HOME/server/lib/weblogic.jar.

Global and application-level configuration is described in the following sections.

Configuring Global Server-Level Data Binding and JAXB Providers

The following jar file is provided in the WebLogic Server distribution to simplify the task of overriding the default data binding configuration:

modules/databinding.override.jar

This jar file is not included in the classpath by default. To restore the Glassfish RI data binding and JAXB provider settings, edit the WebLogic Server start script to prepend this jar file to the classpath.

For the tooling and client, you can apply this jar file globally to Ant scripts or to another build environment.

Note that the modules/databinding.override.jar file overrides both the data binding provider and the JAXB provider. If you desire to override one of these providers, but not both, you can do so by creating a simple jar file containing only the service provider entry that you want to override, and putting this first in the classpath.

For example, to configure *only* the Glassfish RI JAXB provider:

1. Create a file named META-INF/services/javax.xml.bind.JAXBContext that contains a single entry for the Glassfish RI JAXB provider:

   com.sun.xml.bind.v2.ContextFactory

2. Create a jar file, for example jaxb_override.jar, and add the file created in Step 1.
3. Prepend this jar file to the classpath to use the Glassfish JAXB provider.

The same procedure applies if you want to configure only the Glassfish RI data binding provider. In this case, however, name the file you create in Step 1 META-INF/services/com.sun.xml.ws.spi.db.BindingContextFactory containing a single entry for the Glassfish RI data binding provider:

com.sun.xml.ws.db.glassfish.JAXBRIContextFactory.

> **Note:**
>
> Configuring the data binding provider may affect other behavior in addition to runtime data binding. For example, WebLogic Server generates its WSDL at runtime using the data binding provider. Conversely, some runtime SOAP faults are produced by invoking the JAXB provider directly.

As an alternative to placing the override jar file in the classpath, you can edit the Java system properties directly. For more information, see Configuring Java System Properties for JAXB.

Configuring Application-Level Data Binding and JAXB Providers

To configure the data binding and JAXB providers for a single Web application, you can use the filtering loading mechanism provided by WebLogic Server. This mechanism allows the system classpath search to be bypassed when looking for specific application classes and resources that are on the application classpath. Specifically, you use the `<prefer-application-resources>` tag in the `weblogic-application.xml` file for the application EAR or build-out directory.

For example, to configure the Glassfish RI data binding provider for an application:

1. Edit the `weblogic-application.xml` file to include an entry for the data binding resource, as shown in the following example:

   ```
   <prefer-application-resources> <resource-name>META-INF/services/
   com.sun.xml.ws.spi.db.BindingContextFactory</resource-name>
   </prefer-application-resources>
   ```

2. Create a file named `META-INF/services/com.sun.xml.ws.spi.db.BindingContextFactory` containing an entry for the desired provider, in this case, `com.sun.xml.ws.db.glassfish.JAXBRIContextFactory`.

3. Add the file created in step 2 to the build-out directory, or add it as an entry in the EAR file.

Use the same procedure to configure the Glassfish RI JAXB provider using the values appropriate for the JAXB provider. Specifically, add the resource name `META-INF/services/javax.xml.bind.JAXBContext` to the `weblogic-application.xml` file and set the provider name in the file to `com.sun.xml.bind.v2.ContextFactory`.

For more information about the filtering loading mechanism in WebLogic Server, see Filtering Loader Mechanism in *Tuning Performance of Oracle WebLogic Server*.

Configuring Java System Properties for JAXB

You can configure the Java system properties to revert to the Glassfish RI providers and to configure the default EclipseLink MOXy providers if you had previously reverted.

> **Note:**
>
> In certain situations, it can be difficult to propagate the system properties to an indirectly invoked Java instance, such as a client forked from an Ant task. In these situations, it is important to ensure that the environment you are using propagates the properties.

To configure the Glassfish RI data binding and JAXB providers, set the Java system properties as shown in Table 5-15.

Table 5-15 Java System Property Settings for Glassfish RI Providers

Set this Java system property . . .	To this value . . .
com.sun.xml.ws.spi.db.BindingContextFactory	com.sun.xml.ws.db.glassfish.JAXBRIContextFactory
javax.xml.bind.JAXBContext	com.sun.xml.bind.v2.ContextFactory

To configure the default EclipseLink MOXy providers, set the Java system properties as shown in Table 5-16.

Table 5-16 Java System Property Settings for EclipseLink MOXy Providers

Set this Java system property . . .	To this value . . .
com.sun.xml.ws.spi.db.BindingContextFactory	com.sun.xml.ws.db.toplink.JAXBContextFactory
javax.xml.bind.JAXBContext	org.eclipse.persistence.jaxb.JAXBContextFactory

6

Examples of Developing JAX-WS Web Services

This chapter provides some common examples of developing WebLogic web services using Java API for XML-based Web services (JAX-WS).
This chapter includes the following sections:

- Creating a Simple HelloWorld Web Service
- Creating a Web Service With User-Defined Data Types
- Creating a Web Service from a WSDL File

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

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

> **Note:**
>
> For best practice examples demonstrating advanced web service features, see Roadmap for Developing JAX-WS Web Service Clients and Roadmap for Developing Reliable Web Services and Clients.

Creating a Simple HelloWorld Web Service

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

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

1. Set your WebLogic Server environment.

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

`domains/domainName`, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.

2. Create a project directory, as follows:

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

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

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

4. Create the JWS file that implements the web service.

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

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

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

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

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

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

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

```
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="output/helloWorldEar">
       <jws file="examples/webservices/hello_world/HelloWorldImpl.java"
         type="JAXWS"/>
  </jwsc>
</target>
```

   The `jwsc` WebLogic web service Ant task generates the supporting artifacts, compiles the user-created and generated Java code, and archives all the artifacts into an Enterprise Application EAR file that you later deploy to WebLogic Server. You specify the type of web service (JAX-WS) that you want to create using `type="JAXWS"`.

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

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

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

9. Start the WebLogic Server instance to which the web service will be deployed.

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

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

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

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

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

    ```
    prompt> ant deploy
    ```

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

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

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

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

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

To run the web service, you need to create a client that invokes it. See Invoking a Web Service from a WebLogic Web Service for an example of creating a Java client application that invokes a web service.

Sample HelloWorldImpl.java JWS File

```
package examples.webservices.hello_world;
// Import the @WebService annotation
import javax.jws.WebService;
@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
/**
 * This JWS file forms the basis of simple Java-class implemented WebLogic
 * Web Service with a single operation: sayHelloWorld
 */
public class HelloWorldImpl {
  // By default, all public methods are exposed as Web Services operation
  public String sayHelloWorld(String message) {
  try {
    System.out.println("sayHelloWorld:" + message);
  } catch (Exception ex) { ex.printStackTrace(); }

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

Sample Ant Build File for HelloWorldImpl.java

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

```
<project name="webservices-hello_world" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="helloWorldEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/helloWorldEar" />
  <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all" depends="clean,build-service,deploy,client" />
  <target name="clean" depends="undeploy">
    <delete dir="${example-output}"/>
  </target>
  <target name="build-service">
    <jwsc
      srcdir="src"
      destdir="${ear-dir}">
      <jws file="examples/webservices/hello_world/HelloWorldImpl.java"
          type="JAXWS"/>
    </jwsc>
  </target>
  <target name="deploy">
```

```
    <wldeploy action="deploy" name="${ear.deployed.name}"
      source="${ear-dir}" user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>
  <target name="undeploy">
    <wldeploy action="undeploy" name="${ear.deployed.name}"
      failonerror="false"
      user="${wls.username}" password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>
  <target name="client">
    <clientgen
      wsdl="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.hello_world.client"
      type="JAXWS"/>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/hello_world/client/**/*.java"/>
  </target>
  <target name="run">
    <java classname="examples.webservices.hello_world.client.Main"
          fork="true" failonerror="true" >
      <classpath refid="client.class.path"/>
      <arg
 line="http://${wls.hostname}:${wls.port}/HelloWorldImpl/HelloWorldService"  />
    </java> </target>
</project>
```

Creating a Web Service With User-Defined Data Types

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

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

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

1. Set your WebLogic Server environment.

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

`domains/domainName`, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.

2. Create a project directory:

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

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

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

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

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

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

6. Create the JWS file that implements the web service using the Java code specified in Sample ComplexImpl.java JWS File.

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

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

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

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

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

   See Sample Ant Build File for ComplexImpl.java JWS File for a full sample `build.xml` file.

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

```
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="output/ComplexServiceEar" >
    <jws file="examples/webservices/complex/ComplexImpl.java"
       type="JAXWS">
```

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

In the preceding example:

- The `type` attribute of the `<jws>` element specifies the type of web service (JAX-WS or JAX-RPC).
- The `<WLHttpTransport>` child element of the `<jws>` element of the `jwsc` Ant task specifies the context path and service URI sections of the URL used to invoke the web service over the HTTP/S transport, as well as the name of the port in the generated WSDL. For more information about defining the context path, see Defining the Context Path of a WebLogic Web Service.

10. Execute the `jwsc` Ant task:

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

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

11. Start the WebLogic Server instance to which the web service will be deployed.
12. Deploy the web service, packaged in the `ComplexServiceEar` Enterprise Application, to WebLogic Server, using either the WebLogic Server Administration Console or the `wldeploy` Ant task. For example:

```
prompt> ant deploy
```

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

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

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

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

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

```
prompt> ant deploy
```

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

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

To run the web service, you need to create a client that invokes it. See Invoking a Web Service from a WebLogic Web Service for an example of creating a Java client application that invokes a web service.

Sample BasicStruct JavaBean

```
package examples.webservices.complex;
/**
 * Defines a simple JavaBean called BasicStruct that has integer, String,
 * and String[] properties
 */
public class BasicStruct {
  // Properties
  private int intValue;
  private String stringValue;
  private String[] stringArray;
  // Getter and setter methods
  public int getIntValue() {
    return intValue;
  }
  public void setIntValue(int intValue) {
    this.intValue = intValue;
  }
  public String getStringValue() {
    return stringValue;
  }
  public void setStringValue(String stringValue) {
    this.stringValue = stringValue;
  }
  public String[] getStringArray() {
    return stringArray;
  }
  public void setStringArray(String[] stringArray) {
    this.stringArray = stringArray;
  }
  public String toString() {
    return "IntValue="+intValue+", StringValue="+stringValue;
  }
}
```

Sample ComplexImpl.java JWS File

```
package examples.webservices.complex;
// Import the standard JWS annotation interfaces
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
// Import the BasicStruct JavaBean
import examples.webservices.complex.BasicStruct;
// Standard JWS annotation that specifies that the portType name of the Web
// Service is "ComplexPortType", its public service name is "ComplexService",
// and the targetNamespace used in the generated WSDL is "http://example.org"
@WebService(serviceName="ComplexService", name="ComplexPortType",
            targetNamespace="http://example.org")
// Standard JWS annotation that specifies this is a document-literal-wrapped
// Web Service
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
```

```
            use=SOAPBinding.Use.LITERAL,
            parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
/**
 * This JWS file forms the basis of a WebLogic Web Service.  The Web Services
 * has two public operations:
 *
 *  - echoInt(int)
 *  - echoComplexType(BasicStruct)
 *
 * The Web Service is defined as a "document-literal" service, which means
 * that the SOAP messages have a single part referencing an XML Schema element
 * that defines the entire body.
 */
public class ComplexImpl {
  // Standard JWS annotation that specifies that the method should be exposed
  // as a public operation.  Because the annotation does not include the
  // member-value "operationName", the public name of the operation is the
  // same as the method name: echoInt.
  //
  // The WebResult annotation specifies that the name of the result of the
  // operation in the generated WSDL is "IntegerOutput", rather than the
  // default name "return".   The WebParam annotation specifies that the input
  // parameter name in the WSDL file is "IntegerInput" rather than the Java
  // name of the parameter, "input".
  @WebMethod()
  @WebResult(name="IntegerOutput",
            targetNamespace="http://example.org/complex")
  public int echoInt(
      @WebParam(name="IntegerInput",
                targetNamespace="http://example.org/complex")
      int input)
  {
    System.out.println("echoInt '" + input + "' to you too!");
    return input;
  }
  // Standard JWS annotation to expose method "echoStruct" as a public operation
  // called "echoComplexType"
  // The WebResult annotation specifies that the name of the result of the
  // operation in the generated WSDL is "EchoStructReturnMessage",
  // rather than the default name "return".
  @WebMethod(operationName="echoComplexType")
  @WebResult(name="EchoStructReturnMessage",
            targetNamespace="http://example.org/complex")
  public BasicStruct echoStruct(BasicStruct struct)
  {
    System.out.println("echoComplexType called");
    return struct;
  }
}
```

Sample Ant Build File for ComplexImpl.java JWS File

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

```
<project name="webservices-complex" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
```

```
<property name="wls.server.name" value="myserver" />
<property name="ear.deployed.name" value="complexServiceEAR" />
<property name="example-output" value="output" />
<property name="ear-dir" value="${example-output}/complexServiceEar" />
<property name="clientclass-dir" value="${example-output}/clientclass" />
<path id="client.class.path">
  <pathelement path="${clientclass-dir}"/>
  <pathelement path="${java.class.path}"/>
</path>
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JwscTask" />
<taskdef name="clientgen"
  classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy"/>
<target name="all" depends="clean,build-service,deploy,client"/>
<target name="clean" depends="undeploy">
  <delete dir="${example-output}"/>
</target>
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="${ear-dir}"
    keepGenerated="true"
    >
    <jws file="examples/webservices/complex/ComplexImpl.java"
       type="JAXWS">
       <WLHttpTransport
        contextPath="complex" serviceUri="ComplexService"
        portName="ComplexServicePort"/>
    </jws>
  </jwsc>
</target>
<target name="deploy">
  <wldeploy action="deploy"
    name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}"/>
</target>
<target name="undeploy">
  <wldeploy action="undeploy" failonerror="false"
    name="${ear.deployed.name}"
    user="${wls.username}" password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}"/>
</target>
<target name="client">
  <clientgen
    wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
    destDir="${clientclass-dir}"
    packageName="examples.webservices.complex.client"
       type="JAXWS"/>
  <javac
    srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
    includes="**/*.java"/>
  <javac
    srcdir="src" destdir="${clientclass-dir}"
    includes="examples/webservices/complex/client/**/*.java"/>
</target>
```

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

Creating a Web Service from a WSDL File

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

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

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

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

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

The following simple example shows how to create a web service from the WSDL file shown in Sample WSDL File. The web service has one operation, `getTemp`, that returns a temperature when passed a zip code.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory.

The default location of WebLogic Server domains is *ORACLE_HOME*/user_projects/domains/*domainName*, where *ORACLE_HOME* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *domainName* is the name of your domain.

2. Create a working directory:

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

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

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

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

```
<project name="webservices-wsdlc" default="all">
  <taskdef name="wsdlc"
           classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
</project>
```

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

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

```
<target name="generate-from-wsdl">
  <wsdlc
      srcWsdl="wsdl_files/TemperatureService.wsdl"
      destJwsDir="output/compiledWsdl"
      destImplDir="output/impl"
      packageName="examples.webservices.wsdlc"
      type="JAXWS"/>
</target>
```

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

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

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

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

7. Update the generated `output/impl/examples/webservices/wsdlc/TemperatureService_TemperaturePortImpl.java` JWS implementation file using

your favorite Java IDE or text editor to add Java code to the methods so that they behave as you want.

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

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

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

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

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

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

```
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <target name="build-service">
    <jwsc
      srcdir="src"
      destdir="${ear-dir}">
      <jws file="examples/webservices/wsdlc/
TemperatureService_TemperaturePortImpl.java"
           compiledWsdl="${compiledWsdl-dir}/TemperatureService_wsdl.jar"
           type="JAXWS">
         <WLHttpTransport
          contextPath="temp" serviceUri="TemperatureService"
          portName="TemperaturePort">
      </WLHttpTransport>
      </jws>
    </jwsc>
  </target>
```

In the preceding example:

- The `type` attribute of the `<jws>` element specifies the type of web services (JAX-WS or JAX-RPC).

- The `<WLHttpTransport>` child element of the `<jws>` element of the `jwsc` Ant task specifies the context path and service URI sections of the URL used to invoke the web service over the HTTP/S transport, as well as the name of the port in the generated WSDL.

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

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

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

11. Start the WebLogic Server instance to which the web service will be deployed.

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

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

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

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

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

    ```
    prompt> ant deploy
    ```

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

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

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

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

To run the web service, you need to create a client that invokes it. See Invoking a Web Service from a WebLogic Web Service for an example of creating a Java client application that invokes a web service.

Sample WSDL File

```
<?xml version="1.0"?>
<definitions
   name="TemperatureService"
   targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
```

```
   xmlns:tns="http://www.xmethods.net/sd/TemperatureService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/" >
   <types>
      <xsd:schema
         targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
          <xsd:element name="getTempRequest">
             <xsd:complexType>
                <xsd:sequence>
                   <xsd:element name="zip" type="xsd:string"/>
                </xsd:sequence>
             </xsd:complexType>
          </xsd:element>
          <xsd:element name="getTempResponse">
             <xsd:complexType>
                <xsd:sequence>
                   <xsd:element name="return" type="xsd:float"/>
                </xsd:sequence>
             </xsd:complexType>
          </xsd:element>
      </xsd:schema>
   </types>
   <message name="getTempRequest">
      <part name="parameters" element="tns:getTempRequest"/>
   </message>
   <message name="getTempResponse">
      <part name="parameters" element="tns:getTempResponse"/>
   </message>
   <portType name="TemperaturePortType">
      <operation name="getTemp">
         <input message="tns:getTempRequest"/>
         <output message="tns:getTempResponse"/>
      </operation>
   </portType>
   <binding name="TemperatureBinding" type="tns:TemperaturePortType">
      <soap:binding style="document"
           transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="getTemp">
          <soap:operation soapAction=""/>
          <input>
             <soap:body use="literal"/>
          </input>
          <output>
             <soap:body use="literal"/>
          </output>
      </operation>
   </binding>
   <service name="TemperatureService">
      <documentation>
         Returns current temperature in a given U.S. zipcode
      </documentation>
      <port name="TemperaturePort" binding="tns:TemperatureBinding">
         <soap:address
          location="http://localhost:7001/temp/TemperatureService"/>
      </port>
   </service>
</definitions>
```

Sample TemperatureService_TemperaturePortImpl Java Implementation File

```
package examples.webservices.wsdlc;
import javax.jws.WebService;
import javax.xml.ws.BindingType;

/**
 * Returns current temperature in a given U.S. zipcode
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.2.8-b13684
 * Generated source version: 2.2
 *
 */
@WebService(
   portName = "TemperaturePort",
   serviceName = "TemperatureService",
   targetNamespace="http://www.xmethods.net/sd/TemperatureService.wsdl"
   wsdlLocation = "/wsdls/TemperatureService.wsdl",
   endpointInterface = "examples.webservices.wsdlc.TemperaturePortType")
@BindingType("http://schemas.xmlsoap.org/wsdl/soap/http")
public class TemperatureService_TemperaturePortImpl implements
  TemperaturePortType
{
    public TemperatureService_TemperaturePortImpl() { }
    /**
     *
     * @param zip
     * @return
     *     returns float
     */
    public float getTemp(String zip) {
     return 1.234f;
  }
}
```

Sample Ant Build File for TemperatureService

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

```
<project default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="wsdlcEar" />
  <property name="example-output" value="output" />
  <property name="compiledWsdl-dir" value="${example-output}/compiledWsdl" />
  <property name="impl-dir" value="${example-output}/impl" />
  <property name="ear-dir" value="${example-output}/wsdlcEar" />
  <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>
  <taskdef name="wsdlc"
```

```
         classname="weblogic.wsee.tools.anttasks.WsdlcTask"/>
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JwscTask" />
<taskdef name="clientgen"
  classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy"/>
<target name="all"
  depends="clean,generate-from-wsdl,build-service,deploy,client" />
<target name="clean" depends="undeploy">
  <delete dir="${example-output}"/>
</target>
<target name="generate-from-wsdl">
  <wsdlc
      srcWsdl="wsdl_files/TemperatureService.wsdl"
      destJwsDir="${compiledWsdl-dir}"
      destImplDir="${impl-dir}"
      packageName="examples.webservices.wsdlc"
      type="JAXWS"/>
</target>
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="${ear-dir}">
    <jws
 file="examples/webservices/wsdlc/TemperatureService_TemperaturePortImpl.java"
 compiledWsdl="${compiledWsdl-dir}/TemperatureService_wsdl.jar"
 type="JAXWS">
       <WLHttpTransport
        contextPath="temp" serviceUri="TemperatureService"
        portName="TemperaturePort"/>
    </jws>
  </jwsc>
</target>
<target name="deploy">
  <wldeploy action="deploy" name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>
<target name="undeploy">
  <wldeploy action="undeploy" name="${ear.deployed.name}"
    failonerror="false"
    user="${wls.username}" password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>
<target name="client">
  <clientgen
    wsdl="http://${wls.hostname}:${wls.port}/temp/TemperatureService?WSDL"
    destDir="${clientclass-dir}"
    packageName="examples.webservices.wsdlc.client"
    type="JAXWS">
  <javac
    srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
    includes="**/*.java"/>
  <javac
    srcdir="src" destdir="${clientclass-dir}"
    includes="examples/webservices/wsdlc/client/**/*.java"/>
</target>
```

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

Part III

Developing Basic JAX-WS Web Service Clients

Part III describes how to develop basic WebLogic web service clients using Java API for XML-based web services (JAX-WS).

Sections include:

7

Roadmap for Developing JAX-WS Web Service Clients

This chapter presents best practices for developing WebLogic web service clients for Java API for XML Web Services (JAX-WS).
Table 7-1 lists each best practice and is followed by an example that illustrates the best practices presented. The best practices are described in more detail later in this document.

For additional best practices, refer to the following sections:

- For best practices when developing asynchronous web service clients, see Roadmap for Developing Asynchronous Web Service Clients.
- For best practices when developing reliable web service clients, see Roadmap for Developing Reliable Web Services and Clients.

> **Note:**
>
> In the following table, *client instance* can be a port or a Dispatch instance.

Table 7-1 Roadmap for Developing Web Service Clients

Best Practice	Description
Synchronize use of client instances.	Create client instances as you need them; do not store them long term.
Use a stored list of features, including client ID, to create client instances.	Define all features for the web service client instance, including client ID, so that they are consistent each time the client instance is created. For example: `_service.getBackendServicePort(_features);`
Explicitly define the client ID.	Use the `ClientIdentityFeature` to define the client ID explicitly. This client ID is used to group statistics and other monitoring information, and for reporting runtime validations, and so on. For more information, see Managing Client Identity. **Note:** Oracle strongly recommends that you define the client ID explicitly. If not explicitly defined, the server generates the client ID automatically, which may not be user-friendly.
Explicitly close client instances when processing is complete.	For example: `((java.io.Closeable)port).close();` If not closed explicitly, the client instance will be closed automatically when it goes out of scope. **Note**: The client ID remains registered and visible until the container (Web application or EJB) is deactivated. For more information, see Client Identity Lifecycle.

The following example illustrates best practices for developing web service clients.

Example 7-1 Web Service Client Best Practices Example

```
import java.io.IOException;
import java.util.*;

import javax.servlet.*;
import javax.xml.ws.*;

import weblogic.jws.jaxws.client.ClientIdentityFeature;

/**
 * Example client for invoking a web service.
 */
public class BestPracticeClient
  extends GenericServlet {

  private BackendServiceService _service;
  private WebServiceFeature[] _features;
  private ClientIdentityFeature _clientIdFeature;

  @Override
  public void init()
    throws ServletException {

    // Create a single instance of a web service as it is expensive to create repeatedly.
    if (_service == null) {
      _service = new BackendServiceService();
    }

    // Best Practice: Use a stored list of features, per client ID, to create client instances.
    // Define all features for the web service client instance, per client ID, so that they are
    // consistent each time the client instance is created. For example:
    // _service.getBackendServicePort(_features);

    List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

    // Best Practice: Explicitly define the client ID.
    // TODO: Maybe allow ClientIdentityFeature to store other features, and
    //       then create new client instances simply by passing the
    //       ClientIdentityFeature (and the registered features are used).
    _clientIdFeature = new ClientIdentityFeature("MyBackendServiceClient");
    features.add(_clientIdFeature);

    // Set the features used when creating clients with
    // the client ID "MyBackendServiceClient". The features are stored in an array to
    // reinforce that the list should be treated as immutable.
    _features = features.toArray(new WebServiceFeature[features.size()]);
  }

  @Override
  public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException {

    // ... Read the servlet request ...

    // Best Practice: Synchronize use of client instances.
    // Create a web service client instance to talk to the backend service.
    // Note, at this point the client ID is 'registered' and becomes
    // visible to monitoring tools such as the Administration Console and WLST.
    // The client ID *remains* registered and visible until the container
    // (the Web application hosting our servlet) is deactivated (undeployed).
```

```
  //
  // A client ID can be used when creating multiple client instances (port or Dispatch client).
  // The client instance should be created with the same set of features each time, and should
  // use the same service class and refer to the same port type.
  // A given a client ID should be used for a given port type, but not across port types.
  // It can be used for both port and Dispatch clients.
  BackendService port =
    _service.getBackendServicePort(_features);

  // Set the endpoint address for BackendService.
  ((BindingProvider)port).getRequestContext().
    put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        "http://localhost:7001/BestPracticeService/BackendService");

  // Print out the explicit client ID, and compare it to the client ID
  // that would have been generated automatically for the client instance.
  showClientIdentity();

  // Make the invocation on our real port
  String request = "Make a cake";
  System.out.println("Invoking DoSomething with request: " + request);
  String response = port.doSomething(request);
  System.out.println("Got response: " + response);
  res.getWriter().write(response);

  // Best Practice: Explicitly close client instances when processing is complete.
  // If not closed, the client instance will be closed automatically when it goes out of
  // scope. Note, this client ID will remain registered and visible until our
  // container (Web application) is undeployed.
  ((java.io.Closeable)port).close();
}

/**
  // Print out the client's full ID, which is a combination of
  // the client ID provided above and qualifiers from the application and
  // Web application that contain the client. Then compare this with the client ID that
  // would have been generated for the client instance if not explicitly set.
  //
private void showClientIdentity()
  throws IOException {

  System.out.println("Client Identity is: " + _clientIdFeature.getClientId());

  // Create a client instance without explicitly defining the client ID to view the
  // client ID that is generated automatically.
  ClientIdentityFeature dummyClientIdFeature =
    new ClientIdentityFeature(null);
  BackendService dummyPort =
    _service.getBackendServicePort(dummyClientIdFeature);
  System.out.println("Generated Client Identity is: " +
                     dummyClientIdFeature.getClientId());
  // Best Practice: Explicitly close client instances when processing is complete.
  // If not closed, the client instance will be closed automatically when it goes out of
  // scope. Note, this client ID will remain registered and visible until our
  // container (Web application) is undeployed.
  ((java.io.Closeable)dummyPort).close();
}

@Override
public void destroy() {
```

```
    }
}
```

8

Developing Web Service Clients

This chapter describes how to develop Java EE clients to invoke a WebLogic web service using Java API for XML-based Web Services (JAX-WS).
This chapter includes the following sections:

Overview of WebLogic Web Services Client Development

Invoking a web service refers to the actions that a client application performs to use the web service.

There are two types of client applications:

- Java SE client—In its simplest form, a Java SE client is a Java program that has the `Main` public class that you invoke with the `java` command. A Java SE client can be invoked within a WebLogic Server environment (with access to the WebLogic Server classpath) or as a *standalone* client application.
- Java EE component deployed to WebLogic Server—In this type of client application, the web service runs inside a Java Platform, Enterprise Edition (Java EE) component deployed to WebLogic Server, such as an EJB, servlet, or another web service. This type of client application, therefore, runs inside a WebLogic Server container.

The sections that follow describe how to use Oracle's implementation of the JAX-WS specification to invoke a web service from a Java client application. You can use this implementation to invoke web services running on any application server, both WebLogic and non-WebLogic.

WebLogic Server optionally includes examples of creating and invoking WebLogic web services in the *ORACLE_HOME*`/wlserver/samples/server/examples/src/examples` directory, where *ORACLE_HOME* represents the directory in which you installed WebLogic Server. For detailed instructions on how to build and run the examples, open the *ORACLE_HOME*`/wlserver/samples/server/docs/index.html` Web page in your browser and expand the **WebLogic**

Server Examples->Examples->API->Web Services node. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

For more information about:

- Invoking message-secured web services, see Updating a Client Application to Invoke a Message-Secured Web Service in *Securing WebLogic Web Services for Oracle WebLogic Server*.
- Best practices for developing web service clients, see Roadmap for Developing JAX-WS Web Service Clients.
- Invoking web services asynchronously, see Developing Asynchronous Clients.
- Creating a *dynamic proxy client*, using the `javax.xml.ws.Service` API, that enables a web service client to invoke a web service based on a service endpoint interface (SEI) dynamically at run-time (without using `clientgen`), see Developing Dynamic Proxy Clients. This chapter focuses on how to generate a static Java class of the `Service` interface implementation for the particular web service you want to invoke.

Invoking a Web Service from a Java SE Client

The following table summarizes the main steps to create a Java SE application that invokes a web service.

> **Note:**
>
> In this section, it is assumed that:
>
> - When you invoke a web service using the client-side artifacts generated by the `clientgen` or `wsdlc` Ant tasks, you have the entire set of WebLogic Server classes in your CLASSPATH. Support for *standalone* Java applications that are running in an environment where WebLogic Server libraries is described in Invoking a Web Service from a Standalone Java SE Client.
> - You use Ant in your development environment to build your client application, compile Java files, and so on, and that you have an existing `build.xml` file that you want to update with web services client tasks. For general information about using Ant in your development environment, see Creating the Basic Ant build.xml File. For a full example of a `build.xml` file used in this section, see Sample Ant Build File for a Java Client.

Table 8-1 Steps to Invoke a Web Service from a Java SE Client

#	Step	Description
1	Set up the environment.	Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) command, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.
2	Update your `build.xml` file to execute the `clientgen` Ant task to generate the needed client-side artifacts to invoke a web service.	See Using the clientgen Ant Task To Generate Client Artifacts.
3	Get information about the web service, such as the signature of its operations and the name of the ports.	See Getting Information About a Web Service.
4	Write the client application Java code that includes code for invoking the web service operation.	See Writing the Java Client Application Code to Invoke a Web Service.
5	Create a basic Ant build file, `build.xml`.	See Creating the Basic Ant build.xml File.
6	Compile and run your Java client application.	See Compiling and Running the Client Application.

Using the clientgen Ant Task To Generate Client Artifacts

The `clientgen` WebLogic web services Ant task generates, from an existing WSDL file, the client artifacts that client applications use to invoke both WebLogic and non-WebLogic web services. These artifacts include:

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

For additional information about the `clientgen` Ant task, such as all the available attributes, see Ant Task Reference in the *WebLogic Web Services Reference for Oracle WebLogic Server*.

Update your `build.xml` file, adding a call to the `clientgen` Ant task, as shown in the following example:

```
<taskdef name="clientgen"
   classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
<target name="build-client">
   <clientgen
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
     destDir="clientclasses"
     packageName="examples.webservices.simple_client"
```

```
        type="JAXWS"/>
  </target>
```

Before you can execute the `clientgen` WebLogic web service Ant task, you must specify its full Java classname using the standard `taskdef` Ant task.

You must include the `wsdl` and `destDir` attributes of the `clientgen` Ant task to specify the WSDL file from which you want to create client-side artifacts and the directory into which these artifacts should be generated. The `packageName` attribute is optional; if you do not specify it, the `clientgen` task uses a package name based on the `targetNamespace` of the WSDL. The `type` is required in this example; otherwise, it defaults to `JAXRPC`.

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

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

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

If the WSDL file specifies that user-defined data types are used as input parameters or return values of web service operations, `clientgen` automatically generates a JavaBean class that is the Java representation of the XML Schema data type defined in the WSDL. The JavaBean classes are generated into the `destDir` directory.

For a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`, see Sample Ant Build File for a Java Client.

To execute the `clientgen` Ant task, along with the other supporting Ant tasks, specify the `build-client` target at the command line:

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

See the `clientclasses` directory to view the files and artifacts generated by the `clientgen` Ant task.

Getting Information About a Web Service

You need to know the name of the web service and the signature of its operations before you write your Java client application code to invoke an operation. There are a variety of ways to find this information.

The best way to get this information is to use the `clientgen` Ant task to generate the web service-specific `Service` files and look at the generated `*.java` files. These files are generated into the directory specified by the `destDir` attribute, with subdirectories corresponding to either the value of the `packageName` attribute, or, if this attribute is not specified, to a package based on the `targetNamespace` of the WSDL.

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

You can also examine the actual WSDL of the web service; see Browsing to the WSDL of the Web Service for details about the WSDL of a deployed WebLogic web service. The name of the web service is contained in the `<service>` element, as shown in the following excerpt of the `TraderService` WSDL:

```
<service name="TraderService">
  <port name="TraderServicePort"
       binding="tns:TraderServiceSoapBinding">
...
  </port>
</service>
```

The operations defined for this web service are listed under the corresponding `<binding>` element. For example, the following WSDL excerpt shows that the `TraderService` web service has two operations, `buy` and `sell` (for clarity, only relevant parts of the WSDL are shown):

```
<binding name="TraderServiceSoapBinding" ...>
  ...
  <operation name="sell">
  ...
  </operation>
  <operation name="buy">
  </operation>
</binding>
```

Writing the Java Client Application Code to Invoke a Web Service

In the following code example, a Java application invokes a web service operation. The application uses standard JAX-WS API code and the web service-specific implementation of the `Service` interface, generated by `clientgen`, to invoke an operation of the web service.

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

Because the `<clientgen>` `packageName` attribute was set to the same package name as the client application, we are not required to import the `<clientgen>`-generated files.

```
package examples.webservices.simple_client;
/**
 * This is a simple Java application that invokes the
 * the echoComplexType operation of the ComplexService web service.
 */
public class Main {
  public static void main(String[] args) {
    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
    in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
    System.out.println("echoComplexType called. Result: " + result.getIntValue()
+ ", " + result.getStringValue());
  }
}
```

In the preceding example:

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

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

  The `ComplexService` class implements the JAX-WS `Service` interface. The `getComplexServicePortTypePort()` method is used to return an instance of the `ComplexPortType` stub implementation.

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

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

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

Compiling and Running the Client Application

Add `javac` tasks to the `build-client` target in the `build.xml` file to compile all the Java files (both of your client application and those generated by `clientgen`) into class files, as shown by the **bold** text in the following example:

```
<target name="build-client">
  <clientgen
    wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
    destDir="clientclasses"
    packageName="examples.webservices.simple_client"
    type="JAXWS"/>
  <javac
    srcdir="clientclasses"
    destdir="clientclasses"
    includes="**/*.java"/>
  <javac
    srcdir="src"
    destdir="clientclasses"
    includes="examples/webservices/simple_client/*.java"/>
</target>
```

In the example, the first `javac` task compiles the Java files in the `clientclasses` directory that were generated by `clientgen`, and the second `javac` task compiles the Java files in the `examples/webservices/simple_client` subdirectory of the current directory; where it is assumed your Java client application source is located.

In the preceding example, the `clientgen`-generated Java source files and the resulting compiled classes end up in the same directory (`clientclasses`). Although this might be adequate for prototyping, it is often a best practice to keep source code (even generated code) in a different directory from the compiled classes. To do this, set the `destdir` for both `javac` tasks to a directory different from the `srcdir` directory. To run the client application, add a `run` target to the `build.xml` that includes a call to the `java` task, as shown below:

```
<path id="client.class.path">
    <pathelement path="clientclasses"/>
    <pathelement path="${java.class.path}"/>
</path>
<target name="run" >
    <java
       fork="true"
       classname="examples.webServices.simple_client.Main"
       failonerror="true" >
       <classpath refid="client.class.path"/>
</target>
```

The `path` task adds the `clientclasses` directory to the CLASSPATH. The `run` target invokes the `Main` application, passing it the URL of the deployed web service as its single argument.

See Sample Ant Build File for a Java Client for a full sample `build.xml` file that contains additional targets from those described in this procedure, such as `clean`.

Rerun the `build-client` target to regenerate the artifacts and recompile into classes, then execute the `run` target to invoke the `echoStruct` operation:

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

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

Sample Ant Build File for a Java Client

The following example shows a complete `build.xml` file for generating and compiling a Java client. See Using the clientgen Ant Task To Generate Client Artifacts and Compiling and Running the Client Application for explanations of the sections in **bold**.

```
<project name="webservices-simple_client" default="all">
  <!-- set global properties for this build -->
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="example-output" value="output" />
  <property name="clientclass-dir" value="${example-output}/clientclass" />
  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>
  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <target name="clean" >
    <delete dir="${clientclass-dir}"/>
```

```
  </target>
  <target name="all" depends="clean,build-client,run" />
  <target name="build-client">
    <clientgen
      wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.simple_client"
      type="JAXWS"/>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/simple_client/*.java"/>
  </target>
  <target name="run" >
    <java fork="true"
          classname="examples.webservices.simple_client.Main"
          failonerror="true" >
      <classpath refid="client.class.path"/>
    </java>
  </target>
</project>
```

Invoking a Web Service from a Standalone Java SE Client

In Invoking a Web Service from a Java SE Client, it is assumed that when you invoke a Web Service using the client-side artifacts generated by the `clientgen` or `wsdlc` Ant tasks, you have the entire set of WebLogic Server classes in your classpath. If, however, you do not have WebLogic Server installed locally, you can still invoke a Web Service by using one of the standalone WebLogic web services client JAR files.

Table 8-2 summarizes the standalone web service client JAR files that are available in the installation.

Table 8-2 Standalone Web Service Client JAR Files

JAR File	Location	Description
`com.oracle.webservices.wls.jaxws-wlswss-client.jar`	*ORACLE_HOME*`/wlserver/modules/clients/`	Supports basic JAX-WS client-side functionality including: • Using client-side artifacts created by both the `clientgen` Ant tasks • Processing SOAP messages • Using advanced features, such as web services reliable messaging, WS addressing, asynchronous request-response, and MTOM • Using WS-Security • Using client-side SOAP message handlers • Invoking both JAX-WS and JAX-RPC web services • Using SSL The standalone client JAR does not support invoking web services that use the following advanced features: • SOAP over JMS transport • Conversations • Buffering

To use a standalone web services client JAR file with your client application, perform the following steps:

1. Create a Java SE client using your favorite IDE, such as Oracle JDeveloper. For more information, see Developing and Securing Web Services and Clients in *Developing Applications with Oracle JDeveloper*.
2. Copy the required JAR files, defined in Table 8-2, from the computer hosting WebLogic Server to the appropriate directory on the standalone client computer.

   For example, you might copy the files into the directory that contains other classes used by your client application.

Invoking a Web Service from Another WebLogic Web Service

Invoking a web service from a Java EE client, such as another WebLogic web service, is similar to invoking one from a Java SE application, as described in Invoking a Web Service from a Java SE Client, with the following variation:

- Instead of using the `clientgen` Ant task to generate the JAX-WS `Service` interface of the web service to be invoked, you use the `<clientgen>` child element of the `<jws>` element, inside the `jwsc` Ant task that compiles the invoking web service. In the JWS file that invokes the other web service, however, you still use the same standard JAX-WS APIs to get `Service` and `PortType` instances to invoke the web service operations.
- You can use the `@WebServiceRef` annotation to define a reference to a web service, as described in Sample JWS File That Invokes a Web Service.

This section describes the differences between invoking a web service from a client in a Java EE component, specifically another web service, and invoking from a Java SE client. It is assumed that you use Ant in your development environment to build your client application,

compile Java files, and so on, and that you have an existing `build.xml` that builds a web service that you want to update to invoke another web service.

The following list describes the changes you must make to the `build.xml` file that builds your client web service, which will invoke another web service. See Sample build.xml File for a Web Service Client for the full sample `build.xml` file:

- Add a `<clientgen>` child element to the `<jws>` element that specifies the JWS file that implements the web service that invokes another web service. Set the required `wsdl` attribute to the WSDL of the web service to be invoked. Set the required `packageName` attribute to the package into which you want the JAX-WS client stubs to be generated.

The following list describes the changes you must make to the JWS file that implements the client web service; see Sample JWS File That Invokes a Web Service for the full JWS file example.

- Import the files generated by the `<clientgen>` child element of the `jwsc` Ant task. These include the JAX-WS `Service` interface of the invoked web service, as well as the Java representation of any user-defined data types used as parameters or return values in the operations of the invoked web service.

> **Note:**
>
> If the package name set using the `packageName` attribute of `<clientgen>` is set to the same package name as the client application, then you are not required to import the `<clientgen>`-generated files.

- Get the `Service` and `PortType` interface implementation and invoke the operation on the port as usual; see Writing the Java Client Application Code to Invoke a Web Service for details.

Sample build.xml File for a Web Service Client

The following sample `build.xml` file shows how to create a web service that itself invokes another web service; the relevant sections that differ from the `build.xml` for building a simple web service that does not invoke another web service are shown in **bold**.

The `build-service` target in this case is very similar to a target that builds a simple web service; the only difference is that the `jwsc` Ant task that builds the invoking web service also includes a `<clientgen>` child element of the `<jws>` element so that `jwsc` also generates the required JAX-WS client stubs.

```
<project name="webservices-service_to_service" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="ClientServiceEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/ClientServiceEar" />
  <property name="clientclass-dir" value="${example-output}/clientclasses" />
```

```
<path id="client.class.path">
  <pathelement path="${clientclass-dir}"/>
  <pathelement path="${java.class.path}"/>
</path>
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JwscTask" />
<taskdef name="clientgen"
  classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
<taskdef name="wldeploy"
  classname="weblogic.ant.taskdefs.management.WLDeploy"/>
<target name="all" depends="clean,build-service,deploy,client" />
<target name="clean" depends="undeploy">
  <delete dir="${example-output}"/>
</target>
<target name="build-service">
  <jwsc
      srcdir="src"
      destdir="${ear-dir}" >
      <jws
       file="examples/webservices/service_to_service/ClientServiceImpl.java"
       type="JAXWS">
        <clientgen
              wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
              packageName="examples.webservices.complex" />
      </jws>
  </jwsc>
</target>
<target name="deploy">
  <wldeploy action="deploy" name="${ear.deployed.name}"
    source="${ear-dir}" user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>
<target name="undeploy">
  <wldeploy action="undeploy" name="${ear.deployed.name}"
    failonerror="false"
    user="${wls.username}"
    password="${wls.password}" verbose="true"
    adminurl="t3://${wls.hostname}:${wls.port}"
    targets="${wls.server.name}" />
</target>
<target name="client">
  <clientgen
    wsdl="http://${wls.hostname}:${wls.port}/ClientService/ClientService?WSDL"
    destDir="${clientclass-dir}"
    packageName="examples.webservices.service_to_service.client"
    type="JAXWS"/>
  <javac
    srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
    includes="**/*.java"/>
  <javac
    srcdir="src" destdir="${clientclass-dir}"
    includes="examples/webservices/service_to_service/client/**/*.java"/>
</target>
<target name="run">
  <java classname="examples.webservices.service_to_service.client.Main"
        fork="true"
        failonerror="true" >
        <classpath refid="client.class.path"/>
  </java>
```

```
  </target>
</project>
```

Sample JWS File That Invokes a Web Service

The following sample JWS file, called `ClientServiceImpl.java`, implements a web service called `ClientService` that has an operation that in turn invokes the `echoComplexType` operation of a web service called `ComplexService`. This operation has a user-defined data type (`BasicStruct`) as both a parameter and a return value. The relevant code is shown in **bold** and described after the example.

```
package examples.webservices.service_to_service;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.WebServiceRef;

// Import the BasicStruct data type, generated by clientgen and used
// by the ComplexService Web Service
import examples.webservices.complex.BasicStruct;

// Import the JAX-WS stubs generated by clientgen for invoking
// the ComplexService web service.
import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;

@WebService(name="ClientPortType", serviceName="ClientService",
            targetNamespace="http://examples.org")
public class ClientServiceImpl {
// Use the @WebServiceRef annotation to define a reference to a web service.
  @WebServiceRef()
  ComplexService test;

  @WebMethod()
  public String callComplexService(BasicStruct input, String serviceUrl)
  {
    // Create a port stub to invoke ComplexService
    ComplexPortType port = test.getComplexPortTypePort();

    // Invoke the echoComplexType operation of ComplexService
    BasicStruct result = port.echoComplexType(input);
    System.out.println("Invoked ComplexPortType.echoComplexType." );
    return "Invoke went okay!  Here's the result: '" + result.getIntValue() +
           ",  " + result.getStringValue() + "'";
  }
}
```

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

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

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

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

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

- Define a reference to a web service and an injection target for it using the `@WebServiceRef` annotation:

  ```
  @WebServiceRef()
  ComplexService service;
  ```

  For more information about `@WebServiceRef`, see Defining a Web Service Reference Using the @WebServiceRef Annotation.

  Alternatively, you can create a proxy stub to the `ComplexService` web service, as shown below:

  ```
  ComplexService service = new ComplexService();
  ```

- Return an instance of the `ComplexPortType` stub implementation by calling the `getComplexPortTypePort()` operation on the web service reference:

  ```
  ComplexPortType port = service.getComplexPortTypePort();
  ```

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

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

Configuring Web Service Clients

By default, web service clients use the web service configuration defined for the server. You can override the configuration settings used by the web service client using one of the following methods:

- Using the Administration or WLST, if applicable. Only a subset of web service features are configurable on the client.
- Using the `@WebServiceRef` annotation to associate the web service client with the configuration defined for the specified web service reference. The web service reference configuration is defined in the `weblogic.xml` for Web containers and `weblogic-ejb-jar.xml` for EJB containers. For more information about the `@WebServiceRef` annotation, see Defining a Web Service Reference Using the @WebServiceRef Annotation.
- Using the `WsrmClientInitFeature` when creating a web services reliable messaging client. For more information, see Configuring Reliable Messaging on Web Service Clients.

Defining a Web Service Reference Using the @WebServiceRef Annotation

The `@WebServiceRef` annotation enables you to define a reference to a web service and attach the configuration of the web service to the client instance.

For example, in the following code excerpt, `@WebServiceRef` is used to attach the configuration for `ReliableEchoService` to the client's web service instance. The port that is subsequently created and initialized uses the properties defined for `ReliableEchoService` service reference in the `weblogic.xml` for the Web application.

```
package wsrm_jaxws.example;
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
...
@WebService
public class ClientServiceImpl {

    @WebServiceRef(name="MyServiceRef")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;

    @PostConstruct
    public void initPort() {
        port = service.getReliableEchoPort();
    ...
    }
}
```

Example 8-1 shows an example of a `weblogic.xml` file that contains a web service reference description. For information about the reliable messaging properties shown in this example, see Configuring Reliable Messaging.

Example 8-1 Example weblogic.xml File Containing Web Service Reference Description

```
<?xml version='1.0' encoding='UTF-8'?>
<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app">
   <service-reference-description>
     <!-- Any name you want, but use this same name on
          @WebServiceRef(name=<my name>). This anno goes on the service
          field in your client container -->
      <service-ref-name>MyServiceRef</service-ref-name>
      <!-- Use / and any path within the web app to get a local WSDL, or
           use a resource name as defined by the Java ClassLoader, or use an
           absolute/external URL you can guarantee is deployed when this web
           app deploys -->
      <wsdl-url>/WEB-INF/wsdls/ReliableEcho.wsdl</wsdl-url>
      <!-- One or more port-infos, one for each type of port/stub you'll create
           in your JWS -->
      <port-info>
         <!-- The local name of wsdl:port (not portType). The Java type for this
              port, when created from the @WebServiceRef JWS field, will contain,
              in RequestContext, the props you define below -->
         <port-name>ReliableEchoPort</port-name>

         <!-- Any prop name/value pairs you want to show up on you service stub
              The Java type for this port, when created from the @WebServiceRef JWS field,
              will contain, in RequestContext, the stub-props you define below -->

         <!-- RM Source Properties -->

         <stub-property>
            <name>weblogic.wsee.wsrm.BaseRetransmissionInterval</name>
            <value>PT30S</value>
         </stub-property>

         <stub-property>
            <name>weblogic.wsee.wsrm.RetransmissionExponentialBackoff</name>
            <value>true</value>
         </stub-property>
```

```
        <!-- RM Destination Properties -->

         <stub-property>
           <name>weblogic.wsee.wsrm.RetryCount</name>
           <value>5</value>
        </stub-property>

        <stub-property>
           <name>weblogic.wsee.wsrm.RetryDelay</name>
           <value>PT30S</value>
        </stub-property>

        <stub-property>
           <name>weblogic.wsee.wsrm.AcknowledgementInterval</name>
           <value>PT5S</value>
        </stub-property>

        <stub-property>
           <name>weblogic.wsee.wsrm.NonBufferedDestination</name>
           <value>true</value>
        </stub-property>

        <!-- RM Source *or* Destination Properties -->

        <stub-property>
           <name>weblogic.wsee.wsrm.InactivityTimeout</name>
           <value>PT5M</value>
        </stub-property>

        <stub-property>
           <name>weblogic.wsee.wsrm.SequenceExpiration</name>
           <value>PT10M</value>
        </stub-property>

   </port-info>

   </service-reference-description>
   <wl-dispatch-policy>weblogic.wsee.mdb.DispatchPolicy</wl-dispatch-policy>
</weblogic-web-app>
```

Managing Client Identity

Web services enable you to assign any meaningful name to a client, which is represented as the client identity (client ID). This client ID is used to group statistics and other monitoring information, and for reporting runtime validations, and so on.

For on-server clients (clients running in a container within a WebLogic Server instance), the client ID can be generated in one of the following ways:

- By the client when it initializes connection to web service port. This is the recommended approach. See Defining the Client ID During Port Initialization.
- By the server and discovered later by the client. See Accessing the Server-generated Client ID.

> **Note:**
>
> Although optional, Oracle strongly recommends that you define the client ID explicitly.

The `weblogic.wsee.jaxws.persistence.ClientIdentityFeature` client feature enables web service clients to set and access the web service client ID. The following table summarizes the `ClientIdentityFeature` methods.

Table 8-3 Methods of ClientIdentityFeature for Setting and Accessing Client ID

Method	Description
`getClientID()`	Gets the currently defined client ID for the web service port.
`setClientID()`	Sets the client ID for the web service port. In addition, you can set the client ID by passing it as an argument when instantiating the `ClientIdentityFeature` object. For example: `ClientIdentityFeature clientIDFeature = new ClientIdentityFeature("MyBackendServiceAsyncClient");`
`dispose()`	Disposes the client ID. If a client ID is not disposed of explicitly, it will be done when the container for the client instances that use the client ID is deactivated (for example, the host Web application or EJB is deactivated). For more information, see Client Identity Lifecycle.

The following sections describe the methods for managing the client ID:

- Defining the Client ID During Port Initialization
- Accessing the Server-generated Client ID
- Client Identity Lifecycle

Defining the Client ID During Port Initialization

To provide its client ID, the web service client can pass an instance of the `ClientIdentityFeature` containing the client ID to the web service port at initialization time.

The client ID must be unique within the Web application or EJB that contains the client. It is recommended that the client ID appropriately reflect the business purpose. In order to ensure that the client ID is unique, the system prepends the names of the containing server, application, and component (Web application or EJB) to the client ID.

> **Note:**
>
> Care should be taken when choosing a client ID. If a client instance is created with the same client ID as an existing client instance, the two client instances will be treated as the same instance. No exception will be thrown to alert you to the duplication.

The following example demonstrates this method of specifying the client ID. It is recommended that you close the client instance once all processing has been complete, as shown.

This example is excerpted from Roadmap for Developing JAX-WS Web Service Clients.

Example 8-2 Example of Specifying the Client ID During Port Initialization

```
import javax.servlet.*;
import javax.xml.ws.*;
import weblogic.jws.jaxws.client.ClientIdentityFeature;
. . .
public class BestPracticeAsyncClient
  extends GenericServlet {
...
  private BackendServiceService _service;
...
    // Client ID
    ClientIdentityFeature clientIdFeature =
        new ClientIdentityFeature("MyBackendServiceAsyncClient");
    features.add(clientIdFeature);
...
     _features = features.toArray(new WebServiceFeature[features.size()]);
...
    BackendService port = _service.getBackendServicePort(_features);
...
    ((java.io.Closeable)_port).close();
  }
}
```

Accessing the Server-generated Client ID

> **Note:**
>
> As described in this section, in order to ensure that the client ID is unique, the server-generated version may be long and difficult to read. To guarantee that the client ID is presented in a user-friendly format, it is recommended that you define the client ID during port initialization, as described in Defining the Client ID During Port Initialization.

Client IDs that are generated automatically by the server use the following format:

applicationname[*_applicationversion*]:*componentname*:*uniqueID*

Where:

- *applicationname*—Name of the application hosting the client.
- *applicationversion*—Version of the application. Only used if multiple versions of the same application is running simultaneously.
- *componentname*—Name of the component (Web application or EJB) hosting the client.
- *uniqueID*—Calculated based on the information that is available when the client instance is created. The `uniqueID` is constructed by choosing one of the following (whichever is available):
  - Web service reference name, as defined by the `@WebServiceRef` annotation.
  - `[portNamespaceURI:portLocalName][:][endpointAddress]`—port name, endpoint address, or both (separated by a colon).
  - Port class simple name.

  The following information, when available, may also be concatenated to the `uniqueID`, separated by a colon (:), in the order presented below:
  - WSDL location (minus `?wsdl`)
  - Features used to create the client instance, represented by the features class name and separated by dash (-).

For example, assume that you deploy a web service client with the following information associated with it:

- Application name: `example`
- Component: Web application called `BestPracticeClient`
- Port name: `http://example/BackendServicePort`
- Port class: `BackendService`
- WSDL: `jar:file:/C:/example/BackendService.war!/WEB-INF/BackendServiceService.wsdl`

The server-generated client ID will be:

```
example:BestPracticeClient:http://example/:BackendServicePort:jar:file:/C:/
example/BackendService.war!/WEB-INF/
BackendServiceService.wsdl:AsyncClientTransportFeature()-ClientIdentityFeature
```

Each time the code is executed, assuming it is in the same containment hierarchy, the same client ID is generated. This provides a stable client ID that can be used across server VM instances and allows for asynchronous responses to be delivered to the client even after a server restart.

> **Note:**
>
> A given Client ID can be used from multiple locations in the client code, but care should be taken to initialize any port or Dispatch instance that uses that client ID in the same way (same features, service, and so on) as was used in any other location for that client ID.
>
> For best practice information on the recommended approach to client instance (port or Dispatch) initialization, see Roadmap for Developing JAX-WS Web Service Clients.

The following example demonstrates how to access the server-generated client ID. This example is excerpted from Table 7-1.

Example 8-3 Example of Accessing the Server-generated Client ID

```
...
    // Create a port without explicitly defining the client ID to view the client ID that is
    // generated automatically.
    ClientIdentityFeature dummyClientIdFeature = new ClientIdentityFeature(null);
    BackendService dummyPort = _service.getBackendServicePort(dummyClientIdFeature);
    System.out.println("Generated Client Identity is: " + dummyClientIdFeature.getClientId());

    // Best Practice: Explicitly close client instances when processing is complete.
    // If not closed, the port will be closed automatically when it goes out of scope.
    // Note, this client ID will remain registered and visible until our
    // container (Web application) is undeployed.
    ((java.io.Closeable)dummyPort).close();
```

Client Identity Lifecycle

A client ID is *registered* with the web services runtime when the first client instance (port or Dispatch instance) using the client ID is created. Any asynchronous response endpoint associated with the client instances is also tracked along with the registered client ID.

The client ID remains registered until one of the following occurs:

- The client ID is explicitly disposed using the `dispose()` method on `ClientIdentityFeature`, as described in Table 8-3.
- The container for the client instances that use the client ID is deactivated (for example, the host Web application or EJB is deactivated).

Using a Proxy Server When Invoking a Web Service

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

Using the ClientProxyFeature API to Specify the Proxy Server

You can programmatically specify within the Java client application itself the details of the proxy server that will proxy the web service invoke using the `weblogic.wsee.jaxws.proxy.ClientProxyFeature` API. See ClientProxyFeature in *Java API Reference for Oracle WebLogic Server*.

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

> **Note:**
>
> The `ClientProxyFeature` configures the port for WebLogic HTTP over SSL. It is recommended that you configure SSL for WebLogic Server. For more information, see Configuring SSL in *Administering Security for Oracle WebLogic Server*.
>
> The `ClientProxyFeature setUseSunHttpHandler` method forces WebLogic Server to use the Sun HTTP implementation on a per-connection-request basis. You can instead use the `-DUseSunHttpHandler=true` WebLogic Server startup configuration option, which applies the setting for the WebLogic Server instance.

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

Example 8-4 Pass ClientProxyFeature as an Argument When Creating Port

```
package examples.webservices.simple_client;
import weblogic.wsee.jaxws.proxy
public class Main {
  public static void main(String[] args) {
    ComplexService test = new ComplexService();
    ClientProxyFeature cpf = new ClientProxyFeature();
    cpf.setProxyHost("localhost");
    cpf.setProxyPort(8888);
    cpf.setProxyUserName("proxyu");
    cpf.setProxyPassword("proxyp");
    ComplexPortType port = test.getComplexPortTypePort(cpf);
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
    in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
    System.out.println("echoComplexType called. Result: " + result.getIntValue() + ", " +
result.getStringValue());
  }
}
```

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

Example 8-5 Configuring the ClientProxyFeature After Creating the Port

```
package examples.webservices.simple_client;
import weblogic.wsee.jaxws.proxy
public class Main {
  public static void main(String[] args) {
    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();
    ClientProxyFeature cpf = new ClientProxyFeature();
    cpf.setProxyHost("localhost");
    cpf.setProxyPort(8888);
    cpf.setProxyUserName("proxyu");
    cpf.setProxyPassword("proxyp");
    cpf.attachsPort(port);
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
    in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
    System.out.println("echoComplexType called. Result: " + result.getIntValue() + ", " +
result.getStringValue());
  }
}
```

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

Example 8-6 Disabling Client Proxy Settings

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

Using System Properties to Specify the Proxy Server

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

The following table summarizes the Java system properties.

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

Table 8-4 Java System Properties Used to Specify Proxy Server

Property	Description
http.proxyHost=*proxyHost* or https.proxyHost=*proxyHost*	Name of the host computer on which the proxy server is running. Use https.proxyHost for HTTP over SSL.

Table 8-4 (Cont.) Java System Properties Used to Specify Proxy Server

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

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

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

Client Considerations When Redeploying a Web Service

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

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the web service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older web service.

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

- WSDL that describes the web service
- WS-Policy files attached to the web service

If any of these artifacts have changed, you must regenerate the JAX-WS stubs used by the client application by re-running the `clientgen` Ant task.

For example, if you change the signature of an operation in the new version of the web service, then the WSDL file that describes the new version of the web service will also change. In this case, you must regenerate the JAX-WS stubs. If, however, you simply

change the implementation of an operation, but do not change its public contract, then you can continue using the existing client application.

Client Considerations When Web Service and Client Are Deployed to the Same Managed Server

If a web service and client are deployed to the same Managed Server, and one of the following is true:

- The web service client uses the `@WebServiceRef` annotation, but does not specify a value for the `wsdlLocation` element.
- The web service client uses the `wsdlLocation` element of the `@WebServiceRef` annotation to refer to the live WSDL location (for example, `@WebServiceRef(wsdlLocation="http://xyz.com/myService?WSDL")`), as opposed to a WSDL that is packaged with the web service application (for example, `@WebServiceRef(wsdlLocation="myService.wsdl")`).

Then, when you restart the Managed Server on which the web service and client are deployed, the web service client may fail to redeploy, regardless of the deployment order, because the applications are deployed initially in administration mode, and later transition to production mode to accept HTTP requests. In this situation, you must restart the application manually once the server has restarted.

If a web service and client are deployed to the same Managed Server, to avoid this situation, it is recommended that you package the WSDL as part of the web service application and refer to the packaged version from the `@WebServiceRef` annotation.

9

Examples of Developing JAX-WS Web Service Clients

This chapter provides some common examples of developing WebLogic web service clients using Java API for XML-based Web services (JAX-WS).
This chapter includes the following sections:

- Developing a JAX-WS Java SE Client
- Invoking a Web Service from a WebLogic Web Service

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

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

> **Note:**
>
> For best practice examples demonstrating advanced web service features, see Roadmap for Developing JAX-WS Web Service Clients and Roadmap for Developing Reliable Web Services and Clients.

Developing a JAX-WS Java SE Client

> **Note:**
>
> You can invoke a web service from any Java SE or Java EE application running on WebLogic Server (with access to the WebLogic Server classpath). Invoking a web service from *standalone* Java applications that are running in an environment where WebLogic Server libraries are not available is not supported in this release for JAX-WS web services.

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

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

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

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

> **Note:**
>
> It is assumed in this procedure that you have created and deployed the `ComplexService` web service.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.

2. Create a project directory:

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

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

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

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

   ```
   <project name="webservices-simple_client" default="all">
     <taskdef name="clientgen"
       classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
   </project>
   ```

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

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

   ```
   <target name="build-client">
     <clientgen
       wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
   ```

```
      destDir="output/clientclass"
      packageName="examples.webservices.simple_client"
      type="JAXWS"/>
    <javac
      srcdir="output/clientclass" destdir="output/clientclass"
      includes="**/*.java"/>
 <javac
    srcdir="src" destdir="output/clientclass"
    includes="examples/webservices/simple_client/*.java"/>
</target>
```

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

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

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

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

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

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

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

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

   The application follows standard JAX-WS guidelines to invoke an operation of the web service using the web service-specific implementation of the `Service` interface generated by `clientgen`. For details, see Developing Web Service Clients.

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

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

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

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

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

```
<path id="client.class.path">
  <pathelement path="output/clientclass"/>
  <pathelement path="${java.class.path}"/>
</path>
<target name="run" >
  <java fork="true"
        classname="examples.webservices.simple_client.Main"
        failonerror="true" >
    <classpath refid="client.class.path"/>
</target>
```

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

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

```
prompt> ant run
```

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

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

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

Sample Java Client Application

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

```
package examples.webservices.simple_client;
/**
 * This is a simple Java application that invokes the
 * echoComplexType operation of the ComplexService web service.
 */
public class Main {
  public static void main(String[] args) {
    ComplexService test = new ComplexService();
    ComplexPortType port = test.getComplexPortTypePort();
    BasicStruct in = new BasicStruct();
    in.setIntValue(999);
    in.setStringValue("Hello Struct");
    BasicStruct result = port.echoComplexType(in);
    System.out.println("echoComplexType called. Result: " + result.getIntValue()
+ ", " + result.getStringValue());
  }
}
```

Sample Ant Build File For Building Java Client Application

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

```
<project name="webservices-simple_client" default="all">
  <!-- set global properties for this build -->
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="example-output" value="output" />
  <property name="clientclass-dir" value="${example-output}/clientclass" />
  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>
  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <target name="clean" >
    <delete dir="${clientclass-dir}"/>
  </target>
  <target name="all" depends="clean,build-client,run" />
  <target name="build-client">
    <clientgen
     type="JAXWS"
     wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
      destDir="${clientclass-dir}"
      packageName="examples.webservices.simple_client"/>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/simple_client/*.java"/>
  </target>
  <target name="run" >
    <java fork="true"
          classname="examples.webservices.simple_client.Main"
          failonerror="true" >
      <classpath refid="client.class.path"/>
    </java>
  </target>
</project>
```

Invoking a Web Service from a WebLogic Web Service

You can invoke a web service (WebLogic, Microsoft .NET, and so on) from within a deployed WebLogic web service.

The procedure is similar to that described in Developing a JAX-WS Java SE Client except that instead of running the `clientgen` Ant task to generate the client stubs, you use the `<clientgen>` child element of `<jws>`, inside of the `jwsc` Ant task. The `jwsc` Ant task automatically packages the generated client stubs in the invoking web service WAR file so that the web service has immediate access to them. You then follow standard JAX-WS programming guidelines in the JWS file that implements the web service that invokes the other web service.

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

> **Note:**
>
> It is assumed that you have successfully deployed the `ComplexService` web service.

1. Set your WebLogic Server environment.

   Open a command window and execute the `setDomainEnv.cmd` (Windows) or `setDomainEnv.sh` (UNIX) script, located in the `bin` subdirectory of your domain directory. The default location of WebLogic Server domains is *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*, where *`ORACLE_HOME`* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server and *`domainName`* is the name of your domain.

2. Create a project directory:

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

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

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

4. Create the JWS file that implements the web service that invokes the `ComplexService` web service.

   Open your favorite Java IDE or text editor and create a Java file called `ClientServiceImpl.java` using the Java code specified in Sample ClientServiceImpl.java JWS File.

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

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

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

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

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

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

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

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

```
<target name="build-service">
  <jwsc
    srcdir="src"
    destdir="output/ClientServiceEar" >
    <jws
      file="examples/webservices/service_to_service/ClientServiceImpl.java"
     type="JAXWS">
            <WLHttpTransport
             contextPath="ClientService" serviceUri="ClientService"
             portName="ClientServicePort"/>
     <clientgen
       type="JAXWS"
wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
       packageName="examples.webservices.complex" />
    </jws>
  </jwsc>
</target>
```

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

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

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

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

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

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

9. Start the WebLogic Server instance to which you will deploy the web service.

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

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

```
  <taskdef name="wldeploy"
           classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="deploy">
    <wldeploy action="deploy" name="ClientServiceEar"
      source="ClientServiceEar" user="${wls.username}"
      password="${wls.password}" verbose="true"
```

```
        adminurl="t3://${wls.hostname}:${wls.port}"
        targets="${wls.server.name}" />
    </target>
```

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

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

```
prompt> ant deploy
```

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

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

See Developing a JAX-WS Java SE Client for an example of creating a Java client application that invokes a web service.

Sample ClientServiceImpl.java JWS File

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

```
package examples.webservices.service_to_service;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.WebServiceRef;

// Import the BasicStruct data type, generated by clientgen and used
// by the ComplexService Web Service
import examples.webservices.complex.BasicStruct;

// Import the JAX-WS stubs generated by clientgen for invoking
// the ComplexService web service.
import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;

@WebService(name="ClientPortType", serviceName="ClientService",
            targetNamespace="http://examples.org")
public class ClientServiceImpl {
// Use the @WebServiceRef annotation to define a reference to the
// ComplexService web service.
  @WebServiceRef()
  ComplexService test;

  @WebMethod()
  public String callComplexService(BasicStruct input, String serviceUrl)
  {
    // Create a port stub to invoke ComplexService
    ComplexPortType port = test.getComplexPortTypePort();

    // Invoke the echoComplexType operation of ComplexService
    BasicStruct result = port.echoComplexType(input);
    System.out.println("Invoked ComplexPortType.echoComplexType." );
    return "Invoke went okay!  Here's the result: '" + result.getIntValue() +
           ",  " + result.getStringValue() + "'";
  }
}
```

Sample Ant Build File For Building ClientService

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

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

```
<project name="webservices-service_to_service" default="all">
  <!-- set global properties for this build -->
  <property name="wls.username" value="weblogic" />
  <property name="wls.password" value="weblogic" />
  <property name="wls.hostname" value="localhost" />
  <property name="wls.port" value="7001" />
  <property name="wls.server.name" value="myserver" />
  <property name="ear.deployed.name" value="ClientServiceEar" />
  <property name="example-output" value="output" />
  <property name="ear-dir" value="${example-output}/ClientServiceEar" />
  <property name="clientclass-dir" value="${example-output}/clientclasses" />
  <path id="client.class.path">
    <pathelement path="${clientclass-dir}"/>
    <pathelement path="${java.class.path}"/>
  </path>
  <taskdef name="jwsc"
    classname="weblogic.wsee.tools.anttasks.JwscTask" />
  <taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />
  <taskdef name="wldeploy"
    classname="weblogic.ant.taskdefs.management.WLDeploy"/>
  <target name="all" depends="clean,build-service,deploy,client" />
  <target name="clean" depends="undeploy">
    <delete dir="${example-output}"/>
  </target>
  <target name="build-service">
    <jwsc
        srcdir="src"
        destdir="${ear-dir}" >
       <jws
         file="examples/webservices/service_to_service/ClientServiceImpl.java"
         type="JAXWS">
         <WLHttpTransport
          contextPath="ClientService" serviceUri="ClientService"
          portName="ClientServicePort"/>
         <clientgen
               type="JAXWS"
           wsdl="http://${wls.hostname}:${wls.port}/complex/ComplexService?WSDL"
                packageName="examples.webservices.complex" />
      </jws>
    </jwsc>
  </target>
  <target name="deploy">
    <wldeploy action="deploy" name="${ear.deployed.name}"
      source="${ear-dir}" user="${wls.username}"
      password="${wls.password}" verbose="true"
      adminurl="t3://${wls.hostname}:${wls.port}"
      targets="${wls.server.name}" />
  </target>
  <target name="undeploy">
    <wldeploy action="undeploy" name="${ear.deployed.name}"
      failonerror="false"
```

```
        user="${wls.username}"
        password="${wls.password}" verbose="true"
        adminurl="t3://${wls.hostname}:${wls.port}"
        targets="${wls.server.name}" />
    </target>
    <target name="client">
      <clientgen
        wsdl="http://${wls.hostname}:${wls.port}/ClientService/ClientService?WSDL"
        destDir="${clientclass-dir}"
        packageName="examples.webservices.service_to_service.client"
        type="JAXWS"/>
      <javac
        srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
        includes="**/*.java"/>
      <javac
        srcdir="src" destdir="${clientclass-dir}"
        includes="examples/webservices/service_to_service/client/**/*.java"/>
    </target>
    <target name="run">
      <java classname="examples.webservices.service_to_service.client.Main"
            fork="true"
            failonerror="true" >
            <classpath refid="client.class.path"/>
      </java>
    </target>
  </project>
```

Part IV

Developing Advanced Features of JAX-WS Web Services

Part IV describes how to develop advanced features of WebLogic web services using Java API for XML-based Web services (JAX-WS).

Sections include:

10

Using Web Services Addressing

This chapter describes how to use Web Services Addressing (WS-Addressing) for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of WS-Addressing

WS-Addressing provides a transport-neutral mechanism to address web services and their associated messages. Using WS-Addressing, endpoints are uniquely and unambiguously defined in the SOAP header.

WS-Addressing provides two key components that enable transport-neutral addressing, including:

- **Endpoint reference (EPR)**—Communicates the information required to address a web service endpoint.
- **Message addressing properties**—Communicates end-to-end message characteristics, including addressing for source and destination endpoints and message identity, that allows uniform addressing of messages independent of the underlying transport.

  Message addressing properties can include one or more of the properties defined in Table 10-1. All properties are optional except `wsa:Action`.

Table 10-1 WS-Addressing Message Addressing Properties

Component	Description
`wsa:To`	Destination. If not specified, the destination defaults to `http://www.w3.org/2005/08/addressing/anonymous`.
`wsa:From`	Source endpoint.
`wsa:ReplyTo`	Reply endpoint. If not specified, the reply endpoint defaults to `http://www.w3.org/2005/08/addressing/anonymous`.
`wsa:FaultTo`	Fault endpoint.
`wsa:Action`	Required action. This property is required when WS-Addressing is enabled. It can be implicitly or explicitly configured, as described in Associating WS-Addressing Action Properties.
`wsa:MessageID`	Unique ID of the message.

Table 10-1 (Cont.) WS-Addressing Message Addressing Properties

Component	Description
`wsa:RelatesTo`	Message ID to which the message relates. This element can be repeated if there are multiple related messages. You can specify `RelationshipType` as an attribute of this property, which defaults to `http://www.w3.org/2005/08/addressing/reply`.
`wsa:ReferenceParameters`	Reference parameters that need to be communicated.

Example 10-1 shows a SOAP 1.2 request message sent over HTTP 1.2 with WS-Addressing enabled. As shown in **bold**, WS-Addressing provides a transport-neutral mechanism for defining a unique ID for the message (`wsa:MessageID`), the destination (`wsa:To`) endpoint, the reply endpoint (`wsa:ReplyTo`), and the required action (`wsa:Action`).

A response to this message may appear as shown in Example 10-2. The `RelatesTo` property correlates the response message with the original request message.

WS-Addressing is used by the following advanced WebLogic JAX-WS features:

- Asynchronous client transport, as described in Developing Asynchronous Clients.
- WS-ReliableMessaging, as described in Using Web Services Reliable Messaging.
- Callbacks, as described in Using Callbacks.

The following sections describe how to enable WS-Addressing on the web service or client, and explicitly define the action and fault properties.

A Note About WS-Addressing Standards Supported

WebLogic web services support the following standards for web service addressing:

- W3C WS-Addressing, as described at: `http://www.w3.org/2002/ws/addr/`
- Member Submission, as described at: `http://www.w3.org/Submission/ws-addressing/`

This chapter focuses on the use of W3C WS-Addressing only.

Example 10-1 SOAP 1.2 Message With WS-Addressing—Request Message

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
            xmlns:wsa="http://www.w3.org/2005/08/addressing/">
  <S:Header>
    <wsa:MessageID>
      http://example.com/someuniquestring
    </wsa:MessageID>
    <wsa:ReplyTo
      <wsa:Address>http://example.com/Myclient</wsa:Address>
    </wsa:ReplyTo>
    <wsa:To>
      http://example.com/fabrikam/Purchasing
    </wsa:To>
    <wsa:Action>
      http://example.com/fabrikam/SubmitPO
    </wsa:Action>
  <S:Header>
  <S:Body>
```

```
    ...
   </S:Body>
</S:Envelope>
```

Example 10-2 SOAP 1.2 Message Without WS-Addressing—Response Message

```
<S:Envelope
  xmlns:S="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <S:Header>
    <wsa:MessageID>http://example.com/someotheruniquestring</wsa:MessageID>
    <wsa:RelatesTo>http://example.com/someuniquestring</wsa:RelatesTo>
    <wsa:To>http://example.com/MyClient/wsa:To>
    <wsa:Action>
      http://example.com/fabrikam/SubmitPOAck
    </wsa:Action>
  </S:Header>
  <S:Body>
    ...
  </S:Body>
</S:Envelope>
```

Enabling WS-Addressing on the Web Service

By default, WS-Addressing is disabled on a web service endpoint, and any WS-Addressing headers received are ignored. You can enable WS-Addressing on the Web Service starting from Java or WSDL, as described in the following sections:

- Enabling WS-Addressing on the Web Service (Starting From Java)
- Enabling WS-Addressing on the Web Service (Starting from WSDL)

When you enable WS-Addressing on a web service endpoint:

- All WS-Addressing headers are understood by the endpoint. That is, if any WS-Addressing header is received with `mustUnderstand` enabled, then no fault is thrown.
- WS-Addressing headers received are validated to ensure:
  - Correct syntax
  - Correct number of elements
  - `wsa:Action` header in the SOAP header matches what is required by the operation
- Response messages returned to the client contain the required WS-Addressing headers.

