Oracle Fusion Middleware

Oracle Fusion Middleware Developing Standalone Clients for Oracle WebLogic Server

15c (15.1.1.0.0)
G31335-01
October 2025

Oracle Fusion Middleware Oracle Fusion Middleware Developing Standalone Clients for Oracle WebLogic Server, 15c (15.1.1.0.0)

G31335-01

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation i
Conventions ii

1 Overview of Standalone Clients

Distributing Client JAR Files 1
WebLogic T3 Clients 1
- WebLogic Thin T3 Client 1
- WebLogic Install Client 2

Java IIOP 2
CORBA Clients 2
JMX Clients 3
JMS Clients 3
Web Services Clients 3
WebLogic Tuxedo Connector Clients 3
Clients and Features 4

2 Developing a WebLogic Thin T3 Client

Understanding the WebLogic Thin T3 Client 1
- WebLogic Thin T3 Features 1
- Limitations and Considerations 1
- Interoperability 2
  - Prior WebLogic Server Releases 2
  - Foreign Application Servers 2
- Security 2
- Connection Considerations 2

Developing a Basic WebLogic Thin T3 Client 2
Foreign Server Applications 3

Deployment Considerations 4

3 Reliably Sending Messages Using the JMS SAF Client

Overview of Using Store-and-Forward with JMS Clients 1
Configuring a JMS Client To Use Client-Side SAF 1
Generating a JMS SAF Client Configuration File 1
How the JMS SAF Client Configuration File Works 2
Steps to Generate a JMS SAF Client Configuration File from a JMS Module 2
ClientSAFGenerate Utility Syntax 4
Valid SAF Elements for JMS SAF Client Configurations 4
Default Store Options for JMS SAF Clients 6
Encrypting Passwords for Remote JMS SAF Contexts 7
Steps to Generate Encrypted Passwords 7
ClientSAFEncrypt Utility Syntax 8
Installing the JMS SAF Client JAR Files on Client Machines 8
Modify Your JMS Client Applications To Use the JMS SAF Client's Initial JNDI Provider 8
Required JNDI Context Factory for JMS SAF Clients 9
Optional JNDI Properties for JMS SAF Clients 9
JMS SAF Client Management Tools 9
The JMS SAF Client Initialization API 9
Client-Side Store Administration Utility 10
JMS Programming Considerations with JMS SAF Clients 10
How the JMSReplyTo Field Is Handled In JMS SAF Client Messages 10
No Mixing of JMS SAF Client Contexts and Server Contexts 10
Using Transacted Sessions With JMS SAF Clients 10
JMS SAF Client Interoperability Guidelines 10
Java Runtime 10
WebLogic Server Versions 11
JMS C API 11
Tuning JMS SAF Clients 11
Limitations of Using the JMS SAF Client 11
Behavior Change in JMS SAF Client Message Storage 12
The Upgrade Process, Tools, and System Properties 12
JMS SAF Client Discovery Tool 12
JMS SAF Client Migration Properties 14

4 Developing a CORBA/IDL Client

Guidelines for Developing a CORBA/IDL Client 1
Working with CORBA/IDL Clients 1
IDL Client (Corba object) relationships 2

Java to IDL Mapping 2
WebLogic RMI over IIOP object relationships 3
Objects-by-Value 3
Procedure for Developing a CORBA/IDL Client 4

5 Developing Clients for CORBA Objects

Enhancements and Limitations of CORBA Object Types 1
Making Outbound CORBA Calls: Main Steps 1
Using the WebLogic ORB Hosted in JNDI 1
ORB from JNDI 2
Direct ORB creation 2
Using JNDI 2
Supporting Inbound CORBA Calls 3

6 Developing a WebLogic C++ Client for a Tuxedo ORB

WebLogic C++ Client Advantages and Limitations 1
How the WebLogic C++ Client Works 1
Developing WebLogic C++ Clients 2

7 Using Java EE Client Application Modules

Extracting a Client Application 1
Running a Client Application 2

8 Developing Security-Aware Clients

Developing Clients that use JAAS 1
Developing Clients that use JNDI Authentication 1
Developing Clients that use SSL 1
Thin Client Restrictions for JAAS and SSL 3
Install Client Restrictions for SSL 4
Security Code Examples 4

9 Using EJBs with RMI-IIOP Clients

Accessing EJBs with a Java Client 1
Accessing EJBs with a CORBA/IDL Client 1
Example IDL Generation 1

A Client Application Deployment Descriptor Elements

Overview of Client Application Deployment Descriptor Elements A-1
application-client.xml Deployment Descriptor Elements A-1
application-client A-1
weblogic-appclient.xml Descriptor Elements A-3
application-client A-3

B Accessing WebLogic Server MBeans from JConsole Using WebLogic Install Client JARs

Using JConsole with WebLogic Install Client JARs to Access WebLogic Server MBeans B-1

Preface

This document is a resource for developers who want to create standalone client applications that interoperate with WebLogic Server.

Audience

This document is relevant to the design and development phases of a software project. The document also includes solutions to application problems that are discovered during test and pre-production phases of a project.

It is assumed that the reader is familiar with Jakarta Platform, Enterprise Edition concepts. This document emphasizes the value-added features provided by WebLogic Server and key information about how to use WebLogic Server features and facilities when developing standalone clients.

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 customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

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 Documentation

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

- *Developing RMI Applications for Oracle WebLogic Server* is a guide to using Remote Method Invocation (RMI) and Internet Interop-Orb-Protocol (IIOP) features.

- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- *Tuning Performance of Oracle WebLogic Server* contains information on monitoring and improving the performance of WebLogic Server applications.

Also, see the following sections:

Samples and Tutorials

Oracle provides a variety of code examples and tutorials that show WebLogic Server configuration and API use, and provide practical instructions on how to perform key development tasks. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Oracle recommends that you run some or all examples before developing your own applications.

New and Changed WebLogic Server Features

For a comprehensive listing of the new 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.

1

Overview of Standalone Clients

A standalone client is a client that has a runtime environment independent of WebLogic Server. (Managed clients, such as web services, rely on a server-side container to provide the runtime necessary to access a server.) Standalone clients that access WebLogic Server applications range from simple command-line utilities that use standard I/O to highly interactive GUI applications built using the Java Swing/AWT classes. Learn about types of clients, client features, and how clients are distributed.

> **Note**
>
> The WebLogic Full client and thin IIOP clients are removed in WebLogic Server 14.1.1.0.0. Oracle recommends using the thin T3 client or the install client instead.

Distributing Client JAR Files

Learn about license requirements when using client JARs and other resources provided in Oracle WebLogic Server for creating standalone clients.

See Standalone WebLogic Clients.

WebLogic T3 Clients

The WebLogic T3 clients are Java RMI clients that use Oracle T3 protocol to communicate with WebLogic Server. T3 clients outperform other client types and are the most recommended type of client.

WebLogic Thin T3 Client

The WebLogic thin T3 Java client provides a light-weight alternative to the WebLogic Install client. This client provides the same performance as an install client, but uses a much smaller JAR file. The thin T3 client supports most of the use cases for which the install client can be used.

> **Note**
>
> The WebLogic Full client and thin IIOP client are removed in WebLogic Server 14.1.1.0.0. Oracle recommends using the thin T3 client or the Install client instead.

There are two versions of the thin T3 client, one that supports Java EE and one that supports Jakarta EE. The Java EE thin T3 client can be used in standalone applications, and is also designed for applications running on foreign (non-WebLogic) servers. One common use case is integration with WebLogic JMS destinations. The Jakarta version of the thin T3 client is used in the WebLogic Server with Helidon 3.x JMS integration.

- Developing a WebLogic Thin T3 Client
- Using WebLogic RMI with T3 Protocol in *Developing RMI Applications for Oracle WebLogic Server*

WebLogic Install Client

The WebLogic Install client is available in a full WebLogic Server installation. It uses the `weblogic.jar` file located at `WL_HOME/server/lib` and provides client-side support for all WebLogic Server-specific value-added features. It is the only client that supports server-side operations, such as:

- Operations necessary for development purposes, such as the ejbc compiler.
- Administrative operations, such as deployment.
- WLST and client-side JSR 88 applications that invoke server-side operations.

Java IIOP

IIOP can be a transport protocol for distributed applications with interfaces written in Java RMI. When there is an option, Oracle recommends using T3 clients instead of IIOP clients.

An IIOP protocol Java client works with WebLogic Server 14.1.2.0.0 and later, only if all Java (remote and local) is at JDK17 or all Java is at JDK21, and also when:

- The Java client is another WebLogic Server at any version for which overall interoperability is supported.
- The Java client is a WebLogic Install client (`weblogic.jar`), in a WebLogic Server installation, at any version for which overall interoperability is supported.
- The Java client is a WebLogic Full client (`wlfullclient.jar`), as long as the client is an earlier version of WebLogic Server than WLS 14.1.1.0.0. Note that `wlfullclient.jar` is removed in WLS 14.1.1.0.0.
- The Java client with thin IIOP WebLogic client JARs (`wlclient.jar`), as long as the client is an earlier version of WebLogic Server than WLS 14.1.1.0.0. Note that `wlclient.jar` is removed in WLS 14.1.1.0.0.