Enabling WS-Addressing on the Web Service (Starting From Java)

To enable WS-Addressing on the web service starting from Java, use the `java.xml.ws.soap.Addressing` annotation on the web service class. Optionally, you can pass one or more of the Boolean attributes defined in Table 10-2.

Table 10-2 Attributes of the @Addressing Annotation

Attribute	Description
`enabled`	Specifies whether WS-Addressing is enabled. Valid values include `true` (enabled) and `false` (disabled). This attribute defaults to `true`.

Table 10-2 (Cont.) Attributes of the @Addressing Annotation

Attribute	Description
`required`	Specifies whether WS-Addressing rules are enforced for the inbound message. Valid values include `true` (enforced) and `false` (not enforced). If set to `false`, the inbound message is checked to see if WS-Addressing is enabled, and, if so, the rules are enforced. This attribute defaults to `false`.

Once enabled, the `wsaw:UsingAddressing` element is generated in the corresponding `wsdl:binding` element. For more information, see Enabling WS-Addressing on the Web Service (Starting from WSDL).

The following provides an example of how to enable WS-Addressing starting from Java. In this example, WS-Addressing is enforced on the endpoint (`required` is set to `true`).

Example 10-3 Enabling WS-Addressing on the Web Service (Starting From Java)

```
package examples;
import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;

@WebService(name="HelloWorld", serviceName="HelloWorldService")
@Addressing(enabled=true, required=false)

public class HelloWorld {
   public String sayHelloWorld(String message) throws MissingName { ... }
}
```

Enabling WS-Addressing on the Web Service (Starting from WSDL)

To enable WS-Addressing on the web service starting from WSDL, add the `wsaw:UsingAddressing` element to the corresponding `wsdl:binding` element. Optionally, you can add the `wsdl:required` Boolean attribute to specify whether WS-Addressing rules are enforced for the inbound message. By default, this attribute is `false`.

The following provides an example of how to enable WS-Addressing starting from WSDL. In this example, WS-Addressing is enforced on the endpoint (`wsdl:required` is set to `true`).

Example 10-4 Enabling WS-Addressing on the Web Service (Starting From WSDL)

```
...
  <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
    <wsaw:UsingAddressing wsdl:required="true" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>
    <operation name="sayHelloWorld">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
```

```
      <output>
        <soap:body use="literal"/>
      </output>
      <fault name="MissingName">
        <soap:fault name="MissingName" use="literal"/>
      </fault>
    </operation>
  </binding>
...
```

Enabling WS-Addressing on the Web Service Client

WS-Addressing can be enabled on the web service client implicitly or explicitly. Once enabled on the client:

- All WS-Addressing headers received on the client are understood. That is, if any WS-Addressing header is received with `mustUnderstand` enabled, then no fault is thrown.
- The JAX-WS runtime:
  - Maps all `wsaw:Action` elements, including `input`, `output`, and `fault` elements in the `wsdl:operation` to `javax.xml.ws.Action` and `javax.xml.ws.FaultAction` annotations in the generated service endpoint interface (SEI).
  - Generates `Action`, `To`, `MessageID`, and anonymous `ReplyTo` headers on the outbound request.

The following sections describe how to enable WS-Addressing on the web service client explicitly and implicitly, and how to disable WS-Addressing explicitly.

- Explicitly Enabling WS-Addressing on the Web Service Client
- Implicitly Enabling WS-Addressing on the Web Service Client
- Disabling WS-Addressing on the Web Service Client

Explicitly Enabling WS-Addressing on the Web Service Client

The web service client can enable WS-Addressing explicitly by passing `javax.xml.ws.soap.AddressingFeature` as an argument to the `getPort` or `createDispatch` methods on the `javax.xml.ws.Service` object. Optionally, you can pass one or more of the Boolean parameters defined in Table 10-3.

Table 10-3 Parameters of the AddressingFeature Feature

Parameter	Description
`enabled`	Specifies whether WS-Addressing is enabled for an outbound message. Valid values include `true` (enabled) and `false` (disabled). This attribute defaults to `true`.
`required`	Specifies whether WS-Addressing rules are enforced for the inbound messages. Valid values include `true` (enforced) and `false` (not enforced). If set to `false`, the inbound message is checked to see if WS-Addressing is enabled, and, if so, the rules are enforced. This attribute defaults to `false`.

The following shows an example of enabling WS-Addressing on a web service client when creating a web service proxy, by passing the `AddressingFeature` to the `getPort` method.

Example 10-5 Enabling WS-Addressing on a Web Service Client on the Web Service Proxy

```
package examples.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
import examples.client.MissingName_Exception;
import javax.xml.ws.soap.AddressingFeature;

public class Main {
  public static void main(String[] args) throws MissingName_Exception {
    HelloWorldService service;

    try {
      service = new HelloWorldService(new URL(args[0] + "?WSDL"),
                new QName("http://examples/", "HelloWorldService") );
    } catch (MalformedURLException murl) { throw new RuntimeException(murl); }

      HelloWorld port = service.getHelloWorldPort(
        new AddressingFeature(true, true));
...
  }
}
```

The following shows an example of enabling WS-Addressing on a web service client when creating a Dispatch instance, by passing the `AddressingFeature` to the `createDispatch` method.

Example 10-6 Enabling WS-Addressing on a Web Service Client on the Dispatch Instance

```
...
      HelloWorld port = service.getHelloWorldPort(new AddressingFeature(true));
...
```

Implicitly Enabling WS-Addressing on the Web Service Client

WS-Addressing is enabled implicitly if the `wsaw:UsingAddressing` extensibility element exists in the WSDL For more information, see Enabling WS-Addressing on the Web Service (Starting from WSDL).

Disabling WS-Addressing on the Web Service Client

A web service client may need to disable WS-Addressing processing explicitly, for example, if it has its own WS-Addressing processing module. For example, a Dispatch client in MESSAGE mode may be used to perform non-anonymous ReplyTo and FaultTo processing.

The following shows an example of how to disable explicitly WS-Addressing on a web service client when creating a web service proxy. In this example, the `AddressingFeature` feature is called with `enabled` set to `false`.

Example 10-7 Disabling WS-Addressing on a Web Service Client

```
...
new AddNumbersImplService().getAddNumbersImplPort(new
```

```
javax.xml.ws.AddressingFeature(false));
...
```

Associating WS-Addressing Action Properties

WS-Addressing defines an attribute, `wsaw:Action`, that can be used to explicitly associate WS-Addressing action message addressing properties with the web service. By default, an implicit action association is made when WS-Addressing is enabled and no action is explicitly associated with the web service.

The following sections describe how to associate WS-Addressing Action properties either explicitly or implicitly:

- Explicitly Associating WS-Addressing Action Properties (Starting from Java)
- Explicitly Associating WS-Addressing Action Properties (Starting from WSDL)
- Implicitly Associating WS-Addressing Action Properties

Explicitly Associating WS-Addressing Action Properties (Starting from Java)

To explicitly associate WS-Addressing action properties with the web service starting from Java, use the `javax.xml.ws.Action` and `javax.xml.ws.FaultAction` annotations.

Optionally, you can pass to the `@Action` annotation one or more of the attributes defined in Table 10-4.

Table 10-4 Attributes of the @Action Annotation

Attribute	Description
`input`	Associates Action message addressing property for the input message of the operation.
`output`	Associates Action message addressing property for the output message of the operation.
`fault`	Associates Action message addressing property for the fault message of the operation. Each exception that is mapped to a SOAP fault and requires explicit association must be specified using the `@FaultAction` annotation, as described in Table 10-5.

You can pass to the `@FaultAction` annotation one or more of the attributes defined in Table 10-4.

Table 10-5 Attributes of the @FaultAction Annotation

Attribute	Description
`className`	Name of the exception class. This attribute is required.
`value`	Value of the WS-Addressing Action message addressing property for the exception.

Once you explicitly associate the WS-Addressing action properties, the `wsaw:Action` attribute is generated in the corresponding `input`, `output`, and `fault` elements in the `wsdl:portType`

element. For more information, see Enabling WS-Addressing on the Web Service (Starting from WSDL).

The following provides an example of how to explicitly associate the WS-Addressing action message addressing properties for the input, output, and fault messages on the `sayHelloWorld` method, using the `@Action` and `@FaultAction` annotations.

Example 10-8 Example of Explicitly Associating an Action (Starting from Java)

```
...
@Action(
   input = "http://examples/HelloWorld/sayHelloWorldRequest",
   output = "http://examples/HelloWorld/sayHelloWorldResponse",
   fault = { @FaultAction(className = MissingName.class,
               value = "http://examples/MissingNameFault")})

   public String sayHelloWorld(String message) throws MissingName {
...
```

Once defined, the `wsaw:Action` element is generated in the corresponding `input`, `output`, and `fault` elements of the `wsdl:operation` element for the endpoint. For more information about these elements, see Explicitly Associating WS-Addressing Action Properties (Starting from WSDL).

Explicitly Associating WS-Addressing Action Properties (Starting from WSDL)

To explicitly associate WS-Addressing action properties with the web service starting from WSDL, add the `wsaw:Action` element to the corresponding `wsdl:binding` element. Optionally, you can add the `wsdl:required` Boolean attribute to specify whether WS-Addressing rules are enforced for the inbound message. By default, this attribute is `false`.

The following provides an example of how to, within the WSDL file, explicitly associate the WS-Addressing action message addressing properties for the `input`, `output`, and `fault` messages on the `sayHelloWorld` method of the HelloWorld endpoint.

Example 10-9 Example of Explicitly Associating an Action (Starting from WSDL)

```
...
  <portType name="HelloWorld">
    <operation name="sayHelloWorld">
      <input wsaw:Action="http://examples/HelloWorld/sayHelloWorldRequest"
       message="tns:sayHelloWorld"/>
      <output wsaw:Action="http://examples/HelloWorld/sayHelloWorldResponse"
       message="tns:sayHelloWorldResponse"/>
      <fault message="tns:MissingName" name="MissingName"
       wsaw:Action="http://examples/MissingNameFault"/>
    </operation>
  </portType>
...
```

Implicitly Associating WS-Addressing Action Properties

When WS-Addressing is enabled, if no explicit action is defined in the WSDL, the client sends an implicit `wsa:Action` header using the following formats:

- Input message action: *targetNamespace/portTypeName/inputName*
- Output message action: *targetNamespace/portTypeName/outputName*
- Fault message action: *targetNamespace/portTypeName/operationName*/Fault/*FaultName*

```
targetNamespace/portTypeName/[inputName | outputName]
```

For example, for the following WSDL excerpt:

```
<definitions targetNamespace="http://examples/"...>
...
  <portType name="HelloWorld">
    <operation name="sayHelloWorld">
      <input message="tns:sayHelloWorld" name="sayHelloRequest"/>
      <output message="tns:sayHelloWorldResponse" name="sayHelloResponse"/>
      <fault message="tns:MissingName" name="MissingName" />
    </operation>
  </portType>
...
</definitions>
```

The default input and output actions would be defined as follows:

- Input message action: `http://examples/HelloWorld/sayHelloRequest`
- Output message action: `http://examples/HelloWorld/sayHelloResponse`
- Fault message action: `http://examples/HelloWorld/sayHelloWorld/Fault/MissingName`

If the input or output message name is not specified in the WSDL, the operation name is used with Request or Response appended, respectively. For example: `sayHelloWorldRequest` or `sayHelloWorldResponse`.

Configuring Anonymous WS-Addressing

In some cases, the network technologies used in an environment (such as, firewalls, Dynamic Host Configuration Protocol (DHCP), and so on) may prohibit the use of globally addressed URI for a given endpoint. To enable non-addressable endpoints to exchange messages, the WS-Addressing specification supports "anonymous" endpoints using the `wsaw:Anonymous` element.

The `wsaw:Anonymous` element can be set to one of the values defined in Table 10-6.

Table 10-6 Valid Values for the wsaw:Anonymous Element

Value	Description
`optional`	The response endpoint EPR in a request message may contain an anonymous URI.
`required`	The response endpoint EPR in a request message must contain an anonymous URL. Otherwise, an `InvalidAddressingHeader` fault is returned.
`prohibited`	The response endpoint EPRs in a request message must not contain an anonymous URI. Otherwise, an `InvalidAddressingHeader` fault is returned.

To configure anonymous WS-Addressing:

1. Enable WS-Addressing on the web service, add the `wsaw:UsingAddressing` element to the corresponding `wsdl:binding` element. For more information, see Enabling WS-Addressing on the Web Service (Starting from WSDL).

> **Note:**
>
> When anonymous WS-Addressing is enabled, the `wsdl:required` attribute must not be enabled in the `wsaw:UsingAddressing` element.

2. Add the `wsaw:Anonymous` element to the `wsdl:operation` within the `wsdl:binding` element.

Example 10-10 Enabling Anonymous WS-Addressing on the Web Service

The following provides an example of how to enable anonymous WS-Addressing in the WSDL file. In this example, anonymous addressing is required.

```
...
  <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
    <wsaw:UsingAddressing wsdl:required="true" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>
    <operation name="sayHelloWorld">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <fault name="MissingName">
        <soap:fault name="MissingName" use="literal"/>
      </fault>
    </operation>
    <wsaw:Anonymous>required</wsaw:Anonymous>
  </binding>
...
```

11

Roadmap for Developing Asynchronous Web Service Clients

This chapter presents best practices for developing asynchronous WebLogic web service clients for Java API for XML Web Services (JAX-WS).
Table 11-1 provides best practices for developing asynchronous web service clients, and is followed by an example that illustrates the best practices presented. These guidelines should be used in conjunction with the general guidelines provided in Roadmap for Developing JAX-WS Web Service Clients.

For best practices when developing reliable web service clients, see Roadmap for Developing Reliable Web Services and Clients.

> **Note:**
>
> In the following table, *client instance* can be a port or a Dispatch instance.

Table 11-1 Roadmap for Developing Asynchronous Web Service Clients

Best Practice	Description
Define a port-based asynchronous callback handler, `AsyncClientHandlerFeature`, for asynchronous and dispatch callback handling.	Use of `AsyncClientHandlerFeature` is recommended as a best practice when using asynchronous invocation due to its scalability and ability to survive a JVM restart. It can be used by any client (survivable or not.) For information, see Developing the Asynchronous Handler Interface.
Define a singleton port instance and initialize it when the client container initializes (upon deployment).	Creation of the singleton port: • Triggers the asynchronous response endpoint to be published upon deployment. • Supports failure recovery by re-initializing the singleton port instance after VM restart. Within a cluster, initialization of a singleton port will ensure that all member servers in the cluster publish an asynchronous response endpoint.This ensures that the asynchronous response messages can be delivered to any member server and optionally forwarded to the correct server via in-place cluster routing. For complete details, see Clustering Considerations for Asynchronous Web Service Messaging.
If using Make Connection for clients behind a firewall, set the Make Connection polling interval to a value that is realistic for your scenario.	The Make Connection polling interval should be set as high as possible to avoid unnecessary polling overhead, but also low enough to allow responses to be retrieved in a timely fashion. A recommended value for the Make Connection polling interval is one-half of the expected average response time of the web service being invoked. For more information setting the Make Connection polling interval, see Configuring the Polling Interval. **Note**: This best practice is not demonstrated in Example 11-1.

Table 11-1 (Cont.) Roadmap for Developing Asynchronous Web Service Clients

Best Practice	Description
If using the JAX-WS Reference Implementation (RI), implement the `AsyncHandler<T>` interface.	Use of the `AsyncHandler<T>` interface is more efficient than the `Response<T>` interface. For more information and an example, see Using the JAX-WS Reference Implementation. **Note**: This best practice is not demonstrated in Example 11-1.
Define a Work Manager and set the thread pool minimum size constraint (`min-threads-constraint`) to a value that is at least as large as the expected number of concurrent requests or responses into the service.	For example, if a web service client issues 20 requests in rapid succession, the recommended thread pool minimum size constraint value would be 20 for the application hosting the client. If the configured constraint value is too small, performance can be severely degraded as incoming work waits for a free processing thread. For more information about the thread pool minimum size constraint, see Constraints in *Administering Server Environments for Oracle WebLogic Server*.

The following example illustrates best practices for developing asynchronous web service clients.

Example 11-1 Asynchronous Web Service Client Best Practices Example

```
import java.io.*;
import java.util.*;

import javax.servlet.*
import javax.xml.ws.*

import weblogic.jws.jaxws.client.ClientIdentityFeature;
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
import weblogic.jws.jaxws.client.async.AsyncClientTransportFeature;

import com.sun.xml.ws.developer.JAXWSProperties;

/**
 * Example client for invoking a web service asynchronously.
 */
public class BestPracticeAsyncClient
  extends GenericServlet {

  private static final String MY_PROPERTY = "MyProperty";

  private BackendServiceService _service;
  private WebServiceFeature[] _features;
  private BackendService _singletonPort;

  private static String _lastResponse;
  private static int _requestCount;

  @Override
  public void init()
    throws ServletException {

    // Only create the web service object once as it is expensive to create repeatedly.
    if (_service == null) {
      _service = new BackendServiceService();
    }
```

```
// Best Practice: Use a stored list of features, including client ID, to create client
// instances.
// Define all features for the web service client instance, including client ID, so that they
// are consistent each time the client instance is created. For example:
// _service.getBackendServicePort(_features);

List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

// Best Practice: Explicitly define the client ID.
ClientIdentityFeature clientIdFeature =
  new ClientIdentityFeature("MyBackendServiceAsyncClient");
features.add(clientIdFeature);

// Asynchronous endpoint
AsyncClientTransportFeature asyncFeature =
  new AsyncClientTransportFeature(getServletContext());
features.add(asyncFeature);

// Best Practice: Define a port-based asynchronous callback handler,
// AsyncClientHandlerFeature, for asynchronous and dispatch callback handling.
BackendServiceAsyncHandler handler =
  new BackendServiceAsyncHandler() {
    // This class is stateless and should not depend on
    // having member variables to work with across restarts.
    public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
      // ... Handle Response ...
      try {
        DoSomethingResponse response = res.get();
        res.getContext();
        _lastResponse = response.getReturn();
        System.out.println("Got async response: " + _lastResponse);
        // Retrieve the request property. This property can be used to
        // 'remember' the context of the request and subsequently process
        // the response.
        Map<String, Serializable> requestProps =
          (Map<String, Serializable>)
            res.getContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
        String myProperty = (String)requestProps.get(MY_PROPERTY);
        System.out.println("Got MyProperty value propagated from request: "+
                           myProperty);
      } catch (Exception e) {
        _lastResponse = e.toString();
        e.printStackTrace();
      }
    }
  };
AsyncClientHandlerFeature handlerFeature =
  new AsyncClientHandlerFeature(handler);
features.add(handlerFeature);

// Set the features used when creating clients with
// the client ID "MyBackendServiceAsyncClient".

_features = features.toArray(new WebServiceFeature[features.size()]);

// Best Practice: Define a singleton port instance and initialize it when
// the client container initializes (upon deployment).
// The singleton port will be available for the life of the servlet.
// Creation of the singleton port triggers the asynchronous response endpoint to be published
// and it will remain published until our container (Web application) is undeployed.
// Note, the destroy() method will be called before this.
```

```
  // The singleton port ensures proper/robust operation in both
  // recovery and clustered scenarios.
  _singletonPort = _service.getBackendServicePort(_features);
}

@Override
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException {

  // TODO: ... Read the servlet request ...

  // For this simple example, echo the _lastResponse captured from
  // an asynchronous DoSomethingResponse response message.

  if (_lastResponse != null) {
    res.getWriter().write(_lastResponse);
    _lastResponse = null; // Clear the response so we can get another
    return;
  }

  // Set _lastResponse to NULL to to support the invocation against
  // BackendService to generate a new response.

  // Best Practice: Synchronize use of client instances.
  // Create another client instance using the *exact* same features used when creating _
  // singletonPort. Note, this port uses the same client ID as the singleton port
  // and it is effectively the same as the singleton
  // from the perspective of the web services runtime.
  // This port will use the asynchronous response endpoint for the client ID,
  // as it is defined in the _features list.
  BackendService anotherPort =
    _service.getBackendServicePort(_features);

  // Set the endpoint address for BackendService.
  ((BindingProvider)anotherPort).getRequestContext().
    put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        "http://localhost:7001/BestPracticeService/BackendService");

  // Add a persistent context property that will be retrieved on the
  // response. This property can be used as a reminder of the context of this
  // request and subsequently process the response. This property will *not*
  // be passed over the wire, so the properties can change independent of the
  // application message.
  Map<String, Serializable> persistentContext =
    (Map<String, Serializable>)((BindingProvider)anotherPort).
      getRequestContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
  String myProperty = "Request " + (++_requestCount);
  persistentContext.put(MY_PROPERTY, myProperty);
  System.out.println("Request being made with MyProperty value: " +
                     myProperty);

  // Make the asychronous invocation. The asynchronous handler implementation (set
  // into the AsyncClientHandlerFeature above) receives the response.
  String request = "Dance and sing";
  System.out.println("Invoking DoSomething asynchronously with request: " +
                     request);
  anotherPort.doSomethingAsync(request);

  // Return a canned string indicating the response was not received
  // synchronously. Client will need to invoke the servlet again to get
  // the response.
```

```
    res.getWriter().write("Waiting for response...");

    // Best Practice: Explicitly close client instances when processing is complete.
    // If not closed explicitly, the port will be closed automatically when it goes out of scope.
    ((java.io.Closeable)anotherPort).close();
  }

  @Override
  public void destroy() {

    try {
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the singleton port created during initialization. Note, the asynchronous
      // response endpoint generated by creating _singletonPort *remains*
      // published until our container (Web application) is undeployed.
      ((java.io.Closeable)_singletonPort).close();

      // Upon return, the Web application is undeployed, and the asynchronous
      // response endpoint is stopped (unpublished). At this point,
      // the client ID used for _singletonPort will be unregistered and will no longer be
      // visible from the Administration Console and WLST.
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

12

Developing Asynchronous Clients

This chapter describes how to develop asynchronous WebLogic web service clients using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

> **Note:**
>
> See also Roadmap for Developing Asynchronous Web Service Clients.

Overview of Asynchronous Web Service Invocation

To support asynchronous web services invocation, WebLogic web services can use an asynchronous client programming model, asynchronous transport, or both.

Table 12-1 provides a description and key benefits of the asynchronous client programming model and transport types, and introduces the configuration options available to support asynchronous web service invocation.

> **Note:**
>
> The method of generating a WSDL for the asynchronous web service containing two one-way operations defined as two portTypes—one for the asynchronous operation and one for the callback operation—is not supported in the current release.

Table 12-1 Support for Asynchronous Web Service Invocation

Type	Description	Benefits
Client programming model	Describes the invocation semantics used to call a web service operation: synchronous or asynchronous. 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. By calling a web service *asynchronously*, the client can continue its processing, without interruption, and be notified when the asynchronous response is returned. To support asynchronous invocation, you generate automatically an asynchronous flavor of each operation on a web service port using the `clientgen` Ant task, as described later in Building the Client Artifacts for Asynchronous Web Service Invocation. Then, you add methods in your client, including your business logic, that handle the asynchronous response or failures when it returns later on. Finally, to invoke a web service asynchronously, rather than invoking the operation directly, you invoke the asynchronous flavor of the operation. For example, rather than invoking an operation called `addNumbers` directly, you would invoke `addNumbersAsync` instead.	Asynchronous invocation enables web service clients to initiate a request to a web service, continue processing without blocking, and receive the response at some point in the future.
Transport	There are three transport types: asynchronous client transport, Make Connection transport, and synchronous transport. For a comparison of each transport type, see Table 12-2.	Asynchronous client transport and Make Connection transport deliver the following key benefits: • Improves fault tolerance in the event of network outages. • Enables servers to absorb more efficiently spikes in traffic.
Configuration	Configure web service persistence and buffering (optional) to support asynchronous web service invocation. For more information, see Configuring Your Servers for Asynchronous Web Service Invocation.	Benefits of configuring the web service features include: • Persistence supports long running requests and provides the ability to survive server restarts. • Buffering enables all requests to a web service to be handled asynchronously.

Table 12-2 summarizes the transport types that WebLogic Server supports for invoking a web service asynchronously (or synchronously, if configured) from a web service client.

Table 12-2 Transport Types for Invoking Web Services Asynchronously

Transport Types	Description
Asynchronous Client Transport	Provides a scalable asynchronous client programming model through the use of an *addressable* client-side asynchronous response endpoint and WS-Addressing. Asynchronous client transport decouples the delivery of the response message from the initiating transport request used to send the request message. The response message is sent to the asynchronous response endpoint using a new connection originating from the web service. The client correlates request and response messages through WS-Addressing headers. Asynchronous client transport provides improved fault tolerance and enables servers to better absorb spikes in server load. For details about using asynchronous client transport, see Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport). Asynchronous client transport supports the following programming models: • Asynchronous and dispatch callback handling using one of the following methods: - Port-based asynchronous callback handler, `AsyncClientHandlerFeature`, described in Developing the Asynchronous Handler Interface. This is recommended as a **best practice** when using asynchronous invocation due to its scalability and ability to survive a JVM restart. - Per-request asynchronous callback handler, as described in Using the JAX-WS Reference Implementation. • Asynchronous polling, as described in Using the JAX-WS Reference Implementation. • Synchronous invocation by enabling a flag, as described in Configuring Asynchronous Client Transport for Synchronous Operations.

Table 12-2 (Cont.) Transport Types for Invoking Web Services Asynchronously

Transport Types	Description
Make Connection Transport	Enables asynchronous web service invocation from behind a firewall using Web Services Make Connection 1.1 or 1.0. Make Connection is a client polling mechanism that provides an alternative to asynchronous client transport. As with asynchronous client transport, Make Connection enables the decoupling of the response message from the initiating transport request used to send the request message. However, unlike asynchronous client transport which requires an addressable asynchronous response endpoint to forward the response to, with Make Connection typically the sender of the request message is *non-addressable* and unable to accept an incoming connection. For example, when the sender is located behind a firewall. Make Connection transport provides improved fault tolerance and enables servers to better absorb spikes in server load. For details about Make Connection transport, see Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection). Make Connection transport is recommended as a best practice when using asynchronous invocation from behind a firewall due to its scalability and ability to survive a JVM restart. It supports the following programming models: • Asynchronous and dispatch callback handling using one of the following methods: - Port-based asynchronous callback handler, `AsyncClientHandlerFeature`, described in Developing the Asynchronous Handler Interface. - Per-request asynchronous callback handler, as described in Using the JAX-WS Reference Implementation. • Asynchronous polling, as described in Using the JAX-WS Reference Implementation. • Synchronous invocation by enabling a flag, as described in Configuring Make Connection as the Transport for Synchronous Methods. Use of Make Connection transport with `AsyncClientHandlerFeature` is recommended as a best practice when using asynchronous invocation due to its scalability and ability to survive a JVM restart.
Synchronous Transport	Provides support for synchronous and asynchronous web service invocation with very limited support for WS-Addressing. For details, see Using the JAX-WS Reference Implementation. Synchronous transport is recommended when using synchronous invocation. It can be used for asynchronous invocation, as well, though this is not considered a best practice. It supports the following programming models: • Asynchronous and dispatch callback handling on a per request basis using the standard JAX-WS RI implementation, described in Using the JAX-WS Reference Implementation. • Asynchronous polling, as described in Using the JAX-WS Reference Implementation. • Synchronous invocation.

Steps to Invoke Web Services Asynchronously

This section describes the steps required to invoke web services asynchronously.

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 Developing JAX-WS Web Services.

Table 12-3 Steps to Invoke Web Services Asynchronously

#	Step	Description
1	Configure web service persistence to support asynchronous web service invocation.	Configure web service persistence on the servers hosting the web service and client to retain context information required for processing a message at the web service or client. For more information, see Configuring Your Servers for Asynchronous Web Service Invocation. **Note**: This step is not required if you are programming the web service client using the standard JAX-WS RI implementation and synchronous transport (in Step 3), as described in Using the JAX-WS Reference Implementation.
2	Configure web service buffering to enable the web service to process requests asynchronously. (Optional)	This step is optional. To configure the web service to process requests asynchronously, configure buffering on the server hosting the web service. Buffering enables you to store messages in a JMS queue for asynchronous processing by the web service. For more information, see Configuring Your Servers for Asynchronous Web Service Invocation.
3	Build the client artifacts required for asynchronous invocation.	To generate asynchronous polling and asynchronous callback handler methods in the service endpoint interface, create an external binding declarations that enables asynchronous mappings and pass the bindings file as an argument to the `clientgen` when compiling the client. See Building the Client Artifacts for Asynchronous Web Service Invocation.
4	Implement the web service client based on the transport and programming model required.	Refer to one of the following sections based on the transport and programming model required: • Use asynchronous client transport, as described in Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport). (Recommended as a best practice.) • Enable asynchronous access from behind a firewall using Make Connection. See Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection). • Implement standard JAX-WS programming models, such as asynchronous polling or per-request asynchronous callback handling, using synchronous transport. See Using the JAX-WS Reference Implementation. When using web services in a cluster, review the guidelines described in Clustering Considerations for Asynchronous Web Service Messaging.
5	Compile the web service client and package the client artifacts.	For more information, see Compiling and Running the Client Application.
6	Deploy the web service client.	See Deploying and Undeploying WebLogic Web Services.

Table 12-3 (Cont.) Steps to Invoke Web Services Asynchronously

#	Step	Description
7	Monitor the web service client.	You can monitor runtime information for clients that invoke web services asynchronously, such as number of invocations, errors, faults, and so on, using the WebLogic Server Administration Console or WLST. See Monitoring Asynchronous Web Service Invocation.

Configuring Your Servers for Asynchronous Web Service Invocation

> **Note:**
>
> This step is not required if you are programming the web service client using the standard JAX-WS RI implementation and synchronous transport, as described in Using the JAX-WS Reference Implementation.

To support asynchronous web service invocation, you need to configure the features defined in the following table on the servers to which the web service and client are deployed.

Table 12-4 Configuration for Asynchronous Web Service Invocation

Feature	Description
Persistence	Web service persistence is used to save the following types of information: • Client identity and properties • SOAP message, including its headers and body • Context properties required for processing the message at the web service or client (for both asynchronous and synchronous messages) The Make Connection transport protocol makes use of web service persistence as follows: • Web service persistence configured on the MC Receiver (web service) persists response messages that are awaiting incoming Make Connection messages for the Make Connection anonymous URI to which they are destined. Messages are persisted until either they are returned as a response to an incoming Make Connection message or the message reaches the maximum lifetime for a persistent store object, resulting in the message being cleaned from the store. • web service persistence configured on the MC Initiator (web service client) is used with the asynchronous client handler feature to recover after a VM restart. You can configure web service persistence using the Configuration Wizard to extend the WebLogic Server domain using a web services-specific extension template. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Server Administration Console or WLST. For information about configuring web service persistence, see Configuring Web Service Persistence for Web Service Clients. For information about the APIs available for persisting client and message information, see Propagating Request Context to the Response.
Message buffering	When a buffered operation is invoked by a client, the request is stored in a JMS queue and WebLogic Server processes it asynchronously. If WebLogic Server goes down while the request is still in the queue, it will be processed as soon as WebLogic Server is restarted. Message buffering is configured on the server hosting the web service. For configuration information, see Configuring Message Buffering for Web Services. **Note**: Message buffering is enabled automatically on the web service client.

Building the Client Artifacts for Asynchronous Web Service Invocation

Using the WebLogic Server client-side tooling (for example, `clientgen`), you can generate automatically the client artifacts required for asynchronous web service invocation. Specifically, the following artifacts are generated:

- Service endpoint interfaces for invoking the web service asynchronously with or without a per-request asynchronous callback handler. For example, if the web service defined the following method:

  ```
  public int addNumbers(int opA, int opB) throws MyException
  ```

  Then the following methods will be generated:

  ```
  public Future<?> addNumbersAsync(int opA, int opB,
                   AsyncHandler<AddNumbersResponse>)
  public Response<AddNumbersResponse> addNumbersAsync(int opA, int opB)
  ```

- Asynchronous handler interface for implementing a handler and setting it on the port using `AsyncClientHandlerFeature`. The asynchronous handler interface is named as follows: *`portInterfaceName`*`AsyncHandler`, where *`portInterfaceName`* specifies the name of the port interface.

  For example, for a web service with a port type name `AddNumbersPortType`, an asynchronous handler interface named `AddNumbersPortTypeAsyncHandler` is generated with the following method:

  ```
  public void onAddNumbersResponse(Response<AddNumbersResponse>)
  ```

  The `AsyncClientHandlerFeature` is described later, in Developing the Asynchronous Handler Interface.

To generate asynchronous client artifacts in the service endpoint interface when the WSDL is compiled, enable the `jaxws:enableAsyncMapping` binding declaration in the WSDL file.

Alternatively, you can create an external binding declarations file that contains all binding declarations for a specific WSDL or XML Schema document. Then, pass the binding declarations file to the `<binding>` child element of the `wsdlc`, `jwsc`, or `clientgen` Ant task. For more information, see Creating an External Binding Declarations File Using JAX-WS Binding Declarations.

The following provides an example of a binding declarations file (`jaxws-binding.xml`) that enables the `jaxws:enableAsyncMapping` binding declaration:

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

Then, to update the `build.xml` file to generate client artifacts necessary to invoke a web service operation asynchronously:

1. Use the `taskdef` Ant task to define the full classname of the `clientgen` Ant tasks.
2. Add a target that includes a reference to the external binding declarations file containing the asynchronous binding declaration, as defined above. In this case, the `clientgen` Ant task generates both synchronous and asynchronous flavors of the web service operations in the JAX-WS stubs.

For example:

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

<target name="build_client">

<clientgen
      type="JAXWS"
      wsdl="AddNumbers.wsdl"
      destDir="${clientclasses.dir}"
      packageName="examples.webservices.async.client">
      <binding file="jaxws-binding.xml" />
    </clientgen>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/async/client/**/*.java"/>

</target>
```

Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)

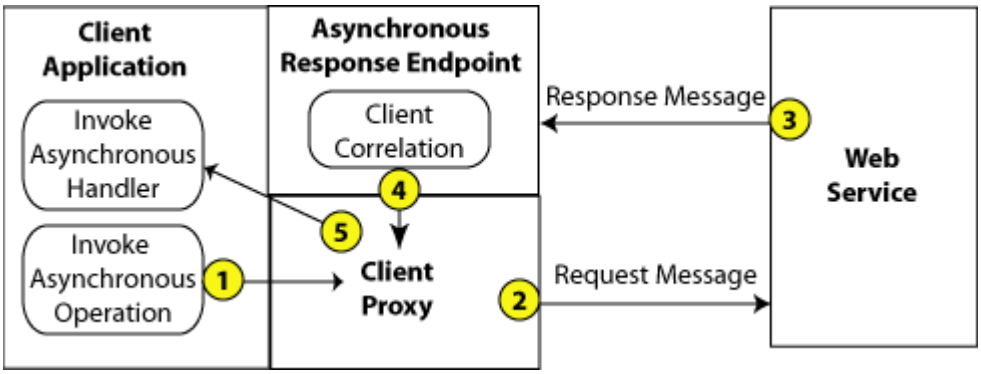
The asynchronous client transport feature provides a scalable asynchronous client programming model. Specifically, this feature:

- Publishes a client-side asynchronous response endpoint, shown in Figure 12-1.
- Creates and publishes a service implementation that invokes the requested asynchronous handler implementation.
- Automatically adds WS-Addressing non-anonymous ReplyTo headers to all non-one-way, outbound messages. This header references the published response endpoint.
- Correlates asynchronous request and response messages using the facilities listed above.

When the asynchronous client transport feature is enabled, all other JAX-WS client programming models (such as asynchronous polling, callback handler, dispatch, and so on) continue to be supported. Synchronous web service operations will, by default, use synchronous transport, unless explicitly configured to use asynchronous client transport feature when enabling the feature.

The following figure shows the message flow used by the asynchronous client transport feature.

Figure 12-1 Asynchronous Client Transport Feature



As shown in the previous figure:

1. The client enables the asynchronous client transport feature on the client proxy and invokes an asynchronous web service operation.
2. The web service operation is invoked via the client proxy.
3. The web service processes the request and sends a response message (at some time in the future) back to the client. The response message is sent to the client's asynchronous response endpoint. The address of the asynchronous response endpoint is maintained in the WS-Addressing headers.
4. The response message is forwarded to the appropriate client via the client proxy.
5. The client asynchronous handler is invoked to handle the response message.

The following sections describe how to develop scalable asynchronous JAX-WS clients using asynchronous client transport:

- Enabling and Configuring the Asynchronous Client Transport Feature
- Developing the Asynchronous Handler Interface
- Propagating User-defined Request Context to the Response

Enabling and Configuring the Asynchronous Client Transport Feature

> **Note:**
>
> The Make Connection and asynchronous client transport features are mutually exclusive. If you attempt to enable both features on the same web service client, an error is returned. For more information about Make Connection, see Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection).

To enable the asynchronous client transport feature on a client, pass an instance of `weblogic.jws.jaxws.client.async.AsyncClientTransportFeature` as a parameter when creating the web service proxy or dispatch.

The asynchronous response endpoint described by the `AsyncClientTransportFeature` is used by all client instances that share the same client ID and is in effect from the time the first client instance using the client ID is published. The asynchronous response endpoint remains published until the client ID is explicitly disposed or the container for the client is deactivated (for example, the host Web application or EJB is undeployed). For more information about managing the client ID, see Managing Client Identity.

The asynchronous response endpoint address is generated automatically using the following format:

`http://`*`contextAddress`*`:`*`port`*`/`*`context`*`/`*`targetPort`*`-AsyncResponse`

In the above:

- *`contextAddress`*`:`*`port`*—Specifies one of the following:
  - If clustered application, cluster address and port.
  - If not clustered application, default WebLogic Server address and port for the selected protocol.
  - If no default address is defined, first network channel address for the given protocol. For more information about network channels, see Configuring Network Resources in *Administering Server Environments for Oracle WebLogic Server*.
- *`context`*—Current servlet context, if running within an existing context. Otherwise, a new context named by the UUID and scoped to the application.
- *`targetPort`*`-AsyncResponse`—Port name of the service accessed by the client appended by `-AsyncResponse`.

You can configure the asynchronous client transport feature, as described in the following sections:

- Configuring the Address of the Asynchronous Response Endpoint
- Configuring the ReplyTo and FaultTo Headers of the Asynchronous Response Endpoint
- Configuring the Context Path of the Asynchronous Response Endpoint
- Publishing the Asynchronous Response Endpoint
- Configuring Asynchronous Client Transport for Synchronous Operations

For more information about the constructor formats, see AsyncClientTransportFeature() in *Java API Reference for Oracle WebLogic Server*.

Configuring the Address of the Asynchronous Response Endpoint

You can configure an address for the asynchronous response endpoint by passing it as an argument to the `AsyncClientTransportFeature`, as follows:

```
String responseAddress = "http://myserver.com:7001/myReliableService/myClientCallback";
AsyncClientTransportFeature asyncFeature = new AsyncClientTransportFeature(responseAddress);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

The specified address must be a legal address for the server or cluster (including the network channels or proxy addresses). Ephemeral ports are not supported. The specified context must be scoped within the current application or refer to an unused context; it cannot refer to a context that is scoped to another deployed application, otherwise an error is thrown.

The following tables summarizes the constructors that can be used to configure the address of the asynchronous response endpoint.

Table 12-5 Constructors for Configuring the Address of the Asynchronous Response Endpoint

Constructor	Description
`AsyncClientTransportFeature(java.lang.String address)`	Configures the address of the asynchronous response endpoint.
`AsyncClientTransportFeature(java.lang.String address, boolean doPublish)`	Configures the following: • Address of the asynchronous response endpoint. • Whether to publish the endpoint at the specified address. For more information, see Publishing the Asynchronous Response Endpoint.
`AsyncClientTransportFeature(java.lang.String address, boolean doPublish, boolean useAsyncWithSyncInvoke)`	Configures the following: • Address of the asynchronous response endpoint. • Whether to publish the endpoint at the specified address. For more information, see Publishing the Asynchronous Response Endpoint. • Whether to enable asynchronous client transport for synchronous operations. For more information, see Configuring Asynchronous Client Transport for Synchronous Operations.

Configuring the ReplyTo and FaultTo Headers of the Asynchronous Response Endpoint

You can configure the address to use for all outgoing ReplyTo and FaultTo headers of type `javax.xml.ws.wsaddressing.W3CEndpointReference` for the asynchronous response endpoint by passing them as arguments to the `AsyncClientTransportFeature`.

For example, to configure only the ReplyTo header address:

```
W3CEndpointReference replyToAddress = "http://myserver.com:7001/myReliableService/
myClientCallback";
AsyncClientTransportFeature asyncFeature = new AsyncClientTransportFeature(replyToAddress);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

To configure both the ReplyTo and FaultTo header addresses:

```
W3CEndpointReference replyToAddress = "http://myserver.com:7001/myReliableService/
myClientCallback";
W3CEndpointReference faultToAddress = "http://myserver.com:7001/myReliableService/FaultTo";
AsyncClientTransportFeature asyncFeature = new AsyncClientTransportFeature(replyToAddress,
faultToAddress);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

The following tables summarizes the constructors that can be used to configure the endpoint reference address for the outgoing ReplyTo and FaultTo headers.

Table 12-6 Constructors for Configuring the ReplyTo and FaultTo Headers

Constructor	Description
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference replyTo)`	Configures the endpoint reference address for the outgoing ReplyTo headers.
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference replyTo, boolean doPublish)`	Configures the following: • Endpoint reference address for the outgoing ReplyTo headers. • Whether to publish the endpoint at the specified address. For more information, see Publishing the Asynchronous Response Endpoint.
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference replyTo, boolean doPublish, boolean useAsyncWithSyncInvoke)`	Configures the following: • Endpoint reference address for the outgoing ReplyTo headers. • Whether to publish the endpoint at the specified address. For more information, see Publishing the Asynchronous Response Endpoint. • Whether to enable asynchronous client transport for synchronous operations. For more information, see Configuring Asynchronous Client Transport for Synchronous Operations.
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference replyTo, javax.xml.ws.wsaddressing.W3CEndpointReference faultTo)`	Configures the endpoint reference address for the outgoing ReplyTo and FaultTo headers
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference replyTo, javax.xml.ws.wsaddressing.W3CEndpointReference faultTo, boolean doPublish)`	Configures the following: • Endpoint reference address for the outgoing ReplyTo and FaultTo headers. • Whether to publish the endpoint at the specified address. For more information, see Publishing the Asynchronous Response Endpoint.
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference replyTo, javax.xml.ws.wsaddressing.W3CEndpointReference faultTo, boolean doPublish, boolean useAsyncWithSyncInvoke)`	Configures the following: • Endpoint reference address for the outgoing ReplyTo and FaultTo headers. • Whether to publish the endpoint at the specified address. For more information, see Publishing the Asynchronous Response Endpoint. • Whether to enable asynchronous client transport for synchronous operations. For more information, see Configuring Asynchronous Client Transport for Synchronous Operations.

Configuring the Context Path of the Asynchronous Response Endpoint

When a client is running within a servlet or Web application-based web service, it can use its ServletContext and context path to construct the asynchronous response endpoint. You pass the information as an argument to the `AsyncClientTransportFeature`, as follows:

- When running inside a servlet:

```
AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(getServletContext());
```

- When running inside a web service or an EJB-based web service:

```
import com.sun.xml.ws.api.server.Container;
...
Container c = ContainerResolver.getInstance().getContainer();
ServletContext servletContext = c.getSPI(ServletContext.class);
AsyncClientTransportFeature asyncFeature =
    new AsyncClientTransportFeature(servletContext);
```

The specified context must be scoped within the current application or refer to an unused context; it cannot refer to a context that is scoped to another deployed application.

> **Note:**
>
> When you use the empty constructor for `AsyncClientTransportFeature`, the web services runtime attempts to discover the container in which the current feature was instantiated and publish the endpoint using any available container context.

The following tables summarizes the constructors that can be used to configure the context path of the asynchronous response endpoint.

Table 12-7 Constructors for Configuring the Context Path of the Asynchronous Response Endpoint

Constructor	Description
`AsyncClientTransportFeature(java.lang.Object context)`	Configures the context path of the asynchronous response endpoint.
`AsyncClientTransportFeature(java.lang.Object context, boolean useAsyncWithSyncInvoke)`	Configures the following: • Context path of the asynchronous response endpoint. • Whether to enable asynchronous client transport for synchronous operations. For more information, see Configuring Asynchronous Client Transport for Synchronous Operations.

Publishing the Asynchronous Response Endpoint

You can configure whether to publish the asynchronous response endpoint by passing the `doPublish` boolean value as an argument to `AsycnClientTransportFeature()` when configuring the following properties:

- Address of the asynchronous response endpoint. See Table 12-5.
- ReplyTo and FaultTo headers. See Table 12-6.
- Context path of the asynchronous response endpoint. See Table 12-7.

If `doPublish` is set to false, then the asynchronous response endpoint is not published automatically, but WS-Addressing headers will be added to outbound non-one-way messages. This scenario supports the following programming models:

- Asynchronous polling (with no attempt to access the Response object)
- Dispatch asynchronous polling (with no attempt to access the Response object)

- Dispatch one-way invocation
- Synchronous invocation using synchronous transport option (default)

For all other asynchronous programming models, the availability of a asynchronous response endpoint is required and the web service client is responsible for publishing it prior to making outbound requests if `doPublish` is set to false.

The following example configures the asynchronous response endpoint address and publishes the asynchronous response endpoint:

```
String responseAddress = "http://localhost:7001/myReliableService/myReliableResponseEndpoint";
boolean doPublish = true;
AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(responseAddress, doPublish);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

Configuring Asynchronous Client Transport for Synchronous Operations

You can enable or disable asynchronous client transport for synchronous operations using the `useAsyncWithSyncInvoke` boolean flag when configuring the following properties:

- Address of the asynchronous response endpoint. See Table 12-5.
- ReplyTo and FaultTo headers. See Table 12-6.
- Context path of the asynchronous response endpoint. See Table 12-7.

The following example configures the asynchronous response endpoint address and enables use of asynchronous client transport for synchronous operations:

```
String responseAddress = "http://localhost:7001/myReliableService/myReliableResponseEndpoint";
boolean useAsyncWithSyncInvoke = true;
AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(responseAddress, useAsyncWithSyncInvoke);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

Developing the Asynchronous Handler Interface

> **Note:**
>
> If you set a single asynchronous handler instance on the port, as described in this section, and subsequently attempt to configure a per-request asynchronous handler, as described in Using the JAX-WS Reference Implementation, then a runtime exception is returned.

As described in Building the Client Artifacts for Asynchronous Web Service Invocation, the asynchronous handler interface, `weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature`, sets a single asynchronous handler instance on the port rather than on a per-request basis.

For example, when you build the client classes using clientgen, as described in Building the Client Artifacts for Asynchronous Web Service Invocation, the asynchronous handler interface is generated, as shown below.

Example 12-1 Example of the Asynchronous Handler Interface

```
import javax.xml.ws.Response;


/**
 * This class was generated by the JAX-WS RI.
 * Oracle JAX-WS 2.1.5
 * Generated source version: 2.1
 *
 */
public interface BackendServiceAsyncHandler {


    /**
     *
     * @param response
     */
    public void onDoSomethingResponse(Response<DoSomethingResponse> response);

}
```

The asynchronous handler interface is generated as part of the same package as the port interface and represents the methods required to accept responses for any operation defined on the service. You can import and implement this interface in your client code to provide a way to receive and process asynchronous responses in a strongly-typed manner.

To set a single asynchronous handler instance on the port, pass an instance of the `weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature` as a parameter when creating the web service proxy or dispatch. You specify the name of the asynchronous handler that will be invoked when a response message is received.

The following example shows how to develop an asynchronous handler interface. The example demonstrates how to initialize the `AsyncClientHandlerFeature` to connect the asynchronous handler implementation to the port used to make invocations on the backend service. This example is excerpted from Example 11-1.

Example 12-2 Example of Developing the Asynchronous Handler Interface

```
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
...
    BackendServiceAsyncHandler handler = new BackendServiceAsyncHandler() {
        public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
            // ... Handle Response ...
            try {
                DoSomethingResponse response = res.get();
                _lastResponse = response.getReturn();
                System.out.println("Got async response: " + _lastResponse);
            } catch (Exception e) {
                _lastResponse = e.toString();
                e.printStackTrace();
              }
        }
      };
    AsyncClientHandlerFeature handlerFeature = new AsyncClientHandlerFeature(handler);
    features.add(handlerFeature);
    _features = features.toArray(new WebServiceFeature[features.size()]);
    BackendService anotherPort = _service.getBackendServicePort(_features);
...
    // Make the invocation. Our asynchronous handler implementation (set
```

```
// into the AsyncClientHandlerFeature above) receives the response.
String request = "Dance and sing";
System.out.println("Invoking DoSomething asynchronously with request: " + request);
anotherPort.doSomethingAsync(request);
```

Propagating User-defined Request Context to the Response

The `weblogic.wsee.jaxws.JAXWSProperties` API defines the following properties that enables users to propagate user-defined request context information to the response message, without relying on the asynchronous handler instance state.

The asynchronous handler instance may be created at any time; for example, if the client's server goes down and is restarted. Therefore, storing request context in the asynchronous handler interface will not be useful.

The `JAXWSProperties` properties are defined in the following table.

Table 12-8 Properties Supported by the JAXWSProperties API

This property . . .	Specifies . . .
`MESSAGE_ID`	Message ID for the request. The client can set this property on the request context to override the auto-generation of the per-request Message ID header.
`PERSISTENT_CONTEXT`	Context properties required by the client or the communication channels. Web service clients can persist context properties, as long as they are Serializable, for the request message. These properties will be available in the response context map available from the `Response` object when the asynchronous handler is invoked. For more information, see Propagating Request Context to the Response.
`RELATES_TO`	Message ID to which the response correlates.
`REQUEST_TIMEOUT`	For synchronous operations using asynchronous client transport, maximum amount of time to block and wait for a response. This property default to 0 indicating no timeout.

In addition, web service clients can persist context properties, as long as they are Serializable, for the request message. Context properties can include those required by the client or the communication channels. Message properties can be stored as part of the `weblogic.wsee.jaxws.JAXWSProperties.PERSISTENT_CONTEXT` Map property and retrieved after the response message is returned. For complete details, see Propagating Request Context to the Response.

Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)

Web Services Make Connection is a client polling mechanism that provides an alternative to asynchronous client transport, typically to provide support for clients that are behind a firewall. WebLogic Server supports WS-MakeConnection version 1.1, as described in the Make Connection specification at: `http://docs.oasis-open.org/ws-rx/wsmc/200702`, and is backwards compatible with version 1.0.

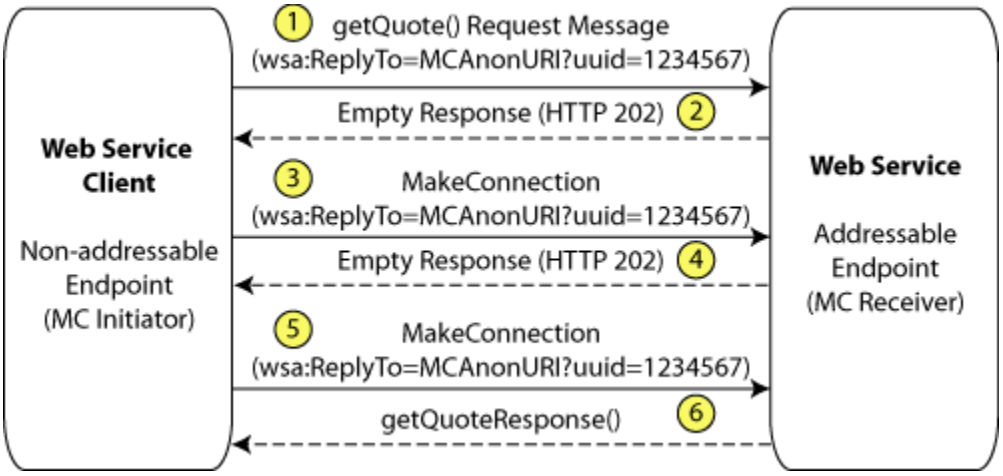
Specifically, Make Connection:

- Enables the decoupling of the response message from the initiating transport request used to send the request message (similar to asynchronous client transport).

- Supports web service clients that are non-addressable and unable to accept an incoming connection (for example, clients behind a firewall).
- Enables a web service client to act as an MC-Initiator and the web service to act as an MC-Receiver, as defined by the WS-MakeConnection specification.

The following figure, borrowed from the Web Services Make Connection specification, shows a typical Make Connection message flow.

Figure 12-2 Make Connection Message Flow



As shown in the previous figure, the Make Connection message flow is as follows:

1. The `getQuote()` request message is sent from the web service client (MC Initiator) to the web service (MC Receiver). The ReplyTo header contains a Make Connection anonymous URI that specifies the UUID for the MC Initiator.

   The MC Receiver receives the `getQuote()` message. The presence of the Make Connection anonymous URI in the ReplyTo header indicates that the response message can be sent back on the connection's back channel or the client will use Make Connection polling to retrieve it.

2. The MC Receiver closes the connection by sending back an empty response (HTTP 202) to the MC Initiator.

   Upon receiving an empty response, the MC Initiator initializes and starts its polling mechanism to enable subsequent polls to be sent to the MC Receiver. Specifically, the MC Initiator polling mechanism starts a timer with expiration set to the interval configured for the time between successive polls.

3. Upon timer expiration, the MC Initiator sends a Make Connection message to the MC Receiver with the same Make Connection anonymous URI information in its message.

4. As the MC Receiver has not completed process the getQuote() operation, no response is available to send back to the MC Initiator. As a result, the MC Receiver closes the connection by sending back another empty response (HTTP 202) indicating that no responses are available at this time.

   Upon receipt of the empty message, the MC Initiator restarts the timer for the Make Connection polling mechanism.

Before the timer expires, the `getQuote()` operation completes. Since the original request contained a Make Connection anonymous URI in its ReplyTo header, the MC Receiver stores the response and waits to receive the next Make Connection message with a matching address.

5. Upon timer expiration, the MC Initiator sends a Make Connection message to the MC Receiver with the same Make Connection anonymous URI information in its message.
6. Upon receipt of the Make Connection message, the MC Receiver retrieves the stored response message and sends it as a response to the received Make Connection message.

   The MC Initiator receives the response message and terminates the Make Connection polling mechanism.

Make Connection transport is recommended when using asynchronous invocation from behind a firewall. For a list of programming models supported, see Table 12-2.

The following sections describe how to enable and configure Make Connection on a web service and client:

- Enabling and Configuring Make Connection on a Web Service
- Enabling and Configuring Make Connection on a Web Service Client

Enabling and Configuring Make Connection on a Web Service

Make Connection can be enabled by attaching a Make Connection policy assertion to the web service and then calling its methods from a client using the standard JAX-WS client APIs. A policy can be attached to a web service in one of the following ways:

- Adding an `@Policy` annotation to the JWS file. You can attach a Make Connection policy at the class level only.
- Adding reference to the policy to the web service WSDL.

The following sections describe the steps required to enable Make Connection on a web service:

- Creating the Web Service Make Connection WS-Policy File (Optional)
- Programming the JWS File to Enable Make Connection

Creating the Web Service Make Connection WS-Policy File (Optional)

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 Make Connection policy assertions.

WebLogic Server includes pre-packaged WS-Policy files that contain typical Make Connection assertions that you can use if you do not want to create your own WS-Policy file. The pre-packaged WS-Policy files that support Make Connection are listed in the following table. In some cases, both reliable messaging and Make Connection are enabled by the policy. For more information, see Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection.

> **Note:**
>
> You can attach Make Connection policies at the class level only; you cannot attach the Make Connection policies at the method level.

Table 12-9 Pre-packaged WS-Policy Files That Support Make Connection

Pre-packaged WS-Policy File	Description
`Mc1.1.xml`	Enables Make Connection support on the web service and specifies usage as optional on the web service client. The WS-Policy 1.5 protocol is used. See Mc1.1.xml (WS-Policy File).
`Mc.xml`	Enables Make Connection support on the web service and specifies usage as optional on the web service client. The WS-Policy 1.2 protocol is used. See Mc.xml (WS-Policy File).
`Reliability1.2_ExactlyOnce_WithMC1.1.xml`	Specifies policy assertions related to quality of service. It enables Make Connection support on the web service and specifies usage as optional on the web service client. See Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File).
`Reliability1.2_SequenceSTR.xml`	Specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the web service and specifies usage as optional on the web service client. See Reliability1.2_SequenceSTR.xml (WS-Policy File).
`Reliability1.0_1.2.xml`	Combines 1.2 and 1.0 WS-Reliable Messaging policy assertions. The policy assertions for the 1.2 version Make Connection support on the web service and specifies usage as optional on the web service client. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Reliability1.0_1.2.xml (WS-Policy File).

You can use one of the pre-packaged Make Connection WS-Policy files included in WebLogic Server; these files are adequate for most use cases. You cannot modify the pre-packaged files. If the values do not suit your needs, you must create a custom WS-Policy file. For example, you may wish to configure support of Make Connection as required on the web service client side. The Make Connection policy assertions conform to the WS-PolicyAssertions specification.

To create a custom WS-Policy file that contains Make Connection assertions, use the following guidelines:

- The root element of a WS-Policy file is always `<wsp:Policy>`.
- To configure web service Make Connection, you simply add a `<wsmc:MCSupported>` child element to define the web service Make Connection support.
- The `<wsmc:MCSupported>` child element contains one policy attribute, `Optional`, that specifies whether Make Connection must be configured on the web service client. This attribute can be set to `true` or `false`, and is set to `true` by default. If set to `false`, then use of Make Connection is required and both the ReplyTo and FaultTo (if specified) headers must contain Make Connection anonymous URIs.

The following example enables Make Connection on the web service and specifies that Make Connection must be enabled on the web service client. In this example, the WS-Policy 1.5 protocol is used.

```
<?xml version="1.0"?>
<wsp15:Policy  xmlns:wsp15="http://www.w3.org/ns/ws-policy"
  xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702">
  <wsmc:MCSupported wsp15:Optional="false" />
</wsp15:Policy>
```

Programming the JWS File to Enable Make Connection

This section describes how to enable Make Connection on the web service using a pre-packaged or custom Make Connection WS-Policy file. For information about creating a custom policy file, see Creating the Web Service Make Connection WS-Policy File (Optional).

Use the `@Policy` annotation in your JWS file to specify that the web service has a WS-Policy file attached to it that contains Make Connection assertions. WebLogic Server delivers a set of pre-packaged WS-Policy files, as described in Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection.

Refer to the following guidelines when using the `@Policy` annotation for web service reliable messaging:

- You can attach the Make Connection policy at the class level only; you cannot attach the Make Connection policy at the method level.
- 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="McPolicy.xml", attachToWsdl=true)
    ```

    In this example, the `McPolicy.xml` file is located in the same directory as the JWS file.
  - To specify one of the pre-packaged WS-Policy files or a WS-Policy file that is packaged in a shared Java EE library, 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.

> **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 in *Developing Applications for Oracle WebLogic Server* for information about creating libraries and setting up your environment so the web service can locate the policy files.

  - To specify that the policy file is published 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"
            attachToWsdl=true)
    ```

- Set the `attachToWsdl` 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`.

For more information about the `@Policy` annotation, see weblogic.jws.Policy in *WebLogic Web Services Reference for Oracle WebLogic Server*.

The following example shows a simple JWS file that enables Make Connection; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.async

import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.Policy;

/**
 * Simple reliable Web Service.
 */

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

@Policy(uri="McPolicy.xml", attachToWsdl=true)
public class HelloWorldImpl {
  private static String onewaySavedInput = null;

/**
 * A one-way helloWorld method that saves the given string for later
 * concatenation to the end of the message passed into helloWorldReturn.
 */
  @WebMethod()
  public void helloWorld(String input) {
    System.out.println(" Hello World " + input);
    onewaySavedInput = input;
   }

/**
 * This echo method concatenates the saved message from helloWorld
 * onto the end of the provided message, and returns it.
 */
  @WebMethod()
  public String echo(String input2) {
    System.out.println(" Hello World " + input2 + onewaySavedInput);
    return input + onewaySavedInput;
  }
}
```

As shown in the previous example, the custom `McPolicy.xml` policy 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. You can attach a Make Connection policy at the class level only; you cannot attach a Make Connection policy at the method level.

The policy file is attached to the WSDL file. For information about the pre-packaged policies available and creating a custom policy, see Creating the Web Service Make Connection WS-Policy File (Optional).

The `echo()` method has been marked with the `@WebMethod` JWS annotation, which means it is a public operation called `echo`. Because of the `@Policy` annotation, the operation using Make Connection transport protocol.

Enabling and Configuring Make Connection on a Web Service Client

> **Note:**
>
> The Make Connection and asynchronous client transport features are mutually exclusive. If you attempt to enable both features on the same web service client, an error is returned. For more information about asynchronous client transport, see Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport).
>
> It is recommended that you use the asynchronous handler feature, `AsyncClientHandlerFeature` when using the asynchronous callback handler programming model. For more information, see Developing the Asynchronous Handler Interface.

To enable Make Connection on a web service client, pass an instance of the `weblogic.wsee.mc.api.McFeature` as a parameter when creating the web service proxy or dispatch. A simple example of how to enable Make Connection is shown below.

> **Note:**
>
> This example will use synchronous transport for synchronous methods. To configure Make Connection as the transport for synchronous methods, see Configuring Make Connection as the Transport for Synchronous Methods.

```
package examples.webservices.myservice.client;

import weblogic.wsee.mc.api.McFeature;
...
    List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();
...
    McFeature mcFeature = new McFeature();
    features.add(mcFeature);
...
    // ... Implement asynchronous handler interface as described in
    // Developing the Asynchronous Handler Interface.
    // ....
    AsyncClientHandlerFeature handlerFeature = new AsyncClientHandlerFeature(handler);
    features.add(handlerFeature);
    _features = features.toArray(new WebServiceFeature[features.size()]);
    BackendService port = _service.getBackendServicePort(_features);
...
    // Make the invocation. Our asynchronous handler implementation (set
```

```
    // into the AsyncClientHandlerFeature above) receives the response.
    String request = "Dance and sing";
    System.out.println("Invoking DoSomething asynchronously with request: " + request);
    anotherPort.doSomethingAsync(request);
..
  }
}
```

To configure specific features of Make Connection on the web service client, as described in the following sections.

- Configuring the Expiration Time for Sending Make Connection Messages
- Configuring the Polling Interval
- Configuring the Exponential Backoff
- Configuring Make Connection as the Transport for Synchronous Methods

Configuring the Expiration Time for Sending Make Connection Messages

Table 12-10 defines that `McFeature` methods for configuring the maximum interval of time before an MC Initiator stops sending Make Connection messages to an MC Receiver.

Table 12-10 Methods for Configuring the Expiration Time for Sending Make Connection Messages

Method	Description
`String getsExpires()`	Returns the expiration value currently configured.
`void setExpires(String` *`expires`*`)`	Set the expiration time. The value specified must be a positive value and conform to the XML schema duration lexical format, `P`*`n`*`Y`*`n`*`M`*`n`*`DT`*`n`*`H`*`n`*`M`*`n`*`S`, where *`n`*`Y` specifies the number of years, *`n`*`M` specifies the number of months, *`n`*`D` specifies the number of days, `T` is the date/time separator, *`n`*`H` specifies the number of hours, *`n`*`M` specifies the number of minutes, and *`n`*`S` specifies the number of seconds. This value defaults to `P1D` (1 day).

Configuring the Polling Interval

Table 12-11 defines that `McFeature` methods for configuring the interval of time that must pass before a Make Connection message is sent by an MC Initiator to an MC Receiver after the receipt of an empty response message. If the MC Initiator does not receive a non-empty response for a given message within the specified interval, the MC Initiator sends another Make Connection message.

Table 12-11 Methods for Configuring the Polling Interval

Method	Description
`String getInterval()`	Gets the polling interval.
`void setInterval(String pollingInterval)`	Set the polling interval. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `nY` specifies the number of years, `nM` specifies the number of months, `nD` specifies the number of days, `T` is the date/time separator, `nH` specifies the number of hours, `nM` specifies the number of minutes, and `nS` specifies the number of seconds. This value defaults to `P0DT5S` (5 seconds).

In the following example, the polling interval is set to 36 hours.

```
...
    McFeature mcFeature = new McFeature();
    mcFeature.setInterval("P0DT36H")
    MyService port = service.getMyServicePort(mcFeature);
...
```

Configuring the Exponential Backoff

Table 12-12 defines the `McFeature` methods for configuring the exponential backoff flag. This flag specifies whether the polling interval, described in Configuring the Polling Interval, will be adjusted using the exponential backoff algorithm. In this case, if the MC Initiator does not receive a non-empty response for the time interval specified by the polling interval, the exponential backoff algorithm is used for timing successive retransmissions by the MC Initiator, should the response not be received.

The exponential backoff algorithm specifies that successive polling intervals should increase exponentially, based on the polling interval. For example, if the polling interval is 2 seconds, and the exponential backoff element is set, successive polling intervals if the response is not received are 2, 4, 8, 16, 32, and so on.

This value defaults to false, the same polling interval is used in successive retries; the interval does not increase exponentially.

Table 12-12 Methods for Configuring the Exponential Backoff

Method	Description
`boolean isExponentialBackoff()`	Returns a boolean value indicating whether exponential backoff is enabled.
`void setExponentialBackoff(boolean backoff)`	Set the exponential backoff flag. Valid values are `true` and `false`. This flag defaults to `false`.

In the following example, enables the exponential backoff flag.

```
...
    McFeature mcFeature = new McFeature();
    mcFeature.setMessageInterval(P0DT36H)
    mcFeature.setExponentialBackoff(true);
    MyService port = service.getMyServicePort(mcFeature);
...
```

Configuring Make Connection as the Transport for Synchronous Methods

By default, synchronous methods use synchronous transport even when Make Connection is enabled on the client. You can configure your client to use Make Connection as the transport for synchronous methods. In this case, Make Connection messages are sent by the MC Initiator based on the configured polling interval (described in Configuring the Polling Interval) until a non-empty response message is received.

To configure Make Connection as the transport protocol to use for synchronous methods, use one of the following methods:

- When instantiating a new `McFeature()` object, you can pass as a parameter a boolean value that specifies whether Make Connection should be used as the transport protocol for synchronous methods. For example:

```
...
    McFeature mcFeature = new McFeature(true);
    MyService port = service.getMyServicePort(mcFeature);
...
```

- Use the `McFeature` methods defined in Table 12-13. For example:

```
...
    McFeature mcFeature = new McFeature();
    mcFeature.setUseMCWithSyncInvoke(true);
    MyService port = service.getMyServicePort(mcFeature);
...
```

Table 12-13 Methods for Configuring Synchronous Method Support

Method	Description
`boolean isUseMCWithSyncInvoke()`	Returns a boolean value indicating whether synchronous method support is enabled.
`void setUseMCWithSyncInvoke(boolean` *`useMCWithSyncInvoke`*`)`	Sets the synchronous method support flag. Valid values are `true` and `false`. This flag defaults to `false`.

You can set the maximum amount of time a synchronous method will block and wait for a response using the `weblogic.wsee.jaxws.JAXWSProperties.REQUEST_TIMEOUT` property. This property default to 0 indicating no timeout. For more information about setting message properties, see Propagating User-defined Request Context to the Response.

Using the JAX-WS Reference Implementation

The JAX-WS Reference Implementation (RI) supports the following programming models:

- Asynchronous client polling through use of the `java.util.concurrent.Future` interface.
- Asynchronous callback handlers on a per request basis. The calling client specifies the callback handler at invocation time. When the response is available, the callback handler is invoked to process the response.

Unlike with asynchronous client transport feature, the JAX-WS RI provides very limited support for WS-Addressing, including:

- Manual support for adding client-side outbound WS-Addressing headers.
- Manual support for publishing the client-side endpoint to receive responses.
- No support for detecting incorrect client-side programming model (resulting in synchronous call hanging, for example).
- No support for surviving a client-side or service-side restart.

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

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

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

import javax.xml.ws.BindingProvider;

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

public class AsyncClient  {

   private AddNumbersPortType port = null;
   protected void setUp() throws Exception {
      AddNumbersService service = new AddNumbersService();
      port = service.getAddNumbersPort();
      String serverURI = System.getProperty("wls-server");
      ((BindingProvider) port).getRequestContext().put(
            BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
             "http://" + serverURI + "/JAXWS_ASYNC/AddNumbersService");
   }

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

```
    }

    public void AddNumbersTestDrive() throws Exception {
       int number1 = 10;
       int number2 = 20;

       // Asynchronous Callback method
       AddNumbersCallbackHandler callbackHandler =
          new AddNumbersCallbackHandler();
       Future<?> resp = port.addNumbersAsync(number1, number2,
          callbackHandler);
       // For the purposes of a test, block until the async call completes
       resp.get(5L, TimeUnit.MINUTES);
       int result = callbackHandler.getResponse().getReturn();

       // Polling method
       Response<AddNumbersResponse> addNumbersResp =
            port.AddNumbersAsync(number1, number2);
       while (!addNumbersResp.isDone()) {
             Thread.sleep(100);
       }
       AddNumbersResponse reply = addNumbersResp.get();
       System.out.println("Server responded through polling with: " +
           reply.getResponseType());
    }
}
```

The example demonstrates the steps to implement both the asynchronous polling and asynchronous callback handler programming models.

To implement an asynchronous callback handler:

1. Create an asynchronous handler that implements the `javax.xml.ws.AsyncHandler<T>` interface (see http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/AsyncHandler.html). The asynchronous handler defines one method, `handleResponse`, that enables clients to receive callback notifications at the completion of service endpoint operations that are invoked asynchronously. The type should be set to `AddNumberResponse`.

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

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

            }
         }

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

2. Instantiate the asynchronous callback handler.

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

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

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

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

   `java.util.concurrent.Future` (see https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) represents the result of an asynchronous computation and provides methods for checking the status of the asynchronous task, getting the result, or canceling the task execution.

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

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

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

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

To implement an asynchronous polling mechanism:

1. Instantiate the `AddNumbersService` web service and call the asynchronous version of the web service method, `addNumbersAsync`.

```
Response<AddNumbersResponse> addNumbersResp =
     port.AddNumbersAsync(number1, number2);
```

2. Sleep until a message is received.

```
while (!addNumbersResp.isDone()) {
        Thread.sleep(100);
```

3. Poll for a response.

```
AddNumbersResponse reply = addNumbersResp.get();
```

Propagating Request Context to the Response

WebLogic Server provides a powerful facility that enables you to attach your business context—for example, a business-level message ID—to the request message and access it when the response is returned, regardless of what the request and response messages convey over the wire. For example, you may have a business-level message ID that will not otherwise be available in the response message. By propagating this information with the message, you can access it when the response message is returned.

Web service clients can store any request message context property, as long as it is Serializable. Message context properties can be stored as part of the `weblogic.wsee.jaxws.JAXWSProperties.PERSISTENT_CONTEXT` Map property and retrieved after the response message is returned.

The following example shows how to use the `PERSISTENT_CONTEXT` Map property to define and set a message context property.

Example 12-3 Setting Message Context Properties

```
import weblogic.wsee.jaxws.JAXWSProperties;
. . .
MyClientPort port = myService.getPort();
Map<String, Serializable> clientPersistProps =
    port.getRequestContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
Serializable obj = <my_property>;
clientPersistProps.put("MyProperty", obj);

port.myOperationAsync(<args>, new AsyncHandler<MyOperationResponse>() {
    public void handleResponse(Response<MyOperationResponse> res) {
        try {
            // Get the actual response
            MyOperationResponse response = res.get().getReturn();

            // Get the property stored when making request. Note, this property did not get
            // passed over the wire with the reuqest. The web services runtime stores it.
            Map<String, Serializable> clientPersistProps =
                res.getContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
            Serializable obj = clientPersistProps.get("MyProperty");
            // Do something with MyProperty
        } catch (Exception e) {
           // Error handling
        }
    }
});
...
```

Monitoring Asynchronous Web Service Invocation

You can monitor runtime information for clients that invoke web services asynchronously, such as number of invocations, errors, faults, and so on, using the WebLogic Server Administration Console. To monitor web service clients, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service client is packaged. Expand the application by clicking the **+** node and click on the application module within which the web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab.

If you use the Make Connection transport protocol, you can monitor the Make Connection anonymous endpoints for a web service or client. For each anonymous endpoint, runtime monitoring information is displayed, such as the number of messages received, the number of messages pending, and so on.

You can customize the information that is shown in the table by clicking **Customize this table**.

To monitor Make Connection anonymous endpoints for a web service, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service is packaged. Expand the application by clicking the **+** node; the web services in the application are listed under the **Web Services** category. Click on the name of the web service and select **Monitoring> Ports> Make Connection**.

To monitor Make Connection anonymous endpoints for a web service client, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service client is packaged. Expand the application by clicking the **+** node and click on the application

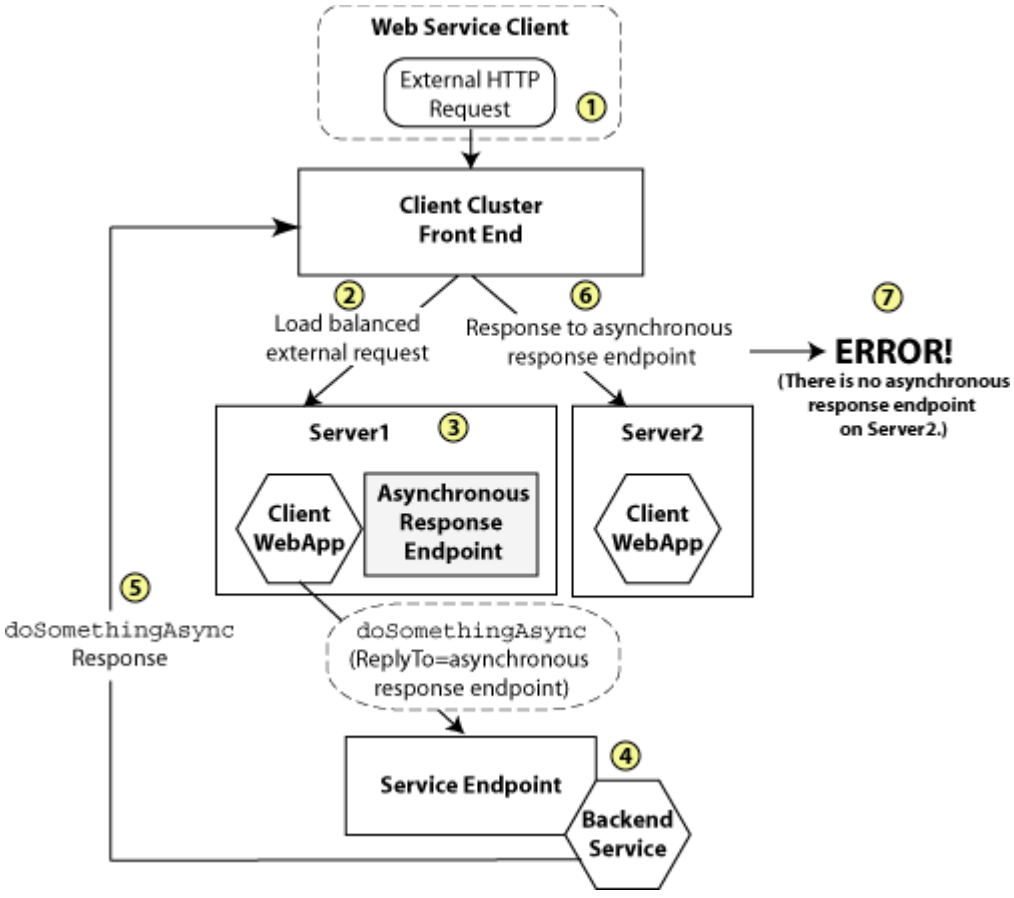
module within which the web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab. Then click **Monitoring> Servers> Make Connection**.

Clustering Considerations for Asynchronous Web Service Messaging

When a web service client runs in a cluster, you need to make special allowances to ensure that the response messages can be delivered properly to the asynchronous response endpoint for asynchronous calls. You defined the asynchronous response endpoint with the AsyncClientTransportFeature, as described in Enabling and Configuring the Asynchronous Client Transport Feature.

Consider the scenario shown in the following figure.

Figure 12-3 Clustering Scenario Resulting in an Error



In the scenario shown in the previous figure:

- A two-node cluster hosts the client application; the nodes are named Server1 and Server2. The cluster has a simple load-balancing front-end proxy.
- The client application is a Web application called ClientWebApp which is deployed homogeneously to the cluster. In other words, the Web application runs on both member servers in the cluster.

- External clients of the ClientWebApp application make requests through the cluster front-end address.

Now consider the following sequence:

1. An external client requests a page from ClientWebApp via the cluster front-end.
2. The cluster front-end load balances the page request and sends it to the ClientWebApp on Server1.
3. ClientWebApp on Server1 creates an instance of a web service client, BackendServiceClient, to communicate with its back-end service, BackendService. The creation of BackendServiceClient causes an asynchronous response endpoint to be published to receive asynchronous responses whenever BackendServiceClient is used to make an asynchronous request.
4. ClientWebApp on Server1 calls `BackendServiceClient.doSomethingAsync()` to perform an operation on the backend service. The address of the asynchronous response endpoint is included in the ReplyTo address. This address starts with the address of the cluster front end, and not the address of Server1.
5. The cluster receives the response to the `doSomething` operation.
6. The cluster load balances the message, this time to Server2.
7. The message delivery fails because there is no asynchronous response endpoint on Server2 to receive the response.

You can use one of the following to resolve this problem:

- Use a SOAP-aware cluster front-end proxy plug-in, such as WebLogic Server HttpClusterServlet. For more information, see Configure Proxy Plug-ins in *Administering Clusters for Oracle WebLogic Server*. This option may not be feasible, for example if your company has standardized on a cluster front-end technology.
- Ensure that all member servers in a cluster publish an asynchronous response endpoint so that the asynchronous response messages can be delivered to any member server and optionally forwarded to the correct server via in-place cluster routing.

To implement the second option, it is recommended that you define a singleton port instance and initialize it when the client container initializes (upon deployment). For an example illustrating the recommended method for initializing the asynchronous response endpoint in a cluster, see Example 11-1.

> **Note:**
>
> You may choose to initialize the endpoint in different ways depending on the container type. For example, if the client is hosted in a web service, a method on the web service container could be annotated with `@PostConstruct` and that method could initialize the singleton port. In an EJB container, you could use the `ejbCreate()` method as the trigger point for creating the singleton port.

13

Roadmap for Developing Reliable Web Services and Clients

This chapter presents best practices for developing WebLogic web services and clients for Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

- Roadmap for Developing Reliable Web Service Clients
- Roadmap for Developing Reliable Web Services
- Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)
- Roadmap for Securing Reliable Web Services

> **Note:**
>
> See also Roadmap for Configuring Web Service Persistence.

Roadmap for Developing Reliable Web Service Clients

Table 13-1 provides best practices for developing reliable web service clients, including an example that illustrates the best practices presented. These guidelines should be used in conjunction with the guidelines provided in Roadmap for Developing JAX-WS Web Service Clients.

Table 13-1 Roadmap for Developing Reliable Web Service Clients

Best Practice	Description
Always implement a reliability error listener.	For more information, see Implementing the Reliability Error Listener.
Group messages into *units of work*.	Rather than incur the RM sequence creation and termination protocol overhead for *every* message sent, you can group messages into business units of work—also referred to as *batching*. For more information, see Grouping Messages into Business Units of Work (Batching). **Note**: This best practice is not demonstrated in Example 13-1.
Set the acknowledgement interval to a realistic value for your particular scenario.	The recommended setting is two times the nominal interval between requests. For more information, see Configuring the Acknowledgement Interval. **Note**: This best practice is not demonstrated in Example 13-1.

Table 13-1 (Cont.) Roadmap for Developing Reliable Web Service Clients

Best Practice	Description
Set the base retransmission interval to a realistic value for your particular scenario.	The recommended setting is two times the acknowledgement interval or nominal response time, whichever is greater. For more information, see Configuring the Base Retransmission Interval. **Note**: This best practice is not demonstrated in Example 13-1.
Set timeouts (inactivity and sequence expiration) to realistic values for your particular scenario.	For more information, see Configuring Inactivity Timeout and Configuring the Sequence Expiration. **Note**: This best practice is not demonstrated in Example 13-1.

The following example illustrates best practices for developing reliable web service clients.

Example 13-1 Reliable Web Service Client Best Practices Example

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.xml.bind.JAXBContext;
import javax.xml.ws.*;

import weblogic.jws.jaxws.client.ClientIdentityFeature;
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
import weblogic.jws.jaxws.client.async.AsyncClientTransportFeature;
import weblogic.wsee.reliability2.api.ReliabilityErrorContext;
import weblogic.wsee.reliability2.api.ReliabilityErrorListener;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;

import com.sun.xml.ws.developer.JAXWSProperties;

/**
 * Example client for invoking a reliable web service asynchronously.
 */
public class BestPracticeAsyncRmClient
  extends GenericServlet {

  private BackendReliableServiceService _service;
  private BackendReliableService _singletonPort;
  private WebServiceFeature[] _features;

  private static int _requestCount;
  private static String _lastResponse;
  private static final String MY_PROPERTY = "MyProperty";

  @Override
  public void init()
    throws ServletException {

    _requestCount = 0;
    _lastResponse = null;

    // Only create the web service object once as it is expensive to create repeatedly.
    if (_service == null) {
      _service = new BackendReliableServiceService();
    }
```

```
// Best Practice: Use a stored list of features, per client ID, to create client instances.
// Define all features for the web service port, per client ID, so that they are
// consistent each time the port is called. For example:
// _service.getBackendServicePort(_features);

List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

// Best Practice: Explicitly define the client ID.
ClientIdentityFeature clientIdFeature =
  new ClientIdentityFeature("MyBackendServiceAsyncRmClient");
features.add(clientIdFeature);

// Best Practice: Always implement a reliability error listener.
// Include this feature in your reusable feature list. This enables you to determine
// a reason for failure, for example, RM cannot deliver a request or the RM sequence fails in
// some way (for example, client credentials refused at service).
WsrmClientInitFeature rmFeature = new WsrmClientInitFeature();
features.add(rmFeature);
rmFeature.setErrorListener(new ReliabilityErrorListener() {
  public void onReliabilityError(ReliabilityErrorContext context) {

    // At a *minimum* do this
    System.out.println("RM sequence failure: " +
                       context.getFaultSummaryMessage());
    _lastResponse = context.getFaultSummaryMessage();

    // And optionally do this...

    // The context parameter conveys whether a request or the entire
    // sequence has failed. If a sequence fails, you will get a notification
    // for each undelivered request (if any) on the sequence.
    if (context.isRequestSpecific()) {
      // Single request failure (possibly as part of a larger sequence failure).
      // Retrieve the original request.
      String operationName = context.getOperationName();
      System.out.println("Failed to deliver request for operation '" +
                         operationName + "'. Fault summary: " +
                         context.getFaultSummaryMessage());
      if ("DoSomething".equals(operationName)) {
        try {
          String request = context.getRequest(JAXBContext.newInstance(),
                                              String.class);
          System.out.println("Failed to deliver request for operation '" +
                             operationName + "' with content: " +
                             request);
          Map<String, Serializable> requestProps =
            context.getUserRequestContextProperties();
          if (requestProps != null) {
            // Retrieve the request property. Use MyProperty
            // to describe the request that failed and print this value
            // during the simple 'error recovery' below.
            String myProperty = (String)requestProps.get(MY_PROPERTY);
            System.out.println("Got MyProperty value propagated from request: "+
                               myProperty);
            System.out.println(myProperty + " failed!");
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
```

```
      } else {
        // The entire sequence has encountered an error.
        System.out.println("Entire sequence failed: " +
                           context.getFaultSummaryMessage());

      }
    }
  });

  // Asynchronous endpoint.
  AsyncClientTransportFeature asyncFeature =
    new AsyncClientTransportFeature(getServletContext());
  features.add(asyncFeature);

  // Best Practice: Define a port-based asynchronous callback handler,
  // AsyncClientHandlerFeature, for asynchronous and dispatch callback handling.
  BackendReliableServiceAsyncHandler handler =
    new BackendReliableServiceAsyncHandler() {
      public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
        // ... Handle Response ...
        try {
          // Show getting the MyProperty value back.
          DoSomethingResponse response = res.get();
          _lastResponse = response.getReturn();
          System.out.println("Got (reliable) async response: " + _lastResponse);
          // Retrieve the request property. This property can be used to
          // 'remember' the context of the request and subsequently process
          // the response.
          Map<String, Serializable> requestProps =
            (Map<String, Serializable>)
              res.getContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
          String myProperty = (String)requestProps.get(MY_PROPERTY);
          System.out.println("Got MyProperty value propagated from request: "+
                             myProperty);
        } catch (Exception e) {
          _lastResponse = e.toString();
          e.printStackTrace();
        }
      }
    };
  AsyncClientHandlerFeature handlerFeature =
    new AsyncClientHandlerFeature(handler);
  features.add(handlerFeature);

  // Set the features used when creating clients with
  // the client ID "MyBackendServiceAsyncRmClient."

  _features = features.toArray(new WebServiceFeature[features.size()]);

  // Best Practice: Define a singleton port instance and initialize it when
  // the client container initializes (upon deployment).
  // The singleton port will be available for the life of the servlet.
  // Creation of the singleton port triggers the asynchronous response endpoint to be published
  // and it will remain published until our container (Web application) is undeployed.
  // Note, we will get a call to destroy() before this.
  _singletonPort = _service.getBackendReliableServicePort(_features);
}

@Override
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException {
```

```
// TODO: ... Read the servlet request ...

// For this simple example, echo the _lastResponse captured from
// an asynchronous DoSomethingResponse response message.

if (_lastResponse != null) {
  res.getWriter().write(_lastResponse);
  _lastResponse = null; // Clear the response so we can get another
  return;
}

// Set _lastResponse to NULL in order to make a new invocation against
// BackendService to generate a new response

// Best Practice: Synchronize use of client instances.
// Create another port using the *exact* same features used when creating _singletonPort.
// Note, this port uses the same client ID as the singleton port and it is effectively the
// same as the singleton from the perspective of the web services runtime.
// This port will use the asynchronous response endpoint for the client ID,
// as it is defined in the _features list.
// NOTE: This is *DEFINITELY* not best practice or ideal because our application is
//       incurring the cost of an RM handshake and sequence termination
//       for *every* reliable request sent. It would be better to send
//       multiple requests on each sequence. If there is not a natural grouping
//       for messages (a business 'unit of work'), then you could batch
//       requests onto a sequence for efficiency. For more information, see
//       Grouping Messages into Business Units of Work (Batching).
BackendReliableService anotherPort =
  _service.getBackendReliableServicePort(_features);

// Set the endpoint address for BackendService.
((BindingProvider)anotherPort).getRequestContext().
  put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
      "http://localhost:7001/BestPracticeReliableService/BackendReliableService");

// Make the invocation. Our asynchronous handler implementation (set
// into the AsyncClientHandlerFeature above) receives the response.
String request = "Protect and serve";
System.out.println("Invoking DoSomething reliably/async with request: " +
                   request);
// Add a persistent context property that will be returned on the response.
// This property can be used to 'remember' the context of this
// request and subsequently process the response. This property will *not*
// get passed over wire, so the properties can change independent of the
// application message.
Map<String, Serializable> persistentContext =
  (Map<String, Serializable>)((BindingProvider)anotherPort).
    getRequestContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
String myProperty = "Request " + (++_requestCount);
persistentContext.put(MY_PROPERTY, myProperty);
System.out.println("Request being made (reliably) with MyProperty value: " +
                   myProperty);
anotherPort.doSomethingAsync(request);

// Return a canned string indicating the response was not received
// synchronously. Client needs to invoke the servlet again to get
// the response.
res.getWriter().write("Waiting for response...");

// Best Practice: Explicitly close client instances when processing is complete.
```

```
    // If not closed, the port will be closed automatically when it goes out of scope.
    // This will force the termination of the RM sequence we created when sending the first
    // doSomething request. For a better way to handle this, see
    // Grouping Messages into Business Units of Work (Batching).
    // NOTE: Even though the port is closed explicitly (or even if it goes out of scope)
    //       the reliable request sent above will still be delivered
    //       under the scope of the client ID used. So, even if the service endpoint
    //       is down, RM retries the request and delivers it when the service endpoint
    //       available. The asynchronous resopnse will be delivered as if the port instance was
    //       still available.
    ((java.io.Closeable)anotherPort).close();
  }

  @Override
  public void destroy() {

    try {
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the singleton port created during initialization. Note, the asynchronous
      // response endpoint generated by creating _singletonPort *remains*
      // published until our container (Web application) is undeployed.
      ((java.io.Closeable)_singletonPort).close();
      // Upon return, the Web application is undeployed, and our asynchronous
      // response endpoint is stopped (unpublished). At this point,
      // the client ID used for _singletonPort will be unregistered and will no longer be
      // visible from the Administration Console and WLST.
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Roadmap for Developing Reliable Web Services

Table 13-2 provides best practices for developing reliable web services. For best practices when accessing reliable web services from behind a firewall, see Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection).

Table 13-2 Roadmap for Developing Reliable Web Services

Best Practice	Description
Set the base retransmission interval to a realistic value for your particular scenario.	For more information, see Configuring the Base Retransmission Interval.
Set the acknowledgement interval to a realistic value for your particular scenario.	The recommended setting is two times the nominal interval between requests. For more information, see Configuring the Acknowledgement Interval.
Set timeouts (inactivity and sequence expiration) to realistic values for your particular scenario.	Consider the following: • For very short-lived exchanges, the default timeouts may be too long and sequence state might be maintained longer than necessary. • Set timeouts to two times the expected lifetime of a given business unit of work. This allows the sequence to live long enough For more information, see Configuring Inactivity Timeout and Configuring the Sequence Expiration.

Table 13-2 (Cont.) Roadmap for Developing Reliable Web Services

Best Practice	Description
Use an reliable messaging policy that reflects the minimum delivery assurance (or quality of service) required.	By default, the delivery assurance is set to Exactly Once, In Order. If you do not require ordering, it can increase performance to set the delivery assurance to simply Exactly Once. Similarly, if your service can tolerate duplicate requests, delivery assurance can be set to At Least Once. For more information about delivery assurance for reliable messaging, see Table 14-1 and Creating the Web Service Reliable Messaging WS-Policy File.

Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)

Table 13-3 provides best practices for accessing reliable web services from behind a firewall using Make Connection. These guidelines should be used in conjunction with the general guidelines provided in Roadmap for Developing Reliable Web Services and Roadmap for Developing Asynchronous Web Service Clients.

Table 13-3 Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)

Best Practice	Description
Coordinate the Make Connection polling interval with the reliable messaging base retransmission interval.	The polling interval you set for Make Connection transport sets the lower limit for the amount of time it takes for reliable messaging protocol messages to make the round trip between the client and service. If you set the reliable messaging base retransmission interval to a value near to the Make Connection polling interval, it will be unlikely that a reliable messaging request will be received by the web service, and the accompanying RM acknowledgement sent for that request (at best one Make Connection polling interval later) before the reliable messaging runtime attempts to retransmit the request. Setting the reliable messaging base retransmission interval to a value that is too low results in unnecessary retransmissions for requests, and potentially a cascading load on the service side as it attempts to process redundant incoming requests and Make Connection poll messages to retrieve the responses from those requests. Oracle recommends setting the base retransmission interval to a value that is at least two times the Make Connection polling interval. **Note**: When web services reliable messaging and Make Connection are used together, the Make Connection polling interval value will be adjusted at runtime, if necessary, to ensure that the value is set at least 3 seconds less than the reliable messaging base transmission interval. If the base transmission interval is three seconds or less, the Make Connection polling interval is set to the value of the base retransmission interval. For more information setting the Make Connection polling interval and reliable messaging base retransmission interval, see Configuring the Polling Interval and Configuring the Base Retransmission Interval, respectively.

Roadmap for Securing Reliable Web Services

Table 13-4 provides best practices for securing reliable web services using WS-SecureConversation. These guidelines should be used in conjunction with the guidelines provided in Roadmap for Developing Reliable Web Services.

Table 13-4 Roadmap for Securing Reliable Web Services

Best Practice	Description
Coordinate the WS-SecureConversation lifetime with the reliable messaging base retransmission and acknowledgement intervals.	A WS-SecureConversation lifetime that is set to a value near to or less than the reliable messaging base retransmission and acknowledgement intervals may result in the WS-SecureConversation token expiring before the reliable messaging handshake message can be sent to the web service. For this reason, Oracle recommends setting the WS-SecureConversation lifetime to a value that is at least two times the base retransmission interval. For more information setting the base retransmission interval, see Configuring the Base Retransmission Interval.

14

Using Web Services Reliable Messaging

This chapter describes how to use web services reliable messaging (WS-ReliableMessaging) for WebLogic web services using Java API for XML Web Services (JAX-WS).
See also Roadmap for Developing Reliable Web Services and Clients.

This chapter includes the following sections:

- Overview of Web Services Reliable Messaging
- Steps to Create and Invoke a Reliable Web Service
- Configuring the Source and Destination WebLogic Server Instances
- Creating the Web Service Reliable Messaging WS-Policy File
- Programming Guidelines for the Reliable JWS File
- Invoking a Reliable Web Service from a Web Service Client
- Configuring Reliable Messaging
- Implementing the Reliability Error Listener
- Managing the Life Cycle of a Reliable Message Sequence
- Monitoring Web Services Reliable Messaging
- Grouping Messages into Business Units of Work (Batching)
- Client Considerations When Redeploying a Reliable Web Service
- Interoperability with WebLogic Web Service Reliable Messaging

The WebLogic Server Examples Server includes three reliable messaging examples:

- Configuring Reliable Messaging for JAX-WS Web Services
- Using Make Connection and Reliable Messaging for JAX-WS Web Services
- Configuring Secure and Reliable Messaging for JAX-WS Web Services

For more information, see Web Services Samples in the WebLogic Serer Distribution in *Understanding WebLogic Web Services for Oracle WebLogic Server*.

Overview of Web Services Reliable Messaging

Web service reliable messaging is a framework that enables an application running on one application server to *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 endpoints (web service and client) in the presence of software component, system, or network failures.

WebLogic web services conform to the WS-ReliableMessaging 1.2 specification (February 2009) at `http://docs.oasis-open.org/ws-rx/wsrm/200702` (and supports version 1.1). This specification describes how two endpoints (web service and client) on different application servers can communicate reliably. 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.

Table 14-1 Delivery Assurances for Reliable Messaging

Delivery Assurance	Description
At Most Once	Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
At Least Once	Every message is delivered at least once. It is possible that some messages are delivered more than once.
Exactly Once	Every message is delivered exactly once, without duplication.
In Order	Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

The following sections describe how to create reliable web services and clients and how to configure WebLogic Server instances to which the web services are deployed.

Using WS-Policy to Specify Reliable Messaging Policy Assertions

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 files are XML files that describe features such as the version of the supported WS-ReliableMessaging specification and quality of service requirements. The WS-Policy specification (`http://www.w3.org/TR/ws-policy/`) provides a general purpose model and syntax to describe and communicate the policies of a web service.

WebLogic Server includes pre-packaged WS-Policy files that contain typical reliable messaging assertions, as described in Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection. If the pre-packaged WS-Policy files do not suit your needs, you must create your own WS-Policy file. See Creating the Web Service Reliable Messaging WS-Policy File for details. See Web Service Reliable Messaging Policy Assertion Reference in the *WebLogic Web Services Reference for Oracle WebLogic Server* for reference information about the reliable messaging policy assertions.

Supported Transport Types for Reliable Messaging

You can use web service reliable messaging asynchronously or synchronously. When delivering messages asynchronously, you can configure buffering to support automatic message delivery retries, if desired.

The following table summarizes the transport type support for web services reliable messaging. For information about transport type support for web service clients, see Invoking a Reliable Web Service from a Web Service Client. For failure recovery information, see Reliable Messaging Failure Recovery Scenarios.

> **Note:**
>
> Message buffering is configurable for web services, as described in Configuring Message Buffering for Web Services. For web service clients, message buffering is enabled by default.

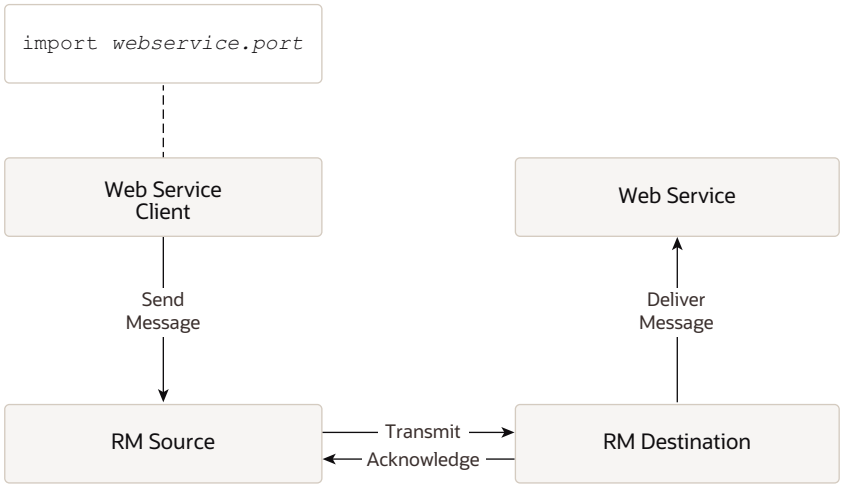
Table 14-2 Transport Types for Web Services Reliable Messaging

Transport Type	Features
Asynchronous transport	For buffered web services: • Most robust usage mode, but requires the most overhead. • Automatically retries message delivery. • Survives network outages. • Enables restart of the source or destination endpoint. • Uses non-anonymous ReplyTo. • Employs asynchronous client transport enabling a single thread to service multiple requests, absorbing load more efficiently. For more information, see Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport). • Web service clients can use asynchronous or synchronous invocation semantics to invoke the web service. For more information, see Table 12-1. For non-buffered web services: • Less overhead than asynchronous, buffered usage mode. • Persists sequence state only. • Uses non-anonymous ReplyTo. • Web service clients can use asynchronous or synchronous invocation semantics to invoke the web service. For more information, see Table 12-1.
Synchronous transport	• Offers the least overhead and simplest programming model. • Uses anonymous ReplyTo. • Web service clients can use asynchronous or synchronous invocation semantics to invoke the web service. For more information, see Table 12-1. • If a web service client invokes a buffered web service using synchronous transport, one of following will result: - If this is the first request of the sequence, the destination sequence will be set to be non-buffered (as though the web service configuration was set as non-buffered). - If this is not the first request of the sequence (that is, the client sent a request using asynchronous transport previously), then the request is rejected and a fault returned.

The Life Cycle of the Reliable Message Sequence

The following figure shows a one-way reliable message exchange.

Figure 14-1 Web Service Reliable Message Exchange



A *reliable message sequence* is used to track the progress of a set of messages that are exchanged reliably between an RM source and RM destination. A sequence can be used to send zero or more messages, and is identified by a string *identifier*. This identifier is used to reference the sequence when using reliable messaging.

The web service client application sends a message for reliable delivery which is transmitted by the RM source to the RM destination. The RM destination acknowledges that the reliable message has been received and delivers it to the web service application. The message may be retransmitted by the RM source until the acknowledgement is received. The RM destination, if configured to buffer requests, may redeliver the request to the web service if the web service fails to process the request.

A web service client sends messages to a target web service by invoking methods on the client instance (port or Dispatch instance). A *port* is associated with the port type of the reliable web service and represents a programmatic interface to that service. The port is created by the `<clientgen>` child element of the `jwsc` Ant task. A *Dispatch instance* is a loosely-typed, general-purpose interface for delivering whole messages from the client to the web service. For more information about Dispatch clients, see Developing a Web Service Dispatch Client.

WebLogic stores the identifier for the reliable message sequence within this client instance. This causes the reliable message sequence to be connected to a single client instance. All messages that are sent using a given client instance will use the same reliable messaging sequence, regardless of the number of messages that are sent. (Unless you using batching, as described in Grouping Messages into Business Units of Work (Batching).)

Because WebLogic Server retains resources associated with the reliable sequence, it is recommended that you take steps to release these resources in a timely fashion. This can be done by managing the lifecycle of the client instance itself, or by using the `weblogic.wsee.reliability2.api.WsrmClient` API. Use the `WsrmClient` API to perform common tasks such as set configuration options, get the sequence id, and terminate a reliable sequence. For more information, see Managing the Life Cycle of a Reliable Message Sequence.

Reliable Messaging Failure Recovery Scenarios

The following sections outline reliable messaging failure recovery for various scenarios.

- RM Destination Down Before Request Arrives
- RM Source Down After Request is Made
- RM Destination Down After Request Arrives
- Failure Scenarios with Non-buffered Reliable Web Services

The first three scenarios assume that buffering is enabled on both the web service and client. The last scenario describes reliable messaging failure recovery for non-buffered web services. Buffering is enabled on web service client by default. To configure buffering on the web service, see Configuring Message Buffering for Web Services.

RM Destination Down Before Request Arrives

Table 14-3 describes the reliable messaging failure recovery scenario when an RM destination is unavailable before a request from the RM source arrives.

It is assumed that web service buffering is enabled on both the web service and client. Buffering is enabled on web service client by default. To configure buffering on the web service, see Configuring Message Buffering for Web Services.

Table 14-3 Reliable Messaging Failure Recovery Scenario—RM Destination Down Before Request Arrives

Transport Type	Scenario Description
Asynchronous Transport	1. Client invokes an asynchronous method. 2. Reliable messaging runtime accepts the request; client returns to do other work. 3. Reliable messaging runtime attempts to deliver the request and fails because the RM destination is down. 4. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery fails again. 5. RM destination comes up. 6. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery succeeds. 7. Acknowledgement is sent to the client which includes the message number of the request. The reliable messaging runtime removes the message from the retry list. 8. Response arrives and the client processes it. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Managing the Life Cycle of a Reliable Message Sequence.

Table 14-3 (Cont.) Reliable Messaging Failure Recovery Scenario—RM Destination Down Before Request Arrives

Transport Type	Scenario Description
Synchronous Transport	1. Client invokes a synchronous method. 2. Reliable messaging runtime accepts the request and blocks the client thread. 3. Reliable messaging runtime attempts to deliver the request and fails because the RM destination is down. 4. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery fails again. 5. RM destination comes up. 6. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery succeeds. 7. Response and acknowledgement are sent to the client via the transport back-channel. The acknowledgement includes the message number of the request. The reliable messaging runtime removes the message from the retry list. 8. Reliable messaging runtime unblocks the client thread and returns the response. 9. Client receives the response as the return value of the method invocation, and processes the response. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Managing the Life Cycle of a Reliable Message Sequence. **Note**: To achieve true reliability with synchronous transport, it is recommended that you use Make Connection. For more information, see Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection).

RM Source Down After Request is Made

Table 14-4 describes the reliable messaging failure recovery scenario when an RM source goes down after a request is made.

It is assumed that web service buffering is enabled on both the web service and client. Buffering is enabled on web service clients by default. To configure buffering on the web service, see Configuring Message Buffering for Web Services.

Table 14-4 Reliable Messaging Failure Recovery Scenario—RM Source Down After Request is Made

Transport Type	Scenario Description
Asynchronous Transport	1. Client invokes an asynchronous method. 2. Reliable messaging runtime accepts the request; client returns to do other work. 3. Client (RM source) goes down. 4. Client comes up. Client must re-initialize the client instance using the same client ID. The runtime will use this client ID to retrieve the reliable sequence ID that was active for the client. For more information, see Managing the Client ID. 5. Reliable messaging runtime detects the reliable sequence ID that was in use prior to the client going down and recovers the accepted requests. **Note**: This step is accomplished only after the client re-initializes the client instance that was used to send the request because delivery of the request depends on resources provided by the client instance. It is recommended that clients initialize the client instance in a static block, or use a `@PostConstruct` annotation or other mechanism to ensure early initialization of the client instance. For more information, see the best practices examples presented in Roadmap for Developing Asynchronous Web Service Clients. 6. Reliable messaging runtime sends the request and succeeds. 7. Acknowledgement is sent to the client which includes the message number of the request. The reliable messaging runtime removes the message from the retry list. 8. Response arrives and the client processes it. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Managing the Life Cycle of a Reliable Message Sequence.

Table 14-4 (Cont.) Reliable Messaging Failure Recovery Scenario—RM Source Down After Request is Made

Transport Type	Scenario Description
Synchronous Transport	1. Client invokes a synchronous method. 2. Reliable messaging runtime accepts the request and blocks the client thread. 3. Reliable messaging runtime attempts to deliver the request. The request delivery succeeds. 4. Before response can be sent, the client (RM source) goes down. Client thread is lost as the VM exits, along with the invocation state and calling stack of the client itself. 5. Client (RM source) comes up. Client must re-initialize the client instance (port or Dispatch) using the same client ID. For more information, see Managing the Client ID. 6. Reliable messaging runtime detects the previous sequence ID for the client, and sees that the last request was made synchronously. 7. Reliable messaging runtime delivers a permanent failure notification for this request, and fails the entire RM sequence associated with the client instance. Any `ReliabilityErrorListener` associated with the client instance will be called at this point. 8. Client is responsible for retrieving the original request (via some client-specific mechanism) and resending it by re-invoking the client instance with the request. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Managing the Life Cycle of a Reliable Message Sequence. **Note**: To achieve true reliability with synchronous transport, it is recommended that you use Make Connection. For more information, see Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection).

RM Destination Down After Request Arrives

Table 14-5 describes the reliable messaging failure recovery scenario when an RM destination is unavailable after a request has been accepted from the RM source.

It is assumed that web service buffering is enabled on both the web service and client. Buffering is enabled on web service client by default. To configure buffering on the web service, see Configuring Message Buffering for Web Services.

Table 14-5 Reliable Messaging Failure Recovery Scenario—RM Destination Down After Request Arrives

Transport Type	Scenario Description
Asynchronous Transport	1. Client invokes an asynchronous method. 2. Reliable messaging runtime accepts the request; client returns to do other work. 3. Reliable messaging runtime attempts to deliver the request and succeeds. 4. The RM destination accepts the request and send an acknowledgement on the back channel. 5. Reliable messaging runtime sees the acknowledgement and removes the message from the retry list. 6. RM destination goes down. 7. Reliable messaging runtime on RM source retries any pending requests during this time. 8. RM destination comes up. 9. RM destination recovers the stored request, processes it, and sends the response. 10. Response arrives and the client processes it. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Managing the Life Cycle of a Reliable Message Sequence.

Table 14-5 (Cont.) Reliable Messaging Failure Recovery Scenario—RM Destination Down After Request Arrives

Transport Type	Scenario Description
Synchronous Transport	**Note**: If you attempt to invoke a buffered web service using synchronous transport, one of following will result: • If this is the first request of the sequence, the destination sequence will be set to be non-buffered (as though the web service configuration was set as non-buffered). • If this is not the first request of the sequence (that is, the client sent a request using asynchronous transport previously), then the request is rejected and a fault returned. The following describes the sequence of this scenario: 1. Client invokes a synchronous method. 2. Reliable messaging runtime accepts the request and blocks the client thread. 3. Reliable messaging runtime attempts to deliver the request. The request delivery succeeds. 4. RM destination accepts the request and sends an acknowledgement via the transport back channel. 5. Client (RM source) detects the acknowledgement and removes the request from the retry list. 6. RM destination goes down. 7. Client thread remains blocked. 8. RM Destination comes up, recovers, and processes the request, and sends the response to the client. 9. Reliable messaging runtime unblocks the client thread and returns the response. 10. Client receives the response as the return value of the method invocation, and processes the response. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Managing the Life Cycle of a Reliable Message Sequence.

Failure Scenarios with Non-buffered Reliable Web Services

A non-buffered web service operates differently than a buffered web service in that it does not buffer a request to hardened storage before acknowledging it and attempting to process it. A non-buffered web service will not attempt to reprocess a request if the service logic fails, whereas a buffered web service will attempt to reprocess the request. In both cases, buffered or non-buffered, any response generated by the web service will be buffered before it is sent back to the client.

A non-buffered web service may be useful in the following cases:

- Web service operates against non-transactional resources and should not process any request more than once (because rolling back the transaction that dequeued the buffered request cannot roll back the side effects of the non-transactional service).

- Web service is relatively light weight, and does not take very long to process requests.
- Web service performance is of paramount importance and risk of losing request or response is acceptable. Non-buffered web services will not incur the overhead of buffering the request to a store, and thus can deliver better throughput than a buffered web service. The performance gain is dependent on how much time and resources are required to buffer the requests (for example, very large request messages may take significant time and resources to buffer).

A non-buffered web service is operationally similar to a buffered web service in most failure scenarios. The exceptions are cases where the service (RM destination) itself fails. For example, in all the RM source failure scenarios described, the behavior is the same for a buffered or a non-buffered web service (RM destination). For non-buffered web services the failure window is open between the following two points:

- The request is accepted for processing.
- The response from the web service is registered for delivery to the client (RM source).

If the web service (RM destination) fails between these two points, the RM source will assume the request has been successfully processed (since it has been acknowledged) but will *never* receive a response, and the request may never have been processed.

Carefully consider this failure window before configuring a web service to run as non-buffered.

Steps to Create and Invoke a Reliable Web Service

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 custom 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 client code is deployed, as well as the *destination* WebLogic Server instance on which the reliable web service itself is deployed.

Table 14-6 summarizes the steps to create a reliable web service and a client that invokes an operation of the reliable web service. The procedure describes how to create the JWS files that implement the web service and client from scratch; if you want to update existing JWS files, use this procedure as a guide. The procedure also describes how to configure the source and destination WebLogic Server instances.

It is assumed that you have completed the following tasks:

- You have created the *destination* and *source* WebLogic Server instances. You deploy the reliable web service to the *destination* WebLogic Server instance, and the client that invokes the reliable web service to the *source* WebLogic Server instance.
- You have set up an Ant-based development environment.
- You have working `build.xml` files that you can edit, for example, to add targets for running the `jwsc` Ant task and deploying the generated reliable web service.

For more information, see Developing JAX-WS Web Services. For best practices for developing asynchronous and reliable web services and clients, see Roadmap for Developing Reliable Web Services and Clients.

Table 14-6 Steps to Create and Invoke a Reliable Web Service

#	Step	Description
1	Configure the *destination* and *source* WebLogic Server instances.	You deploy the reliable web service to the *destination* WebLogic Server instance, and the client that invokes the reliable web service to the *source* WebLogic Server instance. For information about configuring the destination WebLogic Server instance, see Configuring the Source and Destination WebLogic Server Instances.
2	Create the WS-Policy file. (Optional)	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. For details about creating your own WS-Policy file, see Creating the Web Service Reliable Messaging WS-Policy File. **Note**: This step is not required if you plan to use one of the WS-Policy files that are included in WebLogic Server; see Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection for more information.
3	Create or update the JWS file that implements the reliable web service.	This web service will be deployed to the destination WebLogic Server instance. See Programming Guidelines for the Reliable JWS File. For examples demonstrating best practices, see Roadmap for Developing Reliable Web Services and Clients.
4	Update the `build.xml` file that is used to compile the reliable web services.	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.
5	Compile and deploy the reliable JWS file.	Compile the reliable JWS file by calling the appropriate target and deploy to the destination WebLogic Server. For example: `prompt> ant build-reliableService deploy-reliableService`
6	Create or update the web service client.	The web service client invokes the reliable web service and will be deployed to the source WebLogic Server. See Invoking a Reliable Web Service from a Web Service Client.
7	Configure reliable messaging. (Optional)	Configure reliable messaging for the reliable web service using the WebLogic Server Administration Console. The WS-Policy file attached to the reliable web service provides the initial configuration settings. See Configuring Reliable Messaging.
8	Implement a reliability error listener. (Optional)	Implement a reliability error listener to receive notifications if a reliable delivery fails. See Implementing the Reliability Error Listener.
9	Manage the life cycle of a reliable message sequence. (Optional)	WebLogic Server provides a client API, `weblogic.wsee.reliability2.api.WsrmClient`, for use with the web service reliable messaging. Use this API to perform common life cycle tasks such as set configuration options, get the reliable sequence id, and terminate a reliable sequence. See Managing the Life Cycle of a Reliable Message Sequence.
10	Update the `build.xml` file that is used to compile the client web service.	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.
11	Compile and deploy the web service client file.	Compile your client file by calling the appropriate target and deploy to the source WebLogic Server. For example: `prompt> ant build-clientService deploy-clientService`
12	Monitor web services reliable messaging.	Use the WebLogic Server Administration Console to monitor web services reliable messaging. See Monitoring Web Services Reliable Messaging.

Each of these steps is described in more detail in the following sections. In addition, the following topics are discussed:

- Grouping Messages into Business Units of Work (Batching)—Describes how to group messages into business *units of work*—also called batching—to improve performance when using reliable messaging.
- Client Considerations When Redeploying a Reliable Web Service—Describes client considerations for when you deploy a new version of an updated reliable WebLogic web service alongside an older version of the same web service.
- Interoperability with WebLogic Web Service Reliable Messaging—Provides recommendations for interoperating with WebLogic web services reliable messaging.

Configuring the Source and Destination WebLogic Server Instances

You need to configure web service persistence on the destination and source WebLogic Server instances. You deploy the reliable web service to the *destination* WebLogic Server instance, and the client that invokes the reliable web service to the *source* WebLogic Server instance.

When using web services reliable messaging, the web services reliable messaging sequence is saved to the web service persistent store any time its state changes. Examples of state change include:

- Reliable messaging state is updated (creating, created, terminating, terminated, and so on).
- Security property is updated (such as security context token)
- Message is sent on the reliable messaging sequence (if message buffering is enabled)
- Acknowledgement when a message arrives

You can configure web service persistence using the Configuration Wizard to extend the WebLogic Server domain using a web services-specific extension template. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Server Administration Console or WLST. For information about configuring web service persistence, see Configuring Web Service Persistence.

You may also wish to configure buffering for web services. For considerations and steps to configure message buffering, see Configuring Message Buffering for Web Services.

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.

WebLogic Server includes pre-packaged WS-Policy files that contain typical reliable messaging assertions that you can use if you do not want to create your own WS-Policy file.

The pre-packaged WS-Policy files are listed in the following table. This table also specifies whether the WS-Policy file can be attached at the method level; if the value in this column is no, then the WS-Policy file can be attached at the class level only. For more information, see Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection.

> **Note:**
>
> The `DefaultReliability.xml` and `LongRunningReliability.xml` files are deprecated in this release. Use of the `DefaultReliability1.2.xml`, `Reliability1.2_SequenceTransportSecurity`, or `Reliability1.0_1.2.xml` file is recommended and required to comply with the 1.2 version of the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.pdf`.

Table 14-7 Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`DefaultReliability1.2.xml`	Specifies policy assertions related to delivery assurance. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See DefaultReliability1.1.xml (WS-Policy File).	Yes
`DefaultReliability1.1.xml`	Specifies policy assertions related to quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See DefaultReliability1.1.xml (WS-Policy File).	Yes
`Reliability1.2_ExactlyOnce_WithMC1.1.xml`	Specifies policy assertions related to quality of service. It enables Make Connection support on the web service and specifies usage as optional on the web service client. See Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File).	No
`Reliability1.2_SequenceSTRSecurity`	Specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the web service and specifies usage as optional on the web service client. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File).	No
`Reliability1.1_SequenceSTRSecurity`	The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File).	Yes

Table 14-7 (Cont.) Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`Reliability1.2_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File).	Yes
`Reliability1.1_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File).	Yes
`Reliability1.0_1.2.xml`	Combines 1.2 and 1.0 WS-Reliable Messaging policy assertions. The policy assertions for the 1.2 version Make Connection support on the web service and specifies usage as optional on the web service client. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Reliability1.0_1.2.xml (WS-Policy File).	No
`Reliability1.0_1.1.xml`	Combines 1.1 and 1.0 WS Reliable Messaging policy assertions. See Reliability1.0_1.1.xml (WS-Policy.xml File).	Yes
`DefaultReliability.xml`	**Deprecated.** The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf`. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. 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 (WS-Policy) [Deprecated].	Yes
`LongRunningReliability.xml`	**Deprecated.** The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 for long running processes. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. 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 (WS-Policy) [Deprecated].	Yes

You can use one of the pre-packaged reliable messaging WS-Policy files included in WebLogic Server; these files are adequate for most use cases. You cannot modify the pre-

packaged files. If the values do not suit your needs, you must create a custom WS-Policy file. The following sections describe how to create a custom WS-Policy file.

- Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Versions 1.2 and 1.1
- Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.0 (Deprecated)
- Using Multiple Policy Alternatives

Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Versions 1.2 and 1.1

This section describes how to create a custom WS-Policy file that contains web service reliable messaging assertions that are based on the following specifications:

- WS Reliable Messaging Policy Assertion Version 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.2-spec-os.html`
- WS Reliable Messaging Policy Assertion Version 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declaration:

```
<wsp:Policy
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
```

You wrap all web service reliable messaging policy assertions inside of a `<wsrmp:RMAssertion>` element. This element should include the following namespace declaration for using web service reliable messaging policy assertions:

```
<wsrmp:RMAssertion
   xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
```

The following table lists the web service reliable messaging assertions that you can specify in the WS-Policy file. 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:

Table 14-8 Web Service Reliable Messaging Assertions (Versions 1.2 and 1.1)

Assertion	Description
`<wsrmp:SequenceSTR>`	To secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. You can only specify one security assertion; that is, you can specify `wsrmp:SequenceSTR` or `wsrmp:SequenceTransportSecurity`, but not both.

Table 14-8 (Cont.) Web Service Reliable Messaging Assertions (Versions 1.2 and 1.1)

Assertion	Description
`<wsrmp:SequenceTransportSecurity>`	To secure messages in a reliable sequence, the runtime will use the SSL transport session that is used to send the `CreateSequence` message. This assertion must be used in conjunction with the `sp:TransportBinding` assertion that requires the use of some transport-level security mechanism (for example, `sp:HttpsToken`). You can only specify one security assertion; that is, you can specify `wsrmp:SequenceSTR` or `wsrmp:SequenceTransportSecurity`, but not both.
`<wsrm:DeliveryAssurance>`	Delivery assurance (or quality of service) of the web service. Valid values are `AtMostOnce`, `AtLeastOnce`, `ExactlyOnce`, and `InOrder`. You can set one of the delivery assurances defined in the following table. If not set, the delivery assurance defaults to `ExactlyOnce`. For more information about delivery assurance, see Table 14-1.

The following example shows a simple web service reliable messaging WS-Policy file:

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

<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceTransportSecurity/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

For more information about Reliable Messaging policy assertions in the WS-Policy file, see Web Service Reliable Messaging Policy Assertion Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.0 (Deprecated)

This section describes how to create a custom WS-Policy file that contains web service reliable messaging assertions that are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf`.

> **Note:**
>
> Many of the reliable messaging policy assertions described in this section are managed through JWS annotations or configuration.

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

You wrap all web service reliable messaging policy assertions inside of a `<wsrm:RMAssertion>` element. The assertions that use the `wsrm:` namespace are standard ones defined by the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`. The assertions that use the `beapolicy:` namespace are WebLogic-specific. See Web Service Reliable Messaging Policy Assertion Reference in the *WebLogic Web Services Reference for Oracle WebLogic Server* for details.

The following table lists the web service reliable messaging assertions that you can specify in the WS-Policy file. 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,

Table 14-9 Web Service Reliable Messaging Assertions (Version 1.0)

Assertion	Description
`<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 standalone acknowledgment. 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 (see `https://www.w3.org/TR/xmlschema-2/#duration`). For example, if you want to set the expiration time to one day, use the following: `<beapolicy:Expires Expires="P1D" />`.