For Java IIOP limitations, see Table 1-1.

For more information about interoperability, see Protocol Compatibility in *Understanding Oracle WebLogic Server*.

See also Using RMI over IIOP in *Developing RMI Applications for Oracle WebLogic Server* .

CORBA Clients

If you are not working in a Java-only environment, you can use IIOP to connect your Java programs with Common Object Request Broker Architecture (CORBA) clients and run CORBA objects. IIOP can be a transport protocol for distributed applications with interfaces written in Interface Definition Language (IDL) or Java RMI. However, the two models are distinctly different approaches to creating an interoperable environment between heterogeneous systems.

When you program, you must decide to use either IDL or RMI interfaces; you cannot mix them. WebLogic Server supports the following CORBA client models:

- Developing a CORBA/IDL Client

- Developing Clients for CORBA Objects
- Developing a WebLogic C++ Client for a Tuxedo ORB

JMX Clients

You can use a JMX client to access WebLogic Server MBeans.

See Accessing WebLogic Server MBeans With JMX in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

JMS Clients

WebLogic Server provides a number of JMS clients that provide Java EE and WebLogic JMS functionality.

- WebLogic Thin T3 client, see Developing a WebLogic Thin T3 Client.
- WebLogic Install client, see WebLogic Install Client.
- JMS SAF client, see Reliably Sending Messages Using the JMS SAF Client.
- JMS C client, see WebLogic JMS C API in *Developing JMS Applications for Oracle WebLogic Server*.
- JMS .NET client, see *Developing JMS .NET Client Applications for Oracle WebLogic Server*.
- WebLogic AQ JMS client, see Standalone WebLogic AQ JMS Clients in *Administering JMS Resources for Oracle WebLogic Server*. The WebLogic AQ JMS client obtains destination information using WebLogic Server JNDI and provides direct access to Oracle database AQ JMS destinations using an embedded driver. It does not provide access to WebLogic Server JMS destinations.

> **Tip**
>
> Oracle WebLogic JMS clients require using the T3 protocol in all cases.

Web Services Clients

A standalone web services client uses WebLogic client classes to invoke a web service hosted on WebLogic Server or on other application servers.

There are two versions of the web services client, one that supports Java EE and one that supports Jakarta EE. The Jakarta version of the web services client is used in the WebLogic Server with Helidon 3.x web services integration. These clients are located at `WL_HOME/modules/clients`.

WebLogic Tuxedo Connector Clients

WebLogic Tuxedo Connector provides interoperability between WebLogic Server applications and Tuxedo services.

See:

- Developing Oracle WebLogic Tuxedo Connector Client EJBs in the *Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*
- How to Develop RMI/IIOP Applications for the Oracle WebLogic Tuxedo Connector in the *Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*
- How to Develop Oracle WebLogic Tuxedo Connector Client Beans using the CORBA Java API in the *Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*

Clients and Features

Learn about the types of clients and features supported in a WebLogic Server environment.

The following table lists the types of clients supported in a WebLogic Server environment, their characteristics, features, and limitations.

> **Note**
>
> - Oracle does not support combining clients to create extended feature sets. Select a client that best fits your environment and use only the client classes specified for that client type.
> - The following Java clients in WebLogic Server versions 12.2.1.4, 14.1.1, and 14.1.2, are based on `javax.*`. In WebLogic Server version 15.1.1.0.0, the same clients are based on `jakarta.*`.

Table 1-1 WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements/ Bundled Resources	Key Features
WL Thin T3 Client	RMI	Java	T3	`wlthint3client.jar` (Java EE support) `wlthint3client.jakarta.jar` (Jakarta EE support)	• Small Footprint. • Oracle WebLogic Server T3/T3S protocol for Remote Method Invocation (RMI). • Supports WebLogic Server clustering. • Supports JSSE SSL, except with HTTP tunneling. • Faster and more scalable than IIOP clients. • Most WebLogic client JMS features, including the WebLogic Store-and-Forward (SAF) Service using the `wlsaft3client.jar`. • Supports most Jakarta EE 9.1 features. • See Developing a WebLogic Thin T3 Client.

Table 1-1 (Cont.) WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements/ Bundled Resources	Key Features
Install Client	RMI	Java	IIOP	`weblogic.jar`	• Supports JSSE SSL. • Supports most of the Jakarta EE 9.1 features, but does not support WebLogic client JMS. • Operations necessary for development purposes, such as the ejbc compiler. • Supports administrative operations, such as deployment. • Supports WLST and client-side JSR 88 applications that invoke server-side operations. • See WebLogic Install Client.
Install Client	RMI	Java	T3	`weblogic.jar`	• Supports Oracle WebLogic Server T3/T3S protocol for Remote Method Invocation (RMI), including HTTP Tunneling of T3/T3S. • Supports WebLogic Server clustering. • Supports JSSE SSL. • Faster and more scalable than IIOP clients. • All WebLogic client JMS features, including the WebLogic JMS client Store-and-Forward (SAF) Service. • Supports most of the Jakarta 9.1 features. • Supports operations necessary for development purposes, such as the ejbc compiler. • Supports administrative operations, such as deployment. • Supports WLST and client-side JSR 88 applications that invoke server-side operations. • See WebLogic Install Client.
CORBA/IDL	CORBA	Languages that OMG IDL maps to, such as C++, C, Smalltalk, COBOL	IIOP	No WebLogic classes	• Uses CORBA 2.3 ORB. • Does not support WebLogic Server-specific features. • Does not support Java. • See Developing a CORBA/IDL Client.
JMS SAF Client (Introduced in WebLogic Server 9.2)	RMI	Java	T3	`wlsaft3client.jar` and `wlthint3client.jar`	• Locally stores messages on the client and forwards them to server-side JMS destinations when the client is connected. • Supports SSL. • See Reliably Sending Messages Using the JMS SAF Client.
JMS C Client (Introduced in WebLogic Server 9.0)	JNI	C	T3	Any WebLogic JMS capable Java client, such as `wlthint3client.jar`	• C client applications that can access WebLogic JMS applications and resources. • Supports SSL. • See WebLogic JMS C API

Table 1-1 (Cont.) WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements/ Bundled Resources	Key Features
JMS .NET Client (Introduced in WebLogic Server 10.3)	T3	.NET	T3	`WebLogic.Messaging.dll` dynamic library	• Microsoft .NET client applications, written in C#, that can access WebLogic JMS applications and resources. • See *Developing JMS .NET Client Applications for Oracle WebLogic Server* .
WebLogic AQ JMS Client (Introduced in WebLogic Server 10.3.1)	JNDI	Java	IIOP/T3 +	`aqapi.jar`, `o6.jar`, and `orai18n.jar` are required, plus either the `weblogic.jar` (Install client) or the `wlthint3client.jar`.	See Standalone WebLogic AQ JMS Clients in *Administering JMS Resources for Oracle WebLogic Server* .
Web Services	SOAP	Java	HTTP/S	`com.oracle.webservices.wls.jaxws-wlswss-client.jar` (Java EE support) `com.oracle.webservices.wls.jaxws-wlswss-client.jakarta.jar` (Jakarta EE support)	See Invoking a Web Service from a Standalone Java SE Client in *Developing JAX-WS Web Services for Oracle WebLogic Server* .
C++ Client	CORBA	C++	IIOP	Tuxedo libraries	• Interoperability between WebLogic Server applications and Tuxedo clients/services. • Supports SSL. • Uses CORBA 2.3 ORB. • See Developing a WebLogic C++ Client for a Tuxedo ORB.
Tuxedo Server and Native CORBA client	CORBA or RMI	C++	Tuxedo-General-Inter-Orb-Protocol (TGIOP)	Tuxedo libraries	• Interoperability between WebLogic Server applications and Tuxedo clients/services. • Supports SSL and transactions. • Uses CORBA 2.3 ORB. • See Developing Clients for CORBA Objects.
RESTful Webservices Client	JAX-RS	Java	HTTP/S	`jersey-client.jar`	Supports JAX-RS client API.

2

Developing a WebLogic Thin T3 Client

Learn how to develop and use WebLogic thin T3 clients.

Understanding the WebLogic Thin T3 Client

The WebLogic thin T3 client (`wlthint3client.jar` and `wlthint3client.jakarta.jar`) is a light-weight alternative to the full install client (`weblogic.jar`). The thin T3 client has a minimal footprint while providing access to a rich set of APIs that are appropriate for client use. As its name implies, the thin T3 client requires using the WebLogic T3 protocol.

There are two versions of the thin T3 client, one that supports Java EE and one that supports Jakarta EE. The Java EE thin T3 client can be used in standalone applications, and is also designed for applications running on foreign (non-WebLogic) servers. One common use case is integration with WebLogic JMS destinations. The Jakarta version of the thin T3 client is used in the WebLogic Server with Helidon 3.x JMS integration.