Table 14-9 (Cont.) Web Service Reliable Messaging Assertions (Version 1.0)

Assertion	Description
`<beapolicy:QOS>`	Delivery assurance level, as described in Table 14-1. 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>
```

For more information about reliable messaging policy assertions in the WS-Policy file, see Web Service Reliable Messaging Policy Assertion Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Using Multiple Policy Alternatives

You can configure multiple policy alternatives—also referred to as *smart policy alternatives*—for a single web service by creating a custom policy file. At runtime, WebLogic Server selects which of the configured policies to apply. It excludes policies that are not supported or have conflicting assertions and selects the appropriate policy, based on your configured preferences, to verify incoming messages and build the response messages.

The following example provides an example of a security policy that supports both 1.2 and 1.0 WS-Reliable Messaging. Each policy alternative is enclosed in a `<wsp:All>` element.

> **Note:**
>
> The 1.0 web service reliable messaging assertions are prefixed by `wsrmp10`.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrmp10:RMAssertion
```

```
        xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
         <wsrmp10:InactivityTimeout Milliseconds="1200000"/>
         <wsrmp10:BaseRetransmissionInterval Milliseconds="60000"/>
         <wsrmp10:ExponentialBackoff/>
         <wsrmp10:AcknowledgementInterval Milliseconds="800"/>
       </wsrmp10:RMAssertion>
     </wsp:All>
     <wsp:All>
       <wsrmp:RMAssertion
            xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:SequenceSTR/>
         <wsrmp:DeliveryAssurance>
           <wsp:Policy>
             <wsrmp:AtMostOnce/>
           </wsp:Policy>
         </wsrmp:DeliveryAssurance>
       </wsrmp:RMAssertion>
     </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

For more information about multiple policy alternatives, see Smart Policy Selection in *Securing WebLogic Web Services for Oracle WebLogic Server*.

Programming Guidelines for the Reliable JWS File

> **Note:**
>
> For best practices for developing reliable web services, see Roadmap for Developing Reliable Web Services and Clients.

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. WebLogic Server delivers a set of pre-packaged WS-Policy files, as described in Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection.

Follow the following guidelines when using the `@Policy` annotation for web service reliable messaging:

- 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)
    ```

    In this example, the `ReliableHelloWorldPolicy.xml` file is located in the same directory as the JWS file.
  - To specify one of the pre-packaged WS-Policy files or a WS-Policy file that is packaged in a shared Java EE library, 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.

> **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 in *Developing Applications for Oracle WebLogic Server* for information about creating libraries and setting up your environment so the web service can locate the policy files.

  - To specify that the policy file is published 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,
        attachToWsdl=true)
```

- By default, WS-Policy files are applied to both the request (inbound) and response (outbound) SOAP messages. You can change this default behavior with the `direction` attribute by setting the attribute to `Policy.Direction.inbound` or `Policy.Direction.outbound`.
- You can specify whether the web service requires the operations to be invoked reliably and have the responses delivered reliably using the `wsp:optional` attribute within the policy file specified by `uri`.

  Please note:

  - If the client uses synchronous transport to invoke a web service, and the inbound direction of the operation requires reliability (`optional` attribute is `false`), the client must provide an *offer sequence* (`<wsrm: Offer...>` as described in the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`) for use when sending reliable responses.
  - If the client uses asynchronous transport, the client is not required to send an offer sequence. If a request is made reliably, and the outbound direction has any RM policy (optional or not), the reliable messaging runtime will enforce the handshaking of a *new* RM sequence for sending the response. This new sequence will be associated with the request sequence, and all responses from that point onward are sent on the new response sequence. The response sequence is negotiated with the endpoint indicated by the ReplyTo address of the request.
- Set the `attachToWsdl` 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`.

For more information about the `@Policy` annotation, see weblogic.jws.Policy in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Example 14-1 shows a simple JWS file that implements a reliable web service.

Example 14-1 Example of a Reliable Web Service

```
import javax.jws.WebService;

import weblogic.jws.Policies;
import weblogic.jws.Policy;

/**
 * Example web service for reliable client best practice examples
 */
@WebService
// Enable RM on this service.
@Policies( { @Policy(uri = "policy:DefaultReliability1.2.xml") })
public class BackendReliableService {

  public String doSomething(String what) {

    System.out.println("BackendReliableService doing: " + what);

    return "Did (Reliably) '" + what + "' at: " + System.currentTimeMillis();
  }
}
```

In the example, the predefined `DefaultReliability1.2.xml` policy 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 `doSomething()` operation can be invoked reliably. The policy file is applied to both request and response by default. For information about the pre-packaged policies available and creating a custom policy, see Creating the Web Service Reliable Messaging WS-Policy File.

Invoking a Reliable Web Service from a Web Service Client

> **Note:**
>
> For best practices for developing reliable web service clients, see Roadmap for Developing Reliable Web Service Clients.

The following table summarizes how to invoke a reliable web service from a web service client based on the transport type that you want to employ. For a description of transport types, see Table 14-2.

Table 14-10 Invoking a Reliable Web Service Based on Transport Type

Transport Type	Description
Asynchronous transport	To use asynchronous transport, perform the following steps: 1. Implement the web service client, as described in Table 12-3. In step 3 of Table 12-3, implement one of the following transport mechanisms, depending on whether the client is behind a firewall or not: -Asynchronous client transport feature, as described in Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport). - Make Connection if the client is behind a firewall, as described in Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection). 2. Invoke the web service using either asynchronous or synchronous invocation semantics. **Note**: You can invoke synchronous operations when asynchronous client transport or Make Connection is enabled, as described in Configuring Asynchronous Client Transport for Synchronous Operations and Configuring Make Connection as the Transport for Synchronous Methods.
Synchronous transport	To use synchronous transport, invoke an asynchronous or synchronous method on the reliable messaging service port instance using the standard JAX-WS Reference Implementation, as described in Using the JAX-WS Reference Implementation. **Note**: If you attempt to invoke a buffered web service using synchronous transport, one of following will result: • If this is the first request of the sequence, the destination sequence will be set to be non-buffered (as though the web service configuration was set as non-buffered). • If this is not the first request of the sequence (that is, the client sent a request using asynchronous transport previously), then the request is rejected and a fault returned.

For additional control on the client side, you may wish to perform one or more of the following tasks:

- Configure reliable messaging on the client side, as described in Configuring Reliable Messaging.
- Implement the reliability error listener to receive notifications if a reliable delivery fails, as described in Implementing the Reliability Error Listener. Oracle recommends that you always implement the reliability error listener as a best practice.
- Perform common life cycle tasks on the reliable messaging sequence, such as set configuration options, get the reliable sequence id, and terminate a reliable sequence, as described in Managing the Life Cycle of a Reliable Message Sequence.

Configuring Reliable Messaging

> **Note:**
>
> For best practices for configuring reliable web services, see Roadmap for Developing Reliable Web Services and Clients.

You can configure properties for a reliable web service and client at the WebLogic Server, web service endpoint, or web service client level.

The properties that you define at the WebLogic Server level apply to all reliable web services and clients on that server. For information about configuring reliable messaging at the WebLogic Server level, see Configuring Reliable Messaging on WebLogic Server.

If desired, you can override the reliable message configuration options defined at the server level, as follows:

- At the web service endpoint level by updating the application *deployment plan*. The deployment plan associates new values with specific locations in the descriptors for your application, and is stored in the `weblogic-webservices.xml` descriptor. At deployment time, a deployment plan is merged with the descriptors in the application by applying the values in its variable assignments to the locations in the application descriptors to which the variables are linked. For more information, see Configuring Reliable Messaging on the Web Service Endpoint.
- At the web service client level, as described in Configuring Reliable Messaging on Web Service Clients.

The following sections describe how to configure reliable messaging at the WebLogic Server, web service endpoint, and web service client levels.

- Configuring Reliable Messaging on WebLogic Server
- Configuring Reliable Messaging on the Web Service Endpoint
- Configuring Reliable Messaging on Web Service Clients
- Configuring the Base Retransmission Interval
- Configuring the Retransmission Exponential Backoff
- Configuring the Sequence Expiration
- Configuring Inactivity Timeout
- Configuring a Non-buffered Destination for a Web Service
- Configuring the Acknowledgement Interval
- Implementing the Reliability Error Listener

Configuring Reliable Messaging on WebLogic Server

You can configure reliable messaging on WebLogic Server using the WebLogic Server Administration Console or WLST, as described in the following sections.

- Using the Administration Console
- Using WLST

Using the Administration Console

To configure reliable messaging for WebLogic Server using the WebLogic Server Administration Console:

1. Invoke the WebLogic Server Administration Console, as described in Using the Administration Console in *Understanding WebLogic Web Services for Oracle WebLogic Server*.

2. In the left navigation pane, select **Environment**, then **Servers**.
3. Select the **Configuration** tab and in the Server tables, click on the name of the server for which you want to configure reliable messaging.
4. Click the **Configuration** tab, then the **Web Services** tab, then the **Reliable Message** tab.
5. Edit the reliable messaging properties, as described in the following sections:
   - Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint
   - Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint
   - Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint
   - Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint
   - Configuring a Non-buffered Destination for a Web Service
   - Configuring the Acknowledgement Interval
6. Click **Save**.

For more information, see Web Service Reliable Messaging in the *Oracle WebLogic Server Administration Console Online Help*.

Using WLST

Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

Configuring Reliable Messaging on the Web Service Endpoint

By default, web service endpoints use the reliable messaging configuration defined for the server. You can override the reliable messaging configuration used by the web service endpoint using the WebLogic Server Administration Console, as follows:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console, as described in Invoking the Administration Console in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
2. In the left navigation pane, select **Deployments**.
3. Click the name of the web service in the Deployments table.
4. Select the **Configuration** tab, then the Port Components tab.
5. Click the name of the web service endpoint in the Ports table.
6. Select the **Reliable Message** tab.

7. Click **Customize Reliable Message Configuration** and follow the instructions to save the deployment plan, if required.
8. Edit the reliable messaging properties, as described in the following sections:
   - Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint
   - Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint
   - Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint
   - Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint
   - Configuring a Non-buffered Destination for a Web Service
   - Configuring the Acknowledgement Interval
9. Click **Save**.

For more information, see Configure Web Service Reliable Messaging in the *Oracle WebLogic Server Administration Console Online Help*.

Configuring Reliable Messaging on Web Service Clients

For general information about configuring reliable messaging on web service clients, see Configuring Web Service Clients.

For information about using the `weblogic.wsee.reliability2.api.WsrmClientInitFeature` when creating a web services reliable messaging client, refer to the following sections:

- Configuring the Base Retransmission Interval on the Web Service Client
- Configuring the Retransmission Exponential Backoff on the Web Service Client
- Configuring the Sequence Expiration on the Web Service Client
- Configuring the Inactivity Timeout on the Web Service Client

Configuring the Base Retransmission Interval

If the source endpoint does not receive an acknowledgement for a given message within the specified base retransmission interval, the source endpoint retransmits the message. The source endpoint can modify this retransmission interval at any point during the lifetime of the sequence of messages.

This interval can be used in conjunction with the retransmission exponential backoff, described in Configuring the Retransmission Exponential Backoff, to specify the algorithm that is used to adjust the retransmission interval.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `nY` specifies the number of years, `nM` specifies the number of months, `nD` specifies the number of days, `T` is the date/time separator, `nH` specifies the number of hours, `nM` specifies the number of minutes, and `nS` specifies the number of seconds. This value defaults to `P0DT5S` (5 seconds).

The following sections describe how to configure the base retransmission interval:

- Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint
- Configuring the Base Retransmission Interval on the Web Service Client

Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint

To configure the retransmission exponential backoff on WebLogic Server or the web service endpoint level using the WebLogic Server Administration Console, perform the following steps:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console and access the web service reliable messaging pages at the server-level or web service endpoint level, as described in the following sections, respectively:
   - Configuring Reliable Messaging on WebLogic Server
   - Configuring Reliable Messaging on the Web Service Endpoint
2. Set the **Base Retransmission Interval** value, as required.

Configuring the Base Retransmission Interval on the Web Service Client

> **Note:**
>
> For more information about configuring web service clients, see Configuring Web Service Clients.

Table 14-11 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring the interval of time that must pass before a message is retransmitted to the RM destination.

Table 14-11 Methods for Configuring the Base Retransmission Interval

Method	Description
`String getBaseRetransmissionInterval()`	Gets the base retransmission interval.
`void setBaseRetransmissionInterval(String interval)`	Sets the base retransmission interval.

In the following example, the base retransmission interval is set to 3 hours.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
```

```
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setBaseRetransmissionInterval("P0DT3H");
    port = service.getMyReliableServicePort(initFeature);
...
```

The base retransmission interval configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
      <stub-property>
        <name>weblogic.wsee.wsrm.BaseRetransmissionInterval</name>
        <value>PT30S</value>
      </stub-property>
...
  </port-info>
</service-reference-description>
```

Configuring the Retransmission Exponential Backoff

The retransmission exponential backoff is used in conjunction with the base retransmission interval, described in Configuring the Base Retransmission Interval. If a destination endpoint does not acknowledge a sequence of messages for the time interval specified by the base retransmission interval, the exponential backoff algorithm is used for timing successive retransmissions by the source endpoint, should the message continue to go unacknowledged.

The exponential backoff algorithm specifies that successive retransmission intervals should increase exponentially, based on the base retransmission interval. For example, if the base retransmission interval is 2 seconds, and the exponential backoff element is set, successive retransmission intervals if messages continue to go unacknowledged are 2, 4, 8, 16, 32, and so on.

By default, this flag is disabled (false), indicating that the same retransmission interval is used in successive retries; the interval does not increase exponentially.

The following sections describe how to configure the retransmission exponential backoff:

- Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint
- Configuring the Retransmission Exponential Backoff on the Web Service Client

Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint

To configure the retransmission exponential backoff on WebLogic Server or the web service endpoint level using the WebLogic Server Administration Console, perform the following steps:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console and access the web service reliable messaging pages at the server-level or web service endpoint level, as described in the following sections, respectively:
   - Configuring Reliable Messaging on WebLogic Server
   - Configuring Reliable Messaging on the Web Service Endpoint
2. Set the **Enable Retransmission Exponential Backoff** flag, as required.

Configuring the Retransmission Exponential Backoff on the Web Service Client

> **Note:**
>
> For more information about configuring web service clients, see Configuring Web Service Clients.

Table 14-12 defines the `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring whether the message retransmission interval will be adjusted using the retransmission exponential backoff algorithm.

Table 14-12 Methods for Configuring the Retransmission Exponential Backoff

Method	Description
`Boolean isRetransmissionExponentialBackoff()`	Indicates whether retransmission exponential backoff is enabled.
`void setBaseRetransmissionExponentialBackoff(boolean value)`	Specifies whether base retransmission exponential backoff is enabled. Valid values are `true` or `false`.

In the following example, the retransmission exponential backoff is enabled.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
```

```
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setBaseRetransmissionInterval("P0DT3H");
    initFeature.setBaseRetransmissionExponentialBackoff(true);
    port = service.getMyReliableServicePort(initFeature);
...
```

The retransmission exponential backoff configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
            <name>weblogic.wsee.wsrm.RetransmissionExponentialBackoff</name>
            <value>true</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

Configuring the Sequence Expiration

The sequence expiration specifies the expiration time for a sequence regardless of activity.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `nY` specifies the number of years, `nM` specifies the number of months, `nD` specifies the number of days, `T` is the date/time separator, `nH` specifies the number of hours, `nM` specifies the number of minutes, and `nS` specifies the number of seconds. This value defaults to `P1D` (1 day).

The following sections describe how to configure the sequence expiration:

- Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint
- Configuring the Sequence Expiration on the Web Service Client

Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint

To configure the sequence expiration on WebLogic Server or the web service endpoint level using the WebLogic Server Administration Console, perform the following steps:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console and access the web service reliable messaging pages at the server-level or web service endpoint level, as described in the following sections, respectively:
   - Configuring Reliable Messaging on WebLogic Server
   - Configuring Reliable Messaging on the Web Service Endpoint
2. Set the **Sequence Expiration** value, as required.

Configuring the Sequence Expiration on the Web Service Client

> **Note:**
>
> For more information about configuring web service clients, see Configuring Web Service Clients.

Table 14-13 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for expiration time for a sequence regardless of activity.

Table 14-13 Methods for Configuring Sequence Expiration

Method	Description
`String getSequenceExpiration()`	Returns the sequence expiration currently configured.
`void setSequenceExpiration(String expiration)`	Expiration time for a sequence regardless of activity.

In the following example, the sequence expiration is set to 36 hours.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setSequenceExpiration("P0DT36H");
```

```
    port = service.getMyReliableServicePort(initFeature);
...
```

The sequence expiration configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
            <name>weblogic.wsee.wsrm.SequenceExpiration</name>
            <value>PT10M</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

Configuring Inactivity Timeout

If, during the inactivity timeout interval, an endpoint (the RM source or destination) has not received messages application or protocol messages, the endpoint may consider the RM sequence to have been terminated due to inactivity.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `P0DT600S` (600 seconds).

The following sections describe how to configure the inactivity timeout:

- Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint
- Configuring the Inactivity Timeout on the Web Service Client

Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint

To configure the inactivity timeout on WebLogic Server or the web service endpoint level using the WebLogic Server Administration Console, perform the following steps:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console and access the web service reliable messaging pages at the server-level or web service endpoint level, as described in the following sections, respectively:
   - Configuring Reliable Messaging on WebLogic Server
   - Configuring Reliable Messaging on the Web Service Endpoint
2. Set the **Inactivity Timeout** value, as required.

Configuring the Inactivity Timeout on the Web Service Client

> **Note:**
>
> For more information about configuring web service clients, see Configuring Web Service Clients.

Table 14-14 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring the inactivity timeout.

Table 14-14 Methods for Configuring Inactivity Timeout

Method	Description
`String getInactivityTimeout()`	Returns the inactivity timeout currently configured.
`void setInactivityTimeout(String timeout)`	Sets the inactivity timeout.

In the following example, the inactivity timeout interval is set to 1 hour.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setInactivityTimeout("P0DT1H");
    port = service.getMyReliableServicePort(initFeature);
...
```

The inactivity timeout configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
            <name>weblogic.wsee.wsrm.InactivityTimeout</name>
            <value>PT5M</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

Configuring a Non-buffered Destination for a Web Service

You can control whether you want to disable message buffering on a particular destination server to control whether buffering is used when receiving messages. You can configure non-buffering on the destination server at the WebLogic Server or web service endpoint level only, not at the web service client level (buffering is enabled by default on a web service client).

> **Note:**
>
> If you configure a non-buffered destination, any web service client that uses `@WebServiceRef` to define a reference to the configuration will receive responses without buffering them.
>
> The non-buffered destination configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
           <name>weblogic.wsee.wsrm.NonBufferedDestination</name>
           <value>true</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

> For more information about `@WebServiceRef`, see Defining a Web Service Reference Using the @WebServiceRef Annotation.

To configure the destination server to disable message buffering, on WebLogic Server or the web service endpoint level using the WebLogic Server Administration Console, perform the following steps:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console and access the web service reliable messaging pages at the server-level or web service endpoint level, as described in the following sections, respectively:
   - Configuring Reliable Messaging on WebLogic Server
   - Configuring Reliable Messaging on the Web Service Endpoint
2. Set the **Non-buffered Destination** value, to configure the destination server, respectively, as required.

> **Note:**
>
> On the source server, message buffering should always be enabled. That is, the **Non-buffered Source** value should always be disabled.

Configuring the Acknowledgement Interval

The acknowledgement interval specifies the maximum interval during which the destination endpoint must transmit a standalone acknowledgement. You can configure the acknowledgement interval at the WebLogic Server or web service endpoint level only, not at the web service client level.

> **Note:**
>
> A web service client that uses `@WebServiceRef` to define a reference to the web service uses the acknowledgement interval value to control the amount of time that the client's response handling will wait until acknowledging responses that it receives. In other words, the client acts like an RM destination when receiving response messages.
>
> The non-buffered destination configuration appears in the `weblogic.xml` file as follows:
>
> ```
> <service-reference-description>
> ...
>    <port-info>
>          <stub-property>
>           <name>weblogic.wsee.wsrm.AcknowledgementInterval</name>
>           <value>PT5S</value>
>          </stub-property>
> ...
>   </port-info>
> </service-reference-description>
> ```
>
> For more information about `@WebServiceRef`, see Defining a Web Service Reference Using the @WebServiceRef Annotation.

A destination endpoint can send an acknowledgement on the return message immediately after it has received a message from a source endpoint, or it can send one separately as a standalone acknowledgement. If a return message is not available to send an acknowledgement, a destination endpoint may wait for up to the acknowledgement interval before sending a standalone acknowledgement. If there are no unacknowledged messages, the destination endpoint may choose not to send an acknowledgement.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`. Table 14-15 describes the duration format fields. This value defaults to `P0DT0.2S` (0.2 seconds).

Table 14-15 Duration Format Description

Field	Description
*n*Y	Number of years (*n*).
*n*M	Number of months (*n*).
*n*D	Number of days (*n*).
T	Date and time separator.
*n*H	Number of hours (*n*).
*n*M	Number of minutes (*n*).
*n*S	Number of seconds (*n*).

To configure the acknowledgement interval, on WebLogic Server or the web service endpoint level using the WebLogic Server Administration Console, perform the following steps:

> **Note:**
>
> Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

1. Invoke the WebLogic Server Administration Console and access the web service reliable messaging pages at the server-level or web service endpoint level, as described in the following sections, respectively:
   - Configuring Reliable Messaging on WebLogic Server
   - Configuring Reliable Messaging on the Web Service Endpoint
2. Set the **Acknowledgement Interval** value, as required.

Implementing the Reliability Error Listener

To receive notifications related to reliability delivery failures in the event that a request cannot be delivered, you can implement the following `weblogic.wsee.reliability2.api.ReliabilityErrorListener` interface:

```
public interface ReliablityErrorListener {

   public void onReliabilityError(ReliabilityErrorContext context);
}
```

Table 14-16 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring the reliability error listener.

Table 14-16 Methods for Configuring the Reliability Error Listener

Method	Description
`ReliabilityErrorListener getReliabilityListener()`	Gets the reliability listener currently configured.
`void setErrorListener(ReliabilityErrorListener errorListener)`	Sets the reliability error listener.

The following provides an example of how to implement and use a reliability error listener in your web service client. This example is excerpted from Example 13-1.

```
import weblogic.wsee.reliability2.api.ReliabilityErrorListener;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    WsrmClientInitFeature rmFeature = new WsrmClientInitFeature();
    features.add(rmFeature);

    ReliabilityErrorListener listener = new ReliabilityErrorListener() {
      public void onReliabilityError(ReliabilityErrorContext context) {

        // At a *minimum* do this
        System.out.println("RM sequence failure: " +
                           context.getFaultSummaryMessage());
        _lastResponse = context.getFaultSummaryMessage();

        // And optionally do this...

        // The context parameter tells you whether a request or the entire
        // sequence has failed. If a sequence fails, you'll get a notification
        // for each undelivered request (if any) on the sequence.
        if (context.isRequestSpecific()) {
          // We have a single request failure (possibly as part of a larger
          // sequence failure).
          // We can get the original request back like this:
          String operationName = context.getOperationName();
          System.out.println("Failed to deliver request for operation '" +
                             operationName + "'. Fault summary: " +
                             context.getFaultSummaryMessage());
          if ("DoSomething".equals(operationName)) {
            try {
              String request = context.getRequest(JAXBContext.newInstance(),
                                                  String.class);
              System.out.println("Failed to deliver request for operation '" +
                                 operationName + "' with content: " +
                                 request);
              Map<String, Serializable> requestProps =
                context.getUserRequestContextProperties();
              if (requestProps != null) {
                // Fetch back any property you sent in
                // JAXWSProperties.PERSISTENT_CONTEXT when you sent the
                // request.
                String myProperty = (String)requestProps.get(MY_PROPERTY);
                System.out.println(myProperty + " failed!");
              }
            } catch (Exception e) {
```

```
            e.printStackTrace();
          }
        }
      } else {
        // The entire sequence has encountered an error.
        System.out.println("Entire sequence failed: " +
                           context.getFaultSummaryMessage());

      }
    }
  };

  rmFeature.setReliabilityErrorListener(listener);

  _features = features.toArray(new WebServiceFeature[features.size()]);

  BackendReliableService anotherPort =
    _service.getBackendReliableServicePort(_features);
...
```

Managing the Life Cycle of a Reliable Message Sequence

WebLogic Server provides a client API, `weblogic.wsee.reliability2.api.WsrmClient`, for use with the web service reliable messaging. Use this API to perform common life cycle tasks such as set configuration options, get the reliable sequence id, and terminate a reliable sequence.

An instance of the `WsrmClient` API can be accessed from the reliable web service port using the `weblogic.wsee.reliability2.api.WsrmClientFactory` method, as follows:

```
package wsrm_jaxws.example;
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    port = service.getReliableEchoPort();
    WsrmClient wsrmClient = WsrmClientFactory.getWsrmClientFromPort(port);
...
```

The following sections describe how to manage the life cycle of a reliable message sequence using `WsrmClient`.

- Managing the Reliable Sequence
- Managing the Client ID
- Managing the Acknowledged Requests
- Accessing Information About a Message
- Identifying the Final Message in a Reliable Sequence
- Closing the Reliable Sequence

- Terminating the Reliable Sequence
- Resetting a Client to Start a New Message Sequence

For complete details on the web service reliable messaging client API, see `weblogic.wsee.reliability2.api.WsrmClient` in *Java API Reference for Oracle WebLogic Server*.

Managing the Reliable Sequence

To manage the reliable sequence, you can perform one or more of the following tasks.

- Get and set the reliable sequence ID, as described in Getting and Setting the Reliable Sequence ID.
- Access the state of the reliable sequence, for example, to determine if it is active or terminated, as described in Accessing the State of the Reliable Sequence.

Getting and Setting the Reliable Sequence ID

The sequence ID is used to identify a specific reliable sequence. You can get and set the sequence ID using the `weblogic.wsee.reliability2.api.WsrmClient.getSequenceID()` and `weblogic.wsee.reliability2.api.WsrmClient.setSequenceID()` methods, respectively. If no messages have been sent when you issue the `getSequenceID()` method, the value returned is null.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
   _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   // Will be null
   String sequenceId = rmClient.getSequenceId();
   // Send first message
   anotherPort.doSomething("Bake a cake");
   // Will be non-null
   sequenceId = rmClient.getSequenceId();
```

During recovery from a server failure, you can set the reliable sequence on a newly created web service port or dispatch instance after a client or server restart. Setting the sequence ID for a client instance is an advanced feature. Advanced clients may use `setSequenceId` to connect a client instance to a known RM sequence.

Accessing the State of the Reliable Sequence

To access the state of a sequence, use `weblogic.wsee.reliability2.api.WsrmClient.getSequenceState()`. This method returns an `java.lang.Enum` constant of the type `weblogic.wsee.reliability2.api.SequenceState`.

The following table defines valid values that may be returned for sequence state.

Table 14-17 Sequence State Values

Sequence State	Description
CLOSED	Reliable sequence is closed. **Note:** Closing a sequence should be considered a *last resort*, and only to prepare to close down a reliable messaging sequence for which you do not expect to receive the full range of requests. For more information, see Closing the Reliable Sequence.
CLOSING	Reliable sequence is in the process of being closed. **Note:** Closing a sequence should be considered a *last resort*, and only to prepare to close down a reliable messaging sequence for which you do not expect to receive the full range of requests. For more information, see Closing the Reliable Sequence.
CREATED	Reliable sequence has been created and the initial handshaking is complete.
CREATING	Reliable sequence is being created; the initial handshaking is in progress.
LAST_MESSAGE	**Deprecated**. WS-ReliableMessaging 1.0 only. The last message in the sequence has been received.
LAST_MESSAGE_PENDING	**Deprecated**. WS-ReliableMessaging 1.0 only. The last message in the sequence is pending.
NEW	Reliable sequence is in its initial state. Initial handshaking has not started.
TERMINATED	Reliable sequence is terminated. Under normal processing, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated. Though not recommended, you can force the termination of a reliable sequence, as described in Terminating the Reliable Sequence.
TERMINATING	Reliable sequence is in the process of being terminated. Under normal processing, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated. Though not recommended, you can force the termination of a reliable sequence, as described in Terminating the Reliable Sequence.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
import weblogic.wsee.reliability2.api.SequenceState;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
        _service.getBackendReliableServicePort(_features);
...
```

```
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   SequenceState rmState = rmClient.getSequenceState();
   if (rmState == SequenceState.TERMINATED) {
         ... Do some work or log a message ...
   }
...
```

Managing the Client ID

The client ID identifies the web service client. Each client has its own unique ID. The client ID can be used to access saved requests that may exist for a reliable sequence after a client or server restart.

The client ID is configured automatically by WebLogic Server. You can set the client ID to a custom value when creating the port using the `weblogic.wsee.jaxws.persistence.ClientIdentityFeature`. For more information, see Managing Client Identity.

Reliable messaging uses the client ID to find any requests that were sent prior to a VM restart that were not sent before the VM exited. When you establish the first client instance using the prior client ID, reliable messaging uses the resources associated with that port to begin sending requests on behalf of the restored client ID.

You can get the client ID using the `weblogic.wsee.reliability2.api.WsrmClient.getID()` method.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
      _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   String clientId = rmClient.getId();
...
```

Managing the Acknowledged Requests

Use the `weblogic.wsee.reliability2.api.WsrmClient.ackRanges()` method to display the requests that have been acknowledged during the life cycle of a reliable message sequence. The `ackRanges()` method returns a set of `weblogic.wsee.reliability.MessageRange` objects.

After reviewing the range of requests that have been acknowledged, the client may choose to:

- Send an acknowledgement request to the RM destination using the `weblogic.wsee.reliability2.api.WsrmClient.requestAcknowledgement()` method.

- Close the sequence (see Closing the Reliable Sequence) and perform error handling to account for unacknowledged messages after a specific amount of time.

Note: Clients may call `getAckRanges()` repeatedly, to keep track of the reliable message sequence over time. However, you should take into account that there is a certain level of additional overhead associated each call.

Accessing Information About a Message

Use the `weblogic.wsee.reliability2.api.WsrmClient.getMessageInfo()` method to get information about a reliable message sent from the client based on the message number. This method accepts a long value representing the sequential message number of a request message sent from the client instance, and returns information about the message of type `weblogic.wsee.reliability2.sequence.SourceMessageInfo`. You can use the `WsrmClient.getMostRecentMessageNumber()` method to determine the maximum value of the message number value to pass to `getMessageInfo()`.

The returned `SourceMessageInfo` object should be treated as immutable, and only the get methods should be used.

The following table list the `SourceMessageInfo` methods that you can use to access specific details about the source message.

Table 14-18 Methods for SourceMessageInfo()

Method	Description
`getMessageID()`	Gets the message ID as a String value.
`getMessageNum()`	Gets the number of the message as a long value.
`getResponseMessageInfo()`	Returns a `weblogic.wsee.reliability2.sequence.DestinationMessageInfo` object representing the response that has been correlated to the request represented by the current `SourceMessageInfo()` object. Returns NULL if no response has been received for this request or if none is expected (for example, request was one way).
`isAck()`	Indicates whether the message has been acknowledged.

The following table lists the `DestinationMessageInfo` methods that you can use to access specific details about the destination message.

Table 14-19 Methods for DestinationMessageInfo()

Method	Description
`getMessageID()`	Gets the message ID as a String value.
`getMessageNum()`	Gets the number of the message as a long value.

The `getMessageInfo()` method can be used in conjunction with `weblogic.wsee.reliability2.api.WsrmClient.getMostRecentMessageNumber()` to obtain information about the most recently sent reliable message. This method returns

a monotonically increasing long value, starting from 1. This method will return -1 in the following circumstances:

- If the reliable sequence ID has not been established (`getSequenceID()` returns null).
- The first reliable message has not been sent yet.
- The reliable sequence has been terminated.

Identifying the Final Message in a Reliable Sequence

Because WebLogic Server retains resources associated with the reliable sequence, it is recommended that you take steps to release these resources in a timely fashion. Under normal circumstances, a reliable sequence should be retained until all messages have been sent and acknowledged by the RM destination. To facilitate the timely and proper termination of a sequence, it is recommended that you identify the final message in a reliable message sequence. Doing so indicates you are done sending messages to the RM destination and that WebLogic Server can begin looking for the final acknowledgement before automatically terminating the reliable sequence. Indicate the final message using the `weblogic.wsee.reliability2.api.WsrmClient.setFinalMessage()` method.

When you identify a final message, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated, and all resources are released. Otherwise, the sequence is terminated automatically after the configured sequence expiration period is reached.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
   ...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
        _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   anotherPort.doSomething("One potato");
   anotherPort.doSomething("Two potato");
   anotherPort.doSomething("Three potato");
   // Indicate this next invoke marks the 'final' message for the sequence
   rmClient.setFinalMessage();
   anotherPort.doSomething("Four");
...
```

Closing the Reliable Sequence

Use the `weblogic.wsee.reliability2.api.WsrmClient.closeMessage()` to close a reliable messaging sequence.

> **Note:**
>
> This method is valid for WS-ReliableMessaging 1.1 only; it is not supported for WS-ReliableMessaging 1.0.

When a reliable messaging sequence is closed, no new messages will be accepted by the RM destination or sent by the RM source. A closed sequence is still tracked by the RM destination and continues to service acknowledgment requests against it. It allows the RM source to get a full and final accounting of the reliable messaging sequence before terminating it.

Note: Closing a sequence should be considered a *last resort*, and only to prepare to close down a reliable messaging sequence for which you do not expect to receive the full range of requests. For example, after reviewing the range of requests that have been acknowledged (see Managing the Acknowledged Requests), the client may decide it necessary to close the sequence and perform error handling to account for unacknowledged messages after a specific amount of time.

Once a reliable messaging sequence is closed, it is up to the client to terminate the sequence; it will no longer be terminated automatically by the server after a configured timeout has been reached. See Terminating the Reliable Sequence.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
      _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   anotherPort.doSomething("One potato");
   anotherPort.doSomething("Two potato");
   // ... Wait some amount of time, and check for acks
   // ... using WsrmClient.getAckRanges() ...
   // ... If we don't find all of our acks ...
   rmClient.closeSequence();
   // ... Do some error recovery like telling our
   // ... client we couldn't deliver all requests ...
   rmClient.terminateSequence();
...
```

Terminating the Reliable Sequence

Although not recommended, you can terminate the reliable message sequence regardless of whether all messages have been acknowledged using the `weblogic.wsee.reliability2.api.WsrmClient.terminateSequence()` method.

> **Note:**
>
> It is recommended that, instead, you use the `setFinalMessage()` method to identify the final message in a reliable sequence. When you identify a final message, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated, and all resources are released. For more information, see Identifying the Final Message in a Reliable Sequence.

Terminating a sequence causes the RM source and RM destination to remove all state associated with that sequence. The client can no longer perform any action on a terminated sequence. When a sequence is terminated, any pending requests being delivered through server-side retry (SAF agents) for the sequence are rejected and sent as a notification on the `ReliablityErrorListener`.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
      _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   anotherPort.doSomething("One potato");
   anotherPort.doSomething("Two potato");
   // ... Wait some amount of time, and check for acks
   // ... using WsrmClient.getAckRanges() ...
   // ... If we don't find all of our acks ...
   rmClient.closeSequence();
   // ... Do some error recovery like telling our
   // ... client we couldn't deliver all requests ...
   rmClient.terminateSequence();
...
```

Resetting a Client to Start a New Message Sequence

Use the `weblogic.wsee.reliability2.api.WsrmClient.reset()` method to clear all `RequestContext` properties related to reliable messaging that do not need to be retained once the reliable sequence is closed. Typically, this method is called when you want to initiate another sequence of reliable messages from the same client.

For an example of using `reset()`, see Example B-1.

Monitoring Web Services Reliable Messaging

You can monitor reliable messaging sequences for a web service or client using the WebLogic Server Administration Console. For each reliable messaging sequence, runtime monitoring information is displayed, such as the sequence state, the source and destination

servers, and so on. You can customize the information that is shown in the table by clicking **Customize this table**.

In particular, you can use the monitoring pages to determine:

- Whether or not you are cleaning up sequences in a timely fashion. If you view a large number of sequences in the monitoring tab, you may wish to review your client code to determine why.
- Whether an individual sequence has unacknowledged requests, or has not received expected responses.

To monitor reliable messaging sequences for a web service, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service is packaged. Expand the application by clicking the **+** node; the web services in the application are listed under the **Web Services** category. Click on the name of the web service and select **Monitoring> Ports> Reliable Messaging**.

To monitor reliable messaging sequences for a web service client, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service client is packaged. Expand the application by clicking the **+** node and click on the application module within which the web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab. Then click **Monitoring> Servers> Reliable Messaging**.

Grouping Messages into Business Units of Work (Batching)

Often, the messages flowing between a web service client and service are part of a single business transaction or *unit of work*. An example might be a travel agency reservation process that requires messages between the agency, airline, hotel, and rental car company. All of the messages flowing between any two endpoints could be considered a business unit of work.

Reliable messaging is tailored to handling messages related to a unit of work by grouping them into an RM sequence. The entire unit of work (or sequence) is treated as a whole, and error recovery, and so on can be applied to the entire sequence (see the `IncompleteSequenceBehavior` element description in the WS-ReliableMessaging 1.2 specification (February 2009) at `http://docs.oasis-open.org/ws-rx/wsrm/200702`). For example, an RM sequence can be configured to discard requests that occur after a gap in the sequence, or to discard the entire sequence of requests if any request is missing from the sequence.

You can indicate that a message is part of a business unit of work by creating a new client instance before sending the first message in the unit, and by disposing of the client instance after the last message in the unit. Alternatively, you can use the `WsrmClient` API (obtained by passing a client instance to the `WsrmClientFactory.getWsrmClientFromPort()` method) to identify the *final* request in a sequence is about to be sent. This is done by calling `WsrmClient.setFinalMessage()` just before performing the invoke on the client instance, as described in Identifying the Final Message in a Reliable Sequence.

There is some significant overhead associated with the RM protocol. In particular, creating and terminating a sequence involves a round-trip message exchange with the service (RM destination). This means that *four* messages must go across the wire to establish and then terminate an RM sequence. For this reason, it is to your advantage

to send the requests within a single business unit of work on a single RM sequence. This allows you to amortize the cost of the RM protocol overhead over a number of business messages.

In some cases, the client instance being used to talk to the reliable service runs in an environment where there is no intrinsic notion of the business unit of work to which the messages belong. An example of this is an intermediary such as a message broker. In this case, the broker is often aware only of the message itself, and not the context in which the message is being sent. The broker may not do anything to demarcate the start and end of a business unit of work (or sequence); as a result, when using reliable messaging to send requests, the broker will incur the RM sequence creation and termination protocol overhead for *every* message it sends. This can result in a serious negative performance impact.

In cases where no intrinsic business unit of work is known for a message, you can choose to arbitrarily group (or batch) messages into an artificially created unit of work (called a *batch*). Batching of reliable messages can overcome the performance impact described above and can be used to tune and optimize network usage and throughput between a reliable messaging client and service. Testing has shown that batching otherwise unrelated requests into even small batches (say 10 requests) can as much as triple the throughput between the client and service when using reliable messaging (when sending small messages).

> **Note:**
>
> Oracle does not recommend batching requests that already have an association with a business unit of work. This is because error recovery can become complicated when RM sequence boundaries and unit of work boundaries do not match. For example, when you add a `ReliabilityErrorListener` to your client instance (via `WsrmClientInitFeature`), as described in Implementing the Reliability Error Listener, this listener can be used to perform error recovery for single requests in a sequence or whole-sequence failures. When batching requests, this error recovery logic would need to store some information about each request in order to properly handle the failure of a request. A client that does not employ batching will likely have more context about the request given the business unit of work it belongs to.

The following code excerpt shows an example class called `BatchingRmClientWrapper` that can be used to make batching of RM requests simple and effective. This class batches requests into groups of a specified number of requests. It allows you to create a dynamic proxy that takes the place of your regular client instance. When you make invocations on the client instance, the batching wrapper seamlessly groups the outgoing requests into batches, and assigns each batch its own RM sequence. The batching wrapper also takes a duration specification that indicates the maximum lifetime of any given batch. This allows incomplete batches to be completed in a timely fashion even if there are not enough outgoing requests to completely fill a batch. If the batch has existed for the maximum lifetime specified, it will be closed as if the last message in the batch had been sent.

An example of the client wrapper class that can be used for batching reliable messaging is provided in Example Client Wrapper Class for Batching Reliable Messages. You can use this class as-is in your own application code, if desired.

Example 14-2 Example of Grouping Messages into Units of Work (Batching)

```
import java.io.IOException;
import java.util.*;
```

```
import java.util.*;

import javax.servlet.*;
import javax.xml.ws.*;

import weblogic.jws.jaxws.client.ClientIdentityFeature;
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
import weblogic.jws.jaxws.client.async.AsyncClientTransportFeature;
import weblogic.wsee.reliability2.api.ReliabilityErrorContext;
import weblogic.wsee.reliability2.api.ReliabilityErrorListener;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;

/**
 * Example client for invoking a reliable web service and 'batching' requests
 * artificially into a sequence. A wrapper class called
 * BatchingRmClientWrapper is called to begin and end RM sequences for each batch of
 * requests. This avoids per-message RM sequence handshaking
 * and termination overhead (delivering better performance).
 */
public class BestPracticeAsyncRmBatchingClient
  extends GenericServlet {

  private BackendReliableServiceService _service;
  private BackendReliableService _singletonPort;
  private BackendReliableService _batchingPort;

  private static int _requestCount;
  private static String _lastResponse;

  @Override
  public void init()
    throws ServletException {

    _requestCount = 0;
    _lastResponse = null;

    // Only create the web service object once as it is expensive to create repeatedly.
    if (_service == null) {
      _service = new BackendReliableServiceService();
    }

    // Best Practice: Use a stored list of features, per client ID, to create client instances.
    // Define all features for the web service port, per client ID, so that they are
    // consistent each time the port is called. For example:
    // _service.getBackendServicePort(_features);

    List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

    // Best Practice: Explicitly define the client ID.
    ClientIdentityFeature clientIdFeature =
      new ClientIdentityFeature("MyBackendServiceAsyncRmBatchingClient");
    features.add(clientIdFeature);

    // Best Practice: Always implement a reliability error listener.
    // Include this feature in your reusable feature list. This enables you to determine
    // a reason for failure, for example, RM cannot deliver a request or the RM sequence fails in
    // some way (for example, client credentials refused at service).
    WsrmClientInitFeature rmFeature = new WsrmClientInitFeature();
    features.add(rmFeature);
    rmFeature.setErrorListener(new ReliabilityErrorListener() {
      public void onReliabilityError(ReliabilityErrorContext context) {
```

```
    // At a *minimum* do this
    System.out.println("RM sequence failure: " +
                       context.getFaultSummaryMessage());
    _lastResponse = context.getFaultSummaryMessage();
  }
});

// Asynchronous endpoint
AsyncClientTransportFeature asyncFeature =
  new AsyncClientTransportFeature(getServletContext());
features.add(asyncFeature);

// Best Practice: Define a port-based asynchronous callback handler,
// AsyncClientHandlerFeature, for asynchronous and dispatch callback handling.
BackendReliableServiceAsyncHandler handler =
  new BackendReliableServiceAsyncHandler() {
    public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
      // ... Handle Response ...
      try {
        DoSomethingResponse response = res.get();
        _lastResponse = response.getReturn();
        System.out.println("Got reliable/async/batched response: " + _lastResponse);
      } catch (Exception e) {
        _lastResponse = e.toString();
        e.printStackTrace();
      }
    }
  };
AsyncClientHandlerFeature handlerFeature =
  new AsyncClientHandlerFeature(handler);
features.add(handlerFeature);

// Set the features used when creating clients with
// this client ID "MyBackendServiceAsyncRmBatchingClient"

WebServiceFeature[] featuresArray =
  features.toArray(new WebServiceFeature[features.size()]);

// Best Practice: Define a singleton port instance and initialize it when
// the client container initializes (upon deployment).
// The singleton port will be available for the life of the servlet.
// Creation of the singleton port triggers the asynchronous response endpoint to be published
// and it will remain published until our container (Web application) is undeployed.
// Note, we will get a call to destroy() before this.
_singletonPort = _service.getBackendReliableServicePort(featuresArray);

// Create a wrapper class to 'batch' messages onto RM sequences so
// a client with no concept of which messages are related as a unit can still achieve
// good performance from RM. The class will send a given number of requests on
// the same sequence, and then terminate that sequence before starting
// another to carry further requests. A batch has both a max size and
// lifetime so no sequence is left open for too long.
// The example batches 10 messages or executes for 20 seconds, whichever comes
// first. Assuming there were 15 total requests to send, the class would start and complete
// one full batch of 10 requests, then send the next batch of five requests.
// Once the batch of five requests has been open for 20 seconds, it will be closed and the
// associated sequence terminated (even though 10 requests were not sent to fill the batch).
BackendReliableService batchingPort =
  _service.getBackendReliableServicePort(featuresArray);
BatchingRmClientWrapper<BackendReliableService> batchingSeq
  = new BatchingRmClientWrapper<BackendReliableService>(batchingPort,
```

```
                                                BackendReliableService.class,
                                                10, "PT20S",
                                                System.out);
    _batchingPort = batchingSeq.createProxy();
  }

  @Override
  public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException {

    // TODO: ... Read the servlet request ...

    // For this simple example, echo the _lastResponse captured from
    // an asynchronous DoSomethingResponse response message.

    if (_lastResponse != null) {
      res.getWriter().write(_lastResponse);
      System.out.println("Servlet returning _lastResponse value: " + _lastResponse);
      _lastResponse = null; // Clear the response so we can get another
      return;
    }

    // Synchronize on _batchingPort since it is a class-level variable and it might
    // be in this method on multiple threads from the servlet engine.

    synchronized(_batchingPort) {

      // Use the 'batching' port to send the requests instead of creating a
      // new request each time.
      BackendReliableService port = _batchingPort;

      // Set the endpoint address for BackendService.
      ((BindingProvider)port).getRequestContext().
        put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
            "http://localhost:7001/BestPracticeReliableService/BackendReliableService");

    // Make the invocation. Our asynchronous handler implementation (set
    // into the AsyncClientHandlerFeature above) receives the response.
      String request = "Protected and serve " + (++_requestCount);
      System.out.println("Invoking DoSomething reliably/async/batched with request: " +
                         request);
      port.doSomethingAsync(request);
    }

    // Return a canned string indicating the response was not received
    // synchronously. Client needs to invoke the servlet again to get
    // the response.
    res.getWriter().write("Waiting for response...");
  }

  @Override
  public void destroy() {

    try {
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the singleton port created during initialization. Note, the asynchronous
      // response endpoint generated by creating _singletonPort *remains*
      // published until our container (Web application) is undeployed.
      ((java.io.Closeable)_singletonPort).close();
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the batching port created during initialization. Note, this will close
```

```
      // the underlying client instance used to create the batching port.
      ((java.io.Closeable)_batchingPort).close();

      // Upon return, the Web application is undeployed, and the asynchronous
      // response endpoint is stopped (unpublished). At this point,
      // the client ID used for _singletonPort will be unregistered and will no longer be
      // visible from the Administration Console and WLST.
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

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 message sequence is explicitly ended by the client or as a result of a timeout.

For additional information about production redeployment and web service clients, see Client Considerations When Redeploying a Web Service.

Interoperability with WebLogic Web Service Reliable Messaging

The WebLogic web services reliable messaging implementation will interoperate with the web service reliable messaging implementations provided by the following third-party vendor web services: IBM and Microsoft .NET. For best practices when interoperating with Microsoft .NET, see Interoperability with Microsoft WCF/.NET in *Understanding WebLogic Web Services for Oracle WebLogic Server*.

15

Using Web Services Atomic Transactions

This chapter describes how to use web services atomic transactions for WebLogic web services using Java API for XML Web Services (JAX-WS) to enable interoperability with other external transaction processing systems.
This chapter includes the following sections:

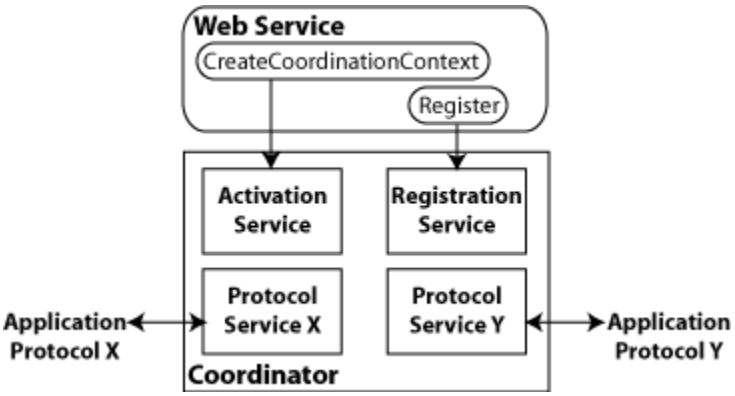
Overview of Web Services Atomic Transactions

WebLogic web services enable interoperability with other external transaction processing systems, such as Websphere, Microsoft .NET, and so on, through the support of the following specifications:

- Web Services Atomic Transaction (WS-AtomicTransaction) Versions 1.0, 1.1, and 1.2: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-cs-01/wstx-wsat-1.2-spec-cs-01.html
- Web Services Coordination (WS-Coordination) Versions 1.0, 1.1, and 1.2: http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-cs-01/wstx-wscoor-1.2-spec-cs-01.html

These specifications define an extensible framework for coordinating distributed activities among a set of participants. The coordinator, shown in the following figure, is the central component, managing the transactional state (coordination context) and enabling web services and clients to register as participants.

Figure 15-1 Web Services Atomic Transactions Framework



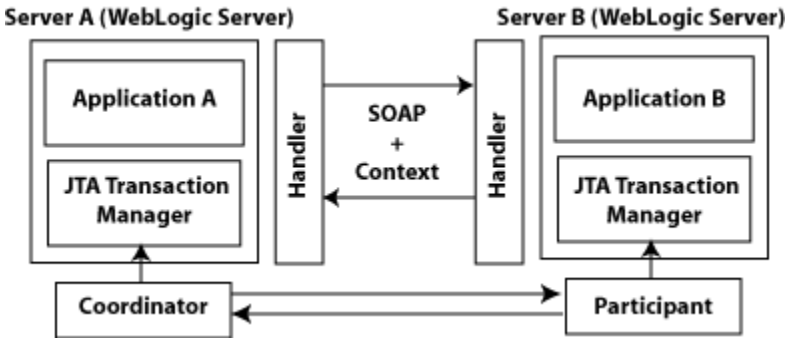
The following table describes the components of web services atomic transactions, shown in the previous figure.

Table 15-1 Components of Web Services Atomic Transactions

Component	Description
Coordinator	Manages the transactional state (coordination context) and enables web services and clients to register as participants.
Activation Service	Enables the application to activate a transaction and create a coordination context for an activity. Once created, the coordination context is passed with the transaction flow.
Registration Service	Enables an application to register as a participant.
Application Protocol X, Y	Supported coordination protocols, such as WS-AtomicTransaction.

The following figure shows two instances of WebLogic Server interacting within the context of a web services atomic transaction. For simplicity, two WebLogic web service applications are shown.

Figure 15-2 Web Services Atomic Transactions in WebLogic Server Environment



Please note the following:

- Using the local JTA transaction manager, a transaction can be imported to or exported from the local JTA environment as a *subordinate transaction*, all within the context of a web service request.
- Creation and management of the coordination context is handled by the local JTA transaction manager.
- All transaction integrity management and recovery processing is done by the local JTA transaction manager.

For more information about JTA, see *Developing JTA Applications for Oracle WebLogic Server.*

The following describes a sample end-to-end web services atomic transaction interaction, illustrated in Figure 15-2:

1. Application A begins a transaction on the current thread of control using the JTA transaction manager on Server A.
2. Application A calls a web service method in Application B on Server B.
3. Server A updates its transaction information and creates a SOAP header that contains the coordination context, and identifies the transaction and local coordinator.
4. Server B receives the request for Application B, detects that the header contains a transaction coordination context and determines whether it has already registered as a participant in this transaction. If it has, that transaction is resumed and if not, a new transaction is started.

   Application B executes within the context of the imported transaction. All transactional resources with which the application interacts are enlisted with this imported transaction.
5. Server B enlists itself as a participant in the WS-AtomicTransaction transaction by registering with the registration service indicated in the transaction coordination context.
6. Server A resumes the transaction.
7. Application A resumes processing and commits the transaction.

Configuring the Domain Resources Required for Web Service Advanced Features

When creating or extending a domain, if you expect that you will be using other web service advanced features in addition to web service atomic transactions (either now or in the future), you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to configure automatically the resources required to support the advanced web service features. Although use of this extension template is not required, it makes the configuration of the required resources much easier. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Server Administration Console or WLST. For more information, see Configuring Your Domain For Advanced Web Services Features.

> **Note:**
>
> If you do not expect to use other web service advanced features with web service atomic transactions, application of this extension template is not required, minimizing start-up times and memory footprint.

Enabling the Web Services Atomic Transactions Feature

Web services atomic transactions feature (WSAT) is disabled by default in WebLogic Server version 12.2.1.0 and later.

> **Note:**
>
> If you do not enable the WSAT feature, WebLogic Server ignores the WSAT configurations that you set using the WSDL and WebLogic Server Administration console. Therefore, before you configure WSAT you must enable it.

You can use WLST both offline and online to enable WSAT as shown in the following examples. See Invoking WLST in *Understanding the WebLogic Scripting Tool*.

Example 15-1 Enabling WSAT Using WLST Online

In the following example, WebLogic Server is running. The arguments *username* and *password* represent the credentials for the user who is connecting WLST to the server.

```
#set flag
bEnabled=true

connect()
// At the system prompts, enter the username, password, and server
URL, for example t3://host:port.
edit()
startEdit()

#get OptionalFeatureDeployment MBean
switcher=cmo.getOptionalFeatureDeployment()
print 'switcher=',switcher
print 'switcher.getOptionalFeatures='
for iam in switcher.getOptionalFeatures():
if iam !=None:
print iam.getName(),'enabled=',iam.isEnabled()

#get an OptionalFeature
abean=switcher.lookupOptionalFeature(WSAT)
if abean !=None:
abean.setEnabled(bEnabled)
else:
abean=switcher.createOptionalFeature(WSAT)
```

```
abean.setEnabled(bEnabled)

save()
activate()
disconnect()
exit()
```

> **Note:**
>
> Restart WebLogic Server after running the WLST script.

Example 15-2 Enabling WSAT Using WLST Offline

In the following example, *domain* represents the path of your domain (for example, *Oracle_Home*/user_projects/domains/*mydomain*). Also note that *Oracle_Home* and *mydomain* and must match the folder name and the domain name. The parameter *ofs* represents the optional features such as WSAT.

```
readDomain('domain')
create('ofs','OptionalFeatureDeployment')
cd('OptionalFeatureDeployment/ofs')
create('WSAT', 'OptionalFeature') cd('OptionalFeature/WSAT')
set('Enabled',true)
updateDomain()
 closeDomain()
exit()
```

Enabling Web Services Atomic Transactions on Web Services

The Web services atomic transactions feature (WSAT) is disabled by default in WebLogic Server version 12.2.1.0 and later.

> **Note:**
>
> If you do not enable the WSAT feature, WebLogic Server ignores the WSAT configurations that you set using the WSDL and WebLogic Server Administration console. Therefore, before you configure WSAT you must enable it. For more information see, Enabling the Web Services Atomic Transactions Feature.

To enable web services atomic transactions on a web service:

- When starting from Java (bottom-up), add the `@weblogic.wsee.wstx.wsat.Transactional` annotation to the web service endpoint implementation class or method. For more information, see Using the @Transactional Annotation in Your JWS File.
- When starting from WSDL (top-down), use `wsdlc` to generate a web service from an existing WSDL file. In this case, The WS-AtomicTransaction policy assertions that are advertised in the WSDL are carried forward and are included in the WSDL file for the new

web service generated by `wsdlc`. See Enabling Web Services Atomic Transactions Starting From WSDL.

- At deployment time, enable and configure web services atomic transactions at the web service endpoint or method level using the WebLogic Server Administration Console. For more information, see Configuring Web Services Atomic Transactions Using the Administration Console.

The following tables summarizes the configuration options that you can set when enabling web services atomic transactions.

Table 15-2 Web Services Atomic Transactions Configuration Options

Attribute	Description
`Version`	Version of the web services atomic transaction coordination context that is used for web services and clients. For clients, it specifies the version used for outbound messages only. The value specified must be consistent across the entire transaction. Valid values include `WSAT10`, `WSAT11`, `WSAT12`, and `DEFAULT`. The `DEFAULT` value for web services is all three versions (driven by the inbound request); the `DEFAULT` value for web service clients is `WSAT10`.
`Flow type`	Whether the web services atomic transaction coordination context is passed with the transaction flow. For valid values, see Table 15-3.

The following table summarizes the valid values for flow type and their meaning on the web service and client. The table also summarizes the valid value combinations when configuring web services atomic transactions for an EJB-style web service that uses the `@TransactionAttribute` annotation.

Table 15-3 Flow Types Values

Value	Web Service Client	Web Service	Valid EJB `@TransactionAttribute` Values
`NEVER`	**JTA transaction**: Do not export transaction coordination context. **No JTA transaction**: Do not export transaction coordination context.	**Transaction flow exists**: Do not import transaction coordination context. If the CoordinationContext header contains `mustunderstand="true"`, a SOAP fault is thrown. **No transaction flow**: Do not import transaction coordination context.	`NEVER`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`
`SUPPORTS` (Default)	**JTA transaction**: Export transaction coordination context. **No JTA transaction**: Do not export transaction coordination context.	**Transaction flow exists**: Import transaction context. **No transaction flow**: Do not import transaction coordination context.	`REQUIRED`, `SUPPORTS`
`MANDATORY`	**JTA transaction**: Export transaction coordination context. **No JTA transaction**: An exception is thrown.	**Transaction flow exists**: Import transaction context. **No transaction flow**: Service-side exception is thrown.	`MANDATORY`, `REQUIRED`, `SUPPORTS`

Using the @Transactional Annotation in Your JWS File

To enable web services atomic transactions, specify the `@weblogic.wsee.wstx.wsat.Transactional` annotation on the web service endpoint implementation class or method.

> **Note:**
>
> This annotation is not to be mistaken with `weblogic.jws.Transactional`, which ensures that the annotated class or operation runs inside of a transaction, but not an *atomic* transaction.

Please note the following:

- If you specify the `@Transactional` annotation at the web service class level, the settings apply to all two-way methods defined by the service endpoint interface. You can override the flow type value at the method level; however, the version must be consistent across the entire transaction.
- You cannot explicitly specify the `@Transactional` annotation on a Web method that is also annotated with `@Oneway`.
- web services atomic transactions cannot be used with the client-side asynchronous programming model.

The format for specifying the `@Transactional` annotation is as follows:

```
@Transactional(
     version=Transactional.Version.[WSAT10|WSAT11|WSAT12|DEFAULT],
     value=Transactional.TransactionFowType.[MANDATORY|SUPPORTS|NEVER]
)
```

For more information about the version and flow type configuration options, see Table 15-2.

The following sections provide examples of using the `@Transactional` annotation at the web service implementation class and method levels, and with the EJB @TransactionAttribute annotation.

- Example: Using @Transactional Annotation on a Web Service Class
- Example: Using @Transactional Annotation on a Web Service Method
- Example: Using the @Transactional and the EJB @TransactionAttribute Annotations Together

Example: Using @Transactional Annotation on a Web Service Class

The following example shows how to add `@Transactional` annotation on a web service class. Relevant code is shown in **bold**. As shown in the example, there is an active JTA transaction.

> **Note:**
>
> The following excerpt is borrowed from the web services atomic transaction example that is delivered with the WebLogic Server Samples Server. For more information, see More Examples of Using Web Services Atomic Transactions.

```
package examples.webservices.jaxws.wsat.simple.service;
. . .
import weblogic.jws.Policy;
import javax.transaction.UserTransaction;
. . .
import javax.jws.WebService;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;

/**
 * This JWS file forms the basis of a WebLogic WS-Atomic Transaction Web Service with the
 * operations: createAccount, deleteAccount, transferMonet, listAccount
 *
 */

@WebService(serviceName = "WsatBankTransferService", targetNamespace = "http://tempuri.org/",
            portName = "WSHttpBindingIService")
@Transactional(value=Transactional.TransactionFlowType.MANDATORY,
               version=weblogic.wsee.wstx.wsat.Transactional.Version.WSAT10)
public class WsatBankTransferService {

    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;
        UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-demoXA-2");
                String sql = "insert into wsat_acct_remote (acctno, amount) values (" + acctNo +
                           ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
            throw new java.lang.Exception("insert fail at remote.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating. ";
            } catch (SQLException e) {
              System.out.println("**** Exception caught *****");
               e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at remote.");
        }
        } catch (java.lang.Exception e) {
          System.out.println("**** Exception caught *****");
          e.printStackTrace();
          throw new java.lang.Exception(e);
        }
    }
    public String deleteAccount(String acctNo) throws java.lang.Exception{
      ...
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
```

```
       java.lang.Exception{
      ...
    }
    public String listAccount() throws java.lang.Exception{
      ...
    }
}
```

```
package jaxws.interop.rsp;
...
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.TransactionalFlowType;
import weblogic.wsee.wstx.wsat.Transactional.Version;
...
@WebService(
     portName = "FlightServiceBindings_Basic",
     serviceName = "FlightService",
     targetNamespace = "http://wsinterop.org/samples",
     wsdlLocation = "/wsdls/FlightService.wsdl",
     endpointInterface = "jaxws.interop.rsp.IFlightService"
)
@BindingType("http://schemas.xmlsoap.org/wsdl/soap/http")
@javax.xml.ws.soap.Addressing
@Transactional(value = Transactional.TransactionFlowType.SUPPORTS,
   version = Transactional.Version.WSAT12)
public class FlightServiceImpl implements IFlightService {
   ...
}
```

Example: Using @Transactional Annotation on a Web Service Method

The following example shows how to add `@Transactional` annotation on a web service implementation method. Relevant code is shown in **bold**.

```
package jaxws.interop.rsp;
...
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.TransactionalFlowType;
import weblogic.wsee.wstx.wsat.Transactional.Version;
...
@WebService(
     portName = "FlightServiceBindings_Basic",
     serviceName = "FlightService",
     targetNamespace = "http://wsinterop.org/samples",
     wsdlLocation = "/wsdls/FlightService.wsdl",
     endpointInterface = "jaxws.interop.rsp.IFlightService"
)
@BindingType("http://schemas.xmlsoap.org/wsdl/soap/http")
@javax.xml.ws.soap.Addressing
public class FlightServiceImpl implements IFlightService {
...
    @Transactional(value = Transactional.TransactionFlowType.SUPPORTS,
                   version = Transactional.Version.WSAT12)
    public FlightReservationResponse reserveFlight(FlightReservationRequest request) {
        //replace with your impl here
        FlightReserverationEnitity entity = new FlightReserverationEnitity();
        entity.setAirlineID(request.getAirlineID());
```

```
        entity.setFlightNumber(request.getFlightNumber());
        entity.setFlightType(request.getFlightType());
        boolean successful = saveRequest(entity);
        FlightReservationResponse response = new FlightReservationResponse();
        if (!successful) {
           response.setConfirmationNumber("OF" +  CONF_NUMBER++ + "-" + request.getAirlineID() +
                   String.valueOf(entity.getId()));
        } else if (request.getFlightNumber() == null ||
                   request.getFlightNumber().trim().endsWith("LAS"))   {
           successful = false;
           response.setConfirmationNumber("OF" + "- No flight available for " +
                   request.getAirlineID());
        } else {
           response.setConfirmationNumber("OF" +  CONF_NUMBER++ + "-" + request.getAirlineID() +
                  String.valueOf(entity.getId()));
        }
        response.setSuccess(successful);
        return response;
    }

package examples.webservices.jaxws.wsat.simple.service;
. . .
import weblogic.jws.Policy;
import javax.transaction.UserTransaction;
. . .
import javax.jws.WebService;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;

/**
 * This JWS file forms the basis of a WebLogic WS-Atomic Transaction Web Service with the
 * operations: createAccount, deleteAccount, transferMonet, listAccount
 *
 */

@WebService(serviceName = "WsatBankTransferService", targetNamespace = "http://tempuri.org/",
            portName = "WSHttpBindingIService")
public class WsatBankTransferService {

@Transactional(value=Transactional.TransactionFlowType.MANDATORY,
               version=weblogic.wsee.wstx.wsat.Transactional.Version.WSAT10)
    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;
        UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-demoXA-2");
                String sql = "insert into wsat_acct_remote (acctno, amount) values (" + acctNo +
                           ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
            throw new java.lang.Exception("insert fail at remote.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating. ";
            } catch (SQLException e) {
              System.out.println("**** Exception caught *****");
               e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at remote.");
        }
```

```
        } catch (java.lang.Exception e) {
          System.out.println("**** Exception caught *****");
          e.printStackTrace();
          throw new java.lang.Exception(e);
        }
    }
    public String deleteAccount(String acctNo) throws java.lang.Exception{
      ...
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
       java.lang.Exception{
      ...
    }
    public String listAccount() throws java.lang.Exception{
      ...
    }
}
```

Example: Using the @Transactional and the EJB @TransactionAttribute Annotations Together

The following example illustrates how to use the `@Transactional` and EJB `@TransactionAttribute` annotations together. In this case, the flow type values must be compatible, as outlined in Table 15-3. Relevant code is shown in **bold**.

```
package examples.webservices.jaxws.wsat.simple.service;
. . .
import weblogic.jws.Policy;
import javax.transaction.UserTransaction;
. . .
import javax.jws.WebService;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;

/**
 * This JWS file forms the basis of a WebLogic WS-Atomic Transaction Web Service with the
 * operations: createAccount, deleteAccount, transferMonet, listAccount
 *
 */

@WebService(serviceName = "WsatBankTransferService", targetNamespace = "http://tempuri.org/",
            portName = "WSHttpBindingIService")
@Transactional(value=Transactional.TransactionFlowType.MANDATORY,
               version=weblogic.wsee.wstx.wsat.Transactional.Version.WSAT10)
@TransactionAttribute(TransactionAttributeType.REQUIRED
public class WsatBankTransferService {
. . .
}
```

Enabling Web Services Atomic Transactions Starting From WSDL

When enabled, web services atomic transactions are advertised in the WSDL file using a policy assertion.

Table 15-4 summarizes the WS-AtomicTransaction 1.2 policy assertions that correspond to a set of common web services atomic transaction flow type and EJB Transaction attribute combinations. All other combinations result in a build-time error.

Table 15-4 Web Services Atomic Transaction Policy Assertion Values (WS-AtomicTransaction 1.2)

Atomic Transaction Flow Type	EJB `@TransactionAttribute`	WS-AtomicTransaction 1.2 Policy Assertion
`MANDATORY`	`MANDATORY`, `REQUIRED`, `SUPPORTS`	`<wsat:ATAssertion/>`
`SUPPORTS`	`REQUIRED`, `SUPPORTS`	`<wsat:ATAssertion wsp:Optional="true"/>`
`NEVER`	`REQUIRED`, `REQUIRES_NEW`, `NEVER`, `SUPPORTS`, `NOT_SUPPORTED`	No policy advertisement

You can use `wsdlc` Ant task to generate, from an existing WSDL file, a set of artifacts that together provide a partial Java implementation of the web service described by the WSDL file. The WS-AtomicTransaction policy assertions that are advertised in the WSDL are carried forward and are included in the WSDL file for the new web service generated by `wsdlc`.

The `wsdlc` Ant tasks creates a JWS file that contains a partial (stubbed-out) implementation of the generated JWS interface. You need to modify this file to include your business code. After you have coded the JWS file with your business logic, run the `jwsc` Ant task to generate a complete Java implementation of the web service. Use the `compiledWsdl` attribute of `jwsc` to specify the JAR file generated by the `wsdlc` Ant task which contains the JWS interface file and data binding artifacts. By specifying this attribute, the `jwsc` Ant task does not generate a new WSDL file but instead uses the one in the JAR file. Consequently, when you deploy the web service and view its WSDL, the deployed WSDL will look just like the one from which you initially started (with the WS-AtomicTransaction policy assertions).

For complete details about using `wsdlc` to generate a web service from a WSDL file, see Developing WebLogic Web Services Starting From a WSDL File: Main Steps.

Enabling Web Services Atomic Transactions on Web Service Clients

On a web service client, enable web services atomic transactions using one of the following methods:

- Add the `@weblogic.wsee.wstx.wsat.Transactional` annotation on the web service reference injection point for a client. For more information, see Using @Transactional Annotation with the @WebServiceRef Annotation.
- Pass an instance of the `weblogic.wsee.wstx.wsat.TransactionalFeature` as a parameter when creating the web service proxy or dispatch. For more information, see Passing the TransactionalFeature to the Client.

- At deployment time, enable and configure web services atomic transactions at the web service client endpoint or method level using the WebLogic Server Administration Console. For more information, see Configuring Web Services Atomic Transactions Using the Administration Console.
- At run-time, if the non-atomic transactional web service client calls an atomic transaction-enabled web service, then based on the flow type advertised in the WSDL:
  - If the flow type is set to `SUPPORTS` or `NEVER` on the service-side, then the call is included as part of the transaction.
  - If the flow type is set to `MANDATORY`, then an exception is thrown.

> **Note:**
>
> Web services atomic transactions are not supported by Java SE clients.

For information about the configuration options that you can set when enabling web services atomic transactions, see Table 15-2.

Using @Transactional Annotation with the @WebServiceRef Annotation

To enable web services atomic transactions, specify the `@weblogic.wsee.wstx.wsat.Transactional` annotation on the web service client at the web service reference (`@WebServiceRef`) injection point.

The format for specifying the `@Transactional` annotation is as follows:

```
@Transactional(
     version=Transactional.Version.[WSAT10|WSAT11|WSAT12|DEFAULT],
     value=Transactional.TransactionFlowType.[MANDATORY|SUPPORTS|NEVER]
)
```

For more information about the version and flow type configuration options, see Table 15-2.

The following example illustrates how to annotate the web service reference injection point. Relevant code is shown in **bold**. As shown in the example, the active JTA transaction becomes a part of the atomic transaction.

> **Note:**
>
> The following excerpt is borrowed from the web services atomic transaction example that is delivered with the WebLogic Server Samples Server. For more information, see More Examples of Using Web Services Atomic Transactions.

```
package examples.webservices.jaxws.wsat.simple.client;
. . .
import javax.servlet.*;
import javax.servlet.http.*;
. . .
import java.net.URL;
import javax.xml.namespace.QName;

import javax.transaction.UserTransaction;
```

```
import javax.transaction.SystemException;

import javax.xml.ws.WebServiceRef;
import weblogic.wsee.wstx.wsat.Transactional;
*/

/**
 * This example demonstrates using a WS-Atomic Transaction to create or delete an account,
 * or transfer money via web service as a single atomic transaction.
 */

public class WsatBankTransferServlet extends HttpServlet {
. . .
        String url = "http://localhost:7001";
        URL wsdlURL = new URL(url + "/WsatBankTransferService/WsatBankTransferService");
. . .
        DataSource ds = null;
        UserTransaction utx = null;

        try {
            ctx = new InitialContext();
            utx = (UserTransaction) ctx.lookup("javax.transaction.UserTransaction");
            utx.setTransactionTimeout(900);
        } catch (java.lang.Exception e) {
            e.printStackTrace();
        }

        WsatBankTransferService port = getWebService(wsdlURL);

        try {
            utx.begin();
            if (remoteAccountNo.length() > 0) {
                if (action.equals("create")) {
                    result = port.createAccount(remoteAccountNo, amount);
                } else if (action.equals("delete")) {
                    result = port.deleteAccount(remoteAccountNo);
                } else if (action.equals("transfer")) {
                    result = port.transferMoney(remoteAccountNo, amount, direction);
                }
            }
            utx.commit();
            result = "The transaction is committed " + result;
       } catch (java.lang.Exception e) {
           try {
                  e.printStackTrace();
                  utx.rollback();
                  result = "The transaction is rolled back.    " + e.getMessage();
           } catch(java.lang.Exception ex) {
             e.printStackTrace();
             result = "Exception is caught. Check stack trace.";
           }
      }
      request.setAttribute("result", result);
 . . .
    @Transactional(value = Transactional.TransactionFlowType.MANDATORY,
                   version = Transactional.Version.WSAT10)
     @WebServiceRef(wsdlLocation =
          "http://localhost:7001/WsatBankTransferService/WsatBankTransferService?WSDL", value =
          examples.webservices.jaxws.wsat.simple.service.WsatBankTransferService.class)
    WsatBankTransferService_Service service;
    private WsatBankTransferService getWebService() {
```

```
        return service.getWSHttpBindingIService();
    }

    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;         UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-dataSource-demoXAPool");
                String sql = "insert into wsat_acct_local (acctno, amount) values (
                    " + acctNo + ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
                    throw new java.lang.Exception("insert fail at local.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating.. ";
            } catch (SQLException e) {
                System.out.println("**** Exception caught *****");
                e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at local.");
            }
        } catch (java.lang.Exception e) {
        System.out.println("**** Exception caught *****");
            e.printStackTrace();
        throw new java.lang.Exception(e);
        }
    }

    public String deleteAccount(String acctNo) throws java.lang.Exception{
    . . .
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
         java.lang.Exception{
    . . .
    }
    public String listAccount() throws java.lang.Exception{
    . . .
    }
}
```

```
package examples.webservices.service_to_service;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.WebServiceRef;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.TransactionalFlowType;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import examples.webservices.complex.ComplexPortType;
import examples.webservices.complex.ComplexService;
...
@WebService(name="ClientPortType", serviceName="ClientService",
            targetNamespace="http://examples.org")
public class ClientServiceImpl {
  @Transactional(value = Transactional.TransactionFlowType.SUPPORTS,
       version = Transactional.Version.WSAT12)
  @WebServiceRef()
  ComplexService service;
  @WebMethod()
  public String callComplexService(BasicStruct input)
  {
    // Create service and port stubs to invoke ComplexService
```

```
        ComplexPortType port = service.getComplexPortTypePort();
        // Include your implementation here.
      }
    }
```

Passing the TransactionalFeature to the Client

To enable web services atomic transactions on the client of the web service, you can pass an instance of the `weblogic.wsee.wstx.wsat.TransactionalFeature` as a parameter when creating the web service proxy or dispatch, as illustrated in the following example. Relevant code is shown in **bold**.

> **Note:**
>
> The following excerpt is borrowed from the web services atomic transaction example that is delivered with the WebLogic Server Samples Server. For more information, see More Examples of Using Web Services Atomic Transactions.

```
package examples.webservices.jaxws.wsat.simple.client;
. . .
import javax.servlet.*;
import javax.servlet.http.*;
. . .
import java.net.URL;
import javax.xml.namespace.QName;

import javax.transaction.UserTransaction;
import javax.transaction.SystemException;

import weblogic.wsee.wstx.wsat.TransactionalFeature;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;
*/

/**
 * This example demonstrates using a WS-Atomic Transaction to create or delete an account,
 * or transfer money via web service as a single atomic transaction.
 */

public class WsatBankTransferServlet extends HttpServlet {
. . .
        String url = "http://localhost:7001";
        URL wsdlURL = new URL(url + "/WsatBankTransferService/WsatBankTransferService");
. . .
        DataSource ds = null;
        UserTransaction utx = null;

        try {
            ctx = new InitialContext();
            utx = (UserTransaction) ctx.lookup("javax.transaction.UserTransaction");
            utx.setTransactionTimeout(900);
        } catch (java.lang.Exception e) {
            e.printStackTrace();
        }
```

```
        WsatBankTransferService port = getWebService(wsdlURL);

        try {
            utx.begin();
            if (remoteAccountNo.length() > 0) {
                if (action.equals("create")) {
                    result = port.createAccount(remoteAccountNo, amount);
                } else if (action.equals("delete")) {
                    result = port.deleteAccount(remoteAccountNo);
                } else if (action.equals("transfer")) {
                    result = port.transferMoney(remoteAccountNo, amount, direction);
                }
            }
            utx.commit();
            result = "The transaction is committed " + result;
       } catch (java.lang.Exception e) {
           try {
                  e.printStackTrace();
                  utx.rollback();
                  result = "The transaction is rolled back.    " + e.getMessage();
           } catch(java.lang.Exception ex) {
             e.printStackTrace();
             result = "Exception is caught. Check stack trace.";
           }
      }
      request.setAttribute("result", result);
. . .
   // Passing the TransactionalFeature to the Client
   private WsatBankTransferService getWebService(URL wsdlURL) {
        TransactionalFeature feature = new TransactionalFeature();
        feature.setFlowType(TransactionFlowType.MANDATORY);
        feature.setVersion(Version.WSAT10);
        WsatBankTransferService_Service service = new WsatBankTransferService_Service(wsdlURL,
              new QName("http://tempuri.org/", "WsatBankTransferService"));
        return service.getWSHttpBindingIService(new javax.xml.ws.soap.AddressingFeature(),
              feature);
   }

   public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;         UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-dataSource-demoXAPool");
                String sql = "insert into wsat_acct_local (acctno, amount) values (
                    " + acctNo + ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
                    throw new java.lang.Exception("insert fail at local.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating.. ";
            } catch (SQLException e) {
                System.out.println("**** Exception caught *****");
                e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at local.");
            }
        } catch (java.lang.Exception e) {
        System.out.println("**** Exception caught *****");
            e.printStackTrace();
        throw new java.lang.Exception(e);
        }
```

```
    }

    public String deleteAccount(String acctNo) throws java.lang.Exception{
    . . .
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
         java.lang.Exception{
    . . .
    }
    public String listAccount() throws java.lang.Exception{
    . . .
    }
}

package jaxws.interop.rsp;
...
import javax.jws.WebService;
import javax.xml.ws.*;
import weblogic.wsee.wstx.wsat.TransactionalFeature;
...
@WebService(
     portName = "TravelAgencyServiceBindings_Basic",
     serviceName = "TravelAgencyService",
     targetNamespace ="http://wsinterop.org/samples",
     wsdlLocation = "/wsdls/TravelAgencyService.wsdl",
     endpointInterface = "jaxws.interop.rsp.ITravelAgencyService"
)
@BindingType("http://schemas.xmlsoap.org/wsdl/soap/http")
@javax.xml.ws.soap.Addressing()
public class TravelAgencyServiceImpl implements ITravelAgencyService {
...
   private IFlightService getFlightProxy(String endpoint, String stsEndpoint) throws Exception {
        TransactionalFeature feature = new TransactionalFeature();
        // Optional setting.
        feature.setVersion(Transactional.Version.WSAT12);
        // Optional setting.
        feature.setEnabled("ReserveFlight", true);
        IFlightService flightProxy = flightService.getFlightServiceBindingsBasic(feature);
...
   }
}
```

Configuring Web Services Atomic Transactions Using the Administration Console

The following sections describe how to configure web services atomic transactions using the WebLogic Server Administration Console.

- Securing Messages Exchanged Between the Coordinator and Participant
- Enabling and Configuring Web Services Atomic Transactions

Securing Messages Exchanged Between the Coordinator and Participant

Using transport-level security, you can secure messages exchanged between the web services atomic transaction coordinator and participant by configuring the properties

defined in the following table using the WebLogic Server Administration Console. These properties are configured at the domain level. For detailed steps, see Configure web services atomic transactions in the *Oracle WebLogic Server Administration Console Online Help*.

Table 15-5 Securing Web Services Atomic Transactions

Property	Description
Web Services Transactions Transport Security Mode	Specifies whether two-way SSL is used for the message exchange between the coordinator and participant. This property can be set to one of the following values: • SSL Not Required—All web service transaction protocol messages are exchanged over the HTTP channel. • SSL Required—All web service transaction protocol messages are exchanged over the HTTPS channel. This flag must be enabled when invoking Microsoft .NET web services that have atomic transactions enabled. • Client Certificate Required—All web service transaction protocol messages are exchanged over HTTPS and a client certificate is required. For more information, see Configure two-way SSL in the *Oracle WebLogic Server Administration Console Online Help*.
Web Service Transactions Issued Token Enabled	Flag the specifies whether to use an issued token to enable authentication between the coordinator and participant. The `IssuedToken` is issued by the coordinator and consists of a security context token (SCT) and a session key used for signing. The participant sends the signature, signed using the shared session key, in its registration message. The coordinator authenticates the participant by verifying the signature using the session key.

Enabling and Configuring Web Services Atomic Transactions

To enable web services atomic transactions and configure the version and flow type, you can customize the configuration at the endpoint or method level for the web service or client. For detailed steps, see Configure web services atomic transactions in the *Oracle WebLogic Server Administration Console Online Help*.

Using Web Services Atomic Transactions in a Clustered Environment

For considerations when using atomic transaction-enabled web services in a clustered environment, see Managing Web Services in a Cluster.

More Examples of Using Web Services Atomic Transactions

Refer to the following sections for additional examples of using web services atomic transactions:

- For an example of how to sign and encrypt message headers exchanged during the web services atomic transaction, see Securing Web Services Atomic Transactions in *Securing WebLogic Web Services for Oracle WebLogic Server*.

  > **Note:**
  >
  > You can secure applications that enable web service atomic transactions using *only* WebLogic web service security policies. You cannot secure them using Oracle Web Services Manager (WSM) policies.

- A detailed example of web services atomic transactions is provided as part of the WebLogic Server sample application. For more information about running the sample application and accessing the example, see Sample Application and Code Examples in *Understanding Oracle WebLogic Server*.

16

Optimizing XML Transmission Using Fast Infoset

This chapter describes how to use Fast Infoset for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of Fast Infoset

Fast Infoset is a compressed binary encoding format that provides a more efficient serialization than the text-based XML format. Fast Infoset optimizes both document size and processing performance.

When enabled, Fast Infoset converts the XML Information Set in the SOAP envelope into a compressed binary format before transmitting the data. Fast Infoset optimizes encrypted and signed messages, MTOM-enabled messages, and SOAP attachments, and supports both HTTP and JMS transports.

The Fast Infoset specification, *ITU-T Rec. X.891 and ISO/IEC 24824-1 (Fast Infoset)* is defined by both the ITU-T and ISO standards bodies. The specification can be downloaded from the ITU Web site: `http://www.itu.int/rec/T-REC-X.891-200505-I/en`

The Fast Infoset capability is enabled on all web services, by default. For web service clients, Fast Infoset is enabled if it is enabled on the web service and advertised in the WSDL.

You can explicitly enable and configure Fast Infoset on a web service or client, as described in the following sections.

Enabling Fast Infoset on Web Services

The Fast Infoset capability is enabled on a web service and advertised in the WSDL, by default. You can enable Fast Infoset explicitly on a web service as follows:

- At design time, using `com.oracle.webservices.api.FastInfosetService` annotation, as shown in Example Using @FastInfosetService Annotation at Design Time.

Example Using @FastInfosetService Annotation at Design Time

The following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetService` annotation to enable and configure Fast Infoset on a web service at design time.

```
package examples.webservices.hello_world;
import javax.jws.WebService;
import com.oracle.webservices.api.FastInfosetService;

@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
@FastInfosetService(enabled=true)

public class HelloWorldImpl {
  public String sayHelloWorld(String message) {
  try {
    System.out.println("sayHelloWorld:" + message);
  } catch (Exception ex) { ex.printStackTrace(); }
    return "Message from FI Enabled Service: '" + message + "'";
  }
}
```

Enabling and Configuring Fast Infoset on Web Services Clients

You can explicitly enable and configure Fast Infoset on a web service client as follows:

- At design time, using `com.oracle.webservices.api.FastInfosetClientFeature` feature class, as shown in Example Using FastInfosetClientFeature Feature Class at Design Time.
  - `com.oracle.webservices.api.FastInfosetClient` annotation, as shown in Example Using @FastInfosetClient Annotation at Design Time.
  - `com.oracle.webservices.api.FastInfosetClientFeature` feature class, as shown in Example Using FastInfosetClientFeature Feature Class at Design Time.

Configuring the Content Negotiation Strategy

When enabling Fast Infoset on the client, you can configure the content negotiation policy. Table 16-1 summarizes the valid content negotiation strategies defined by `com.oracle.webservices.api.FastInfosetContentNegotiationType`.

Table 16-1 Content Negotiation Strategy

Value	Description
`OPTIMISTIC`	Assumes that Fast Infoset is enabled on the service. All requests will be sent using Fast Infoset.
`PESSIMISTIC`	Initial request from client is sent without Fast Infoset enabled, but with an HTTP Accept header that indicates that the client supports the Fast Infoset capability. If the service response is in Fast Infoset format, confirming that Fast Infoset is enabled on the service, then subsequent requests from the client will be sent in Fast Infoset format.
`NONE`	Client requests will not use Fast Infoset.

Please note:

- If the content negotiation strategy is configured explicitly on the client:

  - It takes precedence over the negotiation strategy advertised in the WSDL.
  - If the configured content negotiation strategy conflicts with the capabilities advertised by the service (for example, if the client configures `OPTIMISTIC` and the service has Fast Infoset disabled), then an exception is generated.
- If the content negotiation strategy is not configured explicitly by the client:
  - If Fast Infoset is enabled and advertised on the service, the `OPTIMISTIC` content negotiation strategy is used.
  - If Fast Infoset is disabled and not advertised on the service, the `NONE` content negotiation strategy is used.

Example Using @FastInfosetClient Annotation at Design Time

The following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetClient` annotation to enable and Fast Infoset on a Web service client at design time and configure the content negotiation strategy.

THIS EXAMPLE NEEDS TO BE UPDATED.

```
package examples.webservices.fastinfoset.client;
import com.oracle.webservices.api.FastInfosetClient;
import com.oracle.webservices.api.FastInfosetContentNegotiationType;
import javax.xml.ws.WebServiceRef;
...
public class HelloServicePortClient {
    @WebServiceRef
    @FastInfosetClient(fastInfosetContentNegotiation =
        FastInfosetContentNegotiationType.OPTIMISTIC)
    private static HelloServiceService helloServiceService;
...
}
```

Example Using FastInfosetClientFeature Feature Class at Design Time

The following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetClientFeature` feature class to enable and configure Fast Infoset on a web service at design time.

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

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
import com.oracle.webservices.api.FastInfosetClientFeature;
import com.oracle.webservices.api.FastInfosetContentNegotiationType;

public class Main {

public static void main(String[] args) {
    HelloWorldService service;
    FastInfosetContentNegotiationType clientNeg  =
        FastInfosetContentNegotiationType.PESSIMISTIC;
    FastInfosetClientFeature feature =
FastInfosetClientFeature.builder().fastInfosetContentNegotiation(clientNeg).enabled(tru
e).build();
          try {
      service = new HelloWorldService(new URL(args[0] + "?WSDL"), new QName("http://
```

```
hello_world.webservices.examples/", "HelloWorldService") );
    } catch (MalformedURLException murl) { throw new RuntimeException(murl); }
      HelloWorldPortType port = service.getHelloWorldPortTypePort(feature);

      String result = null;
      result = port.sayHelloWorld("Hi there!");
      System.out.println( "Got result: " + result );
  }
}
```

Disabling Fast Infoset on Web Services and Clients

At design time, to disable Fast Infoset explicitly:

- On a web service, set the `enabled` flag to `false` on the annotation, as described in Enabling Fast Infoset on Web Services.
- On a web service client, set the `enabled` flag to `false` or set the content negotiation strategy to `NONE` on the annotation or Feature class, as described in Enabling and Configuring Fast Infoset on Web Services Clients.

The following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetService` annotation to disable Fast Infoset on a web service at design time.

```
package examples.webservices.hello_world;
import javax.jws.WebService;
import com.oracle.webservices.api.FastInfosetService;

@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
@FastInfosetService(enabled=false)

public class HelloWorldImpl {
  public String sayHelloWorld(String message) {
  try {
    System.out.println("sayHelloWorld:" + message);
  } catch (Exception ex) { ex.printStackTrace(); }
    return "Message from FI Enabled Service: '" + message + "'";
  }
}
```

17

Using SOAP Over JMS Transport

This chapter describes how to use SOAP over Java Messaging Service (JMS) transport as the connection protocol for WebLogic web services using Java API for XML Web Services (JAX-WS).

> **Note:**
>
> SOAP over JMS transport is not compatible with the following web service features: reliable messaging and HTTP transport-specific security.

This chapter includes the following sections:

Overview of SOAP Over JMS Transport

Typically, web services and clients communicate using SOAP over HTTP/S as the connection protocol. You can, however, configure a WebLogic web service so that client applications use JMS as the transport.

Using SOAP over JMS transport, web services and clients communicate using JMS destinations instead of HTTP connections, offering the following benefits:

- Reliability
- Scalability
- Quality of service

As with web service reliable messaging, if WebLogic Server goes down while the method invocation is still in the queue, it will be handled as soon as WebLogic Server is restarted. When a client invokes a web service, the client does not wait for a response, and the execution of the client can continue. Using SOAP over JMS transport does require slightly more overhead and programming complexity than HTTP/S.

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

configure each transport. You configure transports using JWS annotations or child elements of the `jwsc` Ant task.

If you configure JMS transport only, 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-WS stubs for the web service.

> **Note:**
>
> Using JMS transport is a WebLogic feature; non-WebLogic client applications, such as a .NET client, may not be able to invoke the web service using the JMS port.

Figure 17-1 shows the flow of request and response messages for a web service invocation using SOAP over JMS transport.

- The client stub invokes the web service and sends the SOAP request message to the web service via the JMS request queue, and then waits for the response.
- On the server side, the MDB listener receives the request and invokes the service endpoint.
- Once processed, the service endpoint sends the response to the JMS response queue.
- The JMS listener on the client side receives the response and passes it to the client.
- The JMS response endpoint and listener are removed when the client issues the `java.io.Closable.close()` command.

Figure 17-1 Web Service Invocation Using SOAP Over JMS Transport

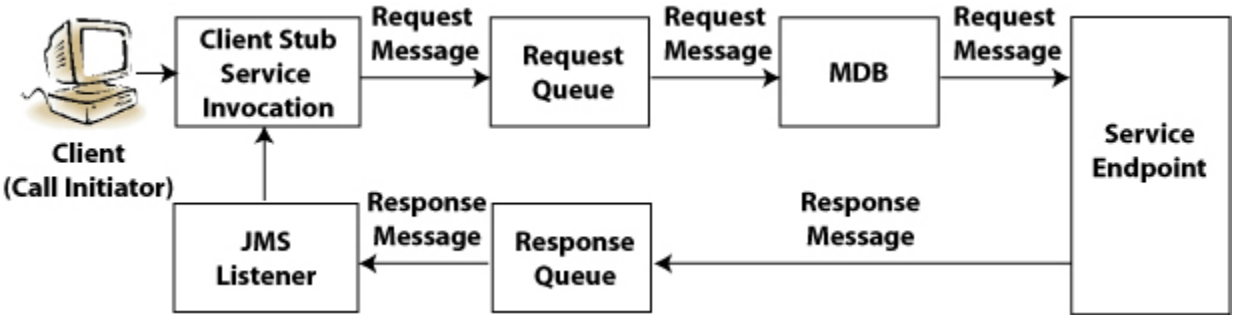
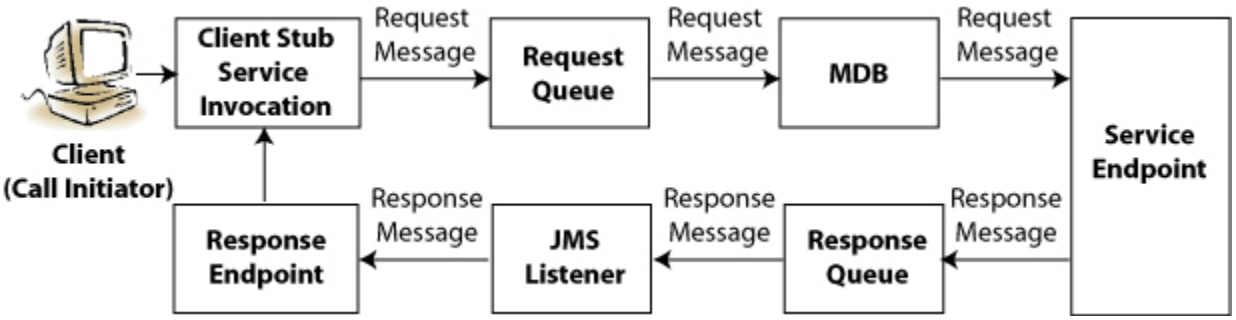


Figure 17-2 shows the flow of request and response messages for an asynchronous web service invocation using SOAP over JMS transport.

- The client stub invokes the web service asynchronously and sends the SOAP request message to the web service via the JMS request queue. The client stub returns a `javax.xml.ws.Response` or `java.util.concurrent.Future<T>` instance, and does not wait for the response.
- On the server side, the MDB listener receives the request and invokes the service endpoint.

- Once processed, the service endpoint sends the response to the JMS response queue.
- The JMS listener invokes the response endpoint which populates the `Response` or `Future<T>` instance for the client.
- The JMS response endpoint and listener are removed when the client issues the `java.io.Closable.close()` command.

Figure 17-2 Asynchronous Web Service Invocation Using SOAP Over JMS Transport



Before sending the request message to the JMS destination, the client sets the JMS message properties defined in Table 17-1.

Table 17-1 JMS Message Properties Defined in the Request Message

JMS Message Property	Description
`SOAPJMS_bindingVersion`	Version of the SOAP JMS binding. This value must be set to `1.0` for this release.
`SOAPJMS_contentType`	MIME content type of the message
`SOAPJMS_requestURI`	JMS request URI. For more information about how the value is configured, see Configuring the JMS Request URI.
`SOAPJMS_soapAction`	SOAP action which defines the intent of the request.
`SOAPJMS_targetService`	Port component name of the web service.
`messageType`	Message type to use with the request message. A value of `BYTES` indicates the `com.oracle.webservices.api.jms.JMSMessageType.BYTES` object is used. A value of `TEXT` indicates `com.oracle.webservices.api.jms.JMSMessageType.TEXT` object is used. This value defaults to `BYTES`.
`JMSMessageID`	ID that uniquely identifies the JMS message and that is used to correlate the response message with the request. The `JMSCorrelationID` property of the response message must match the `JMSMessageID` of the request message.
`ECIDContext`	Execution Context Identifier (ECID), wrapper code, and encoding details. This content is similar to what is provided for the HTTP header, and is required for the client only.

Before sending the response message to the JMS destination, the service sets the JMS message properties defined in Table 17-1.

Table 17-2 JMS Message Properties Defined in the Response Message

JMS Message Property	Description
`SOAPJMS_bindingVersion`	Version of the SOAP JMS binding. This value must be set to `1.0` for this release.
`SOAPJMS_contentType`	MIME content type of the message
`JMSCorrelationID`	ID used to correlate the request and response messages. The `JMSCorrelationID` must match the `JMSMessageID` of the request message.
`JMSMessageID`	ID that uniquely identifies the JMS message and that is used to correlate the response message with the request.

Configuring the WebLogic Server Domain for JMS Transport

Table 17-3 lists the default resources used by JMS transport in your WebLogic Server domain, by default.

Table 17-3 Default Resources Used by JMS Transport

Resource Name (Default)	Resource Type	Description
`WseeSoapjmsJmsServer`	JMS server	JMS server management container. To configure the JMS server manually, see JMS Configuration in *Administering JMS Resources for Oracle WebLogic Server*.
`WseeSoapjmsFileStore`	File store	File store, or physical store, used by the WebLogic Server to handle the I/O operations to save and retrieve data from the physical storage (such as file, DBMS, and so on). To configure the file stores manually, see Using Custom File Stores in *Administering the WebLogic Persistent Store*.
`WseeSoapjmsJmsModule`	JMS Module	JMS module that defines the JMS resources needed for SOAP over JMS transport. To configure the JMS module manually, see JMS Configuration in *Administering JMS Resources for Oracle WebLogic Server*.
`WseeSoapjmsJmsServerSub`	JMS subdeployment	JMS subdeployment for targeting the JMS resources to the `WseeSoapJmsServer`. To configure the JMS subdeployment manually, see JMS Configuration in *Administering JMS Resources for Oracle WebLogic Server*.
`com.oracle.webservices.api.jms.ConnectionFactory`	JMS Connection Factory	Default JMS connection factory used to create connections for SOAP over JMS transport. You can configure a different connection factory using the `jndiConnectionFactoryName` JMS transport property, as described in Configuring JMS Transport Properties.

Table 17-3 (Cont.) Default Resources Used by JMS Transport

Resource Name (Default)	Resource Type	Description
`com.oracle.webservices.api.jms.RequestQueue`	JMS Queue	Default JMS request queue. You can configure a different JMS request queue using the `destinationName` JMS transport property, as described in Configuring JMS Transport Properties.
`com.oracle.webservices.api.jms.ResponseQueue`	JMS Queue	Default JMS response queue. You can configure a different JMS response queue, as described in Configuring the JMS Response Queue.

When creating or extending a domain, you can apply the WebLogic JAX-WS SOAP/JMS Extension template (`wls_webservice_soapjms.jar`) to configure automatically the JMS resources required to support JMS transport.

To configure automatically the JMS resources required to support JMS transport, use one of the following methods:

- Use the Configuration Wizard to create or extend a domain, as described in Creating a WebLogic Domain in the *Creating WebLogic Domains Using the Configuration Wizard*. When prompted to specify a template to use to create or extend the domain, select the **WebLogic JAX-WS SOAP/JMS Extension** template.
- Use WLST to extend a domain, using the `wls_webservice_soapjms.jar` extension template JAR file, as described in Editing a WebLogic Domain (Offline) in *Understanding the WebLogic Scripting Tool*.

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

To configure manually the resources required to support JMS transport, use one of the following methods:

- Use the WebLogic Server Administration Console to create the resources, as described in Table 17-3. For more information, see JMS Configuration in *Administering JMS Resources for Oracle WebLogic Server*.
- Use WLST to create the resources defined in Table 17-3. For more information, see Creating Existing WebLogic Domains in *Understanding the WebLogic Scripting Tool*.

Developing Web Services Using JMS Transport—Starting From Java

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

- Add the `@com.oracle.webservices.api.jms.JMSTransportService` annotation to your JWS file.
- Add a `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task. This setting overrides the transports defined in the JWS file.

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.

Table 17-4 Steps to Develop Web Services With JMS Transport—Starting From Java

#	Step	Description
1	Complete the prerequisites.	It is assumed that you have created a basic JWS file that implements a web service and that you want to configure the web service to be invoked using JMS transport 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 Developing JAX-WS Web Services.
2	Configure the WebLogic Server domain for the required JMS components.	See Configuring the WebLogic Server Domain for JMS Transport.
3	Add the `@com.oracle.webservices.api.jms.JMSTransportService` annotation to your JWS file. (Optional)	This step is optional. If you do not add the `@JMSTransportService` annotation to your JWS file, then you must add a `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task, as described in Step 4. See Using the @JMSTransportService Annotation.
4	Add a `<jmstransportservice>` child element to the `jwsc` Ant task. (Optional)	Use the `<jmstransportservice>` child element to override the transports defined in the JWS file. This step is required if you did not add the `@JmsTransportService` annotation to your JWS file in Step 3. Otherwise, this step is optional. See Using the <jmstransportservice> Child Element in the Ant build.xml File for details.
5	Build your web service by 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` See Running the jwsc WebLogic Web Services Ant Task.
6	Deploy your web service to WebLogic Server.	See Deploying and Undeploying WebLogic Web Services.

See Invoking a WebLogic Web Service Using JMS Transport for information about updating your client application to invoke the web service using JMS transport.

Using the @JMSTransportService 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 `@com.oracle.webservices.api.jms.JMSTransportService` annotation to specify the details of the invocation.

You can include only *one* `@JMSTransportService` annotation in a JWS file.

Optionally, you can configure the destination name, connection factory, delivery mode, and other JMS transport properties using the `@JMSTransportService` annotation. For more information, see Configuring JMS Transport Properties.

Later, at build-time, you can override the invocation defined in the JWS file and add additional JMS transport specifications, by specifying the `<jmstransportservice>` child element in the `<jws>` element of "jwsc" `jwsc` Ant task, as described in Using the <jmstransportservice> Child Element in the Ant build.xml File.

Example 17-1 shows an excerpt from a JWS file, implemented as a stateless EJB, that uses the `@JMSTransportService` annotation, with the relevant code in **bold**.

Example 17-1 Enabling JMS Transport for a Stateless EJB Using @JMSTransportService Annotation

```
package jaxws.ejb;
...
import javax.ejb.Stateless;
import javax.jws.WebService;
import com.oracle.webservices.api.jms.JMSTransportService;

@WebService(name = "Simple", targetNamespace = "http://example.org")
@JMSTransportService(
  targetService="SimpleEjbService",
  destinationName="com.oracle.webservices.api.jms.RequestQueue",
  jndiConnectionFactoryName="weblogic.jms.ConnectionFactory",
  mdbPerDestination=false,
  activationConfig=("transAttribute=Never;maxBeansInFreePool=1000;
                    dispatchPolicy=weblogic.wsee.jaxws.mdb.DispatchPolicy"
  )
@Stateless
public class SimpleEjb { ... }
```

Example 17-2 shows an excerpt from a provider-based web service that uses the `@JMSTransportService` annotation, with the relevant code in **bold**.

Example 17-2 Enabling JMS Transport for a Provider-based Web Service Using @JMSTransportService Annotation

```
package examples.webservices.jaxws;
...
import javax.xml.transform.Source;
import javax.xml.ws.Provider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.Service;
import java.io.ByteArrayInputStream;
import com.oracle.webservices.api.jms.JMSTransportService;

@ServiceMode(value=Service.Mode.PAYLOAD)
@WebServiceProvider(portName="WarehouseServicePort",
                    serviceName="WarehouseService",targetNamespace="http://examples.org/")
@JMSTransportService(destinationName="myQueue")
public class WarehouseServiceImpl implements Provider<Source> {

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

Using the <jmstransportservice> Child Element in the Ant build.xml File

You can specify the JMS transport at build-time by adding the `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task. You may want to configure JMS transport at build-time for one of the following reasons:

- You want to override the attribute values specified in the JWS file using the `@JMSTransportService` annotation.
- The JWS file does not include a `@JMSTransportService` annotation and you determine at build-time that you want clients to use the JMS transport to invoke the web service.

The `<jmstransportservice>` child element of the `jwsc` Ant task takes precedence over the `@JMSTransportService` transport annotation in the JWS file.

Optionally, you can configure the destination name, destination type, delivery mode, and other JMS transport properties, using the `<jmstransportservice>` element. For a complete list of JMS transport properties supported, see Configuring JMS Transport Properties.

Example 17-3 shows an excerpt from a `build.xml` file that shows how to enable and configure JMS transport using the `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task. The relevant code is shown in **bold**.

Example 17-3 Enabling JMS Transport Using the <jmstransportservice> Child Element

```
<?xml version="1.0"?>
<project name="jaxws.jms.jwsc" default="all">
   <import file="../build-jms.xml"/>
      <path id="client.class.path">
         <pathelement path="${clientclasses.dir}"/>
         <pathelement path="${java.class.path}"/>
      </path>
   <target name="jwsc">
      <jwsc srcdir="." sourcepath="client" destdir="${output.dir}" debug="on"
            keepGenerated="yes">
         <jws file="JWSCEndpoint.java" type="JAXWS" explode="true">
            <jmstransportservice
               targetService="JWSCEndpointService"
               destinationName="com.oracle.webservices.api.jms.RequestQueue"
               jndiInitialContextFactory="weblogic.jndi.WLInitialContextFactory"
               jndiConnectionFactoryName="weblogic.jms.XAConnectionFactory"
               jndiURL="t3://localhost:7001"
               deliveryMode=com.oracle.webservices.api.jms.JMSDeliveryMode.PERSISTENT
               timeToLive=60000
               priority=1
               messageType=com.oracle.webservices.api.jms.JMSMessageType.BYTES
               activationConfig = "transAttribute=Supports"
            />
      </jws>
   </jwsc>
</target>
</project>
```

For more information about using the `jwsc` Ant task, see jwsc in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Developing Web Services Using JMS Transport—Starting From WSDL

To use JMS transport for web services 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 `@com.oracle.webservices.api.jms.JMSTransportService` annotation.
- Add a `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task used to build the JWS implementation file. This setting overrides the transports defined in the JWS file.

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.

Table 17-5 Steps to Developing Web Services With JMS Transport—Starting From WSDL

#	Step	Description
1	Complete the prerequisites.	It is assumed in this procedure that you have an existing WSDL file.
2	Configure the WebLogic Server domain for the required JMS components.	See Configuring the WebLogic Server Domain for JMS Transport.
3	Update the WSDL to use JMS transport. (Optional)	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 `@JMSTransportService` annotation to your JWS file, as described in Step 5. • Add a `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task, as described in Step 7. See Updating the WSDL to Use JMS Transport.
4	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` See Running the wsdlc WebLogic Web Services Ant Task.
5	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. See Updating the Stubbed-out JWS Implementation Class File Generated By wsdlc. If you updated the WSDL to use the JMS transport in Step 3, the JWS file includes the `@JMSTransportService` annotation that defines the JMS transport. If the `@JMSTransportService` annotation is not included in the JWS file, you must do at least one of the following: • Edit the JWS file to add the `@JMSTransportService` annotation to your JWS file, as described in Using the @JMSTransportService Annotation. • Add a `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task, as described in Step 7.
6	Add a `<jmstransportservice>` child element to the `jwsc` Ant task. (Optional)	Use the `<jmstransportservice>` child element to override the transports defined in the JWS file. This step is required if the JWS file does not include the `@JMSTransportService` annotation, as noted in Step 5. Otherwise, this step is optional. See Using the <jmstransportservice> Child Element in the Ant build.xml File for details.
7	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.

Table 17-5 (Cont.) Steps to Developing Web Services With JMS Transport—Starting From WSDL

#	Step	Description
8	Deploy the web service to WebLogic Server.	See Deploying and Undeploying WebLogic Web Services.

See Invoking a WebLogic Web Service Using JMS Transport for information about updating your client application to invoke the web service using JMS transport.

Updating the WSDL to Use JMS Transport

To update the WSDL to use JMS transport, you need to add the `<wsdl:binding>` definition that defines JMS transport information. You can add the definition in one of the following ways, depending on whether you want to specify multiple transport options:

- Edit the existing HTTP `<wsdl:binding>` definition.
- To specify multiple transport options in the WSDL (such as HTTP and JMS transport), copy the existing HTTP `<wsdl:binding>` definition and edit it to use JMS transport.

Optionally, you can configure JMS transport properties at the binding or JMS URI level.

The following sections describe how to update the WSDL to use JMS transport:

- Enabling JMS Transport at the WSDL Binding Level
- Configuring JMS Transport Properties in the WSDL
- Example of Enabling JMS Transport in WSDL

Enabling JMS Transport at the WSDL Binding Level

To enable JMS transport at the WSDL binding level, set the `transport` attribute of the `<soapwsdl:binding>` child element of the `<wsdl:binding>` element to `http://www.w3.org/2010/soapjms`.

Optionally, you can configure JMS transport properties within the `<wsdl:binding>` element definition, as described in Configuring JMS Transport Properties in the WSDL.

Example 17-4 provides an example of the `<wsdl:binding>` element for JMS transport. In this example, an HTTP binding is also defined.

Example 17-4 Enabling JMS Transport at the WSDL Binding Level

```
...
<binding xmlns:soapjms="http://www.w3.org/2010/soapjms/"
     name="AddNumbersJMSBinding" type="tns:AddNumbersPortType">
    <soap:binding transport="http://www.w3.org/2010/soapjms/" style="document" />
    <operation name="addNumbers">
        <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
```

```
            <soap:body use="literal" />
          </output>
    </operation>
</binding>
<binding name="AddNumbersSOAPBinding" type="tns:AddNumbersPortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
     ...
</binding>
...
```

Configuring JMS Transport Properties in the WSDL

Optionally, you can configure a subset of JMS transport properties within the following WSDL elements:

- `<wsdl:binding>` element—Propagates to all ports using the binding.
- `<wsdl:service>` element—Propagates to all ports.
- `<wsdl:port>` element—Used only by the port.
- JMS URI, as described in Configuring the JMS URI.

Specifically, you can configure the following JMS transport properties in the WSDL. For a description of the properties, see Table 17-6.

- `deliveryMode`
- `jndiConnectionFactoryName`
- `jndiContextParameters`
- `jndiInitialContextFactory`
- `jndiURL`
- `priority`
- `replyToName`
- `timeToLive`

Example 17-5 provides an example of the `<wsdl:binding>` element with JMS transport properties defined. In this case, the JMS transport properties propagate to all ports that use the binding.

Example 17-5 Configuring JMS Transport Properties in the WSDL

```
...
<binding xmlns:soapjms="http://www.w3.org/2010/soapjms/"
     name="AddNumbersBinding" type="tns:AddNumbersPortType">
   <soap:binding transport="http://www.w3.org/2010/soapjms/"
                 style="document" />
      <soapjms:jndiInitialContextFactory>
         weblogic.jndi.WLInitialContextFactory
      </soapjms:jndiInitialContextFactory>
      <soapjms:jndiConnectionFactoryName>
         weblogic.jms.XAConnectionFactory
      </soapjms:jndiConnectionFactoryName>
      <soapjms:bindingVersion>1.0</soapjms:bindingVersion>
      <soapjms:destinationName>
         com.oracle.webservices.api.jms.RequestQueue
      </soapjms:destinationName>
      <soapjms:targetService>AddNumbersService</soapjms:targetService>
```

```
      <soapjms:deliveryMode>
         com.oracle.webservices.api.jms.JMSDeliveryMode.PERSISTENT
      </soapjms:deliveryMode>
      <soapjms:priority>0</soapjms:priority>
      <soapjms:messageType>
         com.oracle.webservices.api.jms.JMSMessageType.BYTES
      </soapjms:messageType>
      <soapjms:destinationType>
         com.oracle.webservices.api.jms.JMSDestinationType.QUEUE
      </soapjms:destinationType>
            <operation name="addNumbers">
               <soap:operation soapAction="" />
                  <input>
                     <soap:body use="literal" />
                  </input>
                  <output>
                     <soap:body use="literal" />
                  </output>
               </operation>
</binding>
...
```

Example of Enabling JMS Transport in WSDL

Example 17-6 provides an example of a WSDL that is configured for SOAP over JMS transport.

Example 17-6 Enabling JMS Transport in WSDL

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

<definitions
   name="AddNumbers"
   targetNamespace="http://example.org"
   xmlns:tns="http://example.org"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
   <types>
      <xsd:schema
         xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://example.org"
         elementFormDefault="qualified">

         <complexType name="addNumbersResponse"
            <sequence>
               <element name="return" type="xsd:int" />
            </sequence>
         </complexType>
         <element name="addNumbersResponse" type="tns:addNumbersResponse"/>

         <complexType name="addNumbers">
            <sequence>
               <element name="arg0" type="xsd:int" />
               <element name="arg1" type="xsd:int" />
            </sequence>
         </complexType>
         <element name="addNumbers" type="tns:addNumbers"/>
      </xsd:schema>
   </types>
   <message name="addNumbers">
```

```
      <part name="parameters" element="tns:addNumbers" />
   </message>
   <message name="addNumbersResponse">
      <part name="result" element="tns:addNumbersResponse" />
   </message>
   <portType name="AddNumbersPortType">
      <operation name="addNumbers">
         <input message="tns:addNumbers" />
         <output message="tns:addNumbersResponse" />
      </operation>
   </portType>
         <binding xmlns:soapjms="http://www.w3.org/2010/soapjms/" name="AddNumbersBinding"
            type="tns:AddNumbersPortType">
            <soap:binding transport="http://www.w3.org/2010/soapjms/" style="document" />
            <soapjms:jndiInitialContextFactory>weblogic.jndi.WLInitialContextFactory
            </soapjms:jndiInitialContextFactory>
            <soapjms:jndiConnectionFactoryName>weblogic.jms.XAConnectionFactory
            </soapjms:jndiConnectionFactoryName>
            <soapjms:bindingVersion>1.0</soapjms:bindingVersion>
            <soapjms:destinationName>com.oracle.webservices.api.jms.RequestQueue
            </soapjms:destinationName>
            <soapjms:targetService>AddNumbersService</soapjms:targetService>
            <soapjms:deliveryMode>com.oracle.webservices.api.jms.JMSDeliveryMode.PERSISTENT
            </soapjms:deliveryMode>
            <soapjms:priority>0</soapjms:priority>
            <soapjms:messageType>com.oracle.webservices.api.jms.JMSMessageType.BYTES
            </soapjms:messageType>
            <soapjms:destinationType>com.oracle.webservices.api.jms.JMSDestinationType.QUEUE
            </soapjms:destinationType>
               <operation name="addNumbers">
                  <soap:operation soapAction="" />
                     <input>
                        <soap:body use="literal" />
                     </input>
                     <output>
                        <soap:body use="literal" />
                     </output>
                  </operation>
      </binding>
      <service name="AddNumbersService">
         <port name="AddNumbersPort" binding="tns:AddNumbersBinding">
            <soap:address location="jms:jndi:com.oracle.webservices.api.jms.RequestQueue?
targetService=AddNumbersService&amp;jndiInitialContextFactory=weblogic.jndi.WLInitialContextFactory&amp;
jndiConnectionFactoryName=weblogic.jms.XAConnectionFactory"/>

      </port>
   </service>
</definitions>
```

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. In the case of JMS transport, the client sends SOAP request messages to the JMS request destination and receives SOAP response messages from the JMS response destination. For examples of invoking a web service, see Examples of Developing JAX-WS Web Service Clients.

You enable and optionally configure JMS transport on the web service client using one of the following methods:

- Use the `<jmstransportclient>` element of the `clientgen` Ant task to generate automatically client artifacts with JMS transport enabled, as described in Using the <jmstransportclient> Element in the Ant build.xml File.
- Update the web service client to configure JMS transport, using one of the following methods:
  - Adding `@com.oracle.webservices.api.jms.JMSTransportClient` annotation, as described in Using the @JMSTransportClient Annotation.
  - Adding `com.oracle.webservices.api.jms.JMSTransportClientFeature` feature client API, as described in Using the JMSTransportClientFeature Client API.
  - Configure the JMS URI as the target endpoint address for synchronous clients, as described in Configuring the JMS URI as the Target Endpoint Address.
- Update the asynchronous web service client to enable and configure JMS transport, as described in Using AsyncClientTransportFeature to Configure Asynchronous Clients.

Using the <jmstransportclient> Element in the Ant build.xml File

The `clientgen` tool generates a JMS transport client proxy from a WSDL file containing a JMS transport binding. When generating the client proxy using `clientgen`, you can enable JMS transport by adding the `<jmstransportclient>` element in `clientgen` Ant task.

> **Note:**
>
> 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-WS artifacts for the web service.

Optionally, you can configure the destination name, destination type, delivery mode, request and response queues, and other JMS transport properties, using the `<jmstransportclient>` element. For a complete list of JMS transport properties supported, see Configuring JMS Transport Properties.

Example 17-7 shows an excerpt from a `build.xml` file that shows how to enable and configure JMS transport using the `<jmstransportclient>` element of the `clientgen` Ant task. The relevant code is shown in **bold**.

Example 17-7 Using the <jmstransportclient> Element in the Ant build.xml File

```
<target name="clientgen">
<clientgen
      wsdl="./WarehouseService.wsdl"
      destDir="clientclasses"
      packageName="client.warehouse"
      type="JAXWS">
      <jmstransportclient
                targetService="JWSCEndpointService"
                destinationName="com.oracle.webservices.api.jms.RequestQueue"
                jndiInitialContextFactory="weblogic.jndi.WLInitialContextFactory"
```

```
                jndiConnectionFactoryName="weblogic.jms.ConnectionFactory"
                jndiURL="t3://localhost:7001"
                timeToLive=60000
                priority=1
                messageType=com.oracle.webservices.api.jms.JMSMessageType.TEXT
                replyToName="com.oracle.webservices.api.jms.ResponseQueue"
    />
</clientgen>
```

For more information about using the `clientgen` Ant task, see clientgen in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Using the @JMSTransportClient Annotation

When you run `clientgen` to generate the web service client artifacts from the WSDL file, the `@com.oracle.webservices.api.jms.JMSTransportClient` annotation is included automatically to the generated client proxy if JMS transport is enabled in the build file using the `<jmstransportclient>` element, as described in Using the <jmstransportclient> Element in the Ant build.xml File.

If the `@JMSTransportClient` annotation is not configured automatically through `clientgen`, you can add it to the file manually.

Optionally, you can configure the following JMS transport properties using the `@JMSTransportClient` annotation. For a description of the properties, see Table 17-6.

- `destinationName`
- `destinationType`
- `enabled`
- `jmsMessageHeader`
- `jmsMessageProperty`
- `jndiConnectionFactoryName`
- `jndiContextParameters`
- `jndiInitialContextFactory`
- `jndiURL`
- `messageType`
- `priority`
- `replyToName`
- `targetService`
- `timeToLive`

Example 17-8 shows an excerpt from a client file that uses the `@JMSTransportClient` annotation, with the relevant code in **bold**.

Example 17-8 Enabling JMS Transport for a Client Proxy Using the @JMSTransportClient Annotation

```
...
import javax.xml.ws.WebServiceClient;
import com.oracle.webservices.api.jms.JMSTransportClient;
...
```

```
@WebServiceClient(name = "WarehouseService", targetNamespace = "http://oracle.com/samples/",
                  wsdlLocation="WarehouseService.wsdl")
@JMSTransportClient (
    destinationName="myQueue",
    replyToName="myReplyToQueue",
    jndiURL="t3://localhost:7001",
    jndiInitialContextFactory="weblogic.jndi.WLInitialContextFactory" ,
    jndiConnectionFactoryName="weblogic.jms.ConnectionFactory" ,
    timeToLive=1000, priority=1,
    messageType=com.oracle.webservices.api.jms.JMSMessageType.TEXT
)

public class WarehouseService  extends Service { ... }
```

Using the JMSTransportClientFeature Client API

You can use the `com.oracle.webservices.api.jms.JMSTransportClientFeature` client API to configure JMS transport in the web service client.

Optionally, you can configure the following JMS transport properties using the `com.oracle.webservices.api.jms.JMSTransportClientFeature`. For a description of the properties, see Table 17-6.

- `destinationName`
- `destinationType`
- `enabled`
- `jmsMessageHeader`
- `jmsMessageProperty`
- `jndiConnectionFactoryName`
- `jndiContextParameters`
- `jndiInitialContextFactory`
- `jndiURL`
- `messageType`
- `priority`
- `replyToName`
- `targetService`
- `timeToLive`

Example 17-9 shows an excerpt from a Web client that uses `JMSTransportClientFeature`, with the relevant code in **bold**.

Example 17-9 Enabling JMS Transport for a Client Proxy Using JMSTransportClientFeature

```
...
import javax.xml.namespace.QName;
import java.net.URL;
import com.oracle.webservices.api.jms.JMSTransportClientFeature;
...
URL url = new URL("http://localhost:7001/WarehouseServicePort/WarehouseService?WSDL");
QName serviceName = new QName("http://www.oracle.com/samples/", "WarehouseService");
WarehouseService service = new WarehouseService (url, serviceName);
```

```
JMSTransportClientFeature feature =
JMSTransportClientFeature.builder().jndiInitialContextFactory("weblogic.jndi.WLInitialContextFactory").j
ndiURL("t3://localhost:7001").build();
port = service.getWarehouseShipmentsPort(new WebServiceFeature[]{feature});
Item item = new Item();
item.setProductNumber(10001);
item.setQuantity(100);
port.shipGoods(item, "BEA");
...
```

Example 17-10 shows an excerpt from a Dispatch client that uses `JMSTransportClientFeature`, with the relevant code in **bold**.

Example 17-10 Enabling JMS Transport for a Dispatch Client Using JMSTransportClientFeature

```
...
import javax.xml.namespace.QName;
import java.net.URL;
import javax.xml.bind.JAXBContext;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.Dispatch;
import com.oracle.webservices.api.jms.JMSTransportClientFeature;
...
Service service = Service.create(new URL(wsdl), new QName(nameSpace, serviceName));
JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
JMSTransportClientFeature feature =
JMSTransportClientFeature.builder().jndiURL("t3://adc2170585:7003").build();
Dispatch dispatch =
      service.createDispatch(new QName(nameSpace, "WarehouseServicePort"), jaxbContext,
      Service.Mode.PAYLOAD, new WebServiceFeature[]{feature});
...
```

Configuring the JMS URI as the Target Endpoint Address

You can specify the JMS URI as the target endpoint address for the client binding to enable and configure JMS transport in the web service client. For information about constructing the JMS URI, see Configuring the JMS URI.

Example 17-11 shows an excerpt from a Web client that sets the target endpoint address to the JMS URI with the relevant code in **bold**. In this example, if the `replyToName` had been configured using `JMSTransportClientFeature`, it would take precedence over the target endpoint address value.

Example 17-11 Enabling JMS Transport for a Client Proxy Using JMS URI

```
...
import javax.xml.namespace.QName;
import java.net.URL;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.MessageContext;
import com.oracle.webservices.api.jms.JMSTransportClientFeature;
...
URL url = new URL("http://localhost:7001/WarehouseServicePort/WarehouseService?WSDL");
QName serviceName = new QName("http://www.oracle.com/samples/", "WarehouseService");
WarehouseService service = new WarehouseService (url, serviceName);
JMSTransportClientFeature feature = new JMSTransportClientFeature().build();
feature.setJndiInitialContextFactory ("weblogic.jndi.WLInitialContextFactory");
feature.setJndiUrl("t3://localhost:7001");
port = service.getWarehouseShipmentsPort(new WebServiceFeature[]{feature});
BindingProvider bp = (BindingProvider) port;
```

```
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
     "jms:jndi:myQueue?targetService=WarehouseService&replyToName=myReplyToQueue");
Item item = new Item();
item.setProductNumber(10001);
item.setQuantity(100);
port.shipGoods(item, "BEA");
...
```

Example 17-12 shows an excerpt from a Dispatch client that uses `JMSTransportClientFeature`, with the relevant code in **bold**. In this example, the JMS transport properties specified in the `JMSTransportClientFeature` take precedence over the JMS URI.

Example 17-12 Example of Enabling JMS Transport for a Dispatch Client Using JMS URI

```
...
String uri = "jms:jndi:myQueue?
targetService=WarehouseService&jndiConnectionFactoryName=weblogic.jms.ConnectionFactory&jndiURL=t
3://adc2170585:7003&jndiInitialContextFactory=weblogic.jndi.WLInitialContextFactory";
Service service = Service.create(new URL(wsdl), new QName(nameSpace, serviceName));
JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
JMSTransportClientFeature feature = new JMSTransportClientFeature().build();
feature.setJndiUrl("t3://adc2170585:7003");

Dispatch dispatch =
      service.createDispatch(new QName(nameSpace, "WarehouseServicePort"), jaxbContext,
      Service.Mode.PAYLOAD, new WebServiceFeature[]{feature});

dispatch.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, uri );
...
```

Using AsyncClientTransportFeature to Configure Asynchronous Clients

The asynchronous client transport feature, described in Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport), provides a scalable asynchronous client programming model.

To enable SOAP over JMS transport for an asynchronous client:

1. Specify the JMS URI as the target endpoint address for the client binding. For information about constructing the JMS URI, see Configuring the JMS URI.

   > **Note:**
   >
   > When using JMS transport, the context path of the asynchronous response endpoint is ignored if specified using `AsyncClientTransportFeature`.

2. Optionally, configure a *permanent* response queue by configuring the `address` or `ReplyTo` header using the `AsyncClientTransportFeature`, as described in Enabling and Configuring the Asynchronous Client Transport Feature.

   If you do not configure the address of the JMS response queue or if the `destinationName` property is set to anonymous (which is not supported by JMS

transport), then a temporary response queue is used. For more information about configuring the JMS response queue, see Configuring the JMS Response Queue.

Example 17-13 Example of Enabling JMS Transport and Configuring Permanent Queue for an Asynchronous Client

```
...
WarehouseService service = new WarehouseService(url, serviceName);
AsyncClientTransportFeature replyTo = new AsyncClientTransportFeature (
                            "jms:jndi:myReplyToQueue?targetService=WarehouseService");
AsyncClientTransportFeature faultTo = new AsyncClientTransportFeature (
                            "jms:jndi:myFaultToQueue?targetService=WarehouseService");
AsyncClientTransportFeature callbackFeature =  new AsyncClientTransportFeature (
               replyTo.getEndpointReference(W3CEndpointReference.class),
               faultTo.getEndpointReference(W3CEndpointReference.class));
port = service.getWarehouseServicePort(new WebServiceFeature[] { callbackFeature });
status = port.shipGoods(item, "BEAN");
(BindingProvider) port.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                  "jms:jndi:myQueue?targetService=WarehouseService");
((Closable)port).close();
...
```

When the endpoint is invoked, the client runtime environment publishes the response endpoint and deploys the JMS listener on the response JMS queue. Once attached to the client, the `AsyncClientTransportFeature` instance determines the response endpoint of all client invocations; the `replyToName` property in the target endpoint address and `JMSTransportClientFeature` are ignored.

Example 17-13 shows an excerpt from an asynchronous client that uses `AsyncClientTransportFeature`, with the relevant code in **bold**. In this example, the `replyTo` and `faultTo` addresses are defined and passed to the client.

Configuring JMS Transport Properties

Optionally, you can configure JMS transport properties when enabling JMS transport, as described in the following sections:

- Summary of JMS Transport Configuration Properties
- Configuration Methods and Order of Precedence
- Configuring JMS Transport Using the Administration Console
- Configuring JMS Transport Using WLST
- Configuring the JMS URI
- Configuring the JMS Request URI
- Configuring the WS-Addressing Headers
- Configuring the JMS Response Queue
- Configuring the JMS Message Type
- Configuring HTTP Access to the WSDL File

Summary of JMS Transport Configuration Properties

Table 17-6 lists the JMS transport properties that can be configured and the supported configuration methods, defined in Table 17-7.

Table 17-6 Summary of JMS Transport Configuration Properties

Name	Description	Supported Configuration Methods
`activationConfig`	Activation configuration properties passed to the JMS provider. Each property is specified as name-value pairs using the following format: `"name1=value1;...;nameN=valueN"` For example: `"key1=value1;key2=value2"` For a list of activation configuration properties that are supported by this property, see Table 17-7. This value defaults to: `""`	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`bindingVersion`	Version of the SOAP JMS binding. This value must be set to `1.0` for this release, which equates to `com.oracle.webservices.api.jms.JMSBindingVersion.SOAP_JMS_1_0`. This value maps to the `SOAPJMS_bindingVersion` JMS message property, as defined in Table 17-1.	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`deliveryMode`	Delivery mode indicating whether the request message is persistent. Valid values are `com.oracle.webservices.api.jms.JMSDeliveryMode.PERSISTENT` and `com.oracle.webservices.api.jms.JMSDeliveryMode.NON_PERSISTENT`. This value defaults to: `com.oracle.webservices.api.jms.JMSDeliveryMode.PERSISTENT`	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`destinationName`	JNDI name of the destination queue or topic. This value defaults to: `"com.oracle.webservices.api.jms.RequestQueue"`	All configuration methods in Table 17-8
`destinationType`	Destination type. Valid values include: `com.oracle.webservices.api.jms.JMSDestinationType.QUEUE` or `com.oracle.webservices.api.jms.JMSDestinationType.TOPIC`. This value defaults to: `com.oracle.webservices.api.jms.JMSDestinationType.QUEUE` This value overrides the `destinationType` value specified as an entry in `activationConfig` property (as defined in Table 17-7), if applicable. Topics are supported only for one-way communication.	All configuration methods in Table 17-8

Table 17-6 (Cont.) Summary of JMS Transport Configuration Properties

Name	Description	Supported Configuration Methods
`enabled`	Boolean flag that specifies whether JMS transport is enabled. This value defaults to `true`.	`@JMSTransportService` and `@JMSTransportClient` annotations
`enableHttpWsdlAccess`	Boolean flag that specifies whether to publish the WSDL through HTTP. This flag defaults to `true`.	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`jmsMessageHeader`	JMS header properties. Each property is specified as name-value pairs using the following format: "`name1=value1&...&nameN=valueN`" For example: "`JMSType=car&JMSPriority=4`" This value defaults to: ""	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation • `@JMSTransportClient` annotation
`jmsMessageProperty`	JMS message properties. Each property is specified as name-value pairs using the following format: "`name1=value1&...&nameN=valueN`" For example: "`JMSType=car&JMSPriority=4`" This value defaults to: ""	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation • `@JMSTransportClient` annotation
`jndiConnectionFactoryName`	JNDI name of the connection factory that is used to establish a JMS connection. This value defaults to: "`com.oracle.webservices.api.jms.ConnectionFactory`"	All configuration methods in Table 17-8
`jndiContextParameter`	JNDI properties. Each property is specified as name-value pairs using the following format: "`name1=value1&...&nameN=valueN`" The properties are added to the `java.util.Hashtable` sent to the `InitialContext` constructor for the JNDI provider. This value defaults to: ""	All configuration methods in Table 17-8
`jndiInitialContextFactory`	Name of the initial context factory class used for JNDI lookup. This value maps to the `java.naming.factory.initial` property. This value defaults to: "`weblogic.jndi.WLInitialContextFactory`"	All configuration methods in Table 17-8
`jndiURL`	JNDI provider URL. This value defaults to: "`t3://localhost:7001`" This value maps to the `java.naming.provider.url` property.	All configuration methods in Table 17-8

Table 17-6 (Cont.) Summary of JMS Transport Configuration Properties

Name	Description	Supported Configuration Methods
`lookupVariant`	Method used for looking up the specified destination name. This value must be set to `jndi` to support JMS transport; this is the default.	None (cannot be modified)
`mdbPerDestination`	Boolean flag that specifies whether to create one listening message-driven bean (MDB) for each requested destination. This value defaults to `true`. If set to `false`, one listening MDB is created for each web service port, and that MDB cannot be shared by other ports.	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`messageType`	Message type to use with the request message. Valid values are `com.oracle.webservices.api.jms.JMSMessageType.BYTES` and `com.oracle.webservices.api.jms.JMSMessageType.TEXT`. This value defaults to: `com.oracle.webservices.api.jms.JMSMessageType.BYTES` For more information about configuring the message type, see Configuring the JMS Message Type.	All configuration methods in Table 17-8
`priority`	JMS priority associated with the request and response message. Specify this value as a positive Integer from 0, the lowest priority, to 9, the highest priority. The default value is `0`.	All configuration methods in Table 17-8
`replyToName`	JNDI name of the JMS destination to which the response message is sent. For a two-way operation, a temporary response queue is generated by default. Using the default temporary response queue minimizes the configuration that is required. However, in the event of a server failure, the response message may be lost. This property enables the client to use a previously defined, "permanent" queue or topic rather than use the default temporary queue or topic, for receiving replies. For more information about configuring the JMS response queue, see Configuring the JMS Response Queue. The value maps to the `JMSReplyTo` JMS header in the request message. This value defaults to: `""`	All configuration methods in Table 17-8

Table 17-6 (Cont.) Summary of JMS Transport Configuration Properties

Name	Description	Supported Configuration Methods
`runAsPrincipal`	Principal used to run the listening MDB. This value defaults to: `""`	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`runAsRole`	Role used to run the listening MDB. This value defaults to: `""`	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation
`targetService`	Port component name of the web service. This value is used by the service implementation to dispatch the service request. If not specified, the service name from the WSDL or `@javax.jws.WebService` annotation is used. This value maps to the `SOAPJMS_targetService` JMS message property. This value defaults to: `""`	• `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task • `@JMSTransportService` annotation • `@JMSTransportClient` annotation
`timeToLive`	Lifetime, in milliseconds, of the request message. A value of 0 indicates an infinite lifetime. If not specified, the JMS-defined default value of `180000L` is used. On the service side, `timeToLive` also specifies the expiration time for each MDB transaction.	All configuration methods in Table 17-8

The following table lists the activation properties that are supported by the `activationConfig` property in Table 17-6. For information about using the activation properties to tune MDBs, see Tuning Message-Driven Beans in *Tuning Performance of Oracle WebLogic Server*.

Table 17-7 Activation Properties Supported by the activationConfig Property

Name	Description
`acknowledgeMode`	Acknowledgment mode that controls how the JMS provider is notified that the message was received and processed. Valid values include: • `AUTO_ACKNOWLEDGE`—Message is acknowledged immediately. This is the default. • `DUPS_OK_ACKNOWLEDGE`—Acknowledgement may be delayed, allowing duplicate messages to be received. The acknowledgement mode is ignored if you are using container-managed transactions. (In this case, the acknowledgement is performed within the context of the transaction.)
`connectionFactoryJndiName`	JNDI name of the JMS connection factory that the MDB uses to create its queues and topics. This value defaults to `weblogic.jms.MessageDrivenBeanConnectionFactory`.
`destinationJndiName`	JNDI name used to associate an MDB with an actual JMS queue or topic deployed in the WebLogic Server JNDI tree.
`destinationType`	Type of the JMS destination. Valid values include: `QUEUE` and `TOPIC`.

Table 17-7 (Cont.) Activation Properties Supported by the activationConfig Property

Name	Description
`dispatchPolicy`	Work manager for the MDB. This value defaults to `weblogic.wsee.jaxws.mdb.DispatchPolicy`.
`distributedDestinationConnection`	Connection setting that specifies whether an MDB that accesses a WebLogic JMS distributed destination (topic or queue) in the same cluster consumes from all distributed destination members or only those members local to the current WebLogic Server instance. Valid values include: • `LocalOnly`—MDB consumes JMS distributed destinations from members local to the current WebLogic Server instance. This is the default. • `EveryMember`—MDB consumes JMS distributed destinations from all distributed destination members.
`durableSubscriptionDeletion`	Flag that specifies whether you want durable topic subscriptions to be automatically deleted when an MDB is undeployed or removed. This value defaults to `false`.
`initialContextFactory`	Initial context factory that the EJB container uses to create its connection factories. This value defaults to `weblogic.jndi.WLInitialContextFactory`.
`initSuspendSeconds`	Initial number of seconds to suspend an MDB's JMS connection when the EJB container detects a JMS resource outage. This value can be set to any Integer value and defaults to `5`.
`jmsClientId`	Client ID for the MDB when it connects to a JMS destination. This value is used for durable subscriptions to JMS topics.
`jmsPollingIntervalSeconds`	Number of seconds between attempts by the EJB container to reconnect to a JMS destination that has become unavailable. This value can be set to any Integer value and defaults to `10`.
`maxBeansInFreePool`	Maximum number of inactive MDBs in the free pool. This value can be set to any positive Integer value or 0. This value defaults to `1000`.
`maxMessagesInTransaction`	Maximum number of messages that can be in a transaction for this MDB. This value can be set to any positive Integer value or 0. This value defaults to `1`.
`maxSuspendSeconds`	Maximum number of seconds to suspend an MDB's JMS connection when the EJB container detects a JMS resource outage. This value can be set to any Integer and defaults to `60`.
`messageSelector`	String used by a client to specify, by header field references and property references, the messages it should receive. Only messages whose header and property values match the specified selector are delivered. This value can be set to a message header or a conditional expression using message properties. This value defaults to `null`.
`providerURL`	URL provider to be used by the `InitialContext`, typically, `host:port`. This value can be any valid URL and defaults to `null`.
`subscriptionDurability`	Flag that specifies whether a JMS topic subscription is `Durable` or `NonDurable`. This value defaults to `Durable`.
`topicMessagesDistributionMode`	Distribution mode for topic messages. Valid values include: `One-Copy-Per-Application`, `One-Copy-Per-Server`, `Compatibilty`. This value defaults to `Compatibility`. For more information about the valid values, see Topic Deployment Scenarios in *Developing Message-Driven Beans for Oracle WebLogic Server*.
`transAttribute`	Transaction setting that specifies how the container must manage the transaction boundaries when delegating a method invocation to an enterprise bean's business method. Valid values include: `Required`, `NotSupported`, `Supports`, `RequiresNew`, `Mandatory`, and `Never`. This value defaults to `Required`. For more information about the valid values, see Using Web Services Atomic Transactions .

Table 17-7 (Cont.) Activation Properties Supported by the activationConfig Property

Name	Description
`transTimeoutSeconds`	Maximum duration for an EJB's container-initiated transactions, in seconds, after which the transaction is rolled back and the service will return a SOAP fault. This value can be set to any positive Integer or 0. If the transaction timeout is not specified or is set to 0, the transaction timeout configured for the domain is used. If a timeout is not configured for the domain, the default is `30`.
`use81StylePolling`	Flag that specifies whether backwards compatibility for WebLogic Server version 8.1-style polling is supported. Valid values include: `True` or `False`. This value defaults to `False`.

Configuration Methods and Order of Precedence

Optionally, you can configure JMS transport properties when enabling JMS transport using one of the methods defined in Table 17-8.

Table 17-8 Methods Used to Configure JMS Properties.

Configuration Methods	Description
`JMSTransportClientFeature` API	Create the web service client and pass JMS transport properties as arguments to the `weblogic.jws.jaxws.client.JmsTransportClientFeature` API. For more information, see Invoking a WebLogic Web Service Using JMS Transport.
Target service endpoint address	Construct the target service endpoint address and include JMS transport properties as part of the query string. For more information, see Configuring the JMS URI.
`@JMSTransportClient` annotation	Create the web service client and pass JMS transport properties as attributes to the `@com.oracle.webservices.api.jms.JMSTransportClient` annotation in the JWS file, as described in Using the @JMSTransportClient Annotation.
`@JMSTransportService` annotation	Create the web service and pass JMS transport properties as attributes to the `@com.oracle.webservices.api.jms.JMSTransportService` annotation in the JWS file, as described in Using the @JMSTransportService Annotation.
`<jmstransportclient>` element of the `clientgen` Ant task	Build the web service including the `<jmstransportclient>` element in the `clientgen` Ant task For more information, see Using the <jmstransportclient> Element in the Ant build.xml File.
`<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task	Build the web service including the `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task For more information, see Using the <jmstransportservice> Child Element in the Ant build.xml File.
WSDL	Create the web service from a WSDL that includes JMS transport property elements, as defined in Configuring JMS Transport Properties in the WSDL.
Administration Console	Configure the JMS transport properties for the deployed web service using the WebLogic Server Administration Console, as described in Configuring JMS Transport Using the Administration Console.

Table 17-8 (Cont.) Methods Used to Configure JMS Properties.

Configuration Methods	Description
`<soapjms-service-endpoint-address>` element in the `weblogic-webservices.xml` deployment descriptor	You can update the `weblogic-webservices.xml` deployment descriptor manually, though it is not recommended. For more information about the `<soapjms-service-endpoint-address>` elements, see WebLogic Web Service Deployment Descriptor Schema Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*.

The following summarizes the order of precedence for JMS transport property configuration on the web service or client at design time and run time:

- For the web service at **design time** (from high to low):
  - `<jmstransportservice>` child element in the `<jws>` element of the `jwsc` Ant task
  - `@JMSTransportService` annotation
- For the web service at **run time** (from high to low):
  - Administration Console
  - `<soapjms-service-endpoint-address>` element in the `weblogic-webservices.xml` deployment descriptor
  - `@JMSTransportService` annotation
- For the client at **design time** (from high to low):
  - `<jmstransportclient>` child element of clientgen
  - JMS transport properties defined in the WSDL
- For the client at **run time** (from high to low):
  - JMS URI service endpoint address
  - `JMSTransportClientFeature` API
  - `@JMSTransportClient` annotation

Configuring JMS Transport Using the Administration Console

After you have deployed your web service with JMS transport enabled, you can configure JMS transport properties using the WebLogic Server Administration Console.

To configure JMS transport properties using the WebLogic Server Administration Console:

1. Invoke the WebLogic Server Administration Console, as described in Invoking the Administration Console in *Understanding WebLogic Web Services for Oracle WebLogic Server*.
2. In the left navigation pane, select **Deployments**.
3. Click the name of the web service in the Deployments table.
4. Select the **Configuration** tab, then the **Port Components** tab.

5. Click the name of the web service endpoint in the Ports table.
6. Select the **SOAP over JMS Transport** tab.
7. Click **Customize SOAP over JMS Transport Configuration** and follow the instructions to save the deployment plan, if required.
8. Edit the SOAP over JMS transport properties, as described in Configuring JMS Transport Properties.
9. Click **Save**.

For more information, see Configuring SOAP Over JMS Transport in the *Oracle WebLogic Server Administration Console Online Help*.

Configuring JMS Transport Using WLST

Alternatively, you can use WLST to configure JMS transport. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

Configuring the JMS URI

When a WebLogic web service is configured to use SOAP over 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://`.

The JMS URI format is shown below:

```
jms:lookupVariant:destinationName[?targetService=value[&property=value]
[&property=value]...
]
```

The JMS URI is constructed as follows:

- Prefix `jms:`
- Lookup variant type (must be set to `jndi`)
- JMS destination name (`destinationName`)
- Query string containing a list of property-value pairs used to specify JMS endpoint information. The `targetService` property must be specified to define the port component name of the web service.

  Other valid properties include:

  - `bindingVersion`
  - `deliveryMode`
  - `deliveryType`
  - `jndiConnectionFactoryName`
  - `jndiContextParameter`
  - `jndiInitialContextFactory`
  - `jndiURL`
  - `messageType`
  - `priority`

- replyToName
- timeToLive

The `lookupVariant`, `destinationName`, and `targetService` JMS properties are required in the JMS endpoint address.

For more information about the JMS transport properties that construct the JMS URI, see Table 17-6. Optionally, you can configure JMS transport properties when enabling JMS transport using one of the methods defined in Table 17-7.

Examples:

The following provides an example of a JMS endpoint address. In this example, the JMS destination is `myQueue` and the port component name of the web service is `WarehouseServicePort`.

```
jms:jndi:myQueue?targetService=WarehouseServicePort
```

The following example shows the same JMS endpoint address with replyToName property set to specify the JNDI name of the JMS destination to which the response message is sent.

```
jms:jndi:myQueue?targetService=WarehouseServicePort&replyToName=myReplyToQueue
```

The following example shows how to specify additional JNDI environment properties, such as `jndi-com.acme.jndi.enable.tracing` and `jndi-java.naming.referral`.

```
jms:jndi:myQueue?targetService=WarehouseServicePort&jndi-
com.acme.jndi.enable.tracing=true&jndi-java.naming.referral=ignore
```

Configuring the JMS Request URI

Each JMS transport message has a message property defined as `SOAPJMS_requestURI` that is derived from the JMS URI. The JMS Request URI is constructed using the JMS URI and stripping off the query parameters.

The JMS request URI format is shown below:

```
jms:lookupVariant:destinationName
```

The JMS Request URI is constructed as follows:

- Prefix `jms:`
- Lookup variant type (must be set to `jndi`)
- JMS destination name (`destinationName`)

For more information about the JMS transport properties that construct the JMS Request URI, see Table 17-6. Optionally, you can configure JMS transport properties when enabling JMS transport using one of the methods defined in Table 17-7.

Example:

The following provides an example of a JMS endpoint address. In this example, the JMS destination is `myQueue`.

```
jms:jndi:myQueue
```

Configuring the WS-Addressing Headers

Web services and clients that use SOAP over JMS transport populate the WS-Addressing headers `To` and `ReplyTo` of the request and response messages with a value that is derived from the JMS URI.

The WS-Addressing header format is shown below:

`jms:`*`lookupVariant`*`:`*`destinationName`*`?targetService=`*`value`*

For more information about the JMS transport properties that construct the WS-Addressing headers, see Table 17-6. Optionally, you can configure JMS transport properties when enabling JMS transport using one of the methods defined in Table 17-7.

Examples:

The following provides an example of the WS-Addressing headers in a SOAP request message.

```
<S:Header>
   <To xmlns="http://www.w3.org/2005/08/addressing">
      jms:jndi:myQueue?targetService=WarehouseService
   </To>
   <Action xmlns="http://www.w3.org/2005/08/addressing">
      http://www.oracle.com/samples/ShipGoodsRequest
   </Action>
   <ReplyTo xmlns="http://www.w3.org/2005/08/addressing">
      <Address>jms:jndi:myReplyToQueue?targetService=WarehouseService</Address>
   </ReplyTo>
   <MessageID xmlns="http://www.w3.org/2005/08/addressing">
      uuid:3b9e7b20-3aa0-4a4a-9422-470fa7b9ada1
   </MessageID>
</S:Header>
```

The following provides an example of the WS-Addressing headers in a SOAP response message.

```
<S:Header>
   <To xmlns="http://www.w3.org/2005/08/addressing">
      jms:jndi:myReplyToQueue?targetService=WarehouseService
   </To>
   <Action xmlns="http://www.w3.org/2005/08/addressing">
      http://www.oracle.com/samples/ShipGoodsResponse
   </Action>
   <MessageID xmlns="http://www.w3.org/2005/08/addressing">
      uuid:9d0be951-79fc-4a56-b3e6-4775bde2bd82
   </MessageID>
   <RelatesTo xmlns="http://www.w3.org/2005/08/addressing">
      uuid:3b9e7b20-3aa0-4a4a-9422-470fa7b9ada1
   </RelatesTo>
</S:Header>
```

Configuring the JMS Response Queue

For a two-way operation, a temporary response queue is generated by default. Using the default temporary response queue minimizes the configuration that is required. However, in the event of a server failure, the response message may be lost.

You can configure a "permanent" JMS response queue—one that is available after server restart. A permanent JMS response queue provides the following benefits:

- Ensures that the response message can be restored following a server restart.
- Improves performance, avoiding the overhead required to create the temporary queue at initial invocation.
- Enables you to configure the queue for quality of service (QoS).

You can configure the JMS response queue using one of the following methods (in order of precedence):

- Configuring the `address` or `ReplyTo` header using the `AsyncClientTransportFeature`, as described in Enabling and Configuring the Asynchronous Client Transport Feature.
- Configuring the `replyToName` property using one of the following methods:
  - `<jmstransportclient>` element of `clientgen`, as described in Using the <jmstransportclient> Element in the Ant build.xml File.
  - Target endpoint address, as described in Summary of JMS Transport Configuration Properties.
  - `JMSTransportClientFeature`, as described in Using the JMSTransportClientFeature Client API.
  - `@JMSTransportClient` annotation, as described in Using the @JMSTransportClient Annotation.

> **Note:**
>
> If the destinationName property is set to anonymous (which is not supported by JMS transport), then a temporary response queue is used.

By default, the JMS response queue is used as the fault queue for JMS transport service invocation. You can configure the `faultTo` header using the `AsyncClientTransportFeature`, as described in Configuring the ReplyTo and FaultTo Headers of the Asynchronous Response Endpoint.

Configuring the JMS Message Type

You can configure one of the following message types to use with the request message.

- `com.oracle.webservices.api.jms.JMSMessageType.BYTES`—The body of the JMS message is binary data. This is the default.
- `com.oracle.webservices.api.jms.JMSMessageType.TEXT`— The body of the JMS message is String data.

You can configure the `messageType` property using any of the configuration methods defined in Table 17-8.

The web service uses the same message type when sending the response. If the request is received as a `BYTES`, the reply will be sent as a `BYTES`.

When setting the `messageType` property to `TEXT`, consider the following:

- For large payloads, the memory requirements for `TEXT` messages can be significantly greater than `BYTES` messages because the data requirements for the in-memory representation is larger.
- Messages with binary attachments must be base64-encoded, which can also increase the size of the message significantly.

Configuring HTTP Access to the WSDL File

By default, the WSDL of the deployed web service is 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 `enableHttpWsdlAccess` attribute of the `<jmstransportservice>` child element of the `<jws>` element, as described in Using the <jmstransportservice> Child Element in the Ant build.xml File.
- `@JMSTransportService` annotation, as described in Using the @JMSTransportService Annotation.
- Use the WebLogic Server 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 WebLogic Server Administration Console to perform this task, go to the Configuration -> General page of the deployed web service and uncheck the **View Dynamic WSDL Enabled** check box. 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.

Monitoring SOAP Over JMS Transport

You can monitor web services that use SOAP over JMS transport from the following perspectives:

- Monitor web service performance, as described in:
  - Monitor SOAP Web Services in *Oracle WebLogic Server Administration Console Online Help*
  - Monitoring and Auditing Web Services in *Administering Web Services*
- Monitor JMS destination metrics, as described in Monitoring JMS Statistics and Managing Messages in *Administering JMS Resources for Oracle WebLogic Server*.

18

Creating and Using SOAP Message Handlers

This chapter describes how to create and use SOAP message handlers for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of SOAP Message Handlers

Web services and their clients may need to access the SOAP message for additional processing of the message request or response. 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 SOAP message handlers to enable web services and clients to perform additional processing on the SOAP message.

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

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

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

> **Note:**
>
> If SOAP handlers are used in conjunction with policies (security, WS-ReliableMessaging, MTOM, and so on), for inbound messages, the policy interceptors are executed before the user-defined message handlers. For outbound messages, this order is reversed.

Adding Server-side SOAP Message Handlers: Main Steps

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

It is assumed that you have created a basic JWS file that implements a web service and that you want to update the web service by adding SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `jwsc` Ant task. For more information, see:

- Developing JAX-WS Web Services
- Programming the JWS File
- Developing Web Service Clients

Table 18-1 Steps to Add SOAP Message Handlers to a Web Service

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

Adding Client-side SOAP Message Handlers: Main Steps

You can configure client-side SOAP message handlers for both standalone clients and clients that run inside of WebLogic Server. You create the actual Java client-side handler in the same way you create a server-side handler (by creating a Java class that implements the SOAP message handler interface). In many cases you can use the exact same handler class on both the web service running on WebLogic Server

and the client applications that invoke the web service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client.

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

It is assumed that you have created the client application that invokes a deployed web service, and that you want to update the client application by adding client-side SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `clientgen` Ant task. For more information, see Invoking a Web Service from a Java SE Client.

Table 18-2 Steps to Add SOAP Message Handlers to a Web Service Client

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

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

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

Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers, you must decide:

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

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

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

Server-side Handler Execution

When invoking a web service, WebLogic Server executes handlers as follows:

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

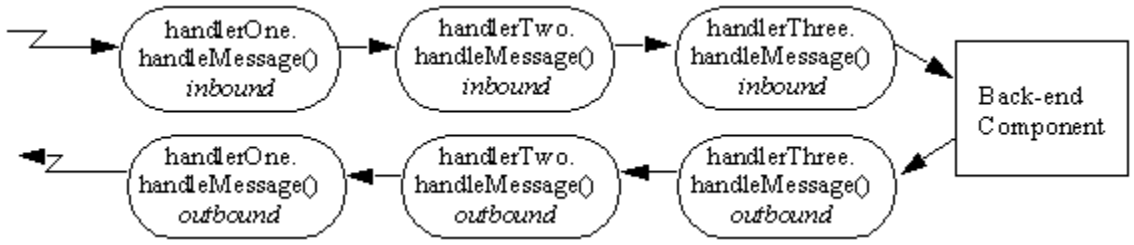
For example, assume that you are going to use the `@HandlerChain` JWS annotation in your JWS file to specify an external configuration file, and the configuration file defines a handler chain called `SimpleChain` that contains three handlers, as shown in the following sample:

```
<?xml version="1.0" encoding="UTF-8" ?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
   <handler-chain>
      <handler>
         <handler-class>
            Handler1
         </handler-class>
      </handler>
   </handler-chain>
   <handler-chain>
      <handler>
         <handler-class>
            Handler2
         </handler-class>
      </handler>
   </handler-chain>
   <handler-chain>
      <handler>
         <handler-class>
            Handler3
         </handler-class>
```

```
      </handler>
   </handler-chain>
</handler-chains>
```

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

Figure 18-1 Order of Execution of Handler Methods



Client-side Handler Execution

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

- Directly before the client application sends the SOAP request to the web service
- Directly after the client application receives the SOAP response from the web service

Creating the SOAP Message Handler

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

Table 18-3 Types of SOAP Message Handlers

Handler Type	Description
SOAP handler	Enables you to access the full SOAP message including headers. SOAP handlers are defined using the `javax.xml.ws.handler.soap.SOAPHandler` interface. They are invoked using the import `javax.xml.ws.handler.soap.SOAPMessageContext` which extends `javax.xml.ws.handler.MessageContext` The `SOAPMessageContext.getMessage()` method returns a `javax.xml.soap.SOAPMessage`.

Table 18-3 (Cont.) Types of SOAP Message Handlers

Handler Type	Description
Logical handlers	Provides access to the payload of the message. Logical handlers cannot change any protocol-specific information (like headers) in a message. Logical handlers are defined using the `javax.xml.ws.handler.LogicalHandler` interface (see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/LogicalHandler.html`). They are invoked using the `javax.xml.ws.handler.LogicalMessageContext` which extends `javax.xml.ws.handler.MessageContext` The `LogicalMessageContext.getMessage()` method returns a `javax.xml.ws.LogicalMessage`. The payload can be accessed either as a JAXB object or as a `javax.xml.transform.Source` object (see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/LogicalMessage.html`).

Each type of message handler extends the `javax.xml.ws.Handler` interface (see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/Handler.html`), which defines the methods defined in the following table.

Table 18-4 Handler Interface Methods

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

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

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the `javax.xml.soap.SOAPMessage` abstract class, which is part of the SOAP With Attachments API for Java 1.3 (SAAJ) specification at `https://jcp.org/en/jsr/detail?id=67` For details, see Directly Manipulating the SOAP Request and Response Message Using SAAJ.

Example of a SOAP Handler

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

```
package examples.webservices.handler;

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

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

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

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

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

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

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

Example of a Logical Handler

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

```
package examples.webservices.handler;

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

public class Handler2 implements LogicalHandler<LogicalMessageContext>
{
```

```
  public Set<QName> getHeaders()
  {
    return Collections.emptySet();
  }

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

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

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

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

Implementing the Handler.handleMessage() Method

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

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

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

`C` extends `javax.xml.ws.handler.MessageContext` (see http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/MessageContext.html). The `MessageContext` properties allow the handlers in a handler chain to determine if a message is inbound or outbound and to share processing state. Use the `SOAPMessageContext` or `LogicalMessageContext` sub-interface of `MessageContext` to get or set the contents of the SOAP or logical message, respectively. For more information, see Using the Message Context Property Values and Methods.

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

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

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

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

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

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

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

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

Implementing the Handler.handleFault() Method

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

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

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

`C` extends `javax.xml.ws.handler.MessageContext` (see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/MessageContext.html`). The `MessageContext` properties allow the handlers in a handler chain to determine if a message is inbound or outbound and to share processing state.Use the `LogicalMessageContext` or `SOAPMessageContext` sub-interface of `MessageContext` to get or set the contents of the logical or SOAP message, respectively. For more information, see Using the Message Context Property Values and Methods.

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

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

Implementing the Handler.close() Method

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

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

Using the Message Context Property Values and Methods

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

Table 18-5 Message Context Property Values

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

Each context object extends `javax.xml.ws.handler.MessageContext`, which enables you to access a set of runtime properties of a SOAP message handler from the client application or web service, or directly from the `javax.xml.ws.WebServiceContext` from a web service (see https://docs.oracle.com/javase/8/docs/api/javax/xml/ws/WebServiceContext.html).

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

For more information about the `MessageContext` property values that are available, see Accessing the Web Service Context.

The `LogicalMessageContext` class defines the following method for processing the Logical message. For more information, see the `java.xml.ws.handler.LogicalMessageContext` Javadoc at http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/LogicalMessageContext.html.

Table 18-6 LogicalMessageContext Class Method

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

The `SOAPMessageContext` class defines the following methods for processing the SOAP message. For more information, see the `java.xml.ws.handler.soap.SOAPMessageContext` Javadoc at http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/soap/SOAPMessageContext.html.

> **Note:**
>
> The SOAP message itself is stored in a `javax.xml.soap.SOAPMessage` object at http://docs.oracle.com/javase/8/docs/api/javax/xml/soap/SOAPMessage.html. For detailed information on this object, see Directly Manipulating the SOAP Request and Response Message Using SAAJ.

Table 18-7 SOAPMessageContext Class Methods

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

Directly Manipulating the SOAP Request and Response Message Using SAAJ

The `javax.xml.soap.SOAPMessage` abstract class is part of the SOAP With Attachments API for Java (SAAJ) specification at https://jcp.org/aboutJava/communityprocess/mrel/jsr067/index4.html. You use the class to manipulate request and response SOAP messages when creating SOAP message handlers. This section describes the basic structure of a `SOAPMessage` object and some of the methods you can use to view and update a SOAP message.

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

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

The SOAPPart Object

> **Note:**
>
> The `setContent` and `getContent` methods of the `SOAPPart` object support `javax.xml.transform.stream.StreamSource` content only; the methods do not support `javax.xml.transform.dom.DOMSource` content.

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 (see `http://docs.oracle.com/javase/8/docs/api/javax/xml/soap/AttachmentPart.html`) contains the optional attachments to the SOAP message. Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file.

> **Note:**
>
> If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see Manipulating Image Attachments in a SOAP Message Handler for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments. For more information, see the `javax.xml.soap.SOAPMessage` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/soap/SOAPMessage.html`.

Table 18-8 SOAPMessage Class Methods to Manipulate Attachments

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

Manipulating Image Attachments in a SOAP Message Handler

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

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

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

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

Configuring Handler Chains in the JWS File

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

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

```
package examples.webservices.handler;

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

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

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

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

For details about creating the external configuration file, see Creating the Handler Chain Configuration File.

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`.

Creating the Handler Chain Configuration File

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-class>examples.webservices.handler.Handler1</handler-class>
    </handler>
  </handler-chain>
  <handler-chain>
    <handler>
      <handler-class>examples.webservices.handler.Handler2</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

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

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

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

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

To ensure that `handler-chain.xml` is added to the WAR file, add the following lines to `build.xml`, inside the JWS file, immediately under `</WLHttpTransport>`:

```
<zipfileset dir="src">
  <include name="**/handler-chain.xml"/>
</zipfileset>
```

See Developing JAX-WS 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 the `@HandlerChain` annotation, you must rerun the `jwsc` Ant task to recompile the JWS file and generate a new web service. This is true anytime you make a change to an annotation in the JWS file.

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

- The `jwsc` Ant task compiles SOAP message handler Java files into handler classes (and then packages them into the generated application) if all the following conditions are true:
  - The handler classes are referenced in the `@HandlerChain` annotation of the JWS file.
  - The Java files are located in the directory specified by the `sourcepath` attribute.
  - The classes are not currently in your CLASSPATH.

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

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

Configuring the Client-side SOAP Message Handlers

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

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

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

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

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

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

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

      Binding binding = ((BindingProvider)port).getBinding();
      List<Handler> handlerList = binding.getHandlerChain();
      handlerList.add(new Handler1());
      handlerList.add(new Handler2());
      binding.setHandlerChain(handlerList);
      String result = null;
      result = port.sayHello("foo bar");
      System.out.println( "Got result: " + result );
  }
}
```

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

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

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

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

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

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

```
        </handler-chain>
    </handler-chains>
</bindings>
```

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

19

Handling Exceptions Using SOAP Faults

This chapter describes how to handle exceptions that occur when a message is being processed using Simple Object Access Protocol (SOAP) faults for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of Exception Handling Using SOAP Faults

When a web service request is being processed, if an error is encountered, the nature of the error needs to be communicated to the client, or sender of the request. Because clients can be written on a variety of platforms using different languages, there must exist a standard, platform-independent mechanism for communicating the error.

The SOAP specification (available at `http://www.w3.org/TR/soap/`) defines a standard, platform-independent way of describing the error within the SOAP message using a *SOAP fault*. In general, a SOAP fault is analogous to an application exception. SOAP faults are generated by receivers to report business logic errors or unexpected conditions.

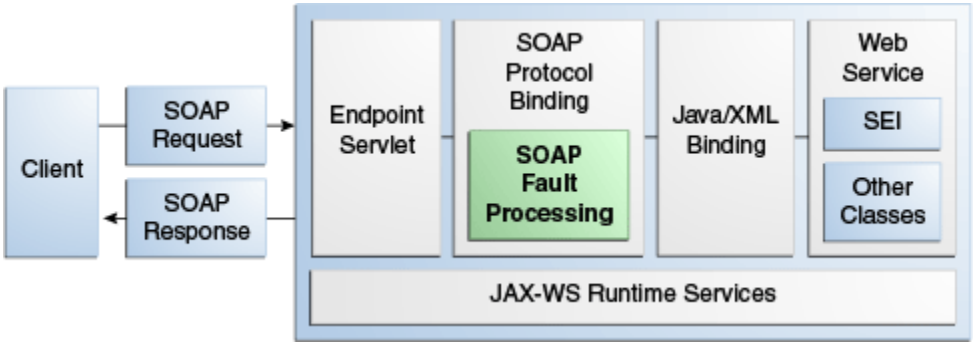
In JAX-WS, Java exceptions (`java.lang.Exception`) that are thrown by your Java web service are mapped to a SOAP fault and returned to the client to communicate the reason for failure. SOAP faults can be one of the following types:

- Modeled—Maps to an exception that is thrown explicitly from the business logic of the Java code and mapped to `wsdl:fault` definitions in the WSDL file, when the web service is deployed. In this case, the SOAP faults are predefined.
- Unmodeled—Maps to an exception (for example, `java.lang.RuntimeException`) that is generated at run-time when no business logic fault is defined in the WSDL. In this case, Java exceptions are represented as generic SOAP fault exceptions, `javax.xml.ws.soap.SOAPFaultException`.

The faults are returned to the sender only if request/response messaging is in use. If a web service operation is configured as one-way, the SOAP fault is not returned to the sender, but stored for further processing.

As illustrated in Figure 19-1, JAX-WS handles SOAP fault processing during SOAP protocol binding. The SOAP binding maps exceptions to SOAP fault messages.

Figure 19-1 How SOAP Faults Are Processed



Contents of the SOAP Fault Element

The SOAP `<Fault>` element is used to transmit error and status information within a SOAP message. The `<Fault>` element is a child of the body element. There can be only one `<Fault>` element in the body of a SOAP message.

The SOAP `<Fault>` element contents for SOAP 1.2 and 1.1 are defined in the following sections:

- SOAP 1.2 <Fault> Element Contents
- SOAP 1.1 <Fault> Element Contents

SOAP 1.2 <Fault> Element Contents

The `<Fault>` element for SOAP 1.2 contains the subelements defined in Table 19-1.

Table 19-1 Subelements of the SOAP 1.2 <Fault> Element

Subelement	Description	Required?
`env:Code`	Information pertaining to the fault error code. The `env:Code` element consists of the following two subelements: • `env:Value` • `env: Subcode` The subelements are defined below.	Required
`env:Value`	Code value that provides more information about the fault. A set of code values is predefined by the SOAP specification, including: • `VersionMismatch`—Invalid namespace defined in SOAP envelope element. The SOAP envelope must conform to the `http://schemas.xmlsoap.org/soap/envelope` namespace. • `MustUnderstand`—SOAP header entry not understood by processing party. • `Sender`—Message was incorrectly formatted or is missing information. • `Receiver`—Problem with the server that prevented the message from being processed. • `DataEncodingUnknown`—Received message has an unrecognized encoding style value. You can define encoding styles for SOAP headerblocks and child elements of the SOAP body, and this encoding style must be recognized by the web services server.	Required

Table 19-1 (Cont.) Subelements of the SOAP 1.2 <Fault> Element

Subelement	Description	Required?
`env:Subcode`	Subcode value that provides more information about the fault. This subelement can have a recursive structure.	Optional
`env:Reason`	Human-readable description of fault. The `<env:Reason>` element contains one or more `<Text>` elements, each of which contains information about the fault in a different language.	Required
`env:Node`	Information regarding the actor (SOAP node) that caused the fault.	Optional
`env:Role`	Role being performed by actor at the time of the fault.	Optional
`env:Detail`	Application-specific information, such as the exception that was thrown.	Optional

The following provides an example of a SOAP 1.2 fault message.

Example 19-1 Example of SOAP 1.2 Fault Message

```
<?xml version="1.0"?>
<env:Envelope xmlns:env=http://www.w3.org/2003/05/soap-envelope>
   <env:Body>
      <env:Fault>
         <env:Code>
            <env:Value>env:Sender</env:Value>
            <env:Subcode>
               <env:Value>rpc:BadArguments</env:Value>
            </env:Subcode>
         </env:Code>
         <env:Reason>
            <env:Text xml:lang=en-US>Processing error<env:Text>
         </env:Reason>
         <env:Detail>
            <e:myFaultDetails
               xmlns:e=http://travelcompany.example.org/faults>
               <e:message>Name does not match card number</e:message>
               <e:errorcode>999</e:errorcode>
            </e:myFaultDetails>
         </env:Detail>
      </env:Fault>
   </env:Body>
</env:Envelope>
```

SOAP 1.1 <Fault> Element Contents

The `<Fault>` element for SOAP 1.1 contains the subelements defined in Table 19-2.

Table 19-2 Subelements of the SOAP 1.1 <Fault> Element

Subelement	Description
`faultcode`	Standard code that provides more information about the fault. A set of code values is predefined by the SOAP specification, as defined below. This set of fault code values can be extended by the application. Predefined fault code values include: • `VersionMismatch`—Invalid namespace defined in SOAP envelope element. The SOAP envelope must conform to the `http://schemas.xmlsoap.org/soap/envelope` namespace. • `MustUnderstand`—SOAP header entry not understood by processing party. • `Client`—Message was incorrectly formatted or is missing information. • `Server`—Problem with the server that prevented message from being processed.
`faultstring`	Human-readable description of fault.
`faultactor`	URI associated with the actor (SOAP node) that caused the fault. In RPC-style messaging, the actor is the URI of the web service.
`detail`	Application-specific information, such as the exception that was thrown. This element can be an XML structure or plain text.

The following provides an example of a SOAP 1.1 fault message.

Example 19-2 Example of SOAP 1.1 Fault Message

```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap='http://schemas.xmlsoap.org/soap/envelope'>
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:VersionMismatch</faultcode>
         <faultstring, xml:lang='en">
            Message was not SOAP 1.1 compliant
         </faultstring>
         <faultactor>
            http://sample.org.ocm/jws/authnticator
         </faultactor>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

Using Modeled Faults

As described previously, a modeled fault is mapped to an exception that is thrown explicitly from the business logic of the Java code. In this case, the exception is mapped to a `wsdl:fault` definitions in the WSDL file, when the web service is deployed.

The following sections provide more information about using modeled faults:

- Creating and Using a Custom Exception
- How Modeled Faults are Mapped in the WSDL File
- How the Fault is Communicated in the SOAP Message
- Creating the Web Service Client

Creating and Using a Custom Exception

To use modeled faults, you need to create a custom Java exception and throw it from within your web service.

Example 19-3 provides a simple example of a custom exception being thrown by a web service. The exception is called `MissingName` and is thrown when the input argument is empty.

Example 19-3 Web Service With Custom Exception

```
package examples;
import javax.jws.WebService;

@WebService(name="HelloWorld", serviceName="HelloWorldService")
public class HelloWorld {
   public String sayHelloWorld(String message) throws MissingName {
        System.out.println("Say Hello World: " + message);
        if (message == null || message.isEmpty()) {
           throw new MissingName();
        }
      return "Here is the message: '" + message + "'";
      }
}
```

Example 19-4 shows the he class for the exception, `MissingName.java`.

Example 19-4 Custom Exception Class (MissingName)

```
package examples;
import java.lang.Exception;

public class MissingName extends Exception {
    public MissingName() {
        super("Your name is required.");
    }
}
```

How Modeled Faults are Mapped in the WSDL File

The JAX-WS Java-to-WSDL mapping binds subclasses of `java.lang.Exception` to `wsdl:fault` messages. Example 19-4 shows the WSDL that is generated from the annotated web service in Example 19-3.

In this example:

- The `<message name="MissingName">` element defines the parts of the `MissingName` message, namely `fault`, and its associated data type, `tns:MissingName`.

  ```
  <message name="MissingName">
    <part name="fault" element="tns:MissingName" />
  </message>
  ```

- The `MissingName` SOAP fault is mapped to the `sayHelloWorld` operation.

  ```
  <operation name="sayHelloWorld">
    <input message="tns:sayHelloWorld" />
    <output message="tns:sayHelloWorldResponse" />
  ```

```
      <fault message="tns:MissingName" name="MissingName" />
    </operation>
```

This `<fault>` subelement in this example is derived from the `throws MissingName` clause of the `sayHelloWorld()` method declaration (see Example 19-3).

```
public String sayHelloWorld(String message) throws MissingName {
...
}
```

- The fault message is mapped to the `sayHelloWorld` operation in the `<binding>` element, as well.

```
<fault name="MissingName">
  <soap:fault name="MissingName" use="literal" />
</fault>
```

Example 19-5 Example of WSDL with Modeled Exceptions

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://examples/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   targetNamespace="http://examples/"
   name="HelloWorldService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://examples/"
     schemaLocation="http://localhost:7001/HelloWorld/HelloWorldService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="sayHelloWorld">
    <part name="parameters" element="tns:sayHelloWorld" />
  </message>
  <message name="sayHelloWorldResponse">
    <part name="parameters" element="tns:sayHelloWorldResponse" />
  </message>
  <message name="MissingName">
    <part name="fault" element="tns:MissingName" />
  </message>
  <portType name="HelloWorld">
    <operation name="sayHelloWorld">
      <input message="tns:sayHelloWorld" />
      <output message="tns:sayHelloWorldResponse" />
      <fault message="tns:MissingName" name="MissingName" />
    </operation>
  </portType>
  <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document" />
    <operation name="sayHelloWorld">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
       </output>
      <fault name="MissingName">
        <soap:fault name="MissingName" use="literal" />
```

```
      </fault>
    </operation>
  </binding>
  <service name="HelloWorldService">
    <port name="HelloWorldPort" binding="tns:HelloWorldPortBinding">
      <soap:address
        location="http://localhost:7001/HelloWorld/HelloWorldService" />
    </port>
  </service>
```

How the Fault is Communicated in the SOAP Message

Example 19-6 shows how the SOAP fault is communicated in the resulting SOAP message when the `MissingName` Java exception is thrown.

Example 19-6 Example SOAP Fault Message for MissingName Exception

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Your name is required.</faultstring>
         <detail>
            <ns2:MissingName xmlns:ns2="http://examples/">
               <message>Your name is required.</message>
            </ns2:MissingName>
            <ns2:exception xmlns:ns2="http://jax-ws.java.net/"
 class="examples.MissingName" note="To disable this feature, set
 com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system
 property to false">
               <message>Your name is required.</message>
               <ns2:stackTrace>
                  <ns2:frame class="examples.HelloWorld" file="HelloWorld.java"
  line="14" method="sayHelloWorld"/>
  ...
               </ns2:stackTrace>
            </ns2:exception>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

Creating the Web Service Client

When you generate a web service client from a WSDL file that contains mapped faults using `clientgen`, the required exception classes are generated automatically, including the mapped exception, fault bean, service implementation classes client implementation class, which you must modify, as described in the following sections.

- Reviewing the Generated Java Exception Class
- Reviewing the Generated Java Fault Bean Class
- Reviewing the Client-side Service Implementation
- Creating the Client Implementation Class

For more information about clientgen, see clientgen in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Reviewing the Generated Java Exception Class

An example of the generated Java exception class is shown in Example 19-7. The `@WebFault` annotation identifies the class as a mapped exception.

Example 19-7 Example of Generated Java Exception Class

```
package examples.client;

import javax.xml.ws.WebFault;

@WebFault(name = "MissingName", targetNamespace = "http://examples/")
public class MissingName_Exception extends Exception {
  private MissingName faultInfo;
  public MissingName_Exception(String message, MissingName faultInfo) { ... }
  public MissingName_Exception(String message, MissingName faultInfo,
     Throwable cause) { ... }
  public MissingName getFaultInfo() { ... }
}
```

Reviewing the Generated Java Fault Bean Class

An example of the generated Java fault bean class is shown in Example 19-8, defining the getters and setters for the fault message.

Example 19-8 Example of Generated Java Fault Bean Class

```
package examples.client;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "MissingName", propOrder = {
    "message"
})
public class MissingName {

    protected String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String value) {
        this.message = value;
    }
}
```

Reviewing the Client-side Service Implementation

An example of the generated client-side service implementation class is shown in Example 19-9.

Example 19-9 Client-side Service Implementation

```
package examples.client;
...
```

```
@WebService(name = "HelloWorld", targetNamespace = "http://examples/")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface HelloWorld {

    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "sayHelloWorld",
       targetNamespace = "http://examples/",
       className = "examples.client.SayHelloWorld")
    @ResponseWrapper(localName = "sayHelloWorldResponse",
       targetNamespace = "http://examples/",
       className = "examples.client.SayHelloWorldResponse")
    public String sayHelloWorld(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0)
        throws MissingName_Exception;
}
```

Creating the Client Implementation Class

Create the client implementation class to call the web service method and throw the custom exception. Then, compile and run the client. For more information about creating web service clients, see Invoking Web Services in *Developing JAX-WS Web Services for Oracle WebLogic Server*.

Example 19-10 shows an example client implementation class.

Example 19-10 Client Implementation Class

```
package examples.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
import examples.client.MissingName_Exception;

public class Main {

  public static void main(String[] args) throws MissingName_Exception {
    HelloWorldService service;

    try {
      service = new HelloWorldService(new URL(args[0] + "?WSDL"),
                    new QName("http://examples/", "HelloWorldService") );
    } catch (MalformedURLException murl) { throw new RuntimeException(murl); }
      HelloWorld port = service.getHelloWorldPort();

      String result = null;
    try {
      result = port.sayHelloWorld("");
    } catch (MissingName_Exception e) {
      System.err.println("Error: " + e);
    }
      System.out.println( "Got result: " + result );
  }
}
```

Using Unmodeled Faults

As noted previously, an unmodeled fault maps to an exception (for example, `java.lang.RuntimeException`) that is generated at run-time when no business logic fault is defined in the WSDL. In this case, Java exceptions are represented as generic SOAP fault exceptions, `javax.xml.ws.soap.SOAPFaultException`.

The following shows an example of an exception that maps to an unmodeled fault.

Example 19-11 Example of Web Service Using Unmodeled Fault

```
package examples;

import javax.jws.WebService;
@WebService(name="HelloWorld", serviceName="HelloWorldService")
public class HelloWorld {
   public String sayHelloWorld(String message) throws MissingName {
        System.out.println("Say Hello World: " + message);
        if (message == null || message.isEmpty()) {
           throw new MissingName(); // Modeled fault
        } else if (message.equalsIgnoreCase("abc")) {
         throw new RuntimeException("Please enter a name."); //Unmodeled fault
        }

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

In this example, if the string "abc" is passed to the method, the following `SOAPFaultException` and `RuntimeException` messages are returned in the log file:

Example 19-12 Example of Log File Message for Unmodeled Fault

```
...
run:
  [java] Exception in thread "main" javax.xml.ws.soap.SOAPFaultException: Please
enter a name.
...
Caused by: java.lang.RuntimeException: Please enter a name.\
...
```

Customizing the Exception Handling Process

You can customize the SOAP fault handling process using *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 SOAP message handlers to enable web services and clients to perform additional processing on the SOAP message. For more information, see Creating and Using SOAP Message Handlers.

Disabling the Stack Trace from the SOAP Fault

> **Note:**
>
> The `com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace` property is supported as an extension to the JDK 6.0. Because this API is not provided as part of the JDK 6.0 kit, it is subject to change.

By default, the entire stack trace, including nested exceptions, is included in the details of the SOAP fault message. For example, the following shows an example of a SOAP fault message that includes the stack trace:

You can disable the inclusion of the stack trace in the SOAP fault message by setting the `com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace` Java startup property to `false`.

To disable the stack trace:

1. Locate the following entry in the *`ORACLE_HOME`*`/user_projects/domains/`*`domainName`*`/startWebLogic.cmd` file, where *`ORACLE_HOME`* is the directory you specified as the Oracle Home when you installed Oracle WebLogic Server:

   ```
   set JAVA_OPTIONS=%SAVE_JAVA_OPTIONS%
   ```

2. Edit the entry as follows:

   ```
   set JAVA_OPTIONS=-
   Dcom.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace=false
   %SAVE_JAVA_OPTIONS%
   ```

3. Save the `startWebLogic.cmd` file.

Example 19-13 Example of Stack Trace in SOAP Fault Message

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope">
   <S:Body>
      <S:Fault xmlns:ns4="http://schemas.xmlsoap.org/soap/envelope/">
         <S:Code>
            <S:Value>S:Receiver</S:Value>
         </S:Code>
         <S:Reason>
            <S:Text xml:lang="en">String index out of range: 3</S:Text>
         </S:Reason>
         <S:Detail>
            <ns2:exception xmlns:ns2="http://jax-ws.java.net/"
             class="java.lang.StringIndexOutOfBoundsException" note="To disable this feature, set
             com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system property
             to false">
               <message>String index out of range: 3</message>
               <ns2:stackTrace>
                  <ns2:frame class="java.lang.String" file="String.java" line="1934"
                    method="substring"/>
                  <ns2:frame class="ratingservice.CreditRating" file="CreditRating.java"
                    line="21" method="processRating"/>
                  <ns2:frame class="sun.reflect.NativeMethodAccessorImpl"
```

```
                  file="NativeMethodAccessorImpl.java" line="native" method="invoke0"/>
                <ns2:frame class="sun.reflect.NativeMethodAccessorImpl"
                  file="NativeMethodAccessorImpl.java" line="39" method="invoke"/>
                <ns2:frame class="sun.reflect.DelegatingMethodAccessorImpl"
                  file="DelegatingMethodAccessorImpl.java" line="25" method="invoke"/>
                <ns2:frame class="java.lang.reflect.Method" file="Method.java" line="597"
                  method="invoke"/>
                ...
              </ns2:stackTrace>
            </ns2:exception>
          </S:Detail>
        </S:Fault>
    </S:Body>
</S:Envelope>
```

Other Exceptions

Note that in addition to the custom exceptions that are thrown explicitly in your web service and the `SOAPFaultExceptions` that are used to map exceptions that are not caught by your business logic, there are two other exceptions that might be communicated to the web service client, and that you should be aware of.

Table 19-3 Other Exceptions

Exception	Description
`javax.xml.ws.WebServiceException`	Base exception for all JAX-WS API runtime exceptions, used when calls to JAX-WS Java classes fail, such as `Service.BindingProvider` and `Dispatch`.
`java.util.concurrent.ExecutionException`	Used by JAX-WS asynchronous calls, when a client tries to get the response from an asynchronous call.

20

Optimizing Binary Data Transmission

This chapter describes how to send SOAP messages as attachments to optimize transmission for WebLogic web services using Java API for XML Web Services (JAX-WS). This chapter includes the following sections:

Optimizing Binary Data Transmission Optimization Using MTOM/XOP

SOAP Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM/XOP) defines a method for optimizing the transmission of XML data of type `xs:base64Binary` or `xs:hexBinary` in SOAP messages. When the transport protocol is HTTP, Multipurpose Internet Mail Extension (MIME) attachments are used to carry that data while at the same time allowing both the sender and the receiver direct access to the XML data in the SOAP message without having to be aware that any MIME artifacts were used to marshal the `base64Binary` or `hexBinary` data.

The binary data optimization process involves the following steps:

1. Encode the binary data.
2. Remove the binary data from the SOAP envelope.
3. Compress the binary data.
4. Attach the binary data to the MIME package.
5. Add references to the MIME package in the SOAP envelope.

MTOM/XOP support is standard in JAX-WS via the use of JWS annotations. The MTOM specification does not require that, when MTOM is enabled, the web service runtime use XOP binary optimization when transmitting `base64binary` or `hexBinary` data. Rather, the specification allows the runtime to choose to do so. This is because in certain cases the runtime may decide that it is more efficient to send the binary data directly in the SOAP Message; an example of such a case is when transporting small amounts of data in which the overhead of conversion and transport consumes more resources than just inlining the data as is.

The following Java types are mapped to the `base64Binary` XML data type, by default: `javax.activation.DataHandler`, `java.awt.Image`, and `javax.xml.transform.Source`. The elements of type `base64Binary` or `hexBinary` are mapped to `byte[]`, by default.

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

Table 20-1 Steps to Use MTOM/XOP to Send Binary Data

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

For more information see:

- MTOM specification at `http://www.w3.org/TR/soap12-mtom`
- XOP specification at `http://www.w3.org/TR/xop10`

Annotating the Data Types

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

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

Table 20-2 Mapping of MIME Content Types to Java Types

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

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

- Annotating the Data Types: Start From Java
- Annotating the Data Types: Start From WSDL

Annotating the Data Types: Start From Java

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

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

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

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

Annotating the Data Types: Start From WSDL

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

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

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

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

Enabling MTOM on the Web Service

You can enable MTOM on the web service using an annotation or WS-Policy file, as described in the following sections:

- Enabling MTOM on the Web Service Using Annotation
- Enabling MTOM on the Web Services by Attaching a WS-Policy File

Enabling MTOM on the Web Service Using Annotation

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

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

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

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

  }
}
```

Enabling MTOM on the Web Services by Attaching a WS-Policy File

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

```
<wsp:Policy wsu:Id="myService_policy">
     <wsp:ExactlyOne>
         <wsp:All>
             <wsoma:OptimizedMimeSerialization
                    xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/
optimizedmimeserialization" />
         </wsp:All>
     </wsp:ExactlyOne>
 </wsp:Policy>
```

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

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

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

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

In addition, you can attach the file at runtime using by the WebLogic Server Administration Console; for details, see Attach a WS-Policy file to a web service in the *Oracle WebLogic Server Administration Console Online Help*. This section describes how to use the JWS annotation.

The following simple JWS file example shows how to use the `@weblogic.jws.Policy` annotation in your JWS file to specify that the pre-packaged `Mtom.xml` file should be applied to your web service (relevant code shown in **bold**):

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

Enabling MTOM on the Client

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

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

import javax.xml.ws.soap.MTOMFeature;

public class Main {
  public static void main(String[] args) {
    String FOO = "FOO";
    MtomService service = new MtomService()
    MtomPortType port = service.getMtomPortTypePort(new MTOMFeature());
    String result = null;
    result = port.echoBinaryAsString(FOO.getBytes());
    System.out.println( "Got result: " + result );
  }
}
```

Setting the Attachment Threshold

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

To set the attachment threshold:

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

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

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

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

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

Enabling HTTP Chunking

You can minimize excessive buffering on the client side when processing MTOM attachments by enabling HTTP chunking on the transport layer using one of the following methods:

- Set the `jaxws.transport.streaming` system property to `true`. In this case, no code modifications are required.
- Set `com.sun.xml.ws.developer.JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE` property on the protocol binding request context. For more information, see the `JAXWSProperties` Javadoc at: https://www.javadoc.io/doc/com.sun.xml.ws/jaxws-rt/2.3.2/com.sun.xml.ws.jaxws/com/sun/xml/ws/developer/JAXWSProperties.html.

It is recommended that you enable HTTP chunking for CPU-intensive applications that are running on a WebLogic Server instance that is participating in web services interactions as a client and is sending out large messages.

The following excerpt from an Ant build script shows an example of setting the system property when invoking a client application called `clients.InvokeMTOMService`:

```
<target name="run-client">
   <java fork="true"
         classname="clients.InvokeMTOMService"
         failonerror="true">
     <classpath refid="client.class.path"/>
     <arg line="${http-endpoint}"/>
     <jvmarg line=
       "-Djaxws.transport.streaming=true"
     />
   </java>
 </target>
```

The following code excerpt shows how to set the `HTTP_CLIENT_STREAMING_CHUNK_SIZE` property.

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

import java.util.Map;
import javax.xml.ws.BindingProvider;
import com.sun.xml.ws.developer.JAXWSProperties;
...

public class Main {
   public static void main(String[] args) {
...
      Map<String, Object> ctxt=((BindingProvider)port).getRequestContext();
      ctxt.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192);
...
}
```

Streaming SOAP Attachments

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

Using MTOM and the `javax.activation.DataHandler` and `com.sun.xml.ws.developer.StreamingDataHandler` APIs you can specify that a web service use a streaming API when reading inbound SOAP messages that include attachments, rather than the default behavior in which the service reads the entire message into memory. This feature increases the performance of web services whose SOAP messages are particularly large.

> **Note:**
>
> Streaming MTOM cannot be used in conjunction with message encryption.

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

Client Side Example

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

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

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

public class Main {
   public static void main(String[] args) {
      MtomStreamingService service = new MtomStreamingService();
      MTOMFeature feature = new MTOMFeature();
      MtomStreamingPortType port = service.getMtomStreamingPortTypePort(
         feature);
      Map<String, Object> ctxt=((BindingProvider)port).getRequestContext();
      ctxt.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192);
      DataHandler dh = new DataHandler(new
                  FileDataSource("/tmp/example.jar"));
      port.fileUpload("/tmp/tmp.jar",dh);

      DataHandler dhn = port.fileDownload("/tmp/tmp.jar");
```

```
      StreamingDataHandler sdh = (StreamingDataHandler)dhn;
      try{
         File file = new File("/tmp/tmp.jar");
         sdh.moveTo(file);
         sdh.close();
       }
       catch(Exception e){
          e.printStackTrace();
       }
  }
}
```

The preceding example demonstrates the following:

- To enable MTOM on the client of the web service, pass an instance of the `javax.xml.ws.soap.MTOMFeature` as a parameter when creating the web service proxy or dispatch.
- Configure HTTP streaming support by enabling HTTP chunking on the MTOM streaming client. For more information, see Enabling HTTP Chunking.

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

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

Server Side Example

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

```
package examples.webservices.mtomstreaming;

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

@StreamingAttachment(parseEagerly=true, memoryThreshold=40000L)
@MTOM
@WebService(name="MtomStreaming",
            serviceName="MtomStreamingService",
            targetNamespace="http://example.org",
            wsdlLocation="StreamingImplService.wsdl")
@Oneway
@WebMethod
public class StreamingImpl {

   // Use @XmlMimeType to map to DataHandler on the client side
   public void fileUpload(String fileName,
```

```
                        @XmlMimeType("application/octet-stream")
                        DataHandler data) {
      try {
            StreamingDataHandler dh = (StreamingDataHandler) data;
            File file = new File(fileName);
            dh.moveTo(file);
            dh.close();
      } catch (Exception e) {
            throw new WebServiceException(e);
   }

  @XmlMimeType("application/octet-stream")
  @WebMethod
  public DataHandler fileDownload(String filename)
  {
    return  new DataHandler(new FileDataSource(filename));
  }
}
```

The preceding example demonstrates the following:

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

Configuring Streaming SOAP Attachments

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

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

Configuring Streaming SOAP Attachments on the Server

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

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

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

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

Configuring Streaming SOAP Attachments on the Client

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

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

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

Sending SOAP Messages With Attachments Using swaRef

Together, the specifications defined in Table 20-3 define a mechanism for sending SOAP messages with attachments using the `swaRef` XML attachment type.

Table 20-3 Specifications Supported for Sending SOAP Messages With Attachments

Specification	Description
SOAP With Attachments (SwA)	Defines a MIME `multipart/related` structure for packaging attachments with SOAP messages. For more information, see http://www.w3.org/TR/SOAP-attachments

Table 20-3 (Cont.) Specifications Supported for Sending SOAP Messages With Attachments

Specification	Description
WS-I Attachments Profile	Defines the `swaRef` schema type that can be used in the WSDL description to represent a reference to an attachment as a content-ID (CID) URL. WS-I publishes a public schema which defines the `swaRef` type, as defined by the following XSD: `http://ws-i.org/profiles/basic/1.1/xsd/swaref.xsd` JAXB maps the `swaRef` schema type to `javax.activation.DataHandler`. For more information, see: http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html

The following shows an example of how to use `swaRef` in a WSDL file to specify that the `claimForm` request and response messages be passed as an attachment.

Example 20-1 Example of WSDL File Using swaRef Data Type

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="SOAPBuilders-mime-cr-test"
    xmlns:types="http://example.org/mime/data"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://example.org/mime"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace="http://example.org/mime">

    <wsdl:types>
        <schema
            xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://example.org/mime/data"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
            elementFormDefault="qualified"
            xmlns:ref="http://ws-i.org/profiles/basic/1.1/xsd">
        <import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
            schemaLocation="WS-ISwA.xsd"/>
. . .
        <complexType name="claimFormTypeRequest">
            <sequence>
                <element name="request" type="ref:swaRef"/>
            </sequence>
        </complexType>
        <complexType name="claimFormTypeResponse">
            <sequence>
                <element name="response" type="ref:swaRef"/>
            </sequence>
        </complexType>

        <element name="claimFormRequest"  type="types:claimFormTypeRequest"/>
        <element name="claimFormResponse" type="types:claimFormTypeResponse"/>
        </schema>

    </wsdl:types>
. . .
     <wsdl:message name="claimFormIn">
        <wsdl:part name="data" element="types:claimFormRequest"/>
    </wsdl:message>
```

```
    <wsdl:message name="claimFormOut">
        <wsdl:part name="data" element="types:claimFormResponse"/>
    </wsdl:message>
. . .
    <wsdl:portType name="Hello">
. . .
        <wsdl:operation name="claimForm">
            <wsdl:input message="tns:claimFormIn"/>
            <wsdl:output message="tns:claimFormOut"/>
        </wsdl:operation>
    </wsdl:portType>
. . .
</wsdl:definitions>
```

As specified in the WSDL example in Example 20-1, the XML content that is tagged as type `swaRef` is sent as a MIME attachment and the element inside the SOAP body holds the reference to this attachment, as shown in Example 20-2.

Example 20-2 Example of SOAP Message with MIME Attachment

```
Content-Type: Multipart/Related; start-info="text/xml"; type="application/xop+xml";
    boundary="----=_Part_4_32542424.1118953563492"Content-Length: 1193SOAPAction: ""
    ------=_Part_5_32550604.1118953563502Content-Type: application/xop+xml; type="text/xml";
charset=utf-8
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
       <soapenv:Body>
       <request xmlns="http://example.org/mtom/data">
          cid:b0a597fd-5ef7-4f0c-9d85-6666239f1d25@example.jaxws.sun.com
       </request>
       </soapenv:Body>
    </soapenv:Envelope>
------=_Part_5_32550604.1118953563502
    Content-Type: application/xmlContent-ID:
   <b0a597fd-5ef7-4f0c-9d85-6666239f1d25@example.jaxws.sun.com>
<?xml
    version="1.0" encoding="UTF-8"?><application xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocaption="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd" version="1.4">
    <display-name>Simple example of application</display-name>
    <description>Simple example</description>
    <module>
        <ejb>ejb1.jar</ejb>
    </module>
    <module>
        <ejb>ejb2.jar</ejb>
    </module>
    <module>
         <web> <web-uri>web.war</web-uri> <context-root>web</context-root></web>
     </module></application>
```

Example 20-3 shows a sample web service that defines the `claimForm` operation. As defined in the WSDL, the request and response messages are sent as MIME attachments.

Example 20-3 Example Web Service

```
package mime.server;

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

```
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.activation.DataHandler;
import java.awt.*;
import java.io.ByteArrayInputStream;

@WebService (endpointInterface = "mime.server.Hello")
public class HelloImpl {
    ...
    public ClaimFormTypeResponse claimForm(ClaimFormTypeRequest data){
        ClaimFormTypeResponse resp = new ClaimFormTypeResponse();
        resp.setResponse(data.getRequest());
        return resp;
    }
    ...
}
```

Example 20-4 shows a sample web service client that calls the `claimForm` operation. Note that the client request data that will be transmitted as an attachment is mapped to the `DataHandler` data type.

Example 20-4 Example Web Service Client With MIME Attachments

```
package mime.client;

import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;
import javax.activation.DataHandler;
import java.io.ByteArrayInputStream;
import java.awt.*;

public class MimeApp {
    public static void main (String[] args){
        try {
            Object port = new HelloService().getHelloPort ();
            testSwaref ((Hello)port);
        } catch (Exception ex) {
            ex.printStackTrace ();
        }
    }

    private static void testSwaref (Hello port) throws Exception{
        DataHandler claimForm = new DataHandler (new StreamSource(
                    new ByteArrayInputStream(sampleXML.getBytes())), "text/xml");
        ClaimFormTypeRequest req = new ClaimFormTypeRequest();
        req.setRequest(claimForm);
        ClaimFormTypeResponse resp = port.claimForm (req);
        DataHandler out = resp.getResponse();
        ...
    }

    private static final String sampleXML = "<?xml version=\"1.0\" encoding=\"UTF-8\" ?> \n" +
            "<NMEAstd>\n" +
            "<DevIdSentenceId>$GPRMC</DevIdSentenceId>\n" +
            "<Time>212949</Time>\n" +
            "<Navigation>A</Navigation>\n" +
            "<NorthOrSouth>4915.61N</NorthOrSouth>\n" +
            "<WestOrEast>12310.55W</WestOrEast>\n" +
            "<SpeedOnGround>000.0</SpeedOnGround>\n" +
            "<Course>360.0</Course>\n" +
            "<Date>030904</Date>\n" +
```

```
            "<MagneticVariation>020.3</MagneticVariation>\n" +
            "<MagneticPoleEastOrWest>E</MagneticPoleEastOrWest>\n" +
            "<ChecksumInHex>*6B</ChecksumInHex>\n" +
            "</NMEAstd>";
}
```

21

Managing Web Service Persistence

This chapter describes how to manage persistence for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of Web Service Persistence

WebLogic Server provides a default web service persistence configuration that provides a built-in, high-performance storage solution for web services. Web service persistence is used by the following advanced features to support long running requests and to survive server restarts:

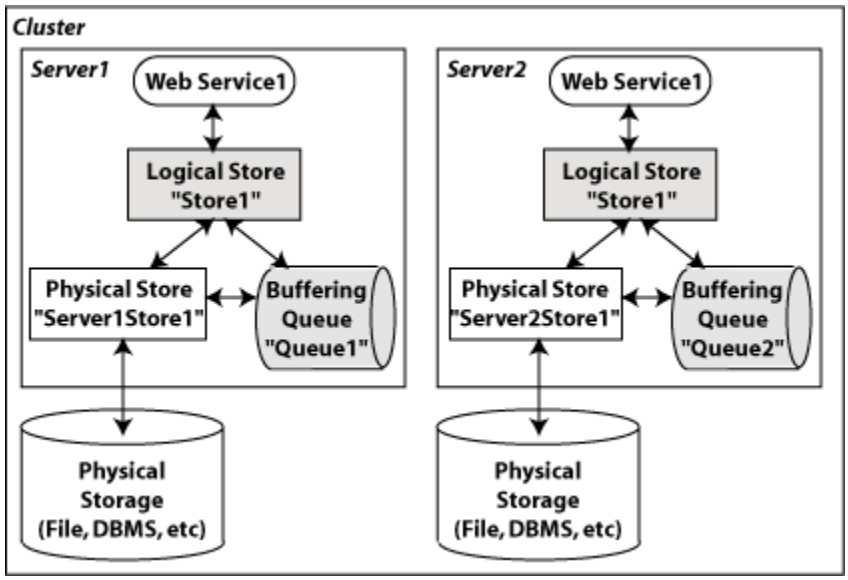
- Asynchronous web service invocation using asynchronous client transport or Make Connection
- Web services reliable messaging
- Message buffering
- Security using WS-SecureConversation

Specifically, web service persistence is used to save the following types of information:

- Client identity and properties
- SOAP message, including its headers and body
- Context properties required for processing the message at the web service or client (for both asynchronous and synchronous messages)

The following figure illustrates an example web service persistence configuration.

Figure 21-1 Example Web Service Persistence Configuration



The following table describes the components of web service persistence, shown in the previous figure.

Table 21-1 Components of the Web Service Persistence

Component	Description
Logical Store	Provides the configuration requirements and connects the web service to the physical store and buffering queue.
Physical store	Handles the I/O operations to save and retrieve data from the physical storage (such as file, DBMS, and so on). The physical store can be a WebLogic Server persistent store, as configured using the WebLogic Server Administration Console or WLST, or in-memory store. **Note**: When using a WebLogic Server persistent store as the physical store for a logical store, the names of the request and response buffering queues are taken from the logical store configuration and not the buffering configuration.
Buffering queue	Stores buffered requests and responses for the web service.

When configuring web service persistence, you associate:

- A logical store with a buffering queue.
- A buffering queue that is associated with a physical store via JMS configuration.

The association between the logical store and buffering queue is used to infer the association between the logical store and physical store. The default logical store is named `WseeStore` and is created automatically when a domain is created using the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`). By default, the physical store that is configured for the server is associated with the buffering queue. This strategy ensures that the same physical store is used for all web service persistence and buffering. Using a single physical store ensures a more efficient, single-phase XA transaction and facilitates migration.

You can configure one or more logical stores for use within your application environment. In Table 21-1, the servers `Server1` and `Server2` use the same logical store. This configuration allows applications that are running in a cluster to be configured globally to access a single store name. As described later in Configuring Web Service Persistence, you can configure web service persistence at various levels for fine-grained management. Best practices are provided in Roadmap for Configuring Web Service Persistence.

Roadmap for Configuring Web Service Persistence

Table 21-2 provides best practices for configuring web service persistence to support web service reliable messaging.

Table 21-2 Roadmap for Configuring Web Service Persistence

Best Practice	Description
Define a logical store for each administrative unit (for example, business unit, department, and so on).	By defining separate logical stores, you can better manage the service-level agreements for each administrative unit. For more information, see Configuring the Logical Store.
Use the correct logical store for each client or service related to the administrative unit.	You can configure the logical store at the WebLogic Server, web service, or web service client level. For more information, see Configuring Web Service Persistence.
Define separate physical stores and buffering queues for each logical store.	For more information, see Figure 21-1.

The best practices defined in Table 21-2 facilitates maintenance, and failure recovery and resource migration.

For example, assume Company X is developing web services for several departments, including manufacturing, accounts payable, accounts receivable. Following best practices, Company X defines a minimum of three logical stores, one for each department.

Furthermore, assume that the manufacturing department has a service-level agreement with the IT department that specifies that it can tolerate system outages that are no longer than a few minutes in duration. The accounts payable and receivable departments, on the other hand, have a more relaxed service-level agreement, tolerating system outages up to one hour in duration. If the systems that host web services and clients for the manufacturing department become unavailable, the IT department is responsible for ensuring that any resources required by those web services and clients are migrated to new active servers within minutes of the failure. Because separate logical stores were defined, the IT department can migrate the file store, JMS servers, and so on, associated with the manufacturing department logical store independently of the resources required for accounts payables and receivables.

Configuring Web Service Persistence

The following table summarizes the information that you can configure for each of the web service persistence components.

Table 21-3 Summary of the Web Service Persistence Component Configuration

Component	Summary of Configuration Requirements
Logical Store	You configure the following information for each logical store: • Name of the logical store. • Maximum lifetime of an object in the store. • The cleaner thread that removes stale objects from the store. For more information, see Cleaning Up Web Service Persistence. • Accessibility from other servers in a network. • Request and response buffering queues. The request buffering queue is used to infer the physical store by association.
Physical store	You configure the following information for the physical store: • Name of the physical store. • Type and performance parameters. • Location of the store. **Note**: You configure the physical store or buffering queue, but not both. If the buffering queue is configured, then the physical store information is inferred.
Buffering queue	You configure the following information for the buffering queue: • Request and response queue details • Retry counts and delays

You can configure web service persistence at the levels defined in the following table.

Table 21-4 Configuring Web Service Persistence

Level	Description
WebLogic Server	The web service persistence configured at the server level defines the default configuration for all web services and clients running on that server. To configure web service persistence for WebLogic Server, use one of the following methods: • When creating or extending a domain using Configuration Wizard, you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to configure automatically the resources required to support web services persistence. Although use of this extension template is not required, it makes the configuration of the required resources much easier. • Configure the resources required for web service persistence using the Oracle WebLogic Server Administration Console or WLST. For more information, see: - WebLogic Server Administration Console: Configure web service persistence in *Oracle WebLogic Server Administration Console Online Help* - WLST: Configuring Existing Domains in *Understanding the WebLogic Scripting Tool* For more information, see Configuring Your Domain For Advanced Web Services Features.
Web service endpoint	Configure the default logical store used by the web service endpoint, as described in Configuring Web Service Persistence for a Web Service Endpoint.

Table 21-4 (Cont.) Configuring Web Service Persistence

Level	Description
Web service client	Configure the default logical store used by the web service client, as described in Configuring Web Service Persistence for Web Service Clients.

The following sections provide more information about configuring web service persistence:

- Configuring the Logical Store
- Configuring Web Service Persistence for a Web Service Endpoint
- Configuring Web Service Persistence for Web Service Clients

Configuring the Logical Store

You can configure one or more logical stores for use within your application environment, and identify the logical store that is used as the default.

The default logical store, `WseeStore`, is generated automatically when you create or extend a domain using the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`), as described in Configuring Your Domain For Advanced Web Services Features.

You can configure the logical store using the WebLogic Server Administration Console, see Configure web service persistence in *Oracle WebLogic Server Administration Console Online Help*. Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

The following table summarizes the properties that you define for the logical store.

Table 21-5 Configuration Properties for the Logical Store

Property	Description
Default Logical Store Name	Name of the logical store. The name must begin with an alphabetical character and can contain alphabetical characters, spaces, dashes, underscores, and numbers only. This field defaults to `LogicalStore_n`. This field is required. If you create or extend a single server domain using the web service extension template, a logical store named `WseeStore` is created by default.
Default Logical Store	Flag that specifies whether the logical store is used, by default, to persist state of all web services on the server. Only one logical store can be set as the default. If you enable this flag on the current logical store, the flag is disabled on the current default store.
Persistence strategy	Persistence strategy. Select one of the following values from the drop-down menu. • `Local Access Only`—Accessible to the local server only. • `In Memory`—Accessible by the local VM only. In this case, the buffering queue and physical store configuration information is ignored.

Table 21-5 (Cont.) Configuration Properties for the Logical Store

Property	Description
Request Buffering Queue JNDI Name	JNDI name for the request buffering queue. The request buffering queue is used to infer the physical store by association. If this property is not set, then the default physical store that is configured for the server is used. **Note**: You configure the physical store or buffering queue, but not both. If the buffering queue is configured, then the physical store information is inferred. It is recommended that the same physical storage resource be used for both persistent state and message buffering to allow for a more efficient, single-phase XA transaction and facilitate service migration. By setting this value, you ensure that the buffering queue and physical store reference the same physical storage resource. If you create or extend a domain using the web service extension template, a buffering queue named `weblogic.wsee.BufferedRequestQueue` is created by default. **Note**: This property is ignored if **Persistence strategy** is set to `In Memory`.
Response Buffering Queue JNDI Name	JNDI name for the response buffering queue. If this property is not set, then the request queue is used, as defined by the **Request Buffering Queue JNDI Name** property. If you create or extend a domain using the web service extension template, a buffering queue named `weblogic.wsee.BufferedRequestErrorQueue` is created by default. **Note**: This property is ignored if **Persistence strategy** is set to `In Memory`.
Cleaner Interval	Interval at which the logical store will be cleaned. For more information, see Cleaning Up Web Service Persistence. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `PT10M` (10 minutes). **Note**: This field is available when editing the logical store only. When creating the logical store, the field is set to the default, `PT10M` (10 minutes).
Default Maximum Object Lifetime	Default value used as the maximum lifetime of an object. This value can be overridden by the individual objects saved to the logical store. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `P1D` (one day). **Note**: This field is available when editing the logical store only. When creating the logical store, the field is set to the default, `P1D` (one day).

Configuring Web Service Persistence for a Web Service Endpoint

By default, web service endpoints use the web service persistent store defined for the server. You can override the logical store used by the web service endpoint using the WebLogic Server Administration Console. For more information, see Configure web service persistence in *Oracle WebLogic Server Administration Console Online Help*.

Configuring Web Service Persistence for Web Service Clients

For information about configuring persistence for web service clients, see Configuring Web Service Clients.

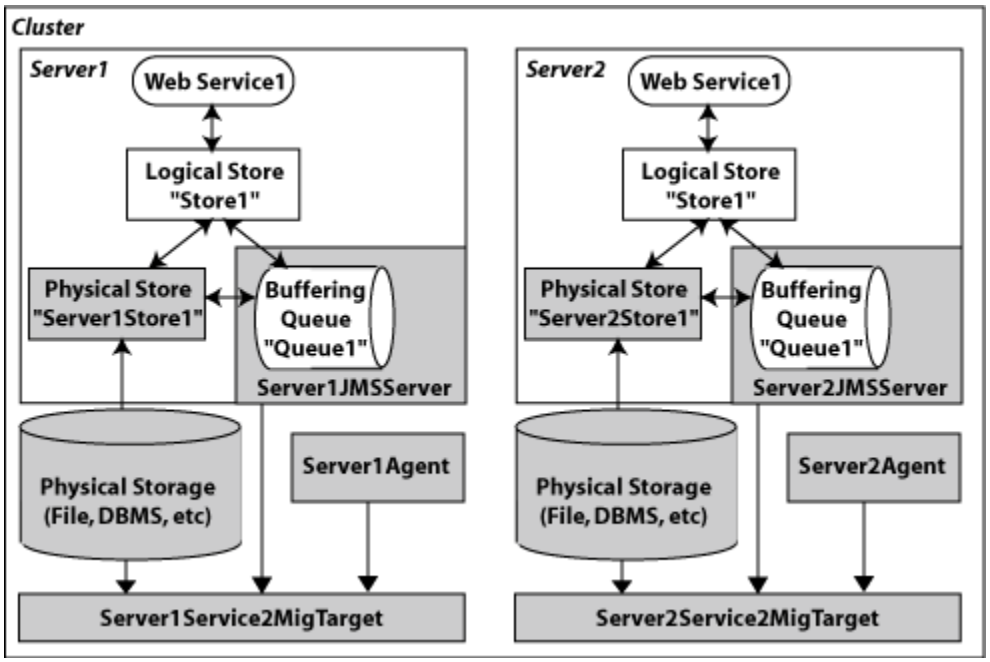
Using Web Service Persistence in a Cluster

The following provides some considerations for using web services persistence in a cluster:

- If you create or extend a clustered domain using the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`), the resources required to support web services persistence in a cluster are automatically created. For more information, see Configuring Your Domain For Advanced Web Services Features.
- To facilitate service migration, it is recommended that the same physical storage resource be used for both persistent state and message buffering. To ensure that the buffering queue and physical store reference the same physical storage resource, you configure the **Request Buffering Queue JNDI Name** property of the logical store, as described in Configuring the Logical Store.
- It is recommended that the buffering queues be defined as JMS uniform distributed destinations (UDDs). JMS defines a member queue for the UDD on each JMS Server that you identify. Because a logical store is associated with a physical store through the defined buffering queue, during service migration, this allows a logical store to use the new physical stores seamlessly for the member queues that migrate onto the new server.
- It is recommended that you target the JMS Server, store-and-forward (SAF) service agent, and physical store (file store) resources to migrateable targets. For more information, see Resources Required by Advanced Web Service Features.

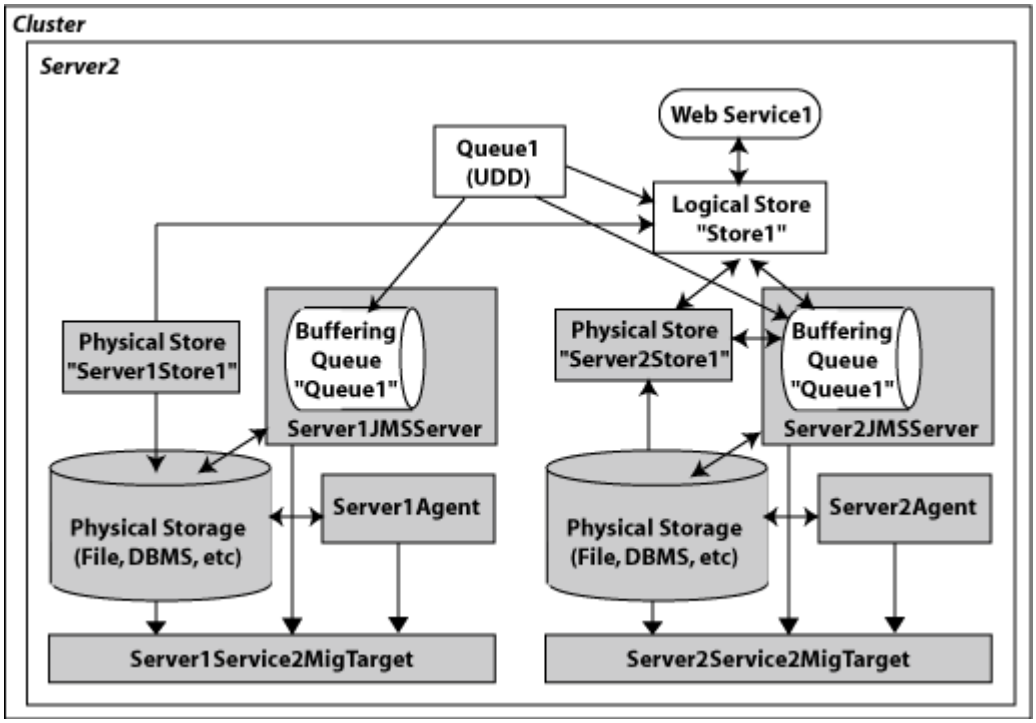
For example, consider the two-node cluster configuration shown in Figure 21-2. The domain resources are configured and targeted using the guidelines provided above.

Figure 21-2 Example of a Two-Node Cluster Configuration (Before Migration)



The following figure shows how the resources on Server1 can be easily migrated to Server2 in the event Server1 fails.

Figure 21-3 Example of a Two-Node Cluster Configuration (After Migration)



Cleaning Up Web Service Persistence

The persisted information is cleaned up periodically to remove expired or stale objects. Typically, an object is associated with a specific expiration time or a maximum lifetime. In addition, a *stale* object may represent a request for which no response was received or a reliable messaging sequence that was not explicitly terminated.

You configure the interval of time at which web service persistence will be cleaned by setting the **Cleaner Interval** configuration property on the logical store. For more information about setting this property, see Configuring the Logical Store.

22

Configuring Message Buffering for Web Services

This chapter describes how to configure message buffering for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

- Overview of Message Buffering
- Configuring Messaging Buffering

Overview of Message Buffering

When an operation on a buffered web service is invoked, the message representing that invocation is stored in a JMS queue. WebLogic Server processes this buffered message asynchronously. If WebLogic Server goes down while the message is still in the queue, it will be processed as soon as WebLogic Server is restarted.

WebLogic Server then processes the request message on a separate thread obtained from a pre-configured and managed pool of threads. This allows WebLogic Server to absorb spikes in client load, and continue to process the requests in an orderly fashion over a period of time. Message buffering is a powerful tool to avoid denial of service attacks and general overload conditions on the server.

To assist you in determining whether to configure message buffering on the web service, it is recommended that you review Failure Scenarios with Non-buffered Reliable Web Services.

Configuring Messaging Buffering

You can configure message buffering for web services at the WebLogic Server or web service endpoint levels. The message buffering configured at the server level defines the default message buffering defined for all web services and clients running on that server, unless explicitly overridden at the web service endpoint level.

For detailed steps to configure message buffering for web services at the WebLogic Server or web service endpoint level using the WebLogic Server Administration Console, see Configure message buffering for web services in *Oracle WebLogic Server Administration Console Online Help*.

When you configure message buffering at the web service endpoint level, select **Customize Buffering Configuration** to indicate that you want to customize the buffering configuration defined in the web service descriptor or deployment plan at the web service endpoint level. If not checked, the buffering configuration specified at the WebLogic Server level is used.

Alternatively, you can use WLST to configure message buffering. For information about using WLST to extend the domain, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

The following sections describe message buffering configuration properties:

Configuring the Request Queue

The following table summarizes the properties used to configure the request queue.

Table 22-1 Configuring the Request Queue

Property	Description
Request Queue Enabled	Flag that specifies whether the request queue is enabled. By default, the request queue is disabled. The request queue name is defined by the logical store enabled at this level. When using a WebLogic Server persistent store as the physical store for a logical store, the names of the request and response buffering queues are taken from the logical store configuration and not the buffering configuration.
Request Queue Connection Factory JNDI Name	JNDI name of the connection factory to use for request message buffering. This value defaults to the default JMS connection factory defined by the server.
Request Queue Transaction Enabled	Flag that specifies whether transactions should be used when storing and retrieving messages from the request buffering queue. This flag defaults to false.

Configuring the Response Queue

The following table summarizes the properties used to configure the response queue.

Table 22-2 Configuring the Response Queue

Property	Description
Response Queue Enabled	Flag that specifies whether the response queue is enabled. By default, the response queue is disabled. The response queue name is defined by the logical store enabled at this level. When using a WebLogic Server persistent store as the physical store for a logical store, the names of the request and response buffering queues are taken from the logical store configuration and not the buffering configuration.
Response Queue Connection Factory JNDI Name	JNDI name of the connection factory to use for response message buffering. This value defaults to the default JMS connection factory defined by the server.
Response Queue Transaction Enabled	Flag that specifies whether transactions should be used when storing and retrieving messages from the response buffering queue. This flag defaults to false.

Configuring Message Retry Count and Delay

The following table summarizes the properties used to configure the message retry count and delay.

Table 22-3 Configuring Message Retry Count and Delay

Property	Description
Retry Count	Number of times that the JMS queue on the invoked WebLogic Server instance attempts to deliver the message to the web service implementation until the operation is successfully invoked. This value defaults to 3.
Retry Delay	Amount of time between retries of a buffered request and response. Note, this value is only applicable when **RetryCount** is greater than 0. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P0DT30S` (30 seconds).

23

Managing Web Services in a Cluster

This chapter describes how to manage WebLogic web services in a cluster. This chapter includes the following sections:

> **Note:**
>
> For considerations specific to using web service persistence in a cluster, see Using Web Service Persistence in a Cluster.

Overview of Web Services Cluster Routing

Clustering of *stateless* web services—services that do not require knowledge of state information from prior invocations—is straightforward and works with existing WebLogic HTTP routing features on a third-party HTTP load balancer.

Clustering of web services that require state information be maintained provides more challenges. Each instance of such a web service is associated with state information that must be managed and persisted. The cluster routing decision is based on whether the message is bound to a specific server in the cluster. For example, if a particular server stores state information that is needed to process the message, and that state information is available only locally on that server.

> **Note:**
>
> Services that use session state replication to maintain their state are a separate class of problem from those that make use of advanced web service features, such as Reliable Secure Profile. The latter require a more robust approach to persistence that may include storing state that may be available only from the local server. For more information, see A Note About Persistence.

In addition to ensuring that the web service requests are routed to the appropriate server, the following general clustering requirements must be satisfied:

- The internal topology of a cluster must be transparent to clients. Clients interact with the cluster only through the front-end host, and do not need to be aware of any particular

server in the cluster. This enables the cluster to scale over time to meet the demands placed upon it.

- Cluster migration must be transparent to clients. Resources within the cluster (including persistent stores and other resources required by a web service or web service client) can be migrated from one server to another as the cluster evolves, responds to failures, and so on.

To meet the above requirements, the following methods are available for routing web services in a cluster:

- **In-place SOAP router**—Assumes request messages arrive on the correct server and, if not, forwards the messages to the correct server ("at most one additional hop"). The routing decision is made by the web service that receives the message. This routing strategy is the simplest to implement and requires no additional configuration. Though, it is not as robust as the next option.
- **Front-end SOAP router** (HTTP cluster servlet only)—Message routing is managed by the front-end host that accepts messages on behalf of the cluster and forwards them onto a selected member server of the cluster. For web services, the front-end SOAP router inspects information in the SOAP message to determine the correct server to which it should route messages.

  This routing strategy is more complicated to configure, but is the most efficient since messages are routed directly to the appropriate server (avoiding any "additional hops").

> **Note:**
>
> When using Make Connection, as described in Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection), only front-end SOAP routing can guarantee proper routing of all messages related to a given Make Connection anonymous URI.

This chapter describes how to configure your environment to optimize the routing of web services within a cluster. Use of the HTTP cluster servlet for the front-end SOAP router is described. The in-place SOAP router is also enabled and is used in the event the HTTP cluster servlet is not available or has not yet been initialized.

A Note About Persistence

While it is possible to maintain state for a web service using the HttpSession as described in Programming Stateful JAX-WS Web Services Using HTTP Session, in some cases this simple persistence may not be robust enough. Advanced web services features like reliable messaging, Make Connection, secure conversation, and so on, have robust persistence requirements that cannot be met by using the HttpSession alone. Advanced web service features use a dedicated persistence implementation based on the concept of a *logical store*. For more information, see Managing Web Service Persistence.

At this time, these two approaches to persistence of web service state are not compatible with each other. If you choose to write a clustered stateful web service using HttpSession persistence and then use the advanced web service features from that service (either as a client or service), Oracle cannot guarantee correct operation of your service in a cluster. This is because HttpSession replication may make the

HttpSession available on a different set of servers than are hosting the persistence for advanced web service features.

Cluster Routing Scenarios

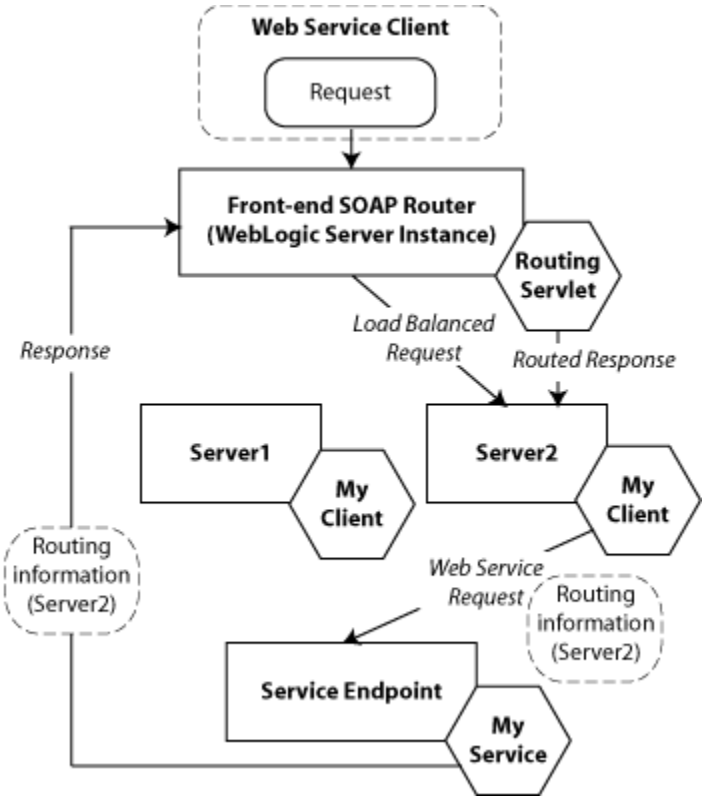
The following sections illustrate several scenarios for routing web service request and response messages within a clustered environment:

- Scenario 1: Routing a Web Service Response to a Single Server
- Scenario 2: Routing Web Service Requests to a Single Server Using Routing Information
- Scenario 3: Routing Web Service Requests to a Single Server Using an ID

Scenario 1: Routing a Web Service Response to a Single Server

In this scenario, an incoming request is load balanced to a server. Any responses to that request must be routed to that same server, which maintains state information on behalf of the original request.

Figure 23-1 Routing a Web Service Response to a Single Server



As shown in the previous figure:

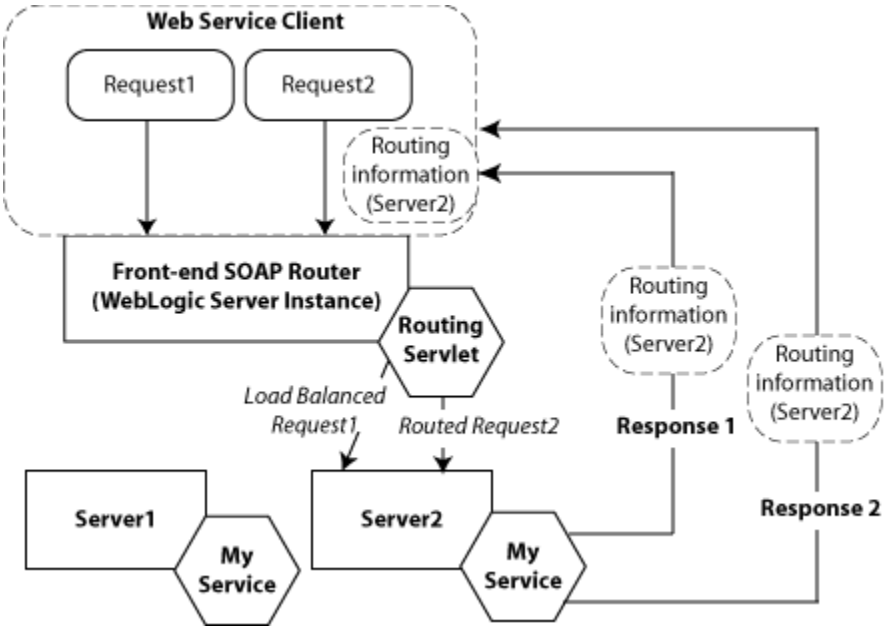
1. The front-end SOAP router routes an incoming HTTP request and sends it to Server2 using standard load balancing techniques.
2. Server2 calls Myservice at the web service endpoint address. The ReplyTo header in the SOAP message contains a pointer back to the front-end SOAP router.

3. MyService returns the response to the front-end SOAP router.
4. The front-end SOAP router must determine where to route the response. Because Server2 maintains state information that is relevant to the response, the front-end SOAP router routes the response to Server2.

Scenario 2: Routing Web Service Requests to a Single Server Using Routing Information

In this scenario, an incoming request is load balanced to a server. The response contains routing information that targets the original server for any subsequent requests

Figure 23-2 Routing Web Service Requests to a Single Server



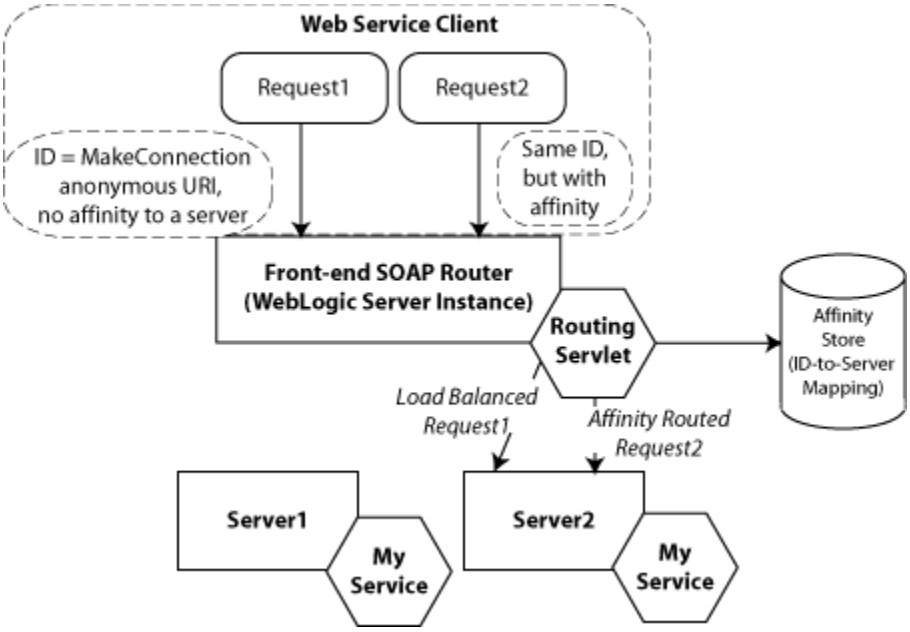
As shown in the previous figure:

1. The front-end SOAP router routes an incoming HTTP request (Request1) and sends it to Server 2 using standard load balancing techniques. The request has no routing information.
2. Server2 calls the Myservice at the web service endpoint address. The ReplyTo header in the SOAP message contains a pointer back to the front-end SOAP router.
3. MyService returns the response to the caller. The response contains routing information that targets Server2 for any subsequent requests. The caller is responsible for passing the routing information contained in the response in any subsequent requests (for example, Request2).
4. The front-end SOAP router uses the routing information passed with Request2 to route the request to Server2.

Scenario 3: Routing Web Service Requests to a Single Server Using an ID

In this scenario, an incoming SOAP request contains an identifier, but no routing information. All subsequent requests with the same identifier must go to the same server.

Figure 23-3 Routing Web Service Requests to a Single Server Using an ID



As shown in the previous figure:

1. A request comes from a web service client that includes an ID (Make Connection anonymous URI) that will be shared by future requests that are relevant to Request1. The form of this ID is protocol-specific.
2. The front-end SOAP router detects an ID in Request1 and checks the affinity store to determine if the ID is associated with a particular server in the cluster. In this case, there is no association defined.
3. The front-end SOAP router load balances the request and sends it to Server 2 for handling.
4. The MyService web service instance on Server2 handles the request (generating a response, if required). Unlike in Scenario 2: Routing Web Service Requests to a Single Server Using Routing Information, routing information cannot be propagated in this case.
5. Request2 arrives at the front-end SOAP router using the same ID as that used in Request1.
6. The front-end SOAP router detects the ID and checks the affinity store to determine if the ID is associated with a particular server. This time, it determines that the ID is mapped to Server2.
7. Based on the affinity information, the front-end SOAP router routes Request2 to Server2.

How Web Service Cluster Routing Works

The following sections describe how web service cluster routing works:

Adding Routing Information to Outgoing Requests

The web services runtime adds routing information to the SOAP header of any outgoing message to ensure proper routing of messages in the following situations:

- The request is sent from a web service client that uses a store that is not accessible from every member server in the cluster.
- The request requires in-memory state information used to process the response.

When processing an outgoing message, the web services runtime:

- Creates a message ID for the outgoing request, if one has not already been assigned, and stores it in the `RelatesTo`/`MessageID` SOAP header using the following format:

  `uuid:WLS`*`format_version`*`:`*`store_name`*`:`*`uniqueID`*

  Where:

  - *`format_version`* specifies the WebLogic Server format version, for example `WLS1`.
  - *`store_name`* specifies the name of the persistent store, which specifies the store in use by the current web service or web service client sending the message. For example, `Server1Store`. This value may be a system-generated name, if the default persistent store is used, or an empty string if no persistent store is configured.
  - *`unique_ID`* specifies the unique message ID. For example: `68d6fc6f85a3c1cb:-2d3b89ab8:12068ad2e60:-7feb`

- Allows other web service components to inject routing information into the message before it is sent.

Detecting Routing Information in Incoming Requests

The SOAP router (in-place or front-end) inspects incoming requests for routing information. In particular, the SOAP router looks for a `RelatesTo`/`MessageID` SOAP header to find the name of the persistent store and routes the message back to the server that hosts that persistent store.

In the event that there is an error in determining the correct server using front-end SOAP routing, then the message is sent to any server within the cluster and the in-place SOAP router is used. If in-place SOAP routing fails, then the sender of the message receives a fault on the protocol-specific back channel.

> **Note:**
>
> SOAP message headers that contain routing information must be presented in clear text; they cannot be encrypted.

Routing Requests Within the Cluster

To assist in making a routing determination, the SOAP router (in-place or front-end) uses a dynamic map of store-to-server name associations. This dynamic map originates on the Managed Servers within a cluster and is accessed in memory by the in-place SOAP router and via HTTP response headers by the front-end SOAP router. The HTTP response headers are included automatically by WebLogic Server in every HTTP response sent by a web service in the cluster.

> **Note:**
>
> For more information about the HTTP response headers, see Maintaining the Routing Map on the Front-end SOAP Router.

Initially, the dynamic map is empty. It is only initialized after receiving its first response back from a Managed Server in the cluster. Until it receives back its first response with the HTTP response headers, the front-end SOAP router simply load balances the requests, and the in-place SOAP router routes the request to the appropriate server.

In the absence of SOAP-based routing information, it defers to the base routing that includes HTTP-session based routing backed by simple load balancing (for example, round-robin).

Maintaining the Routing Map on the Front-end SOAP Router

As noted in Routing Requests Within the Cluster, to assist in making a routing determination, the SOAP router (in-place or front-end) uses a dynamic map of store-to-server name associations.

To generate this dynamic map, two new HTTP response headers are provided, as described in the following sections. These headers are included automatically by WebLogic Server in every HTTP response sent by a web service in the cluster.

> **Note:**
>
> When implementing a third-party front-end to include the HTTP response headers described below, clients should send an HTTP request header with the following variable set to any value: `X-weblogic-wsee-request-storetoserver-list`

X-weblogic-wsee-storetoserver-list HTTP Response Header

A complete list of store-to-server mappings is maintained in the `X-weblogic-wsee-storetoserver-list` HTTP response header. The front-end SOAP router uses this header to populate a mapping that can be referenced at runtime to route messages.

The `X-weblogic-wsee-storetoserver-list` HTTP response header has the following format:

storename1:*host_server_spec* | *storename2*:*host_server_spec* | *storename3*:*host_server_spec*

In the above:

- *storename* specifies the name of the persistent store.
- *host_server_spec* is specifies using the following format: *servername:host:port:sslport*. If not known, the *sslport* is set to -1.

X-weblogic-wsee-storetoserver-hash HTTP Response Header

A hash mapping of the store-to-server list is provided in `X-weblogic-wsee-storetoserver-hash` HTTP response header. This header enables you to determine whether the new mapping list needs to be refreshed.

The `X-weblogic-wsee-storetoserver-hash` HTTP response header contains a String value representing the hash value of the list contained in the `X-weblogic-wsee-storetoserver-list` HTTP response header. By keeping track of the last entry in the list, it can be determined whether the list needs to be refreshed.

Configuring Web Services in a Cluster

The following table summarizes the steps to configure web services in a cluster.

Table 23-1 Steps to Manage Web Services in a Cluster

#	Step	Description
1	Set up the WebLogic cluster.	See Setting Up the WebLogic Cluster.
2	Configure the clustered domain resources required for advanced web service features.	You can configure automatically the clustered domain resources required using the cluster extension template script. Alternatively, you can configure the resources using the Oracle WebLogic Server Administration Console or WLST. See Configuring the Domain Resources Required for Web Service Advanced Features in a Clustered Environment.
3	Extend the front-end SOAP router to support web services.	**Note:** This step is required only if you are using the front-end SOAP router. The web services routing servlet extends the functionality of the WebLogic HTTP cluster servlet to support routing of web services in a cluster. See Extending the Front-end SOAP Router to Support Web Services.

Table 23-1 (Cont.) Steps to Manage Web Services in a Cluster

#	Step	Description
4	Enable routing of web services atomic transaction messages.	See Enabling Routing of Web Services Atomic Transaction Messages.
5	Enable routing of web services Make Connection messages.	See Enabling Routing of Web Services Make Connection Messages.
6	Configure the identity of the front-end SOAP router.	Each WebLogic Server instance in the cluster must be configured with the address and port of the front-end SOAP router. See Configuring the Identity of the Front-end SOAP Router.

Setting Up the WebLogic Cluster

Set up the WebLogic cluster, as described in Setting up WebLogic Clusters in *Administering Clusters for Oracle WebLogic Server*. Please note:

- To configure the clustered domain, see Configuring the Domain Resources Required for Web Service Advanced Features in a Clustered Environment.
- To enable SOAP-based front-end SOAP routing, configure an HTTP cluster servlet, as described in Set Up the HttpClusterServlet in *Administering Clusters for Oracle WebLogic Server*.

Configuring the Domain Resources Required for Web Service Advanced Features in a Clustered Environment

When creating or extending a domain using Configuration Wizard, you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to configure automatically the resources required to support the advanced web service features in a clustered environment. Although use of this extension template is not required, it makes the configuration of the required resources much easier. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Server Administration Console or WLST.

In addition, the template installs scripts into the domain directory that can be used to manage the resource required for advanced web services in-sync as the domain evolves (for example, servers are added or removed, and so on).

For more information about how to configure the domain and run the scripts to manage resources, see Configuring Your Domain For Advanced Web Services Features.

Extending the Front-end SOAP Router to Support Web Services

> **Note:**
>
> If you are not using the front-end SOAP router, then this step is not required.

You extend the front-end SOAP router to support web services by specifying the `RoutingHandlerClassName` parameter shown in the following example (in **bold**), as part of the WebLogic HTTP cluster servlet definition.

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>HttpClusterServlet</servlet-name>
    <servlet-class>weblogic.servlet.proxy.HttpClusterServlet</servlet-class>
    <init-param>
       <param-name>WebLogicCluster</param-name>
       <param-value>Server1:7001|Server2:7001</param-value>
    </init-param>
   <init-param>
      <param-name>RoutingHandlerClassName</param-name>
      <param-value>
         weblogic.wsee.jaxws.cluster.proxy.SOAPRoutingHandler
      </param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HttpClusterServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
. . .
</web-app>
```

Enabling Routing of Web Services Atomic Transaction Messages

High availability and routing of web services atomic transaction messages is automatically enabled in web service clustered environments. However, if the WebLogic HTTP cluster servlet is being used as the front-end server, you need to set the following system property to `false` on the server hosting the WebLogic HTTP cluster servlet:

```
weblogic.wsee.wstx.wsat.deployed=false
```

In addition, when using a WebLogic Server plugin, you should configure the `WLIOTimeoutSecs` parameter value appropriately. This parameter defines the amount of time the plug-in waits for a response to a request from WebLogic Server. If the value is less than the time the servlets take to process, then you may see unexpected results. For more information about the `WLIOTimeoutSecs` parameter, see General Parameters for Web Server Plug-ins in *Using Oracle WebLogic Server Proxy Plug-Ins*.

Enabling Routing of Web Services Make Connection Messages

To support Web Service Make Connection, as described in Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection), you must configure a default logical store on the WebLogic Server that is hosting the WebLogic HTTP cluster servlet. For information about configuring the default logical store, see Configuring the Logical Store.

Configuring the Identity of the Front-end SOAP Router

Each WebLogic Server instance in the cluster must be configured with the address and port of the front-end SOAP router.

You can configure the identity of the front-end SOAP router using one of the following methods, listed in order of precedence:

- Create a network channel, as describe in Configuring the Identity of the Front-end SOAP Router Using Network Channels. *This is the recommended method.*
- Configure the front-end host and port for the cluster, as described in Configure HTTP Settings for a Cluster in *Oracle WebLogic Server Administration Console Online Help*.
- Configure the front-end host and port for the local server, as described in Configure HTTP Protocol in *Oracle WebLogic Server Administration Console Online Help*.
- Define the `Cluster Address` for the cluster, as described in Configure Clusters in *Oracle WebLogic Server Administration Console Online Help*. The `Cluster Address` is required if no other values are set.

Configuring the Identity of the Front-end SOAP Router Using Network Channels

Network channels enable you to provide a consistent way to access the front-end address of a cluster. For more information about network channels, see Understanding Network Channels in *Administering Server Environments for Oracle WebLogic Server*.

To configure the identity of the front-end SOAP router using network channels, 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 in *Oracle WebLogic Server Administration Console Online Help* 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.

Monitoring Cluster Routing Performance

You can monitor the following cluster routing statistics to evaluate the application performance:

- Total number of requests and responses.
- Total number of requests and responses that were routed specifically to the server.
- Routing failure information, including totals and last occurrence.

You can use the WebLogic Server Administration Console or WLST to monitor cluster routing performance. For information about using WebLogic Server Administration Console to monitor cluster routing performance, see Monitor SOAP web services and Monitor SOAP web service clients in *Oracle WebLogic Server Administration Console Online Help*.For

information about using WLST to monitor cluster routing performance, see Configuring Existing Domains in *Understanding the WebLogic Scripting Tool*.

24

Using Provider-based Endpoints and Dispatch Clients to Operate on SOAP Messages

This chapter describes how to develop web service provider-based endpoints and dispatch clients to operate on SOAP messages at the XML message level for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of Web Service Provider-based Endpoints and Dispatch Clients

Although the use of JAXB-generated classes is simpler, faster, and likely to be less error prone, there are times when you may want to generate your own business logic to manipulate the XML message content directly. Message-level access can be accomplished on the server side using web service Provider-based endpoints, and on the client side using Dispatch clients.

A **web service Provider-based endpoint** offers a dynamic alternative to the Java service endpoint interface (SEI)-based endpoint. Unlike the SEI-based endpoint that abstracts the details of converting between Java objects and their XML representation, the Provider interface enables you to access the content directly at the XML message level—without the JAXB binding. web service Provider-based endpoints can be implemented synchronously or asynchronously using the `javax.xml.ws.Provider<T>` or `com.sun.xml.ws.api.server.AsyncProvider<T>` interfaces, respectively. For more information about developing web service Provider-based endpoints, see Developing a Web Service Provider-based Endpoint (Starting from Java).

A **web service Dispatch client**, implemented using the `javax.xml.ws.Dispatch<T>` interface, enables clients to work with messages at the XML level. The steps to develop a web service Dispatch client are described in Developing a Web Service Dispatch Client.

Provider endpoints and Dispatch clients can be used in combination with other WebLogic web services features as long as a WSDL is available, including:

- WS-Security
- WS-ReliableMessaging
- WS-MakeConnection

- WS-AtomicTransaction

In addition, Dispatch clients can be used in combination with the asynchronous client transport and asynchronous client handler features. These features are described in detail in Developing Asynchronous Clients, and a code example is provided in Creating a Dispatch Instance.

Usage Modes and Message Formats for Operating at the XML Level

When operating on messages at the XML level using Provider-based endpoints or Dispatch clients, you use one of the usage modes defined in the following table. You define the usage mode using the `javax.xml.ws.ServiceMode` annotation, as described in Specifying the Usage Mode (@ServiceMode Annotation).

Table 24-1 Usage Modes for Operating at the XML Message Level

Usage Mode	Description
Message	Operates directly with the entire message. For example, if a SOAP binding is used, then the entire SOAP envelope is accessed.
Payload	Operates on the payload of a message only. For example, if a SOAP binding is used, then the SOAP body is accessed.

Provider-based endpoints and Dispatch clients can receive and send messages using one of the message formats defined in Table 24-2. This table also defines the valid message format and usage mode combinations based on the configured binding type (SOAP or XML over HTTP).

Table 24-2 Message Formats Supported for Operating at the XML Message Level

Message Format	Usage Mode Support for SOAP/HTTP Binding	Usage Mode Support for XML/HTTP Binding
`javax.xml.transform.Source`	**Message mode**: SOAP envelope **Payload mode**: SOAP body	**Message mode**: XML content as `Source` **Payload mode**: XML content as `Source`
`javax.activation.DataSource`	Not valid in either mode because attachments in SOAP/HTTP binding are sent using `SOAPMessage` format.	**Message mode**: `DataSource` object Not valid in payload mode because `DataSource` is used for sending attachments.
`javax.xml.soap.SOAPMessage`	**Message mode**: SOAPMessage object Not valid in payload mode because the entire SOAP message is received, not just the payload.	Not valid in either mode because the client can send a non-SOAP message in XML/HTTP binding.

Developing a Web Service Provider-based Endpoint (Starting from Java)

You can develop both synchronous and asynchronous web service Provider-based endpoints, as described in the following sections:

- Developing a Synchronous Provider-based Endpoint
- Developing an Asynchronous Provider-based Endpoint

> **Note:**
>
> To start from WSDL and flag a port as a web service provider, see Developing a Web Service Provider-based Endpoint (Starting from WSDL).

Developing a Synchronous Provider-based Endpoint

A web service Provider-based endpoint, implemented using the `javax.xml.ws.Provider<T>`, enables you to access content directly at the XML message level—without the JAXB binding. The `Provider` interface processes messages synchronously—the service waits to process the response before continuing with its work. For more information about the `javax.xml.ws.Provider<T>` interface, see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Provider.html`.

The following procedure describes the typical steps for programming a JWS file that implements a synchronous web service Provider-based endpoint.

Table 24-3 Steps to Develop a Synchronous Web Service Provider-based Endpoint

#	Step	Description

1	Import the JWS annotations that will be used in your web service Provider-based JWS file.	The standard JWS annotations for a web service Provider-based JWS file include:
		`import javax.xml.ws.Provider;`
		`import javax.xml.ws.WebServiceProvider;`
		`import javax.xml.ws.ServiceMode;`
		
		Import additional annotations, as required. For a complete list of JWS annotations that are supported, see Web Service Annotation Support in *WebLogic Web Services Reference for Oracle WebLogic Server*.

2	Specify one of the message formats supported, defined in Table 24-2, when developing the Provider-based implementation class.	See Specifying the Message Format.

Table 24-3 (Cont.) Steps to Develop a Synchronous Web Service Provider-based Endpoint

#	Step	Description
3	Add the standard required `@WebServiceProvider` JWS annotation at the class level to specify that the Java class exposes a web service provider.	See Specifying that the JWS File Implements a Web Service Provider (@WebServiceProvider Annotation).
4	Add the standard `@ServiceMode` JWS annotation at the class level to specify whether the web service provider is accessing information at the message or message payload level. (Optional)	See Specifying the Usage Mode (@ServiceMode Annotation). The service mode defaults to `Service.Mode.Payload`.
5	Define the `invoke()` method.	The `invoke()` method is called and provides the message or message payload as input to the method using the specified message format. See Defining the invoke() Method for a Synchronous Provider-based Endpoints.

The following sample JWS file shows how to implement a simple synchronous web service Provider-based endpoint. The steps to develop a synchronous web service Provider-based endpoint are described in detail in the sections that follow. To review the JWS file within the context of a complete sample, see "Creating JAX-WS Web Services for Java EE" in the Web Services Samples distributed with Oracle WebLogic Server.

> **Note:**
>
> RESTful Web Services can be built using XML/HTTP binding Provider-based endpoints. For an example of programming a Provider-based endpoint within the context of a RESTful web service, see Programming Web Services Using XML Over HTTP.

Example 24-1 Example of a JWS File that Implements a Synchronous Provider-based Endpoint

```
package examples.webservices.jaxws;

import org.w3c.dom.Node;

import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Provider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.Service;
import java.io.ByteArrayInputStream;
```

```
/**
 * A simple Provider-based web service implementation.
 *
 * @author Copyright (c) 2010, Oracle and/or its affiliates.
 * All Rights Reserved.
 */
// The @ServiceMode annotation specifies whether the Provider instance
// receives entire messages or message payloads.
@ServiceMode(value = Service.Mode.PAYLOAD)

// Standard JWS annotation that configures the Provider-based web service.
@WebServiceProvider(portName = "SimpleClientPort",
    serviceName = "SimpleClientService",
    targetNamespace = "http://jaxws.webservices.examples/",
    wsdlLocation = "SimpleClientService.wsdl")
public class SimpleClientProviderImpl implements Provider<Source> {

  //Invokes an operation according to the contents of the request message.
  public Source invoke(Source source) {
    try {
      DOMResult dom = new DOMResult();
      Transformer trans = TransformerFactory.newInstance().newTransformer();
      trans.transform(source, dom);
      Node node = dom.getNode();
      // Get the operation name node.
      Node root = node.getFirstChild();
      // Get the parameter node.
      Node first = root.getFirstChild();
      String input = first.getFirstChild().getNodeValue();
      // Get the operation name.
      String op = root.getLocalName();
      if ("invokeNoTransaction".equals(op)) {
        return sendSource(input);
      } else {
        return sendSource2(input);
      }
    }
    catch (Exception e) {
      throw new RuntimeException("Error in provider endpoint", e);
    }
  }

  private Source sendSource(String input) {
    String body =
        "<ns:invokeNoTransactionResponse
             xmlns:ns=\"http://jaxws.webservices.examples/\"><return>"
           + "constructed:" + input
           + "</return></ns:invokeNoTransactionResponse>";
    Source source = new StreamSource(new ByteArrayInputStream(body.getBytes()));
    return source;
  }

  private Source sendSource2(String input) {
    String body =
        "<ns:invokeTransactionResponse
            xmlns:ns=\"http://jaxws.webservices.examples/\"><return>"
           + "constructed:" + input
           + "</return></ns:invokeTransactionResponse>";
    Source source = new StreamSource(new ByteArrayInputStream(body.getBytes()));
    return source;
```

```
    }

}
```

Developing an Asynchronous Provider-based Endpoint

As with the `Provider` interface, web service Provider-based endpoints implemented using the `com.sun.xml.ws.api.server.AsyncProvider<T>` interface enable you to access content directly at the XML message level—without the JAXB binding. However, the `AsyncProvider` interface processes messages asynchronously—the service can continue its work and process the request when it becomes available, without blocking the thread.

The following procedure describes the typical steps for programming a JWS file that implements an asynchronous web service Provider-based endpoint.

Table 24-4 Steps to Develop an Asynchronous Web Service Provider-based Endpoint

#	Step	Description

1	Import the JWS annotations that will be used in your web service Provider-based JWS file.	The standard JWS annotations for an asynchronous web service Provider-based JWS file include:
		`import com.sun.xml.ws.api.server.AsyncProvider;`
		`import com.sun.xml.ws.api.server.AsyncProviderCallback;`
		`import javax.xml.ws.ServiceMode;`
		
		Import additional annotations, as required. For a complete list of JWS annotations that are supported, see Web Service Annotation Support in *WebLogic Web Services Reference for Oracle WebLogic Server*.

2	Specify one of the message formats supported, defined in Table 24-2, when developing the Provider-based implementation class.	See Specifying the Message Format.
3	Add the standard required `@WebServiceProvider` JWS annotation at the class level to specify that the Java class exposes a web service provider.	See Specifying that the JWS File Implements a Web Service Provider (@WebServiceProvider Annotation).
4	Add the standard `@ServiceMode` JWS annotation at the class level to specify whether the web service provider is accessing information at the message or message payload level. (Optional)	See Specifying the Usage Mode (@ServiceMode Annotation). The service mode defaults to `Service.Mode.Payload`.
5	Define the `invoke()` method.	The `invoke()` method is called and provides the message or message payload as input to the method using the specified message format. See Defining the invoke() Method for an Asynchronous Provider-based Endpoints.
6	Define the asynchronous handler callback method to handle the response.	The method handles the response when it is returned. See Defining the Callback Handler for the Asynchronous Provider-based Endpoint.

The following sample JWS file shows how to implement a simple asynchronous web service Provider-based endpoint. The steps to develop an asynchronous web service Provider-based endpoint are described in detail in the sections that follow.

Example 24-2 Example of a JWS File that Implements an Asynchronous Provider-based Endpoint

```
package asyncprovider.server;

import com.sun.xml.ws.api.server.AsyncProvider;
import com.sun.xml.ws.api.server.AsyncProviderCallback;

import javax.xml.bind.JAXBContext;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.WebServiceProvider;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

@WebServiceProvider(
    wsdlLocation="WEB-INF/wsdl/hello_literal.wsdl",
    targetNamespace = "urn:test",
    serviceName="Hello")

public class HelloAsyncImpl implements AsyncProvider<Source> {

    private static final JAXBContext jaxbContext = createJAXBContext();
    private int bodyIndex;

    public javax.xml.bind.JAXBContext getJAXBContext(){
        return jaxbContext;
    }

    private static javax.xml.bind.JAXBContext createJAXBContext(){
        try{
            return javax.xml.bind.JAXBContext.newInstance(ObjectFactory.class);
        }catch(javax.xml.bind.JAXBException e){
            throw new WebServiceException(e.getMessage(), e);
        }
    }

    private Source sendSource() {
        System.out.println("**** sendSource ******");

        String[] body  = {
            "<HelloResponse xmlns=\"urn:test:types\">
              <argument xmlns=\"\">foo</argument>
              <extra xmlns=\"\">bar</extra>
             </HelloResponse>",
            "<ans1:HelloResponse xmlns:ans1=\"urn:test:types\">
               <argument>foo</argument>
               <extra>bar</extra>
             </ans1:HelloResponse>",
        };
        int i = (++bodyIndex)%body.length;
        return new StreamSource(
            new ByteArrayInputStream(body[i].getBytes()));
    }

    private Hello_Type recvBean(Source source) throws Exception {
```

```
    System.out.println("**** recvBean ******");
    return (Hello_Type)jaxbContext.createUnmarshaller().unmarshal(source);
}

private Source sendBean() throws Exception {
    System.out.println("**** sendBean ******");
    HelloResponse resp = new HelloResponse();
    resp.setArgument("foo");
    resp.setExtra("bar");
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    jaxbContext.createMarshaller().marshal(resp, bout);
    return new StreamSource(new ByteArrayInputStream(bout.toByteArray()));
}

public void invoke(Source source, AsyncProviderCallback<Source> cbak,
                   WebServiceContext ctxt) {
    System.out.println("**** Received in AsyncProvider Impl ******");
    try {
         Hello_Type hello = recvBean(source);
         String arg = hello.getArgument();
         if (arg.equals("sync")) {
             String extra = hello.getExtra();
             if (extra.equals("source")) {
                    cbak.send(sendSource());
             } else if (extra.equals("bean")) {
                    cbak.send(sendBean());
             } else {
                    throw new WebServiceException("Expected extra =
                         (source|bean|fault), Got="+extra);
             }
         } else if (arg.equals("async")) {
           new Thread(new RequestHandler(cbak, hello)).start();
         } else {
            throw new WebServiceException("Expected Argument =
              (sync|async), Got="+arg);
         }
    } catch(Exception e) {
      throw new WebServiceException("Endpoint failed", e);
    }
}

private class RequestHandler implements Runnable {
    final AsyncProviderCallback<Source> cbak;
    final Hello_Type hello;
    public RequestHandler(AsyncProviderCallback<Source> cbak, Hello_Type hello) {
        this.cbak = cbak;
        this.hello = hello;
    }

    public void run() {
        try {
            Thread.sleep(5000);
        } catch(InterruptedException ie) {
              cbak.sendError(new WebServiceException("Interrupted..."));
              return;
        }
        try {
            String extra = hello.getExtra();
            if (extra.equals("source")) {
                cbak.send(sendSource());
            } else if (extra.equals("bean")) {
```

```
                cbak.send(sendBean());
            } else {
                cbak.sendError(new WebServiceException(
                 "Expected extra = (source|bean|fault), Got="+extra));
            }
        } catch(Exception e) {
            cbak.sendError(new WebServiceException(e));
        }
    }
  }
}
```

Specifying the Message Format

Specify one of the message formats supported, defined in Table 24-2, when developing the Provider-based implementation class.

For example, in the Provider implementation example shown in Example 24-1, the `SimpleClientProviderImpl` class implements the `Provider<Source>` interface, indicating that both the input and output types are `java.xml.transform.Source` objects.

```
public class SimpleClientProviderImpl implements Provider<Source> {
. . .
}
```

Similarly, in the AsyncProvider implementation example shown in Example 24-2, the `HelloAsyncImpl` class implements the `AsyncProvider<Source>` interface, indicating that both the input and output types are `java.xml.transform.Source` objects.

```
public class HelloAsyncImpl implements AsyncProvider<Source>  {
. . .
}
```

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

Use the standard `javax.xml.ws.WebServiceProvider` annotation to specify, at the class level, that the JWS file implements a web service provider, as shown in the following code excerpt:

```
@WebServiceProvider(portName = "SimpleClientPort",
    serviceName = "SimpleClientService",
    targetNamespace = "http://jaxws.webservices.examples/",
    wsdlLocation = "SimpleClientService.wsdl")
```

In the example, the service name is `SimpleClientService`, which will map to the `wsdl:service` element in the generated WSDL file. The port name is `SimpleClientPort`, which will map to the `wsdl:port` element in the generated WSDL. The target namespace used in the generated WSDL is `http://jaxws.webservices.examples/` and the WSDL location is local to the web service provider, at `SimpleClientService.wsdl`.

For more information about the `@WebServiceProvider` annotation, see https://javaee.github.io/metro-jax-ws/doc/user-guide/release-documentation.html#users-guide-annotations.

Specifying the Usage Mode (@ServiceMode Annotation)

The `javax.xml.ws.ServiceMode` annotation is used to specify whether the web service Provider-based endpoint receives entire messages (`Service.Mode.MESSAGE`) or message payloads (`Service.Mode.PAYLOAD`) only.

For example:

```
@ServiceMode(value = Service.Mode.PAYLOAD)
```

If not specified, the `@ServiceMode` annotation defaults to `Service.Mode.PAYLOAD`.

For a list of valid message format and usage mode combinations, see Table 24-2.

For more information about the `@ServiceMode` annotation, see https://javaee.github.io/metro-jax-ws/doc/user-guide/release-documentation.html#users-guide-annotations.

Defining the invoke() Method for a Synchronous Provider-based Endpoints

The `Provider<T>` interface defines a single method that you must define in your implementation class:

```
T invoke(T request)
```

When a web service request is received, the `invoke()` method is called and provides the message or message payload as input to the method using the specified message format.

For example, in the Provider implementation example shown in Example 24-1, excerpted below, the class defines an `invoke` method to take as input the `Source` parameter and return a `Source` response.

```
public Source invoke(Source source) {
  try {
    DOMResult dom = new DOMResult();
    Transformer trans = TransformerFactory.newInstance().newTransformer();
    trans.transform(source, dom);
    Node node = dom.getNode();
    // Get the operation name node.
    Node root = node.getFirstChild();
    // Get the parameter node.
    Node first = root.getFirstChild();
    String input = first.getFirstChild().getNodeValue();
    // Get the operation name.
    String op = root.getLocalName();
    if ("invokeNoTransaction".equals(op)) {
      return sendSource(input);
    } else {
      return sendSource2(input);
    }
  }
  catch (Exception e) {
    throw new RuntimeException("Error in provider endpoint", e);
  }
}
```

Defining the invoke() Method for an Asynchronous Provider-based Endpoints

The `AsycnProvider<T>` interface defines a single method that you must define in your implementation class:

```
void invoke(T request, AsyncProviderCallback<t> callback, WebserviceContext context))
```

You pass the following parameters to the invoke method:

- Request message or message payload in the specified format.
- `com.sun.xml.ws.api.server.AsyncProviderCallback` implementation that will handle the response once it is returned. For more information, see Defining the Callback Handler for the Asynchronous Provider-based Endpoint.
- The `javax.xml.ws.WebServiceContext` that defines the message context for the request being served. An asynchronous Provider-based endpoint cannot use the injected `WebServiceContext` which relies on the calling thread to determine the request it should return information about. Instead, it passes the `WebServiceContext` object which remains usable until you invoke `AsyncProviderCallback`.

For example, in the AysncProvider implementation example shown in Example 24-2, excerpted below, the class defines an `invoke` method as shown below:

```
public void invoke(Source source, AsyncProviderCallback<Source> cbak,
                   WebServiceContext ctxt) {
    System.out.println("**** Received in AsyncProvider Impl ******");
    try {
         Hello_Type hello = recvBean(source);
         String arg = hello.getArgument();
         if (arg.equals("sync")) {
             String extra = hello.getExtra();
             if (extra.equals("source")) {
                    cbak.send(sendSource());
             } else if (extra.equals("bean")) {
                    cbak.send(sendBean());
             } else {
                    throw new WebServiceException("Expected extra =
                         (source|bean|fault), Got="+extra);
             }
         } else if (arg.equals("async")) {
           new Thread(new RequestHandler(cbak, hello)).start();
         } else {
            throw new WebServiceException("Expected Argument =
              (sync|async), Got="+arg);
         }
    } catch(Exception e) {
      throw new WebServiceException("Endpoint failed", e);
    }
}
```

Defining the Callback Handler for the Asynchronous Provider-based Endpoint

The `AsyncProviderCallback` interface enables you to define a callback handler for processing the asynchronous response once it is received.

For example, in the AysncProvider implementation example shown in Example 24-2, excerpted below, the `RequestHandler` method uses the AsyncProviderCallback callback handler to process the asynchronous response.

```
    private class RequestHandler implements Runnable {
        final AsyncProviderCallback<Source> cbak;
        final Hello_Type hello;
        public RequestHandler(AsyncProviderCallback<Source> cbak, Hello_Type
hello) {
            this.cbak = cbak;
            this.hello = hello;
        }

        public void run() {
            try {
                Thread.sleep(5000);
            } catch(InterruptedException ie) {
                  cbak.sendError(new WebServiceException("Interrupted..."));
                  return;
            }
            try {
                String extra = hello.getExtra();
                if (extra.equals("source")) {
                    cbak.send(sendSource());
                } else if (extra.equals("bean")) {
                    cbak.send(sendBean());
                } else {
                    cbak.sendError(new WebServiceException(
                     "Expected extra = (source|bean|fault), Got="+extra));
                }
           } catch(Exception e) {
                cbak.sendError(new WebServiceException(e));
           }
       }
    }
}
```

Developing a Web Service Provider-based Endpoint (Starting from WSDL)

If the Provider-based endpoint is being generated from a WSDL file, the `<provider>` WSDL extension can be used to mark a port as a provider. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings wsdlLocation="SimpleClientService.wsdl"
  xmlns="http://java.sun.com/xml/ns/jaxws">
<bindings node="wsdl:definitions" >
    <package name="provider.server"/>
   <provider>true</provider>
</bindings>
```

Using SOAP Handlers with Provider-based Endpoints

Provider-based endpoints may need to access the SOAP message for additional processing of the message request or response. You can create SOAP message handlers to enable Provider-based endpoints to perform this additional processing on the SOAP message, just as you do for an SEI-based endpoint. For more information about creating the SOAP handler, see Creating the SOAP Message Handler.

Table 18-1 enumerates the steps required to add a SOAP handler to a web service. These steps apply to web service Provider-based endpoints, as well.

For example:

1. Design SOAP message handlers and group them together in a *handler chain*, as described in Designing the SOAP Message Handlers and Handler Chains.
2. For each handler in the handler chain, create a Java class that implements the SOAP message handler interface, as described in Creating the SOAP Message Handler.

   An example of the SOAP handler, `MyHandler`, is shown below.

```
package provider.rootpart_charset_772.server;

import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.xml.namespace.QName;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import java.io.InputStream;
import java.io.OutputStream;.import java.io.ByteArrayInputStream;
import java.util.Set;

public class MyHandler implements SOAPHandler<SOAPMessageContext> {

    public Set<QName> getHeaders() {
        return null;
    }

    public boolean handleMessage(SOAPMessageContext smc) {
        if (!
(Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY))
            return true;
        try {
            SOAPMessage msg = smc.getMessage();
            AttachmentPart part =
msg.createAttachmentPart(getDataHandler());

part.setContentId("SOAPTestHandler@example.jaxws.sun.com");
            msg.addAttachmentPart(part);
            msg.saveChanges();
            smc.setMessage(msg);
        } catch (Exception e) {
            throw new WebServiceException(e);
        }
        return true;
```

```
    }

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

    public void close(MessageContext context) {}

    private DataHandler getDataHandler() throws Exception {
        return new DataHandler(new DataSource() {
            public String getContentType() {
                return "text/xml";
            }

            public InputStream getInputStream() {
                return new ByteArrayInputStream("<a/
>".getBytes());
            }

            public String getName() {
                return null;
            }

            public OutputStream getOutputStream() {
                throw new UnsupportedOperationException();
            }
        });
    }
}
```

3. Add the `@javax.jws.HandlerChain` annotation to the Provider implementation, as described in Configuring Handler Chains in the JWS File.

   For example:

```
package provider.rootpart_charset_772.server;

import javax.jws.HandlerChain;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.*;
import java.io.ByteArrayInputStream;

@WebServiceProvider(targetNamespace="urn:test", portName="HelloPort",
serviceName="Hello")
@ServiceMode(value=Service.Mode.MESSAGE)
@HandlerChain(file="handlers.xml")
public class SOAPMsgProvider implements Provider<SOAPMessage> {

    public SOAPMessage invoke(SOAPMessage msg) {
        try {
    // keeping white space in the string is intentional
            String content = "<soapenv:Envelope
                xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\">
                <soapenv:Body>  <VoidTestResponse
                xmlns=\"urn:test:types\">
                </VoidTestResponse></soapenv:Body></soapenv:Envelope>";
            Source source = new StreamSource(new
                ByteArrayInputStream(content.getBytes()));
            MessageFactory fact = MessageFactory.newInstance();
```

```
            SOAPMessage soap = fact.createMessage();
            soap.getSOAPPart().setContent(source);
            soap.getMimeHeaders().addHeader("foo", "bar");
            return soap;
        } catch(Exception e) {
            throw new WebServiceException(e);
        }
    }
}
```

4. Create the handler chain configuration file, as described in Creating the Handler Chain Configuration File.

   An example of the handler chain configuration file, `handlers.xml`, is shown below.

```
<handler-chains xmlns='http://java.sun.com/xml/ns/javaee'>
   <handler-chain>
      <handler>
         <handler-name>MyHandler</handler-name>
         <handler-class>
             provider.rootpart_charset_772.server.MyHandler
         </handler-class>
      </handler>
   </handler-chain>
</handler-chains>
```

5. Compile all handler classes in the handler chain and rebuild your web service, as described in Compiling and Rebuilding the Web Service .

Developing a Web Service Dispatch Client

A web service Dispatch client, implemented using the `javax.xml.ws.Dispatch<T>` interface, enables clients to work with messages at the XML level.

The following procedure describes the typical steps for programming a web service Dispatch client.

Table 24-5 Steps to Develop a Web Service Provider-based Endpoint

#	Step	Description

1	Import the JWS annotations that will be used in your web service Provider-based JWS file.	The standard JWS annotations for a web service Provider-based JWS file include:
		`import javax.xml.ws.Service;`
		`import javax.xml.ws.Dispatch;`
		`import javax.xml.ws.ServiceMode;`
		
		Import additional annotations, as required. For a complete list of JWS annotations that are supported, see Web Service Annotation Support in *WebLogic Web Services Reference for Oracle WebLogic Server*.

2	Create a `Dispatch` instance.	See Creating a Dispatch Instance.
3	Invoke a web service operation.	You can invoke a web service operation synchronously (one-way or two-way) or asynchronously (polling or asynchronous handler). See Invoking a Web Service Operation.

Example of a Web Service Dispatch Client

The following sample shows how to implement a basic web service Dispatch client. The sample is described in detail in the sections that follow.

Example 24-3 Example of a Web Service Dispatch Client

```
package jaxws.dispatch.client;

import java.io.ByteArrayOutputStream;
import java.io.OutputStream;
import java.io.StringReader;
import java.net.URL;

import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.WebServiceException;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;
import javax.xml.ws.soap.SOAPBinding;

public class WebTest extends TestCase {
   private static String in_str = "wiseking";
   private static String request =
      "<ns1:sayHello xmlns:ns1=\"http://example.org\"><arg0>"+in_str+"</arg0></ns1:sayHello>";

   private static final QName portQName = new QName("http://example.org", "SimplePort");
   private Service service = null;

   protected void setUp() throws Exception {

      String url_str = System.getProperty("wsdl");
      URL url = new URL(url_str);
      QName serviceName = new QName("http://example.org", "SimpleImplService");
      service = Service.create(serviceName);
      service.addPort(portQName, SOAPBinding.SOAP11HTTP_BINDING, url_str);
      System.out.println("Setup complete.");

   }

   public void testSayHelloSource() throws Exception {
      setUp();
      Dispatch<Source> sourceDispatch =
         service.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
      System.out.println("\nInvoking xml request: " + request);
      Source result = sourceDispatch.invoke(new StreamSource(new StringReader(request)));
      String xmlResult = sourceToXMLString(result);
      System.out.println("Received xml response: " + xmlResult);
      assertTrue(xmlResult.indexOf("HELLO:"+in_str)>=0);
```

```
   }

   private String sourceToXMLString(Source result) {
      String xmlResult = null;
      try {
         TransformerFactory factory = TransformerFactory.newInstance();
         Transformer transformer = factory.newTransformer();
         transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
         transformer.setOutputProperty(OutputKeys.METHOD, "xml");
         OutputStream out = new ByteArrayOutputStream();
         StreamResult streamResult = new StreamResult();
         streamResult.setOutputStream(out);
         transformer.transform(result, streamResult);
         xmlResult = streamResult.getOutputStream().toString();
      } catch (TransformerException e) {
         e.printStackTrace();
      }
      return xmlResult;
   }

}
```

Creating a Dispatch Instance

The `javax.xml.ws.Service` interface acts as a factory for the creation of `Dispatch` instances. So to create a Dispatch instance, you must first create a `Service` instance. Then, create the Dispatch instance using the `Service.createDispatch()` method.

You can pass one or more of the following parameters to the `createDispatch()` method:

- Qualified name (QName) or endpoint reference of the target service endpoint.
- Class of the type parameter `T`. In this example, the `javax.xml.transform.Source` format is used. For valid values, see Table 24-2.
- Usage mode. In this example, the message payload is specified. For valid usage modes, see Table 24-1.
- A list of web service features to configure on the proxy.
- JAXB context used to marshal or unmarshal messages or message payloads.

For more information about the valid parameters that can be used to call the `Service.createDispatch()` method, see the `javax.xml.ws.Service` Javadoc at: `https://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Service.html`.

For example:

```
...
      String url_str = System.getProperty("wsdl");
      QName serviceName = new QName("http://example.org", "SimpleImplService");
      service = Service.create(serviceName);
      service.addPort(portQName, SOAPBinding.SOAP11HTTP_BINDING, url_str);
      Dispatch<Source> sourceDispatch =
        service.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
...
```

In the example above, the `createDispatch()` method takes three parameters:

- Qualified name (QName) of the target service endpoint.

- Class of the type parameter `T`. In this example, the `javax.xml.transform.Source` format is used. For valid values, see Table 24-2.
- Usage mode. In this example, the message payload is specified. For valid usage modes, see Table 24-1.

The following example shows how to pass a list of web service features, specifically the asynchronous client transport feature and asynchronous client handler feature. For more information about these features, see Developing Asynchronous Clients.

```
...
  protected Dispatch createDispatch(boolean isSoap12, Class dateType,
                Service.Mode mode, AsyncClientHandlerFeature feature)
                throws Exception {
    String address = publishEndpoint(isSoap12);
    Service service = Service.create(new URL(address + "?wsdl"),
                new QName("http://example.org", "AddNumbersService"));
    QName portName = new QName("http://example.org", "AddNumbersPort");
    AsyncClientTransportFeature transportFeature = new
         AsyncClientTransportFeature("http://localhost:8238/clientsoap12/" +
                UUID.randomUUID().toString());
    Dispatch dispatch = service.createDispatch(portName, dateType, mode,
         feature, transportFeature);
    return dispatch;
  }
...
```

Invoking a Web Service Operation

Once the `Dispatch` instance is created, use it to invoke a web service operation. You can invoke a web service operation synchronously (one-way or two-way) or asynchronously (polling or asynchronous handler). For complete details about the synchronous and asynchronous invoke methods, see the `javax.xml.ws.Dispatch` Javadoc at: https://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Dispatch.html.

For example, in the following code excerpt, the XML message is encapsulated as a `javax.xml.transform.stream.StreamSource` object and passed to the synchronous `invoke()` method. The response XML is returned in the *`result`* variable as a `Source` object, and transformed back to XML. The `sourcetoXMLString()` method used to transform the message back to XML is shown in Example 24-3.

```
...
private static String request = "<ns1:sayHello xmlns:ns1=\"http://
example.org\"><arg0>"+in_str+"</arg0></ns1:sayHello>";
Source result = sourceDispatch.invoke(new StreamSource(new StringReader(request)));
String xmlResult = sourceToXMLString(result);
...
```

25

Sending and Receiving SOAP Headers

This chapter describes how use the methods available from `com.sun.xml.ws.developer.WSBindingProvider` to send outbound or receive inbound SOAP headers for WebLogic web services using Java API for XML Web Services (JAX-WS). This chapter includes the following sections:

- Overview of Sending and Receiving SOAP Headers
- Sending SOAP Headers Using WSBindingProvider
- Receiving SOAP Headers Using WSBindingProvider

> **Note:**
>
> The `com.sun.xml.ws.developer.WSBindingProvider` and `com.sun.xml.ws.api.message.Headers` APIs are supported as an extension to the JDK 6.0. Because the APIs are not provided as part of the JDK 6.0 kit, they are subject to change.

Overview of Sending and Receiving SOAP Headers

When you create a proxy or Dispatch client, the client implements the `javax.xml.ws.BindingProvider` interface. If you need to send or receive a SOAP header, you can downcast the web service proxy or Dispatch client to `com.sun.xml.ws.developer.WSBindingProvider` and use the methods on the interface to send outbound or receive inbound SOAP headers.

Sending SOAP Headers Using WSBindingProvider

Use the `setOutboundHeaders` method to the `com.sun.xml.ws.developer.WSBindingProvider` to send SOAP headers. You create SOAP headers using the `com.sun.xml.ws.api.message.Headers` method.

For example, the following provides a code excerpt showing how to pass a simple string value as a header.

Example 25-1 Sending SOAP Headers Using WSBindingProvider

```
import com.sun.xml.ws.developer.WSBindingProvider;
import com.sun.xml.ws.api.message.Headers;
import javax.xml.namespace.QName;
...
HelloService helloService = new HelloService();
HelloPort port = helloService.getHelloPort();
WSBindingProvider bp = (WSBindingProvider)port;

bp.setOutboundHeaders(
```

```
  // Sets a simple string value as a header
  Headers.create(new QName("simpleHeader"),"stringValue")
);
...
```

Receiving SOAP Headers Using WSBindingProvider

Use the `getInboundHeaders` method to the `com.sun.xml.ws.developer.WSBindingProvider` to receive SOAP headers.

For example, the following provides a code excerpt showing how to get inbound headers.

Example 25-2 Receiving SOAP Headers Using WSBindingProvider

```
import com.sun.xml.ws.developer.WSBindingProvider;
import com.sun.xml.ws.api.message.Headers;
import javax.xml.namespace.QName;
import java.util.List;
...
HelloService helloService = new HelloService();
HelloPort port = helloService.getHelloPort();
WSBindingProvider bp = (WSBindingProvider)port;

List inboundHeaders = bp.getInboundHeaders();
...
```

26
Using Callbacks

This chapter describes how to use callbacks to notify clients of events for WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of Callbacks

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

Example Callback Implementation

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

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

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

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

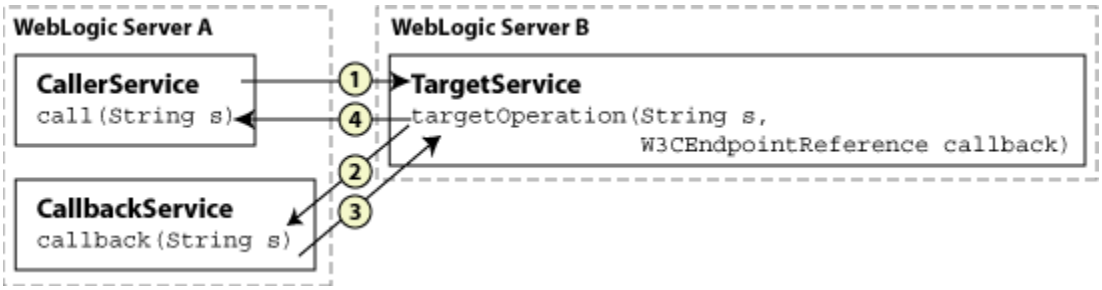
  In the example, this web service is called `TargetService` and it defines a single standard method called `targetOperation()`.

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

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

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

Figure 26-1 Example Callback Implementation



1. The `call()` method of the `CallerService` web service, running in one WebLogic Server instance, explicitly invokes the `targetOperation()` method of the `TargetService` and passes a web service endpoint to the `CallbackService`. Typically, the `TargetService` service is running in a separate WebLogic Server instance.
2. The implementation of the `TargetService.targetOperation()` method explicitly invokes the `callback()` method of the `CallbackService`, which implements the callback service, using the web service endpoint that is passed in from `CallerService` when the method is called.
3. The `CallbackService.callback()` method sends information back to the `TargetService` web service.
4. The `TargetService.targetOperation()` method, in turn, sends the information back to the `CallerService` service, completing the callback sequence.

Steps to Program Callbacks

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

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the web services. For more information, see Programming the JWS File.

Table 26-1 Steps to Program Callbacks

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the target web service.	Use your favorite IDE or text editor. See Programming Guidelines for Target Web Service. **Note:** The JWS file that implements the target web service invokes one or more callback methods of the callback web service. However, the step that describes how to program the callback web service comes later in this procedure. For this reason, programmers typically program the three JWS files at the same time, rather than linearly as implied by this procedure. The steps are listed in this order for clarity only.
2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the target JWS file into a web service.	See Updating the build.xml File for the Target Web Service.
3	Run the Ant target to build the target web service.	For example: `prompt> ant build-target`
4	Deploy the target web service as usual.	See Deploying and Undeploying WebLogic Web Services.
5	Create a new JWS file, or update an existing one, that implements the client web service.	It is assumed that the client web service is deployed to a different WebLogic Server instance from the one that hosts the target web service. See Programming Guidelines for the Callback Client Web Service.
6	Create the JWS file that implements the callback web service.	See Programming Guidelines for the Callback Web Service.

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

8	Run the Ant target to build the client and callback web services.	For example: `prompt> ant build-caller`
9	Deploy the client web service as usual.	See Deploying and Undeploying WebLogic Web Services.

Programming Guidelines for Target Web Service

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

```
package examples.webservices.callback;

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

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

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

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

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

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

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

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

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

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

- Return the result to the `CallerService` service.

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

Programming Guidelines for the Callback Client Web Service

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

```
package examples.webservices.callback;

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

import examples.webservices.callback.target.*;

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

public class CallerImpl
{
   @Resource
   private WebServiceContext context;

   @WebServiceRef()
   private TargetService target;

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

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

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

  ```
  import javax.xml.ws.Endpoint;
  import javax.xml.ws.WebServiceContext;
  import javax.xml.ws.handler.MessageContext;
  ```

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

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

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

  For more information about accessing runtime information using `WebServiceContext` and `MessageContext`, see Accessing Runtime Information About a Web Service.

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

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

  For more information about web service publishing, see Publishing a Web Service Endpoint.

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

> **Note:**
>
> Ensure that the callback web service is deployed during the range of time that callbacks may arrive.

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

- Stop publishing the endpoint:

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

Programming Guidelines for the Callback Web Service

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

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

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

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

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

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

     public CallbackWS() {
     }

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

Updating the build.xml File for the Target Web Service

You update a `build.xml` file to generate a target web service that invokes the callback web service by adding `taskdefs` and a `build-target` target that resemble the following example. See the description after the example for details.

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

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

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

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

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

For more information about `jwsc`, see Running the jwsc WebLogic Web Services Ant Task in *Developing JAX-RPC Web Services for Oracle WebLogic Server*.

27

Developing Dynamic Proxy Clients

This chapter highlights the differences between static and dynamic proxy clients, and describes the steps to develop a dynamic proxy client for WebLogic web services using Java API for XML Web Services (JAX-WS)
This chapter includes the following sections:

- Overview of Static Versus Dynamic Proxy Clients
- Steps to Develop a Dynamic Proxy Client
- Additional Considerations When Specifying WSDL Location

Overview of Static Versus Dynamic Proxy Clients

Table 27-1 highlights the differences between static and dynamic proxy clients.

Table 27-1 Static Versus Dynamic Proxy Clients

Proxy Client Type	Description
Static proxy client	Compile and bind the web service client at development time. This generates a *static stub* for the web service client proxy. The source code for the generated static stub client relies on a specific service implementation. As a result, this option offers the least flexibility. For examples of static proxy clients, see: • Invoking Web Service Clients in *Developing JAX-WS Web Services for Oracle WebLogic Server* • Roadmap for Developing JAX-WS Web Service Clients
Dynamic proxy client	Compile nothing at development time. At runtime, the application retrieves and interprets the WSDL and dynamically constructs calls. A *dynamic proxy client* enables a web service client to invoke a web service based on a service endpoint interface (SEI) dynamically at run-time (without using `clientgen`). This option does not rely upon a specific service implementation, providing greater flexibility, but also a greater performance hit. The steps to develop a dynamic proxy client are described in Steps to Develop a Dynamic Proxy Client.

Steps to Develop a Dynamic Proxy Client

The steps to create a dynamic proxy client are outlined in the following table. For more information, see the `javax.xml.ws.Service` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Service.html`.

Table 27-2 Steps to Create a Dynamic Proxy Client

#	Step	Description

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

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

Additional Considerations When Specifying WSDL Location

If you use HTTPS to get the web service from the WSDL, and the hostname definition in the WebLogic Server SSL certificate does not equal the hostname of the peer HTTPS server or is not one of the following, the action fails with a hostname verification error:

- localhost
- 127.0.0.1
- hostname of localhost
- IP address of localhost

The hostname verification error is as follows:

```
EchoService service = new EchoService(https-wsdl, webservice-qName);
:
:
javax.xml.ws.WebServiceException: javax.net.ssl.SSLKeyException:
Security:090504 Certificate chain received from host.example.com - 192.0.2.1
```

```
failed hostname verification check. Certificate contained {....} but
check expected host.example.com
```

The recommended workaround is to use HTTP instead of HTTPS to get the web service from a WSDL when creating the service, and your own hostname verifier code to verify the hostname after the service is created:

```
EchoService service = Service.create(http_wsdl, qname);
//get Port
EchoPort port = service.getPort(...);
//set self-defined hostname verifier
((BindingProvider) port).getRequestContext().put(
      com.sun.xml.ws.developer.JAXWSProperties.HOSTNAME_VERIFIER,
      new  MyHostNameVerifier());
/*
*/
```

Optionally, you can ignore hostname verification by setting the binding provider property:

```
((BindingProvider) port).getRequestContext().put(
      BindingProviderProperties.HOSTNAME_VERIFICATION_PROPERTY,
      "true");
```

However, if you must use HTTPS to get the web service from the WSDL, there are several possible workarounds:

- Turn off hostname verification if you are using the WebLogic Server HTTPS connection. To do this, set the global system property to ignore hostname verification:

  ```
  weblogic.security.SSL.ignoreHostnameVerification=true
  ```

  The system property does not work for service creation if the connection is a JDK connection or other non-WebLogic Server connection.

- Set your own hostname verifier for the connection before you get the web service from the WSDL, then use HTTPS to get the web service from the WSDL:

  ```
  //set self-defined hostname verifier
  URL url = new URL(https_wsdl);
  HttpsURLConnection connection = (HttpsURLConnection)url.openConnection();
  connection.setHostnameVerifier(new MyHostNameVerifier());

  //then initiate the service
  EchoService service = Service.create(https_wsdl, qname);

  //get port and set self-defined hostname verifier to binding provider
  ...
  ```

For the workarounds in which you set your own hostname verifier, an example hostname verifier might be as follows:

```
public class MyHostnameVerifier implements HostnameVerifier {
    public boolean verify(String hostname, SSLSession session) {
            if (hostname.equals("the host you want"))
          return true;
           else
               return false;
    }
}
```

28
Publishing a Web Service Endpoint

This chapter describes how to create a web service endpoint at runtime *without deploying* the web service to a WebLogic Server instance using the `javax.xml.ws.Endpoint` API.
For more information, see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Endpoint.html`.

The following table summarizes the steps to publish a web service endpoint.

Table 28-1 Steps to Publish a Web Service Endpoint

#	Step	Description
1	Create a web service endpoint.	Use the `javax.xml.ws.Endpoint create()` method to create the endpoint, specify the *implementor* (that is, the web service implementation) to which the endpoint is associated, and optionally specify the binding type. If not specified, the binding type defaults to `SOAP1.1/HTTP`. The endpoint is associated with only one implementation object and one `javax.xml.ws.Binding`, as defined at runtime; these values cannot be changed. For example, the following example creates a web service endpoint for the `CallbackWS()` implementation. `Endpoint callbackImpl = Endpoint.create(new CallbackWS());`
2	Publish the web service endpoint to accept incoming requests.	Use the `javax.xml.ws.Endpoint publish()` method to specify the server context, or the address and optionally the implementor of the web service endpoint. **Note:** If you wish to update the metadata documents (WSDL or XML schema) associated with the endpoint, you must do so before publishing the endpoint. For example, the following example publishes the web service endpoint created in Step 1 using the server context. `Object sc context.getMessageContext().get(MessageContext.SERVLET_CONTEXT); callbackImpl.publish(sc);`
3	Stop the web service endpoint to shut it down and prevent additional requests after processing is complete.	Use the `javax.xml.ws.Endpoint stop()` method to shut down the endpoint and stop accepting incoming requests. Once stopped, an endpoint cannot be republished. For example: `callbackImpl.stop()`

For an example of publishing a web service endpoint within the context of a callback example, see Programming Guidelines for the Callback Client Web Service.

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

- Endpoint metadata documents (WSDL or XML schema) associated with the endpoint. You must define metadata before publishing the web service endpoint.
- Endpoint properties.
- `java.util.concurrent.Executor` that will be used to dispatch incoming requests to the application (see `http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html`).

For more information, see the `javax.xml.ws.Endpoint` Javadoc at `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Endpoint.html`.

29
Using XML Catalogs

This chapter describes how to use XML catalogs with WebLogic web services using Java API for XML Web Services (JAX-WS).
This chapter includes the following sections:

Overview of XML Catalogs

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

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

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

Table 29-1 Support for XML Catalogs in WebLogic Server Ant Tasks

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

Table 29-1 (Cont.) Support for XML Catalogs in WebLogic Server Ant Tasks

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

The following sections describe how to:

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

For more information about XML catalogs, see the *Oasis XML Catalogs* specification at `http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html`.

Defining and Referencing XML Catalogs

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

- **Define an external XML catalog** - Define an external XML catalog file and reference that file from the `clientgen` or `wsdlc` Ant tasks in your `build.xml` file using the `catalogs` attribute. For more information, see Defining an External XML Catalog.

> **Note:**
>
> If you use the catalog option, you cannot define the catalog element in the catalog file using a relative path that starts with "../". If you do so, the element file cannot be copied to the client class directory and it may cause an unexpected exception in the client runtime.

- **Embed an XML catalog** - Embed the XML catalog directly in the `build.xml` file using the `<xmlcatalog>` element and reference it from the `clientgen` or `wsdlc` Ant tasks in your `build.xml` file using the `<xmlcatalog>` child element. For more information, see Embedding an XML Catalog.

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

> **Note:**
>
> You can use the `wsdlget` Ant task to get a local copy of the XML resources, as described in Disabling XML Catalogs in the Client Runtime.

Defining an External XML Catalog

To define an external XML catalog:

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

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

Creating an External XML Catalog File

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

The following provides a sample XML catalog file:

```
<catalog xmln="urn:oasis:names:tc:entity:xmlns:xml:catalog"
    prefer="system">
    <system systemId="http://foo.org/hello?wsdl"
            uri="HelloService.wsdl" />
    <public publicId="ISO 8879:1986//ENTITIES Added Latin 1//EN"
            uri="wsdl/myApp/myApp.wsdl"/>
</catalog>
```

In the above example:

- The `<catalog>` root element defines the XML catalog namespace and sets the `prefer` attribute to `system` to specify that system matches are preferred.

- The `<system>` element associates a URI reference with a system identifier.
- The `<public>` element associates a URI reference with a public identifier.

For a complete description of the XML catalog file syntax, see the *Oasis XML Catalogs* specification at http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html.

Referencing the External XML Catalog File

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

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

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

Embedding an XML Catalog

To embed an XML catalog:

1. Create an embedded XML catalog in the `build.xml` file. See Creating an Embedded XML Catalog.
2. Reference the embedded XML catalog from the `clientgen` or `wsdlc` Ant task using the `xmlcatalog` child element. See Referencing an Embedded XML Catalog.

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

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

Creating an Embedded XML Catalog

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

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

For a complete description of the embedded XML catalog syntax, see the *Oasis XML Catalogs* specification at http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html.

Referencing an Embedded XML Catalog

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

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

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

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

```
<target name="clientgen">
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"/>
     <xmlcatalog refid="wsimportcatalog"/>
</clientgen>
</target>
<xmlcatalog id="wsimportcatalog">
     <entity publicid="http://helloservice.org/types/HelloTypes.xsd"
             location="${basedir}/HelloTypes.xsd"/>
</xmlcatalog>
```

Disabling XML Catalogs in the Client Runtime

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

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

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

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

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

Getting a Local Copy of XML Resources

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

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

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

30

Programming Web Services Using XML Over HTTP

This chapter describes how to program web services using XML over HTTP.
This chapter includes the following sections:

About Programming Web Services Using XML Over HTTP

In addition to standard "SOAP over HTTP" use cases, WebLogic JAX-WS can also be used for some "XML over HTTP" web services. Use of the XML over HTTP style allows you to build simple RESTful web services while still leveraging the convenience of the JAX-WS programming model.

> **Note:**
>
> As a best practice, it is recommended that you develop RESTful web services using the Jersey JAX-RS RI, as described in Developing RESTful Web Services in *Developing and Securing RESTful Web Services for Oracle WebLogic Server*. The Jersey JAX-RS RI provides an open source, production quality RI for building RESTful web services and supports all of the HTTP methods.

When using the HTTP protocol to access web service resources, the resource identifier is the URL of the resource and the standard operation to be performed on that resource is one of the HTTP methods: GET, PUT, DELETE, POST, or HEAD.

> **Note:**
>
> In this JAX-WS implementation, the set of supported HTTP methods is limited to GET and POST. DELETE, PUT, and HEAD are not supported. Any HTTP requests containing these methods will be rejected with a `405 Method Not Allowed` error.
>
> If the functionality of PUT and DELETE are required, the desired action can be accomplished by tunneling the actual method to be executed on the POST method. This is a workaround referred to as *overloaded POST*. (A Web search on "REST overloaded POST" will return a number of ways to accomplish this.)

You build RESTful-like endpoints using the `invoke()` method of the `javax.xml.ws.Provider<T>` interface (see `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/Provider.html`). The `Provider` interface provides a dynamic alternative to building an service endpoint interface (SEI).

The procedure in this section describes how to program and compile the JWS file required to implement web services using XML over HTTP. The procedure shows how to create the JWS file from scratch; if you want to update an existing JWS file, you can also use this procedure as a guide.

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

Table 30-1 Steps to Program RESTful Web Services

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the web service using XML over HTTP.	Use your favorite IDE or text editor. See Programming Guidelines for the Web Service Using XML Over HTTP.

2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the JWS file into a web service.	For example:
		`<jwsc srcdir="." destdir="output/restEar">`
		`<jws file="NearbyCity.java" type="JAXWS"/>`
		`</jwsc>`
		For more information, see Running the jwsc WebLogic Web Services Ant Task.

3	Run the Ant target to build the web service.	For example: `prompt> ant build-rest`
4	Deploy the web service as usual.	See Deploying and Undeploying WebLogic Web Services.
5	Access the web service from your web service client.	See Accessing the Web Service from a Client.

Programming Guidelines for the Web Service Using XML Over HTTP

The following example shows a simple JWS file that implements a web service using XML over HTTP; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.jaxws.rest;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.BindingType;
import javax.xml.ws.Provider;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.ws.http.HTTPException;
```

```
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import java.io.ByteArrayInputStream;
import java.util.StringTokenizer;

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

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

  public Source invoke(Source source)  {
    try {
        MessageContext messageContext = wsContext.getMessageContext();

        // Obtain the HTTP mehtod of the input request.
        javax.servlet.http.HttpServletRequest servletRequest =
          (javax.servlet.http.HttpServletRequest)messageContext.get(
           MessageContext.SERVLET_REQUEST);
        String httpMethod = servletRequest.getMethod();
        if (httpMethod.equalsIgnoreCase("GET"));
        {

            String query =
                 (String)messageContext.get(MessageContext.QUERY_STRING);
            if (query != null && query.contains("lat=") &&
                query.contains("long=")) {
                return createSource(query);
            } else {
                System.err.println("Query String = "+query);
                throw new HTTPException(404);
            }
            } catch(Exception e) {
              e.printStackTrace();
              throw new HTTPException(500);
            }
        }
        } else {
   // This operation only supports "GET"
       throw new HTTPException405);
   }
    private Source createSource(String str) throws Exception {
        StringTokenizer st = new StringTokenizer(str, "=&/");
        String latLong = st.nextToken();
        double latitude = Double.parseDouble(st.nextToken());
        latLong = st.nextToken();
        double longitude = Double.parseDouble(st.nextToken());
        City nearby = City.findNearBy(latitude, longitude);
        String body = nearby.toXML();
        return new StreamSource(new ByteArrayInputStream(body.getBytes()));
    }

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

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

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

        static final City[] cities = {
           new City("San Francisco",37.7749295,-122.4194155,"CA"),
           new City("Columbus",39.9611755,-82.9987942,"OH"),
           new City("Indianapolis",39.7683765,-86.1580423,"IN"),
           new City("Jacksonville",30.3321838,-81.655651,"FL"),
           new City("San Jose",37.3393857,-121.8949555,"CA"),
           new City("Detroit",42.331427,-83.0457538,"MI"),
           new City("Dallas",32.7830556,-96.8066667,"TX"),
           new City("San Diego",32.7153292,-117.1572551,"CA"),
           new City("San Antonio",29.4241219,-98.4936282,"TX"),
           new City("Phoenix",33.4483771,-112.0740373,"AZ"),
           new City("Philadelphia",39.952335,-75.163789,"PA"),
           new City("Houston",29.7632836,-95.3632715,"TX"),
           new City("Chicago",41.850033,-87.6500523,"IL"),
           new City("Los Angeles",34.0522342,-118.2436849,"CA"),
           new City("New York",40.7142691,-74.0059729,"NY")};
        static City findNearBy(double lati, double longi) {
            int n = 0;
            for (int i = 1; i < cities.length; i++)  {
                if (cities[i].distance(lati, longi) <
                     cities[n].distance(lati, longi)) {
                    n = i;
                }
            }
            return cities[n];
        }

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

Follow these guidelines when programming the JWS file that implements the web service using XML over HTTP. Code snippets of the guidelines are shown in **bold** in the preceding example.

- Import the packages required to implement the Provider web service.

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

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

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

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

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

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

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

- Get the request string using the `QUERY_STRING` field in the `javax.xml.ws.handler.MessageContext` for processing (see message URL `http://docs.oracle.com/javase/8/docs/api/javax/xml/ws/handler/MessageContext.html`). The query string is then passed to the `createSource()` method that returns the city, state, longitude, and latitude that is closest to the specified values.

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

Accessing the Web Service from a Client

To access the web service from a web service client, use the resource URI. For example:

```
URL url = new URL (http://localhost:7001/NearbyCity/NearbyCityService?
lat=35&long=-120);
HttpURLConnection conn = (HttpURLConnection)url.openConnection();
connection.setRequestMethod("POST");
// Get result
InputStream is = connection.getInputStream();
```

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

Securing Web Services that Use XML Over HTTP

You can secure web services that use XML over HTTP using the same methods that you use to secure Web applications. For more information, see Options for Securing Web Application and EJB Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

31

Programming Stateful JAX-WS Web Services Using HTTP Session

This chapter describes how you can develop WebLogic web services using Java API for XML Web Services (JAX-WS) that interact with an Oracle database.
This chapter includes the following sections:

Overview of Stateful Web Services

Normally, a JAX-WS web service is stateless: that is, none of the local variables and object values that you set in the web service object are saved from one invocation to the next. Even sequential requests from a single client are treated each as independent, stateless method invocations.

There are web service use cases where a client may want to save data on the service during one invocation and then use that data during a subsequent invocation. For example, a shopping cart object may be added to by repeated calls to the `addToCart` web method and then fetched by the `getCart` web method. In a stateless web service, the shopping cart object would always be empty, no matter how many `addToCart` methods were called. But by using HTTP Sessions to maintain state across web service invocations, the cart may be built up incrementally, and then returned to the client.

Enabling stateful support in a JAX-WS web service requires a minimal amount of coding on both the client and server.

Accessing HTTP Session on the Server

On the server, every web service invocation is tied to an HttpSession object. This object may be accessed from the web service Context that, in turn, may be bound to the web service object using resource injection. Once you have access to your HttpSession object, you can "hang" off of it any stateful objects you want. The next time your client calls the web service, it will find that same HttpSession object and be able to lookup the objects previously stored there. Your web service is stateful!

The steps required on the server:

1. Add the @Resource (defined by Common Annotations for the Java Platform, JSR 250) to the top of your web service.
2. Add a variable of type WebServiceContext that will have the context injected into it.

3. Using the web service context, get the HttpSession object.
4. Save objects in the HttpSession using the setAttribute method and retrieve saved object using getAttribute. Objects are identified by a string value you assign.

Example 31-1 Accessing HTTP Session on the Server

The following snippet shows its usage:

```
@WebService
public class ShoppingCart {
   @Resource    // Step 1
   private WebServiceContext wsContext;    // Step 2
   public int addToCart(Item item) {
      // Find the HttpSession
      MessageContext mc = wsContext.getMessageContext();    // Step 3
      HttpSession session =
((javax.servlet.http.HttpServletRequest)mc.get(MessageContext.SERVLET_REQUEST)).g
etSession();
      if (session == null)
         throw new WebServiceException("No HTTP Session found");
      // Get the cart object from the HttpSession (or create a new one)
      List<Item> cart = (List<Item>)session.getAttribute("myCart");  // Step 4
      if (cart == null)
         cart = new ArrayList<Item>();
      // Add the item to the cart (note that Item is a class defined
      // in the WSDL)
      cart.add(item);
      // Save the updated cart in the HTTPSession (since we use the same
      // "myCart" name, the old cart object will be replaced)
      session.setAttribute("myCart", cart);
      // return the number of items in the stateful cart
      return cart.size();
   }
}
```

Enabling HTTP Session on the Client

The client-side code is quite simple. All you need to do is set the SESSION_MAINTAIN_PROPERTY on the request context. This tells the client to pass back the HTTP Cookies that it receives from the web service. The cookie contains a session ID that allows the server to match the web service invocation with the correct HttpSession, providing access to any saved stateful objects.

Example 31-2 Enabling HTTP Session on the Client

```
ShoppingCart proxy = new CartService().getCartPort();
((BindingProvider)proxy).getRequestContext().put(BindingProvider.SESSION_MAINTAIN
_PROPERTY, true);
// Create a new Item object with a part number of '123456' and an item
// count of 4.
Item item = new Item('123456', 4);
// After first call, we'll print '1' (the return value is the number of objects
// in the Cart object)
System.out.println(proxy.addToCart(item));
// After the second call, we'll print '2', since we've added another
// Item to the stateful, saved Cart object.
System.out.println(proxy.addToCart(item));
```

Developing Stateful Services in a Cluster Using Session State Replication

In a high-availability environment, a JAX-WS web service may be replicated across multiple server instances in a cluster. A stateful JAX-WS web service is supported in this environment through the use of the WebLogic Server HTTP Session State Replication feature. For more information, see HTTP Session State Replication in *Administering Clusters for Oracle WebLogic Server*.

There are a variety of techniques and configuration requirements for setting up a clustered environment using session state replication (for example, supported servers and load balancers, and so on). From the JAX-WS programming perspective, the only new consideration is that the objects you store in the HttpSession using the HttpSession.setAttribute method (as in Example 31-1) must be Serializable. If they are Serializable, then these stateful objects become available to the web service on all replicated web service instances in the cluster, providing both load balancing and failover capabilities for JAX-WS stateful web services.

A Note About the JAX-WS RI @Stateful Extension

The JAX-WS 2.1 Reference Implementation (RI) contains a vendor extension that supports a different model for stateful JAX-WS web services using the @Stateful annotation. It's implementation "pins" the state to a particular instance and is not designed to be scalable or fault-tolerant. This feature is not supported for WebLogic Server JAX-WS web services.

32
Testing and Monitoring Web Services

This chapter introduces you to the tools available for developing and administering WebLogic web services.
This chapter includes the following sections:

- Testing Web Services
- Monitoring Web Services and Clients
- Using Work Managers to Prioritize Web Services Work and Reduce Stuck Execute Threads

Testing Web Services

You can test basic and advanced features of your web service, such as security, quality of service (QoS), HTTP headers, and so on. You can also perform stress testing of the security features. For information about testing web services using the Web Services Test Client or Fusion Middleware Control Test Web Service page, see Testing Web Services in *Administering Web Services*.

Monitoring Web Services and Clients

You can monitor runtime information for web services and clients, such as number of invocations, errors, faults, and so on, using the WebLogic Server Administration Console or WLST.

The following naming convention is used to identify the web service or client in the monitoring pages:

```
<application_name>#<application_version>!<service_name><contextpath><url_pattern>
```

Where:

- *application_name*—Name of the application that contains the web service or client.
- *application_version*—Version of the application that contains the web service or client.
- *service_name*—Name of the web service or client.
- *context_path*—Context path defined for the web service. For more information, see Defining the Context Path of a WebLogic Web Service.
- *url_pattern*—System default or user-defined web service URL pattern. For more information, see Specifying the Transport Used to Invoke the Web Service.

Monitoring Web Services

To monitor a web service using the WebLogic Server Administration Console, click on the **Deployments** node in the left pane and in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service is packaged. Expand the

application by clicking the **+** node; the web services in the application are listed under the **Web Services** category. Click on the name of the web service and click the **Monitoring** tab.

Alternatively, click the **Deployments** node in the left pane, the **Monitoring** tab that appears in the right pane, and then the **Web Service** tab. Click on the name of the web service for which you want to view monitoring statistics.

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

> **Note:**
>
> For JAX-WS web services, the built-in `Ws-Protocol` operation displays statistics that are relevant to the underlying WS-* protocols. This information is helpful in evaluating the application performance.

Table 32-1 Monitoring Web Services

Click this tab . . .	To view . . .
Monitoring> General	General statistics about the web services, including total error and invocations counts.
Monitoring> Invocations	Invocation statistics, such as dispatch and execution times and averages.
Monitoring> WS-Policy	Policies that are attached to the web service, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Ports	Table listing the web service endpoints (ports). The table provides a summary of information for each port. Click a port name to view more details.
Monitoring> Ports > General	General statistics about the web service endpoint. The page displays information such as the web service endpoint name, its URI, and its associated web service, Enterprise application, and application module. Error and invocations counts are aggregated for all web service endpoint operations.
Monitoring> Ports > Invocations	Invocation statistics for the web service endpoint, such as success, fault, and violation counts.
Monitoring> Ports > Cluster Routing	Cluster routing statistics for the web service endpoint, such as request and response, and routing failures.
Monitoring> Ports > Make Connection	MakeConnection anonymous endpoints for a web service. For each anonymous endpoint, runtime monitoring information is displayed, such as the number of messages received, the number of messages pending, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. Click the name of an anonymous endpoint to view more details.
Monitoring> Ports > Reliable Message	Reliable messaging sequences for a web service. For each reliable messaging sequence, runtime monitoring information is displayed, such as the sequence state, the source and destination servers, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. Click the sequence ID to view more details.
Monitoring> Ports > Reliable Message > Requests	Reliable messaging requests for a web service. For each reliable messaging request, runtime monitoring information is displayed. You can customize the information that is shown in the table by clicking **Customize this table**. Click the reliable message ID to view more details.

Table 32-1 (Cont.) Monitoring Web Services

Click this tab . . .	To view . . .
Monitoring> Ports > WS-Policy	Statistics related to the policies that are attached to the web service endpoint, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Ports > Operations	List of operations for the web service endpoint. For each operation, runtime monitoring information is displayed, such as the number of times the operation has been invoked since the WebLogic Server instance started, the average time it took to invoke the web service, the average time it took to respond, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. **Note**: For JAX-WS web services, the built-in Ws-Protocol operation displays statistics that are relevant to the underlying WS-* protocols. For example, for web services reliable messaging, this operation captures message statistics for `CreateSequence` and `AckRequested` messages received or sent by the reliable messaging subsystem on behalf of the web service or client. This information is helpful in evaluating the application performance. Click the name of an operation to view more information. Click the **General** or **Invocations** tab to display general statistics or invocation statistics, respectively, for the selected operation.

Monitoring Web Service Clients

To monitor a web service client using the WebLogic Server Administration Console, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the web service client is packaged. Expand the application by clicking the **+** node and click on the application module within which the web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab.

Alternatively, click the **Deployments** node in the left pane, the **Monitoring** tab that appears in the right pane, and then the **Web Service Clients** tab. Click on the name of the web service client for which you want to view monitoring statistics.

The table provides a summary of runtime information for each web service client. Click the client name in the table to view more information.

> **Note:**
>
> For JAX-WS web services, the web services runtime creates system-defined client instances within a web service endpoint that are used to send protocol-specific messages as required by that endpoint. These client instances are named after the web service endpoint that they serve with the following suffix: `-SystemClient`. Monitoring information relevant to the system-defined client instances is provided to assist in evaluating the application.

Table 32-2 Monitoring Web Service Clients

Click this tab . . .	To view . . .
Monitoring> General	General statistics about the web service clients, including total error and invocations counts. The page displays the web service client name, its associated Enterprise application and application module, and context root. Error and invocations statistics are aggregated for all servers on which the web service is running.
Monitoring> Invocations	Invocation statistics, such as dispatch and execution times and averages.
Monitoring> WS-Policy	Policies that are attached to the web service client, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Servers	Table listing the server on which the client is currently running. Click the client name and then use the tabs in the following steps to view more information about the web service client on that server.
Monitoring> Servers > General	General statistics about the web service client. The page displays information such as the web service client port, its associated Enterprise application, and application module, context root, and so on. Error and invocations counts are aggregated for all web service client operations.
Monitoring> Servers > Invocations	Invocation statistics for the web service client, such as success, fault, and violation counts.
Monitoring> Servers > Cluster Routing	Cluster routing statistics for the web service client, such as request and response, and routing failures. For more information, see Monitoring Cluster Routing Performance.
Monitoring> Servers > Make Connection	MakeConnection anonymous endpoints for a web service client. For each anonymous endpoint, runtime monitoring information is displayed, such as the number of messages received, the number of messages pending, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. Click the name of an anonymous endpoint to view more details.
Monitoring> Servers > Reliable Message	Reliable messaging sequences for a web service client. For each reliable messaging sequence, runtime monitoring information is displayed, such as the sequence state, the source and destination servers, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. Click the name of an anonymous endpoint to view more details.
Monitoring> Servers > WS-Policy	Statistics related to the policies that are attached to the web service client, organized into the following categories: authentication, authorization, confidentiality, and integrity.
Monitoring> Servers > Operations	List of operations for the web service client. For each operation, runtime monitoring information is displayed, such as average response, execution, and dispatch times, response, invocation and error counts, and so on. You can customize the information that is shown in the table by clicking **Customize this table**. Click the name of an operation to view more information. **Click the Genera**l or **Invocations** tab to display general statistics or invocation statistics, respectively, for the selected operation.

Using Work Managers to Prioritize Web Services Work and Reduce Stuck Execute Threads

After a connection has been established between a client application and a web service, the interactions between the two are ideally smooth and quick, whereby the client makes requests and the service responds in a prompt and timely manner. Sometimes, however, a client application might take a long time to make a new request, during which the web service waits to respond, possibly for the life of the

WebLogic Server instance; this is often referred to as a *stuck execute thread*. If, at any given moment, WebLogic Server has a lot of stuck execute threads, the overall performance of the server might degrade.

If a particular web service gets into this state fairly often, you can specify how the service prioritizes the execution of its work by configuring a Work Manager and applying it to the service. For example, you can configure a *response time request class* (a specific type of Work Manager component) that specifies a response time goal for the web service.

The following shows an example of how to define a response time request class in a deployment descriptor:

```
<work-manager>
    <name>responsetime_workmanager</name>
        <response-time-request-class>
            <name>my_response_time</name>
            <goal-ms>2000</goal-ms>
        </response-time-request-class>
</work-manager>
```

You can configure the response time request class using the WebLogic Server Administration Console, as described in Work Manager: Response Time: Configuration in *Oracle WebLogic Server Administration Console Online Help*.

For more information about Work Managers in general and how to configure them for your web service, see Using Work Managers to Optimize Scheduled Work in *Administering Server Environments for Oracle WebLogic Server*.

Part V
Reference

Part V contains the following chapters:

A

Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection

This appendix summarizes the pre-packaged WS-Policy files that support reliable messaging, Make Connection, or both features together, for WebLogic web services using Java API for XML Web Services (JAX-WS).
You cannot change these pre-packaged files. If their values do not suit your needs, you must create your own WS-Policy file. For details, see:

- Creating the Web Service Reliable Messaging WS-Policy File
- Creating the Web Service Make Connection WS-Policy File (Optional)

For reference information about the reliable messaging and Make Connection policy assertions, see:

- Web Service Reliable Messaging Policy Assertion Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*
- Web Service Make Connection Policy Assertion Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*

The following table summarizes the pre-packaged WS-Policy files. This table also specifies whether the WS-Policy file can be attached at the method level; if the value in this column is no, then the WS-Policy file can be attached at the class level only.

Table A-1 Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`DefaultReliability1.2.xml`	Specifies policy assertions related to delivery assurance. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See DefaultReliability1.2.xml (WS-Policy File).	Yes
`DefaultReliability1.1.xml`	Specifies policy assertions related to quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See DefaultReliability1.1.xml (WS-Policy File).	Yes

Table A-1 (Cont.) Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`DefaultRelibility.xml`	**Deprecated.** The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf`. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. 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 (WS-Policy) [Deprecated].	Yes
`LongRunningReliability.xml`	**Deprecated.** The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 for long running processes. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. 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 (WS-Policy) [Deprecated].	Yes
`Mc1.1.xml`	Enables Make Connection support on the web service and specifies usage as optional on the web service client. The WS-Policy 1.5 protocol is used. See Mc1.1.xml (WS-Policy File).	No
`Mc.xml`	Enables Make Connection support on the web service and specifies usage as optional on the web service client. The WS-Policy 1.2 protocol is used. See Mc.xml (WS-Policy File).	No
`Reliability1.2_ExactlyOnce_WithMC1.1.xml`	Specifies policy assertions related to quality of service. It enables Make Connection support on the web service and specifies usage as optional on the web service client. See Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File).	No
`Reliability1.2_SequenceSTRSecurity`	Specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the web service and specifies usage as optional on the web service client. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File).	No

Table A-1 (Cont.) Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`Reliability1.1_SequenceSTRSecurity`	The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File).	Yes
`Reliability1.2_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File).	Yes
`Reliability1.1_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File).	Yes
`Reliability1.0_1.2.xml`	Combines 1.2 and 1.0 WS-Reliable Messaging policy assertions. The policy assertions for the 1.2 version Make Connection support on the web service and specifies usage as optional on the web service client. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Reliability1.0_1.2.xml (WS-Policy File).	No
`Reliability1.0_1.1.xml`	Combines 1.1 and 1.0 WS Reliable Messaging policy assertions. See Reliability1.0_1.1.xml (WS-Policy.xml File).	Yes

DefaultReliability1.2.xml (WS-Policy File)

The `DefaultRealiability1.2.xml` WS-Policy file specifies policy assertions related to delivery assurance. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:DeliveryAssurance>
            <wsp15:Policy>
               <wsrmp:ExactlyOnce/>
               <wsrmp:InOrder/>
            </wsp15:Policy>
          </wsrmp:DeliveryAssurance>
        </wsrmp:RMAssertion>
```

```
   </wsp15:All>
</wsp15:Policy>
```

DefaultReliability1.1.xml (WS-Policy File)

The `DefaultRealiability1.1.xml` WS-Policy file specifies policy assertions related to quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html.

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

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  >
  <wsrmp:RMAssertion
    xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702"
    >
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce />
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

DefaultReliability.xml WS-Policy File (WS-Policy) [Deprecated]

This WS-Policy file is deprecated. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at http://schemas.xmlsoap.org/ws/2005/02/rm/policy/. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The `DefaultReliability.xml` WS-Policy file 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.

```
<?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 (WS-Policy) [Deprecated]

This WS-Policy file is deprecated. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/policy/`. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The `LongRunningRelibility.xml` WS-Policy files specifies values that are similar to the `DefaultReliability.xml` WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours). See LongRunningReliability.xml WS-Policy File (WS-Policy) [Deprecated].

```
<?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>
```

Mc1.1.xml (WS-Policy File)

The `Mc1.1.xml` WS-Policy file enables Make Connection support on the web service and sets usage as optional on the web service client. In this case, the WS-Policy 1.5 protocol is used. The assertions are based on the Make Connection policy assertion defined at `http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html`.

```
<?xml version="1.0"?>
<wsp15:Policy
  xmlns:wsp15="http://www.w3.org/ns/ws-policy"
  xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702">
  <wsmc:MCSupported wsp15:Optional="true" />
</wsp15:Policy>
```

Mc.xml (WS-Policy File)

The `Mc.xml` WS-Policy file enables Make Connection support on the web service and sets usage as optional on the web service client. The assertions are based on the Make Connection policy assertion defined at `http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html`.

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702">
```

```
  <wsmc:MCSupported wsp:Optional="true" />
</wsp:Policy>
```

Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File)

The `Reliability1.2_ExactlyOnce_WithMC1.1.xml` WS-Policy file specifies policy assertions related to quality of service. It enables Make Connection support on the web service and specifies usage as optional on the web service client.

The assertions are based on the following specifications:

- Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html.
- Make Connection assertions are based on the Make Connection policy assertion defined at http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html.

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
       xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:DeliveryAssurance>
            <wsp15:Policy>
               <wsrmp:ExactlyOnce />
            </wsp15:Policy>
         </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
      <wsmc:MCSupported
        xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702"
        wsp15:Optional="true" />
   </wsp15:All>
</wsp15:Policy>
```

Reliability1.2_SequenceSTR.xml (WS-Policy File)

The `Reliability1.2_SequenceSTR.xml` file specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the web service and specifies usage as optional on the web service client.

The assertions are based on the following specifications:

- Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html.
- Make Connection assertions are based on the Make Connection policy assertion defined at http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
       xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
```

```
            <wsrmp:SequenceSTR/>
            <wsrmp:DeliveryAssurance>
               <wsp15:Policy>
                  <wsrmp:ExactlyOnce/>
               </wsp15:Policy>
            </wsrmp:DeliveryAssurance>
         </wsrmp:RMAssertion>
         <wsmc:MCSupported
          xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702"
          wsp15:Optional="true"/>
      </wsp15:All>
</wsp15:Policy>
```

Reliability1.1_SequenceSTR.xml (WS-Policy File)

The `Reliability1.1_SequenceSTR.xml` file specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceSTR/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File)

The `Reliability1.2_SequenceTransportSecurity.xml` file specifies policy assertions related to transport-level security and quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
       xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:SequenceTransportSecurity/>
            <wsrmp:DeliveryAssurance>
               <wsp15:Policy>
                  <wsrmp:ExactlyOnce/>
               </wsp15:Policy>
            </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
   </wsp15:All>
</wsp15:Policy>
```

Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)

The `Reliability1.1_SequenceTransportSecurity.xml` file specifies policy assertions related to transport-level security and quality of service. The web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceTransportSecurity/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

Reliability1.0_1.2.xml (WS-Policy File)

The `Reliability1.0_1.2.xml` WS-Policy file combines 1.2 and 1.0 WS-Reliable Messaging policy assertions.

This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. For more information about smart policy selection, see Using Multiple Policy Alternatives.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:ExactlyOne>
      <wsp15:All>
         <wsrmp:RMAssertion
          xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
            <wsrmp:DeliveryAssurance>
               <wsp15:Policy>
                  <wsrmp:ExactlyOnce/>
               </wsp15:Policy>
             </wsrmp:DeliveryAssurance>
         </wsrmp:RMAssertion>
         <wsmc:MCSupported
           xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702"
           wsp15:Optional="true"/>
      </wsp15:All>
      <wsp15:All>
         <wsrmp10:RMAssertion
          xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
            <wsrmp10:InactivityTimeout Milliseconds="600000"/>
            <wsrmp10:BaseRetransmissionInterval Milliseconds="3000"/>
            <wsrmp10:ExponentialBackoff/>
            <wsrmp10:AcknowledgementInterval Milliseconds="200"/>
         </wsrmp10:RMAssertion>
      </wsp15:All>
   </wsp15:ExactlyOne>
</wsp15:Policy>
```

Reliability1.0_1.1.xml (WS-Policy.xml File)

The `Reliability1.0_1.1.xml` WS-Policy file combines 1.1 and 1.0 WS-Reliable Messaging policy assertions. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. For more information about smart policy selection, see Using Multiple Policy Alternatives.

> **Note:**
>
> The 1.0 web service reliable messaging assertions are prefixed by `wsrmp10`.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrmp:RMAssertion
           xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
        <wsrmp:DeliveryAssurance>
          <wsp:Policy>
            <wsrmp:ExactlyOnce/>
          </wsp:Policy>
        </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
    </wsp:All>
    <wsp:All>
      <wsrmp10:RMAssertion
       xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp10:InactivityTimeout Milliseconds="600000"/>
        <wsrmp10:BaseRetransmissionInterval Milliseconds="3000"/>
        <wsrmp10:ExponentialBackoff/>
        <wsrmp10:AcknowledgementInterval Milliseconds="200"/>
      </wsrmp10:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

B

Example Client Wrapper Class for Batching Reliable Messages

This appendix provides an example client wrapper class that can be used for batching reliable messaging for WebLogic web services using Java API for XML Web Services (JAX-WS).
For more information about batching reliable messages, see Grouping Messages into Business Units of Work (Batching).

> **Note:**
>
> This client wrapper class is example code only; it is not an officially supported production class.

Example B-1 Example Client Wrapper Class for Batching Reliable Messages

```
package example.servlet;

import java.io.PrintStream;
import java.io.PrintWriter;
import java.lang.ref.WeakReference;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Date;
import java.util.SortedSet;
import java.util.Timer;
import java.util.TimerTask;

import javax.xml.datatype.DatatypeFactory;
import javax.xml.datatype.Duration;
import javax.xml.ws.BindingProvider;

import weblogic.wsee.jaxws.JAXWSProperties;
import weblogic.wsee.jaxws.spi.ClientInstance;
import weblogic.wsee.reliability.MessageRange;
import weblogic.wsee.reliability2.api.WsrmClient;
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.property.WsrmInvocationPropertyBag;
import weblogic.wsee.reliability2.tube.WsrmClientImpl;

/**
 * Example wrapper class to batch reliable requests into fixed size 'batches'
 * that can be sent using a single RM sequence. This class allows a client to
 * send requests that have no natural common grouping or
 * 'business unit of work' and not pay the costs associated with creating and
 * terminating an RM sequence for every message.
 * NOTE: This class is an *example* of how batching might be performed. To do
 *       batching correctly, you should consider error recovery and how to
 *       report sequence errors (reported via ReliabilityErrorListener) back
```

```
 *       to the clients that made the original requests.
 * <p>
 * If your web service client code knows of some natural business-oriented
 * grouping of requests (called a 'business unit of work'), it should make the
 * RM subsystem aware of this unit of work by using the
 * WsrmClient.setFinalMessage() method to demarcate the end of a unit (just
 * before sending the actual final request via an invocation on
 * the client instance). In some cases, notably when the client code represents
 * an intermediary in the processing of messages, the client code may not be
 * aware of any natural unit of work. In the past, if no business unit of work
 * could be determined, clients often just created the client instance, sent the
 * single current message they had, and then allowed the sequence to terminate.
 * This is functionally workable, but very inefficient. These clients pay the
 * cost of an RM sequence handshake and termination for every message they send.
 * The BatchingRmClientWrapper class can be used to introduce an artificial
 * unit of work (a batch) when no natural business unit of work is available.
 * <p>
 * Each instance of BatchingRmClientWrapper is a wrapper instance around a
 * client instance (port or Dispatch instance). This wrapper can be used to
 * obtain a Proxy instance that can be used in place of the original client
 * instance. This allows this class to perform batching operations completely
 * invisibly from the perspective of the client code.
 * <p>
 * This class is used for batching reliable requests into
 * batches of a given max size that will survive for a given maximum
 * duration. If a batch fills up or times out, it is ended, causing the
 * RM sequence it represents to be ended/terminated. The timeout ensures that
 * if the flow of incoming requests stops the batch/sequence will still
 * end in a timely manner.
 */
public class BatchingRmClientWrapper<T>
  implements InvocationHandler {

  private Class<T> _clazz;
  private int _batchSize;
  private long _maxBatchLifetimeMillis;
  private T _clientInstance;
  private PrintWriter _out;
  private WsrmClient _rmClient;
  private int _numInCurrentBatch;
  private int _batchNum;
  private Timer _timer;
  private boolean _closed;
  private boolean _proxyCreated;

  /**
   * Create a wrapper instance for batching reliable requests into
   * batches of the given max size that will survive for the given maximum
   * duration. If a batch fills up or times out, it is ended, causing the
   * RM sequence it represents to be ended/terminated.
   * @param clientInstance The client instance that acts as the source object
   *        for the batching proxy created by the createProxy() method. This
   *        is the port/Dispatch instance returned from the call to
   *        getPort/createDispatch. The BatchingRmClientWrapper will take over
   *        responsibility for managing the interaction with and cleanup of
   *        the client instance via the proxy created from createProxy.
   * @param clazz of the proxy instance we'll be creating in createProxy.
   *        This should be the class of the port/Dispatch instance you would
   *        use to invoke operations on the service. BatchingRmClientWrapper will
   *        create (via createProxy) a proxy of the given type that can be
   *        used in place of the original client instance.
```

```
 * @param batchSize Max number of requests to put into a batch. If the
 *        max number of requests are sent for a given batch, that batch is
 *        ended (ending/terminating the sequence it represents) and a new
 *        batch is started.
 * @param maxBatchLifetime A duration value (in the lexical form supported
 *        by java.util.Duration, e.g. PT30S for 30 seconds) representing
 *        the maximum time a batch should exist. If the batch exists longer
 *        than this time, it is ended and a new batch is begun.
 * @param out A print stream that can be used to print diagnostic and
 *        status messages.
 */
public BatchingRmClientWrapper(T clientInstance, Class<T> clazz,
                               int batchSize, String maxBatchLifetime,
                               PrintStream out) {
  _clazz = clazz;
  _batchSize = batchSize;
  try {
    if (maxBatchLifetime == null) {
      maxBatchLifetime = "PT5M";
    }
    Duration duration =
      DatatypeFactory.newInstance().newDuration(maxBatchLifetime);
    _maxBatchLifetimeMillis = duration.getTimeInMillis(new Date());
  } catch (Exception e) {
    throw new RuntimeException(e.toString(), e);
  }
  _clientInstance = clientInstance;
  _out = new PrintWriter(out, true);
  _rmClient = WsrmClientFactory.getWsrmClientFromPort(_clientInstance);
  _closed = false;
  _proxyCreated = false;
  _timer = new Timer(true);
  _timer.schedule(new TimerTask() {
    @Override
    public void run() {
      terminateOrEndBatch();
    }
  }, _maxBatchLifetimeMillis);
}

/**
 * Creates the dynamic proxy that should be used in place of the client
 * instance used to create this BatchingRmClientWrapper instance. This method
 * should be called only once per BatchingRmClientWrapper.
 */
public T createProxy() {
  if (_proxyCreated) {
    throw new IllegalStateException("Already created the proxy for this BatchingRmClientWrapper
      instance which wraps the client instance: " + _clientInstance);
  }
  _proxyCreated = true;
  return (T) Proxy.newProxyInstance(getClass().getClassLoader(),
                                    new Class[] {
                                      _clazz,
                                      BindingProvider.class,
                                      java.io.Closeable.class
                                    }, this);
}

private void terminateOrEndBatch() {
  synchronized(_clientInstance) {
```

```
    if (_rmClient.getSequenceId() != null) {
      if (terminateBatchAllowed()) {
        _out.println("Terminating batch " + _batchNum + " sequence (" + _
              rmClient.getSequenceId() + ") for " + _clientInstance);
        try {
          _rmClient.terminateSequence();
        } catch (Exception e) {
          e.printStackTrace(_out);
        }
      } else {
        _out.println("Batch " + _batchNum + " sequence (" + _rmClient.getSequenceId() + ")
          for " + _clientInstance + " timed out but has outstanding requests to send and
          cannot be terminated now");
      }
    }
    endBatch();
  }
}

/**
 * Check to see if we have acks for all requests sent. If so,
 * we can terminate.
 */
private boolean terminateBatchAllowed() {
  try {
    synchronized(_clientInstance) {
      if (_rmClient.getSequenceId() != null) {
        long maxMsgNum = _rmClient.getMostRecentMessageNumber();
        if (maxMsgNum < 1) {
          // No messages sent, go ahead and terminate.
          return true;
        }
        SortedSet<MessageRange> ranges = _rmClient.getAckRanges();
        long maxAck = -1;
        boolean hasGaps = false;
        long lastRangeUpper = -1;
        for (MessageRange range: ranges) {
          if (lastRangeUpper > 0) {
            if (range.lowerBounds != lastRangeUpper + 1) {
              hasGaps = true;
            }
          } else {
            lastRangeUpper = range.upperBounds;
          }
          maxAck = range.upperBounds;
        }
        return !(hasGaps || maxAck < maxMsgNum);
      }
    }
  } catch (Exception e) {
    e.printStackTrace(_out);
  }
  return true;
}

private void endBatch() {
  synchronized(_clientInstance) {
    if (_numInCurrentBatch > 0) {
      _out.println("Ending batch " + _batchNum + " sequence (" + _rmClient.getSequenceId() + ")
         for " + _clientInstance + "...");
    }
```

```
      /**
       * _rmClient.reset() resets a WsrmClient instance (and the client instance it represents)
       * so it can track a new WS-RM sequence for the next invoke on the client
       * instance. This method effectively *disconnects* the RM sequence from the
       * client instance and lets them continue/complete separately.
       */
      _rmClient.reset();
      _numInCurrentBatch = 0;
      if (!_closed) {
        _timer.schedule(new TimerTask() {
          @Override
          public void run() {
            terminateOrEndBatch();
          }
        }, _maxBatchLifetimeMillis);
      }
    }
  }

  public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    boolean operationInvoke = method.getDeclaringClass() == _clazz;
    boolean closeableInvoke = method.getDeclaringClass() ==
                              java.io.Closeable.class;
    boolean endOfBatch = false;
    if (operationInvoke) {
      synchronized(_clientInstance) {
        // Check our batch size
        if (_numInCurrentBatch == 0) {
          _batchNum++;
        }
        endOfBatch = _numInCurrentBatch >= _batchSize - 1;
        if (endOfBatch) {
          _rmClient.setFinalMessage();
        }
        _out.println("Making " + (endOfBatch ? "final " : "") + "invoke " +
           (_numInCurrentBatch+1) + " of batch " + _batchNum + " sequence (" + _
           rmClient.getSequenceId() + ") with operation: " + method.getName());
      }
    } else if (closeableInvoke && method.getName().equals("close")) {
      synchronized(_clientInstance) {
        // Make sure we don't try to schedule the timer anymore
        _closed = true;
        _timer.cancel();
      }
    }
    Object ret = method.invoke(_clientInstance, args);
    if (operationInvoke) {
      synchronized(_clientInstance) {
        _numInCurrentBatch++;
        if (endOfBatch) {
          endBatch();
        }
      }
    }
    return ret;
  }
}
```

C

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

This appendix describes how to migrate Java API for XML-based RPC (JAX-RPC) web services and clients to Java API for XML-based Web Services (JAX-WS).
When migrating your JAX-RPC web services, to preserve the original WSDL file, use the top-down approach, starting from a WSDL file, to generate the JAX-WS web service. For more information, see Developing WebLogic Web Services Starting From a WSDL File: Main Steps.

> **Note:**
>
> In some cases, a JAX-RPC feature may not be supported currently by JAX-WS. In this case, the application cannot be migrated unless it is re-architected.

The following table summarizes the topics that are covered.

Table C-1 Tips for Migrating JAX-RPC Web Services and Clients to JAX-WS

Topic	Description
Setting the Final Context Root of a WebLogic Web Service	Describes the methods that can be used to set the final context root of a WebLogic web service. The use of @WLXXXTransport JWS annotations is not supported for JAX-WS; these annotations are supported by JAX-RPC only.
Using WebLogic-specific Annotations	Describes the WebLogic-specific annotations that are supported by JAX-WS.
Generating a WSDL File	Describes how to generate a WSDL file when you are generating a JAX-WS web service using the jwsc Ant task.
Using JAXB Custom Types	Describes the use of Java Architecture for XML Binding (JAXB) for managing all of the data binding tasks.
Using EJB 3.0	Describes changes in EJB 3.0 from EJB 2.1. JAX-WS supports EJB 3.0. JAX-RPC supports EJB 2.1 only.
Migrating from RPC Style SOAP Binding	Provides guidelines for setting the SOAP binding. RPC style is supported, but not recommended for JAX-WS.
Updating SOAP Message Handlers	Explains how you must re-write your JAX-RPC SOAP message handlers when migrating to JAX-WS.
Invoking JAX-WS Clients	Explains how you must re-write your JAX-RPC client to invoke JAX-WS clients.

Setting the Final Context Root of a WebLogic Web Service

You can set the final context root of a WebLogic web service using a variety of methods, as described in Specifying the Transport Used to Invoke the Web Service in *Developing JAX-RPC Web Services for Oracle WebLogic Server*.

As described in this section, when defining a JAX-RPC web service, you can use the @WLXXXTransport JWS annotations to specify the context root. For JAX-WS web services, the @WLXXXTransport JWS annotations are not valid. If used in the JAX-RPC web service, the JWS file needs to be updated to remove the annotations in favor of one of the other methods.

Using WebLogic-specific Annotations

JAX-WS supports the following WebLogic-specific annotations:

- @Policy
- @Policies
- @SecurityPolicy
- @SecurityPolicies
- @WssConfiguration

All other WebLogic-specific annotations must be removed from your JAX-RPC applications when migrating to JAX-WS. For more information, see WebLogic-specific Annotations in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Generating a WSDL File

When you run the jwsc file on a JAX-RPC web service, a WSDL file is generated in the specified output directory. For JAX-WS web services, the WSDL file is generated when the service endpoint is deployed. In order to generate a WSDL file in the output directory, you must specify the wsdlOnly attribute of the <jws> child element of the jwsc Ant task. For more information, see jwsc in the *WebLogic Web Services Reference for Oracle WebLogic Server*.

Using JAXB Custom Types

JAX-WS uses Java Architecture for XML Binding (JAXB), described at `http://jcp.org/en/jsr/detail?id=222`, to manage all of the data binding tasks. If your application supports custom types using XMLBeans or Tylar, you will need to modify them to use JAXB. For more information about using JAXB, see Using JAXB Data Binding.

Using EJB 3.0

JAX-WS supports EJB 3.0. JAX-RPC supports EJB 2.1 only.

EJB 3.0 introduced metadata annotations that enable you to automatically generate, rather than manually create, the EJB Remote and Home interface classes and deployment descriptor files needed when implementing an EJB.

For more information about EJB 3.0 bean class requirements and changes from 2.x, see Programming the Bean File: Requirements and Changes from 2.X in *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

Migrating from RPC Style SOAP Binding

Use of the SOAPBinding.Style.RPC style, although supported, is not recommended with JAX-WS. It is recommended that you change the style to SOAPBinding.Style.DOCUMENT.

Updating SOAP Message Handlers

Although the SOAP APIs are similar, JAX-RPC SOAP handlers will need to be modified to run with JAX-WS. For more information, see Sending and Receiving SOAP Headers.

Invoking JAX-WS Clients

JAX-RPC clients will need to be re-written as the JAX-RPC and JAX-WS client APIs are completely different. For more information about writing JAX-WS clients, see Developing Web Service Clients in *Developing JAX-WS Web Services for Oracle WebLogic Server*.