The thin T3 client is the recommended option for most remote client use cases. There are some limitations in the thin T3 client as outlined below.

WebLogic Thin T3 Features

This release supports:

- Oracle WebLogic Server T3/T3S protocol for Remote Method Invocation (RMI), including RMI over HTTP (HTTP tunneling) and RMI over HTTPS (HTTP tunneling over SSL). For more information on WebLogic T3 communication, see Using WebLogic RMI with T3 Protocol in *Developing RMI Applications for Oracle WebLogic Server*.
- Access to JMS, JMX, JNDI, and EJB resources available in WebLogic Server.
- The WebLogic Store-and-Forward (SAF) Service when used in combination with the `wlsaft3client.jar`. See Reliably Sending Messages Using the JMS SAF Client.
- Transaction initiation and termination (rollback or commit) using JTA.
- WebLogic client JMS features, including Unit-of-Order, Unit-of-Work, message compression, XML messages, JMS automatic client reconnect, and Destination Availability Helper APIs.
- Client-side clustering allowing a client application to participate in failover and load balancing of a WebLogic Server instance. See Clustered RMI Applications in *Developing RMI Applications for Oracle WebLogic Server*.
- JAAS authentication and JSSE SSL. See Security.
- Network class loading. By default, the network class loading for the thin T3 client is disabled. Use the following system property to enable network classloading:

  ```
  -Dweblogic.rmi.networkclassloadingenabled=true
  ```

Limitations and Considerations

This release does not support:

- MBean-based utilities (such as JMS Helper, JMS Module Helper), and JMS multicast. You can use JMX calls as an alternative to "mbean-based helpers."
- JDBC resources, including WebLogic JDBC extensions.
- Running a WebLogic RMI server in the client.

The thin T3 client uses JDK classes to connect to the host, including when connecting to dual-stacked machines. If multiple addresses available on the host, the connection may attempt to go to the wrong address and fail if the host is not properly configured.

Interoperability

This release of the WebLogic thin T3 client has the following interoperability support:

Prior WebLogic Server Releases

For information on WebLogic Thin T3 client support for communicating with previous WebLogic Server releases, see Protocol Compatibility in *Understanding Oracle WebLogic Server*.

Foreign Application Servers

The WebLogic Thin T3 client JAR is supported on the following application servers:

- GlassFish
- IBM WebSphere Application Server
- Red Hat JBoss Application Server

Security

For general information on client security see:

- The Java Secure Socket Extension (JSSE) in *Understanding Security for Oracle WebLogic Server*.
- Java Authentication and Authorization Services (JAAS) in *Understanding Security for Oracle WebLogic Server*.
- Using SSL Authentication in Java Clients in *Developing Applications with the WebLogic Security Service*.
- Using JAAS Authentication in Java Clients in *Developing Applications with the WebLogic Security Service*.

Connection Considerations

The WebLogic Thin T3 client uses JDK classes to connect to the host. If your host has multiple addresses (Dual-Stack) available, your client may connect to the wrong IP address if the host is not configured properly.

Developing a Basic WebLogic Thin T3 Client

Learn how to create a basic WebLogic thin T3 client using a WebLogic initial context.

Use the following steps to create a basic WebLogic Thin T3 client:

1. Obtain a reference to the remote object.

a. Get the initial context of the server that hosts the service using a T3 URL in the form of `t3://ip address:port` or `t3s://ip address:port`.

b. Obtain an instance of the service object by performing a lookup using the initial context. This instance can then be used just like a local object reference.

2. Call the remote objects methods.

3. Place the `wlthint3client.jar` in your client classpath. It is located in the *`WL_HOME`*`\server\lib` directory of your WebLogic Server installation.

> **Note**
>
> Oracle does not support combining clients to create extended feature sets. Never add the `wlfullclient.jar`, `wlthint3client.jar`, or `wlclient.jar` to a WebLogic Server classpath or a classpath that references the `weblogic.jar` file in a full WebLogic install. The behavior is undefined. WebLogic Server applications already have full access to WebLogic client functionality.

Sample code for a basic WebLogic Thin T3 client is provided in Example 2-1.

Example 2-1 Creating and Using a WebLogic Initial Context

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
  "weblogic.jndi.WLInitialContextFactory");
env.put("java.naming.provider.url","t3://host:7001");
env.put("java.naming.security.principal","user");
env.put("java.naming.security.credentials","password");
Context ctx = new InitialContext(env);
try {
  Object homeObject =
    context.lookup("EmployeeBean");
//use the EmployeeBean
}
catch (NamingException e) {
// a failure occurred
}
finally {
  try {ctx.close();}
  catch (Exception e) {
// a failure occurred
}
}
```

Foreign Server Applications

A foreign server hosted application can use the `wlthint3client.jar` to act as a remote client to a WebLogic Server instance. To provide access to remote services such as JMS, servlets, EJBs, and startup classes, deploy any necessary application code along with the `wlthint3client.jar`. If your application server supports Java EE (use of `javax` packages), then use `wlthint3client.jar`. If your application server supports Jakarta EE (use of `jakarta` packages), then use `wlthint3client.jakarta.jar`.

The following steps provide a guideline to connect to and access WebLogic Server resources from a foreign application server using JNDI:

1. Include the `wlthint3client.jar` or `wlthint3client.jakarta.jar` on the class path of your client.
2. In your client application, create a WebLogic initial context and use the context to look up and use a resource. See Example 2-1 for more details.
3. It may be necessary to explicitly set the initial context factory as a system property in the client code, to the following value:

   ```
   env.put("java.naming.factory.initial", "weblogic.jndi.WLInitialContextFactory"
   );
   ```

4. Deploy any necessary application code along with the `wlthint3client.jar` or `wlthint3client.jakarta.jar` file to your application server using standard Java EE or Jakarta EE methods, such as embedding the `wlthint3client.jar` or `wlthint3client.jakarta.jar` file in a servlet or using a shared library. See Deployment Considerations.
5. Start or deploy the client.

The following section outlines specific items to consider when interoperating with a foreign servers.

Deployment Considerations

> **Note**
>
> Never deploy the thin T3 client as part of a WebLogic Server application or library. A WebLogic Server instance already has the necessary T3 client classes.

You can deploy the `wlthint3client.jar` using standard Java EE methods or deploy the `wlthint3client.jakarta.jar` using standard Jakarta EE methods. However, when determining what deployment method to use, you must account for client footprint, class loading, performance, and tolerance of the risk for code incompatibility. For example:

- If you embed the `wlthint3client.jar` or `wlthint3client.jakarta.jar` in your application, such as a servlet, the application footprint is increased by the size of the `wlthint3client.jar` or `wlthint3client.jakarta.jar` file, but the risk of code incompatibility is limited to the scope of your application.
- If you deploy the `wlthint3client.jar` or `wlthint3client.jakarta.jar`file to your `lib` directory, the application footprint is not affected but the risk of code incompatibility can include the entire foreign server container.

3

Reliably Sending Messages Using the JMS SAF Client

Learn how to configure and use the JMS SAF client to reliably send JMS messages from standalone JMS clients to server-side JMS destinations.
This chapter includes the following sections:

Overview of Using Store-and-Forward with JMS Clients

The JMS SAF client extends the JMS store-and-forward service to standalone JMS clients. JMS clients can reliably send messages to server-side JMS destinations even when the client cannot reach a destination (for example, due to a temporary network connection failure). While disconnected from the server, messages sent by a JMS SAF client are stored locally on the client file system and are forwarded to server-side JMS destinations when the client reconnects.

The JMS SAF client consists of two main parts:

- The JMS SAF client implementation, which writes messages directly to a client-side persistent store on the local file system.
- A SAF forwarder, which takes the messages written to the store and sends them to a WebLogic Server instance.

An optional `ClientSAF` initialization API is also available that allows JMS SAF clients to turn the SAF forwarder mechanism on and off whenever necessary, as described in `weblogic.jms.extensions`. See The JMS SAF Client Initialization API.

> **Note**
>
> For information about using server-side WebLogic JMS SAF for reliably sending JMS messages to potentially unavailable destinations. See Configuring SAF for JMS Messages in *Administering the Store-and-Forward Service for Oracle WebLogic Server*.

Configuring a JMS Client To Use Client-Side SAF

No configuration is required on the server-side, but running client-side SAF does require some configuration on each client. Learn how to configure a JMS client to use client-side SAF.

These sections describe how to configure a JMS client to use client-side SAF.

Generating a JMS SAF Client Configuration File

Each client machine requires a JMS SAF client configuration file that specifies information about the server-side connection factories and destinations needed by the JMS SAF client environment to operate. You generate the JMS SAF client configuration file from a specified

JMS module's configuration file by using the `ClientSAFGenerate` utility bundled with your WebLogic Server installation.

The `ClientSAFGenerate` utility creates entries for all connection factories, standalone destinations, and distributed destinations found in the source JMS configuration file, as described in Steps to Generate a JMS SAF Client Configuration File from a JMS Module. The generated file defines the connection factories and imported destinations that the JMS SAF client will interact with directly through the initial JNDI context described in Modify Your JMS Client Applications To Use the JMS SAF Client's Initial JNDI Provider. However, the generated file will not contain entries for any foreign JMS destinations or SAF destinations in server-side JMS modules. Furthermore, only JMS destinations with their SAF Export Policy set to `All` are added to the file (the default setting for destinations).

How the JMS SAF Client Configuration File Works

The JMS SAF client XML file conforms to the WebLogic Server `weblogic-jms.xsd` schema for JMS modules and contains the root element `weblogic-client-jms`. The `weblogic-jms.xsd` schema contains several top-level elements that correspond to server-side WebLogic JMS SAF features, as described in Valid SAF Elements for JMS SAF Client Configurations.

The top-level elements in the file describe the connection factory and imported destination elements that the JMS SAF client will interact with directly. The SAF sending agent, remote SAF context, and SAF error handling elements describe the function of the SAF forwarder. The persistent store element is used by both the JMS SAF client API and the SAF forwarder.

Steps to Generate a JMS SAF Client Configuration File from a JMS Module

Use the `ClientSAFGenerate` utility to generate a JMS SAF client configuration file from a JMS module configuration file in a WebLogic Server domain. You can also generate a configuration file from an existing JMS SAF client configuration file, as described in ClientSAFGenerate Utility Syntax.

> **Note**
>
> Running the `ClientSAFGenerate` utility on a client machine to generate a configuration file from an existing JMS SAF client configuration file requires using the install client (`weblogic.jar`) in the CLASSPATH instead of the thin T3 JMS client and JMS SAF clients. See Installing the JMS SAF Client JAR Files on Client Machines.

These steps demonstrate how to use the `ClientSAFGenerate` utility to generate a JMS SAF client configuration file from the `examples-jms.xml` module file bundled in WebLogic Server installations.

1. Navigate to the directory in the WebLogic Server domain containing the JMS module file that you want to use as the basis for the JMS SAF client configuration file:

   ```
   $DOMAIN_HOME/domains/wl_server/config/jms
   ```

2. From a Java command line, run the `ClientSAFGenerate` utility:

   ```
   > java weblogic.jms.extensions.ClientSAFGenerate -url http://10.61.6.138:7001 -
   username myusername -moduleFile examples-jms.xml -outputFile d:\temp\ClientSAF-
   jms.xml
   ```

   Table 3-1 explains the valid `ClientSAFGenerate` arguments.

3. A configuration file named `SAFClient-jms.xml` is created in the current directory. Here is a representative example of its contents:

```
<weblogic-client-jms xmlns="http://www.bea.com/ns/weblogic/100" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
  <connection-factory name="exampleTrader">
    <jndi-name>jms.connection.traderFactory</jndi-name>
    <transaction-params>
      <xa-connection-factory-enabled>false
      </xa-connection-factory-enabled>
    </transaction-params>
  </connection-factory>
  <saf-imported-destinations name="examples">
    <saf-queue name="exampleQueue">
      <remote-jndi-name>weblogic.examples.jms.exampleQueue
      </remote-jndi-name>
      <local-jndi-name>weblogic.examples.jms.exampleQueue
      </local-jndi-name>
    </saf-queue>
    <saf-topic name="quotes">
      <remote-jndi-name>quotes</remote-jndi-name>
      <local-jndi-name>quotes</local-jndi-name>
    </saf-topic>
  </saf-imported-destinations>
  <saf-remote-context name="RemoteContext0">
    <saf-login-context>
      <loginURL>t3://localhost:7001</loginURL>
      <username>weblogic</username>
    </saf-login-context>
  </saf-remote-context>
</weblogic-client-jms>
```

> **Tip**
>
> To include additional remote SAF connection factories and destinations from other JMS modules deployed in a cluster or domain, re-run the `ClientSAFGenerate` utility against these JMS module files and specify the same JMS SAF configuration file name in the `-outputFile` parameter. See ClientSAFGenerate Utility Syntax.

4. The generated configuration file does not contain any encrypted passwords for the SAF remote contexts used to connect to remote servers. To create encrypted passwords for the remote SAF contexts and add them to the configuration file, follow the directions in Encrypting Passwords for Remote JMS SAF Contexts.
5. Copy the generated configuration file to the client machines where you will run your JMS SAF client applications. See Installing the JMS SAF Client JAR Files on Client Machines.

> **Note**
>
> `ClientSAF.xml` is the default name expected in the current working directory of the JMS client, but you can explicitly specify a file name by passing an argument in the JMS client, as described in Modify Your JMS Client Applications To Use the JMS SAF Client's Initial JNDI Provider.

ClientSAFGenerate Utility Syntax

The `weblogic.jms.extensions.ClientSAFGenerate` utility generates a JMS SAF client configuration file, using either a JMS module file or an existing JMS SAF client configuration file.

```
java [ weblogic.jms.extensions.ClientSAFGenerate ]
[ -url server-url ]
[ -username name-of-user ]
[ -existingClientFile file-path ]
[ -moduleFile file-path ['@' plan-path ]]*
[ -outputFile file-path  ]
```

Table 3-1 ClientSAFGenerate Arguments

Argument	Definition
`url`	The URL of the WebLogic Server instance where the JMS SAF client instance should connect.
`username`	The name of a valid user that this JMS SAF client instance should use when forwarding messages.
`existingClientFile`	The name of an existing JMS SAF client configuration file. If this parameter is specified, then the existing file will be read and new entries will be added. If any conflicts are detected between items being added and items already in the JMS SAF client configuration file, a warning will be given and the new item will not be added. If a JMS SAF client configuration file is specified but the file cannot be found, then an error is printed and the utility exits.
`moduleFile`	The name of a JMS module configuration file and optional plan file.
`outputFile`	`stdout`. `ClientSAF.xml` is the default name expected in the current working directory of the JMS client, but you can also explicitly specify a file name by passing an argument in the JMS client.

Valid SAF Elements for JMS SAF Client Configurations

The `weblogic-client-jms` root element of the `weblogic-jms.xsd` schema contains several top-level elements that correspond to server-side WebLogic JMS SAF features. Table 3-2 identifies the management MBean to which each top-level element in the schema corresponds.

Table 3-2 weblogic-client-saf Elements

weblogic-client-jms Element	WebLogic Server Management Bean
`connection-factory`	`JMSConnectionFactoryBean`
`saf-agent`	`SAFAgentMBean`
`saf-imported-destinations`	`SAFImportedDestinationsBean`
`saf-remote-context`	`SAFRemoteContextBean`
`saf-error-handling`	`SAFErrorHandlingBean`
`persistent-store`	See Default Store Options for JMS SAF Clients.

> **Note**
>
> You can only specify one `persistent-store` and `saf-agent` element in a JMS SAF client configuration file.

All of the properties in these management MBeans work the same in the JMS SAF client implementation as they do in server-side SAF JMS configurations, except for those described in the following tables.

Table 3-3 describes the differences between the standard `SAFAgentMBean` fields and the fields in the JMS SAF client configuration file.

Table 3-3 Modified SAFAgentMBean Fields

Server-side SAF Fields	Difference in JMS SAF Client Configuration File
`PersistentStore`	Not available. There is only one persistent store defined.
`ServiceType`	Not available. This can only be a sending agent.
`BytesThresholdHigh`	Threshold properties are not available.
`BytesThresholdLow`	Threshold properties are not available.
`MessagesThresholdHigh`	Threshold properties are not available.
`MessagesThresholdLow`	Threshold properties are not available.
`ConversationIdleTimeMaximum`	Not available. This field is only valid for receiving messages.
`AcknowledgeInterval`	Not available. Only valid for receiving messages.
`IncomingPausedAtStartup`	Not available. No way to unpause; same effect achieved by not setting the JMS SAF client property.
`ForwardingPausedAtStartup`	Not available. No way to unpause; same effect achieved by not setting the JMS SAF client property.
`ReceivingPausedAtStartup`	Not available. No way to unpause; same effect achieved by not setting the JMS SAF client property.

> **Note**
>
> You can only specify one `saf-agent` element in a JMS SAF client configuration file.

Table 3-4 describes the differences between the standard `JMSConnectionFactoryBean` fields and the fields in the JMS SAF client configuration file.

Table 3-4 Modified JMSConnectionFactoryBean Fields

Server-side SAF Fields	Difference in JMS SAF Client Configuration File
`SubDeploymentName`	Ignored. These connection factories are not targeted.
`ClientParamsBean: MulticastOverrunPolicy`	Ignored. This client cannot do multicast receives.
`TransactionParamsBean: XAConnectionFactoryEnabled`	Ignored. JMS SAF client cannot do XA transactions.

Table 3-4 (Cont.) Modified JMSConnectionFactoryBean Fields

Server-side SAF Fields	Difference in JMS SAF Client Configuration File
`FlowControlParamsBean`	All fields are ignored. JMS SAF client cannot receive messages.
`LoadBalancingParamsBean`	All fields are ignored. JMS SAF client cannot load balance because it is not connected to a server.

Table 3-5 describes the differences between the standard `SAFImportedDestinationsBean` fields and the fields in the JMS SAF client configuration file.

Table 3-5 Modified SAFImportedDestinationsBean Fields

Server-side SAF Fields	Difference in JMS SAF Client Configuration File
`SubDeploymentName`	Ignored. These are targeted to the single SAF agent defined in this file.
`UnitOfOrderRouting`	Ignored. Message unit-of-order is not supported.

Default Store Options for JMS SAF Clients

Each JMS SAF client has a default store that requires no configuration, that can be shared by multiple JMS SAF clients. The default store is a file-based store that maintains its data in a group of files directly under the JMS SAF client configuration directory.

Using the `persistent-store` element, you can specify another location for the default store and also change its default write policy by specifying the following elements in the JMS SAF client configuration file:

Table 3-6 persistent-store Elements

Element Name	What it does
`directory-path`	Specifies the path to the directory on the file system where the file store is kept.
`synchronous-write-policy`	Defines how hard a file store will try to flush records to the disk. Values are: Direct-Write (default), Cache-Flush, and Disabled.

> **Note**
>
> You can only specify one `persistent-store` element in a JMS SAF client configuration file.

The following is an example of a customized JMS SAF client default store in a JMS SAF client configuration file:

```
<persistent-store>
  <directory-path>config/jms/storesdom</directory-path>
  <synchronous-write-policy>Disabled</synchronous-write-policy>
</persistent-store>
```

For more information on using the Synchronous Write Policy for a file store, see Using the WebLogic Persistent Store in *Administering the WebLogic Persistent Store*.

Encrypting Passwords for Remote JMS SAF Contexts

The generated SAF configuration file does not contain any encrypted passwords for its generated SAF remote contexts, regardless of whether any were configured in the source JMS module file. If security credentials are configured for the remote cluster or server contexts defined in the JMS SAF client configuration file, then encrypted passwords are required to connect to the remote servers or cluster.

To create encrypted passwords for your remote SAF contexts, you must use the `ClientSAFEncrypt` utility bundled with your WebLogic Server installation, which encrypts clear text strings for use with the JMS SAF client feature.

> **Note**
>
> The existing `weblogic.security.Encrypt` command-line utility cannot be used because it expects access to the domain security files, which are not available on the client.

Steps to Generate Encrypted Passwords

The following steps demonstrate how to use `ClientSAFEncrypt` to generate encrypted passwords:

1. From a Java command line, run the `ClientSAFEncrypt` utility:

   ```
   > java -Dweblogic.management.allowPasswordEcho=true
   weblogic.jms.extensions.ClientSAFEncrypt [ key-password ] [ remote-password ]*
   ```

2. If the `key-password` or the `remote-password` fields are not specified, then you will be prompted for the `key-password` and the `remote-password` interactively.
3. The following is an example of obtaining an encrypted password:

   ```
   Password Key ("quit" to end):
   Password ("quit" to end):
   <password-
   encrypted>{Algorithm}PBEWithMD5AndDES{Salt}9IsTPAuZdcQ={Data}d6SSPp3GwPAfEXn8izyZA0IR
   CV/izT8H</password-encrypted>
   Password ("quit" to end):
   ```

4. Continue generating as many remote passwords as necessary for the remote contexts defined in the JMS SAF client configuration file.
5. Copy the encrypted remote password before the closing `</saf-login-context>` stanza in the JMS SAF client configuration file. For example:

   ```
   <saf-remote-context name="RemoteContext0">
   <saf-login-context>
   <loginURL>http://10.61.6.138:7001</loginURL>
   <username>weblogic</username>
   <password-
   encrypted>{Algorithm}PBEWithMD5AndDES{Salt}dWENfrgXh8U={Data}u8xZ968dElHckso/
   ZYm2LQ6xVNBPpBGQ</password-encrypted>
   </saf-login-context>
   </saf-remote-context>
   ```

Use the `ClientSAFEncrypt` utility for all passwords (with the same `key-password`) required by the remote contexts defined in the JMS SAF client configuration file. When a client starts using the JMS SAF client, it must supply the same `key-password` that was provided to the `ClientSAFEncrypt` utility.

6. Type `quit` to exit the `ClientSAFEncrypt` utility.

ClientSAFEncrypt Utility Syntax

The `weblogic.jms.extensions.ClientSAFEncrypt` utility encrypts clear text strings for use with JMS SAF clients in order to access remote SAF contexts.

```
java [ -Dweblogic.management.allowPasswordEcho=true ]
weblogic.jms.extensions.ClientSAFEncrypt [ key-password ]
weblogic.jms.extensions.ClientSAFEncrypt [ remote-password ]
```

Table 3-7 ClientSAFEncrypt Arguments

Argument	Definition
`weblogic.management.allowPasswordEcho`	Optional. Allows echoing characters entered on the command, `weblogic.jms.extensions.ClientSAFEncrypt`. Expects that no-echo is available; if no-echo is not available, set this property to `true`.
`key-password`	The key to use when encrypting all remote passwords needed for the remote contexts defined in the JMS SAF client configuration file. If omitted from the command line, you will be prompted to enter a `key-password`.
`remote-password`	Clear text string to be encrypted. Multiple passwords for each remote context can be generated in one session. If omitted from the command line, you are prompted to enter a `remote-password`.

Installing the JMS SAF Client JAR Files on Client Machines

WebLogic Server provides two JMS SAF client options:

- `weblogic.jar`, see WebLogic Install Client.
- Standalone T3 client (`wlthint3client.jar` and `wlsaft3client.jar`). For information on how to create JMS SAF clients using the WebLogic Thin T3 client, see Developing a WebLogic Thin T3 Client.

The required JAR files are located in the *`WL_HOME`*`\server\lib` subdirectory of the WebLogic Server installation directory, where *`WL_HOME`* is the top-level installation directory for the entire WebLogic Server product installation (for example, `c:\Oracle\Middleware\Oracle_Home\wlserver\server\lib`).

Modify Your JMS Client Applications To Use the JMS SAF Client's Initial JNDI Provider

The JMS SAF client requires a special initial JNDI provider to look up the server-side JMS connection factories and destinations specified in the JMS SAF client configuration file that was generated during Steps to Generate a JMS SAF Client Configuration File from a JMS Module.

Required JNDI Context Factory for JMS SAF Clients

Modify your JMS client applications to use the JMS SAF client JNDI context factory in place of the standard server initial context. The name used for the JMS SAF client JNDI property is `java.naming.factory.initial` is `weblogic.jms.safclient.jndi.InitialContextFactoryImpl`.

An example JNDI initial context factory could look like this in a JMS SAF client application:

```
   public final static String
JNDI_FACTORY="weblogic.jms.safclient.jndi.InitialContextFactoryImpl";
```

With the standard JNDI lookup, the JMS SAF client is started automatically and looks up the server-side JMS connection factories and destinations specified in the configuration file. For the configuration file, `ClientSAF.xml` is the default name expected in the current working directory of the JMS client, but you can also explicitly specify a configuration file name by passing an argument in the JMS client.

Items returned from the initial context created with the JMS SAF client do not work in JMS calls from third-party JMS providers. Also, there can be no mixing of JMS SAF client initial contexts with server initial contexts, as described in No Mixing of JMS SAF Client Contexts and Server Contexts.

You can also update your JMS client applications to use the `weblogic.jms.extensions.ClientSAF` extension class, which allows the JMS client to control when the JMS SAF client system is in use. See The JMS SAF Client Initialization API.

Optional JNDI Properties for JMS SAF Clients

There are also two optional JMS SAF client JNDI properties:

- `Context.PROVIDER_URL` – This must be a URL that points to your JMS SAF client configuration file. If one is not specified, it defaults to a file named `ClientSAF.xml` in the current working directory of the JVM.
- `Context.SECURITY_CREDENTIALS` – If you are using security, specify a key password used to encrypt the remote context passwords in the configuration file.

The local JNDI provider only supports the `lookup(String)` and `close()` APIs. All other APIs throw an exception stating that the functionality is not supported.

JMS SAF Client Management Tools

Learn about the JMS SAF client initialization API and client-side store administration utility features, which are available for use with the JMS SAF client implementation.

The following management features are available for use with the JMS SAF client implementation:

The JMS SAF Client Initialization API

The `weblogic.jms.extensions.ClientSAF` extension class allows the JMS client to control when the JMS SAF client system is in use. JMS clients do not need to use this extension mechanism, but can do so in order to get finer control of the JMS SAF client system. For example, the `close()` method can be used to stop a JMS client from forwarding messages.

Client-Side Store Administration Utility

The JMS SAF client provides a utility to administer the default file store used by JMS SAF clients. Similar to the server-side WebLogic Store utility, it enables you to troubleshoot a JMS SAF client store or extract its data. Run the utility from a Java command line or from the WebLogic Scripting Tool (WLST). The store utility operates only on a store that is not currently opened by a running JMS SAF client.

The most common uses-cases for store administration are for compacting a file store to reduce its size and for dumping the contents of a file store to an XML file for troubleshooting purposes. See Administering a Persistent Store in *Administering the WebLogic Persistent Store*.

JMS Programming Considerations with JMS SAF Clients

Learn about the JMS programming considerations while using the JMS SAF client.

The following JMS programming considerations apply when you use the JMS SAF client:

How the JMSReplyTo Field Is Handled In JMS SAF Client Messages

Generally, JMS applications can use the `JMSReplyTo` header field to advertise its temporary destination name to other applications. However, as with server-side JMS SAF imported destinations, the use of temporary destinations with a `JMSReplyTo` field is not supported for JMS SAF clients.

For more information on using JMS temporary destinations, see Using Temporary Destinations in *Developing JMS Applications for Oracle WebLogic Server*.

No Mixing of JMS SAF Client Contexts and Server Contexts

When items returned from the JMS SAF client naming context are used in conjunction with items returned from a server initial context, the JMS API fails with a reasonable exception message. Likewise, when items returned from a server initial context is used in conjunction with items returned from the JMS SAF client naming context, the JMS API fails with a reasonable exception message.

Using Transacted Sessions With JMS SAF Clients

Transacted sessions are supported with JMS SAF clients, but client SAF operations do not participate in any global (XA) transactions. If there is an XA transaction, the message send operation is done outside the XA transaction and no exception is thrown.

JMS SAF Client Interoperability Guidelines

The interoperability guidelines apply when using the JMS SAF client to forward messages to server-side WebLogic JMS destinations.

The topic has following sections:

Java Runtime

Each client machine must have Java SE 1.4 runtime or later installed.

WebLogic Server Versions

The WebLogic JMS SAF client system only works with WebLogic Server 9.2 and later.

On the client-side, the WebLogic JMS SAF client code must be running with WebLogic Server JAR files that are release 9.2 or later. For more information on installing WebLogic Server JAR files, see Installing the JMS SAF Client JAR Files on Client Machines.

JMS C API

Client-side SAF is usable from C environments using the JMS C API. This implementation of the JMS C API uses JNI in order to access a Java Virtual Machine (JVM). However, the JMS C API cannot use the `weblogic.jms.extensions.ClientSAF` interface because it is a non-standard JMS API.

To use SAF with the JMS C API, set the SAF context on the `jndiFactory`. By default, if you pass `NULL` as the `jndiFactory` you would get the normal WebLogic Server context. For example:

```
int JmsContextCreate(JmsString *uri, JmsString *jndiFactory, JmsString *username,
JmsString *password, JmsContext **context, JMS64I flags)
```

See WebLogic C API in *Developing JMS Applications for Oracle WebLogic Server*.

Tuning JMS SAF Clients

JMS SAF clients can take advantage of the tuning parameters available with the server-side SAF service.

See Tuning WebLogic JMS Store-and-Forward in the *Tuning Performance of Oracle WebLogic Server*.

Limitations of Using the JMS SAF Client

Learn about the non-supported features and exceptions while using the JMS SAF Client.

In addition to the field-level limitations discussed in Valid SAF Elements for JMS SAF Client Configurations, the following limitations apply to the JMS SAF client:

- The JMS Message Unit-of-Order and Unit-of-Work JMS Message Group features are not supported.
- A destination consumer of an imported SAF destination is not supported. An exception is thrown if you attempt to create such a consumer in JMS SAF client environment.
- A destination browser of an imported SAF destination is not supported. An exception is thrown if you attempt to create such a browser in JMS SAF client environment.
- Transacted sessions are supported, but not user (XA) transactions. Client SAF operations do not participate in any global transactions. See Using Transacted Sessions With JMS SAF Clients.
- JMS SAF clients are not supported in Java Applets.
- You can only specify one `persistent-store` and `saf-agent` element in a JMS SAF client configuration file.

- The WebLogic Server CMP 2.x extension that allows users to return a `java.sql.ResultSet` to a client is not supported.

Behavior Change in JMS SAF Client Message Storage

In the Weblogic JMS SAF client, messages are kept in local storage before being forwarded to the remote destinations. Each remote destination corresponds to a local storage unit, which is called a kernel queue. In releases prior to Oracle WebLogic Server 10.3.3.0, a JMS SAF client instance used a different kernel queue each time it closed and reopened. This behavior allowed multiple kernel queues to correspond to a destination.

> **Note**
>
> - If the destination was a single remote destination, under some circumstances a JMS SAF client may not have forwarded messages or may have forwarded them out of order.
> - If the destination was a distributed destination, under some circumstances some messages could be permanently lost or duplicate messages could be sent.

In this release, the same kernel queue is used for a remote destination regardless of how many times the JMS SAF client is opened and closed. For application environments in which a JMS client SAF instance is opened only once, there is no change in behavior.

The Upgrade Process, Tools, and System Properties

The following sections provide information on process, tools, and system properties used to upgrade JMS SAF clients to use one kernel queue for each destination, regardless of how many times the client opens and closes the kernel queue.

- If your application environment opens a JMS SAF client only once, no action is required.
- New JMS SAF clients require no changes.
- If your application environment opens and close a JMS SAF client more than once, existing messages can be located in multiple kernel queues in the client. Oracle provides a user-tunable process to migrate messages from multiple kernel queues to a single kernel queue when a JMS SAF client starts for the first time after being upgraded. Although the migration ensures messages are not lost, there is a small possibility that message duplication can occur. Any message that is migrated retains its normal SAF QoS. You can opt out of migrating existing messages by either removing the local store or specifying `weblogic.jms.safclient.MigrateExistingMessages=false`. See JMS SAF Client Migration Properties. If the message migration fails for any reason, the JMS SAF client does not start.

JMS SAF Client Discovery Tool

The JMS SAF client discovery tool is a Java program packaged in the WebLogic Server JMS client library that can be used to survey existing local SAF messages before upgrading. This tool:

1. Reviews the client configuration, including checking each remote destination and the corresponding kernel queues.
2. Prints the number of messages in each kernel queue.

3. Prints select header information from the first message in each kernel queue; for example, message ID, correlation ID, SAF sequence name, SAF sequence number and Unit-of-Order.

You can use the results of the survey to tune upgraded system properties. See JMS SAF Client Migration Properties.

Usage: `java weblogic.jms.extensions.ClientSAFDiscover` *`options`*

In the preceding syntax, *`options`* represents one or more of the values described in the following table:

Table 3-8 Descriptions and Default values for the JMS SAF Client Discovery Tool

Option	Description
`-help`	Print usage information.
`-clientSAFRootDir` *`client-saf-root-directory`*	Optional. Defaults to current directory. Specifies the root directory of the target SAF client to discover. Any relative paths in the SAF client configuration file are relative to this directory.
`-configurationFile` *`config-file`*	Optional. Defaults to `ClientSAF.xml`. Specifies the location of the configuration file used by the targeted JMS SAF client. This option is required if the `clientSAFRootDir` option is specified. If the `clientSAFRootDir` option or this option is specified, the `ClientSAF.xml` file under the current working directory is used. If the specified configuration file does not exist, an exception is thrown.
`-cutoffFormat` *`pattern`*	Optional. Defaults to *`yyyy-MM-dd`*`'T'`*`HH:mm:ss.SSSZ`*. Specifies the date and time pattern for the optional cutoff time used. See `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/text/DateFormat.html`.
`-cutoffTime` *`cutoff-time`*	Optional. Defaults to null set. Prints data on messages that are discarded during upgrade if `weblogic.jms.safclient.MigrationCutoffTime` is set. No messages are discarded. The cutoff time format depends on the value of the `-cutoffFormat` property. An exception is thrown if the specified cutoff time does not match the `cutoffFormat` pattern. If a cutoff time is not specified, no messages are discarded and no messages are printed.
`-discoveryFile` *`discovery-file`*	Optional. Defaults to `SAF_DISCOVERY`. Specifies the file that contains the output generated by the JMS SAF client discovery tool. The output is placed relative to the root directory unless an absolute path is specified. If the specified file already exists, it is deleted and a new file is created.

Example

If you created a JMS SAF CLient using:

```
ClientSAFFactory.getClientSAF(new File("c:\\foo"), new FileInputStream("c:\
\ClientSAF-jms.xml"));
```

You can survey the existing messages using the `ClientSAFDiscover` tool before upgrading the JMS SAF client. For example:

```
java weblogic.jms.client.ClientSAFDiscover -clientSAFRootDir c:\foo -
configurationFile c:\ClientSAF-jms.xml
```

The discovery information will be written to the default location at `c:\foo\SAF_DISCOVERY`.

JMS SAF Client Migration Properties

Because message migration can be a complex issue even when automated, Oracle provides the following system properties to manage the process:

- `weblogic.jms.safclient.MigrateExistingMessages`—If set to `false`, this property prevents the migration of messages from multiple queues to a single queue. The default is `true`.
- `weblogic.jms.safclient.MigrationCutoffTime`—Use this property to specify a time after which messages are migrated to a single kernel queue. Any remaining messages are discarded. If this property is not specified, all existing messages are upgraded. Use this property in conjunction with the `weblogic.jms.safclient.MigrationCutoffTimeFormat`property to specify the time format.

  For example, if the cutoff time format is the default, a valid cutoff time is `2009-12-16T10:34:17.887-0800`. If the specified time does not match the format pattern, then an exception is thrown and the JMS SAF client stops all message processing.
- `weblogic.jms.safclient.MigrationCutoffTimeFormat`—Specifies the format of the `weblogic.jms.safclient.MigrationCutoffTime`.

  The default is *`yyyy-MM-dd'T'HH:mm:ss.SSSZ`*. See the description of the java.text.SimpleDateFormat class for more information.

4

Developing a CORBA/IDL Client

Learn how to develop clients for heterogeneous distributed applications. RMI over IIOP with CORBA/IDL clients involves an Object Request Broker (ORB) and a compiler that creates an interoperating language called IDL. C, C++, and COBOL are examples of languages that ORBs may compile into IDL. A CORBA programmer can use the interfaces of the CORBA Interface Definition Language (IDL) to enable CORBA objects to be defined, implemented, and accessed from the Java programming language.
This chapter includes the following sections:

Guidelines for Developing a CORBA/IDL Client

Using RMI-IIOP with a CORBA/IDL client enables interoperability between non-Java clients and Java objects. If you have existing CORBA applications, you should program according to the RMI-IIOP with CORBA/IDL client model. Basically, you will be generating IDL interfaces from Java. Your client code will communicate with WebLogic Server through these IDL interfaces. This is basic CORBA programming.

The following sections provide some guidelines for developing RMI-IIOP applications with CORBA/IDL clients.

For further reference see the following Object Management Group (OMG) specifications:

- Java Language to IDL Mapping Specification at http://www.omg.org/cgi-bin/doc?formal/01-06-07
- CORBA/IIOP 2.4.2 Specification at http://www.omg.org/cgi-bin/doc?formal/01-02-33

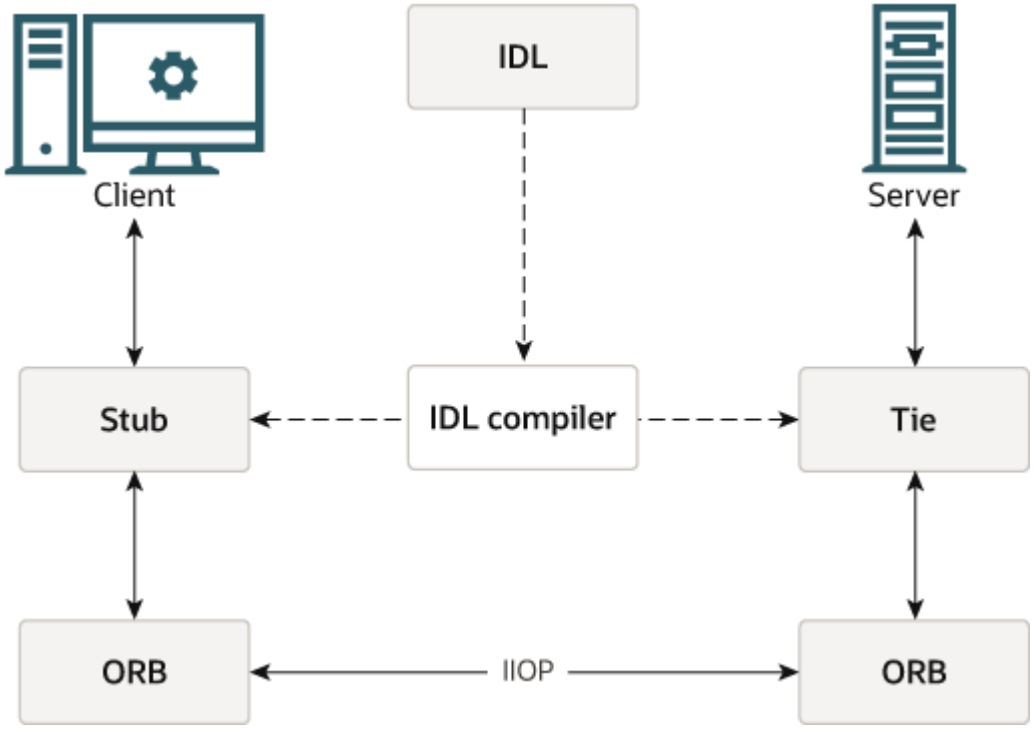
Working with CORBA/IDL Clients

In CORBA, interfaces to remote objects are described in a platform-neutral interface definition language (IDL). To map the IDL to a specific language, you compile the IDL with an IDL compiler. The IDL compiler generates a number of classes such as stubs and skeletons that the client and server use to obtain references to remote objects, forward requests, and marshall incoming calls. Even with IDL clients it is strongly recommended that you begin programming with the Java remote interface and implementation class, then generate the IDL to allow interoperability with WebLogic and CORBA clients, as illustrated in the following sections. Writing code in IDL that can be then reverse-mapped to create Java code is a difficult and bug-filled enterprise, and Oracle does not recommend it.

IDL Client (Corba object) relationships

Learn how the IDL takes part in the RMI-IIOP model.

Figure 4-1 IDL Client relationships



Java to IDL Mapping

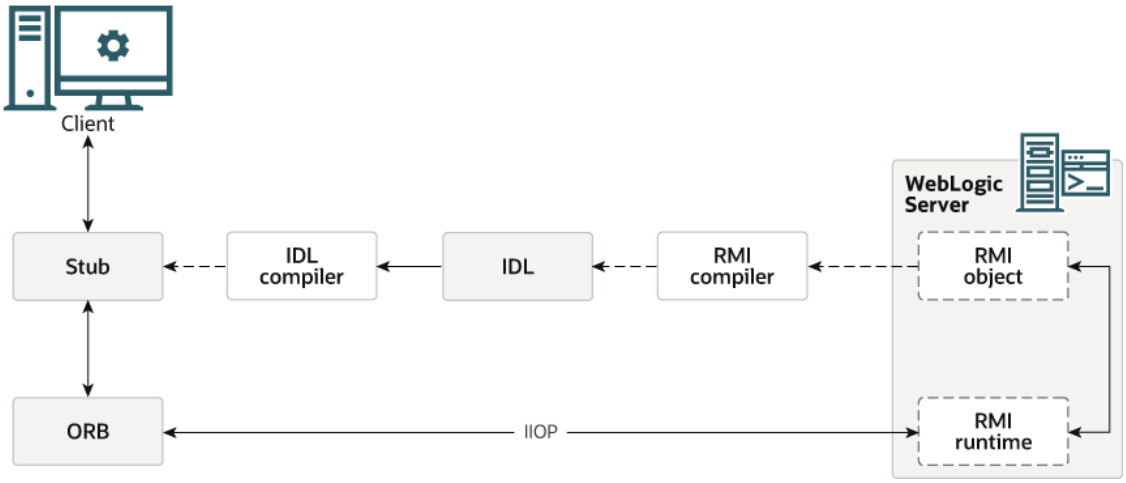
In WebLogic RMI, interfaces to remote objects are described in a Java remote interface that extends `java.rmi.Remote`. The Java-to-IDL mapping specification defines how an IDL is derived from a Java remote interface. In the WebLogic RMI over IIOP implementation, you run the implementation class through the WebLogic RMI compiler or WebLogic EJB compiler with the `-idl` option. This process creates an IDL equivalent of the remote interface. You then compile the IDL with an IDL compiler to generate the classes required by the CORBA client.

The client obtains a reference to the remote object and forwards method calls through the stub. WebLogic Server implements a `CosNaming` service that parses incoming IIOP requests and dispatches them directly into the RMI run-time environment.

WebLogic RMI over IIOP object relationships

Learn about the object relationships when using RMI-IIOP to connect a client and server.

Figure 4-2 WebLogic RMI over IIOP object relationships



Objects-by-Value

The Objects-by-Value specification allows complex data types to be passed between the two programming languages involved. In order for an IDL client to support Objects-by-Value, you develop the client in conjunction with an Object Request Broker (ORB) that supports Objects-by-Value. To date, relatively few ORBs support Objects-by-Value correctly.

When developing an RMI over IIOP application that uses IDL, consider whether your IDL clients will support Objects-by-Value, and design your RMI interface accordingly. If your client ORB does not support Objects-by-Value, you must limit your RMI interface to pass only other interfaces or CORBA primitive data types. The following table lists ORBs that Oracle has tested with respect to Objects-by-Value support:

Table 4-1 ORBs Tested with Respect to Objects-by-Value Support

Vendor	Versions	Objects-by-Value
Oracle	Tuxedo 8.x C++ Client ORB	Supported
Borland	VisiBroker 3.3, 3.4	Not supported
Borland	VisiBroker 4.x, 5.x	Supported
Iona	Orbix 2000	Supported (Oracle has encountered problems with this implementation)

For more information on Objects-by-Value, see Limitations of Passing Objects by Value in *Developing RMI Applications for Oracle WebLogic Server*.

Procedure for Developing a CORBA/IDL Client

Learn how to develop RMI over IIOP application with CORBA/IDL.

To develop an RMI over IIOP application with CORBA/IDL:

1. Define your remote object's public methods in an interface that extends `java.rmi.Remote`.

   This remote interface may not require much code. All you need are the method signatures for methods you want to implement in remote classes. For example:

   ```
   public interface Pinger extends java.rmi.Remote {
   public void ping() throws java.rmi.RemoteException;
   public void pingRemote() throws java.rmi.RemoteException;
   public void pingCallback(Pinger toPing) throws java.rmi.RemoteException;
   }
   ```

2. Implement the interface in a class named `interfaceNameImpl` and bind it into the JNDI tree to be made available to clients.

   This class should implement the remote interface that you wrote, which means that you implement the method signatures that are contained in the interface. All the code generation that will take place is dependent on this class file. Typically, you configure your implementation class as a WebLogic startup class and include a main method that binds the object into the JNDI tree. For example:

   ```
   public static void main(String args[]) throws Exception {
     if (args.length > 0)
     remoteDomain = args[0];
     Pinger obj = new PingImpl();
     Context initialNamingContext = new InitialContext();
     initialNamingContext.rebind(NAME,obj);
     System.out.println("PingImpl created and bound to "+ NAME);
   }
   ```

3. Compile the remote interface and implementation class with a Java compiler. Developing these classes in an RMI-IIOP application is no different than doing so in normal RMI. For more information on developing RMI objects, see *Developing RMI Applications for Oracle WebLogic Server*.

4. Generate an IDL file by running the WebLogic RMI compiler or WebLogic EJB compiler with the `-idl` option.

   The required stub classes will be generated when you compile the IDL file. For general information on the these compilers, refer to Understanding WebLogic RMI and *Developing RMI Applications for Oracle WebLogic Server*. Also reference the Java IDL specification at Java Language Mapping to OMG IDL Specification at http://www.omg.org/technology/documents/index.htm.

   The following compiler options are specific to RMI over IIOP:

   Table 4-2 RMI-IIOP Compiler Options

Option	Function
`-idl`	Creates an IDL for the remote interface of the implementation class being compiled

Table 4-2 (Cont.) RMI-IIOP Compiler Options

Option	Function
`-idlDirectory`	Target directory where the IDL will be generated
`-idlFactories`	Generate factory methods for value types. This is useful if your client ORB does not support the factory value type.
`-idlNoValueTypes`	Suppresses generation of IDL for value types.
`-idlOverwrite`	Causes the compiler to overwrite an existing idl file of the same name
`-idlStrict`	Creates an IDL that adheres strictly to the Objects-By-Value specification. (not available with appc)
`-idlVerbose`	Display verbose information for IDL generation
`-idlVisibroker`	Generate IDL somewhat compatible with Visibroker 4.1 C++

The options are applied as shown in this example of running the RMI compiler:

```
> java weblogic.rmic -idl -idlDirectory /IDL rmi_iiop.HelloImpl
```

The compiler generates the IDL file within sub-directories of the idlDirectoy according to the package of the implementation class. For example, the preceding command generates a `Hello.idl` file in the `/IDL/rmi_iiop` directory. If the `idlDirectory` option is not used, the IDL file is generated relative to the location of the generated stub and skeleton classes.

5. Compile the IDL file to create the stub classes required by your IDL client to communicate with the remote class. Your ORB vendor will provide an IDL compiler.
6. The IDL file generated by the WebLogic compilers contains the directives: `#include orb.idl`. This IDL file should be provided by your ORB vendor. An `orb.idl` file is shipped in the `/lib` directory of the WebLogic distribution. This file is only intended for use with the ORB included in the JDK.
7. Develop the IDL client.

   IDL clients are pure CORBA clients and do not require any WebLogic classes. Depending on your ORB vendor, additional classes may be generated to help resolve, narrow, and obtain a reference to the remote class. In the following example of a client developed against a VisiBroker 4.1 ORB, the client initializes a naming context, obtains a reference to the remote object, and calls a method on the remote object.

   Code segment from C++ client of the RMI-IIOP example

```
// string to object
CORBA::Object_ptr o;
cout << "Getting name service reference" << endl;
if (argc >= 2 && strncmp (argv[1], "IOR", 3) == 0)
  o = orb->string_to_object(argv[1]);
else
  o = orb->resolve_initial_references("NameService");
// obtain a naming context
cout << "Narrowing to a naming context" << endl;
CosNaming::NamingContext_var context =
CosNaming::NamingContext::_narrow(o);
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Pinger_iiop");
name[0].kind = CORBA::string_dup("");
// resolve and narrow to RMI object
```

```
cout << "Resolving the naming context" << endl;
CORBA::Object_var object = context->resolve(name);
cout << "Narrowing to the Ping Server" << endl;
::examples::iiop::rmi::server::wls::Pinger_var ping =
  ::examples::iiop::rmi::server::wls::Pinger::_narrow(object);
// ping it
cout << "Ping (local) ..." << endl;
ping->ping();
}
```

Notice that before obtaining a naming context, initial references were resolved using the standard Object URL (see the CORBA/IIOP 2.4.2 Specification, section 13.6.7). Lookups are resolved on the server by a wrapper around JNDI that implements the COS Naming Service API.

The Naming Service allows WebLogic Server applications to advertise object references using logical names. The CORBA Name Service provides:

- An implementation of the Object Management Group (OMG) Interoperable Name Service (INS) specification.
- Application programming interfaces (APIs) for mapping object references into an hierarchical naming structure (JNDI in this case).
- Commands for displaying bindings and for binding and unbinding naming context objects and application objects into the namespace.

8. IDL client applications can locate an object by asking the CORBA Name Service to look up the name in the JNDI tree of WebLogic Server. In the example above, you run the client by entering:

```
Client.exe -ORBInitRef NameService=iioploc://localhost:7001/NameService
```

5

Developing Clients for CORBA Objects

Learn how to use the CORBA API to develop clients using CORBA objects.
This chapter includes the following sections:

Enhancements and Limitations of CORBA Object Types

Learn about the enhancements and limitations of CORBA. The RMI-IIOP run time is extended to support all CORBA object types (as opposed to RMI valuetypes) and CORBA stubs.

Enhancements include:

- Support for out and in-out parameters
- Support for a call to a CORBA service from WebLogic Server using transactions and security
- Support for a WebLogic ORB hosted in JNDI rather than an instance of the JDK ORB used in previous releases

CORBA Object Type support has the following limitations:

- It should not be used to make calls from one WebLogic Server instance to another WebLogic Server instance.
- Clustering is not supported. If a clustered object reference is detected, WebLogic Server uses internal RMI-IIOP support to make the call. Out and in-out parameters will not be supported.
- CORBA services created by `ORB.connect()` result in a second object hosted inside the server. It is important that you use `ORB.disconnect()`to remove the object when it is no longer needed.

Making Outbound CORBA Calls: Main Steps

Learn how to implement a development model for customers using CORBA to make outbound calls.

Follow these steps to implement a typical development model for customers wanting to use the CORBA API for outbound calls.

1. Generate CORBA stubs from IDL using idlj, the JDKs IDL compiler.
2. Compile the stubs using javac.
3. Build EJB(s) including the generated stubs in the jar.
4. Use the WebLogic ORB hosted in JNDI to reference the external service.

Using the WebLogic ORB Hosted in JNDI

Learn about various mechanisms to access the WebLogic ORB with the help of examples provided in this section. Each mechanism achieves the same effect and their constituent components can be mixed to some degree.

The object returned by `narrow()` will be a CORBA stub representing the external ORB service and can be invoked as a normal CORBA reference. In the following code examples it is assumed that the CORBA interface is called MySvc and the service is hosted at "where" in a foreign ORB's CosNaming service located at `exthost:extport`:

ORB from JNDI

The following code listing provides information on how to access the WebLogic ORB from JNDI.

Example 5-1 Accessing the WebLogic ORB from JNDI

```
.
.
.
ORB orb = (ORB)new InitialContext().lookup("java:comp/ORB");
NamingContext nc =
NamingContextHelper.narrow(orb.string_to_object("corbaloc:iiop:exthost:extport/
NameService"));
MySvc svc = MySvcHelper.narrow( nc.resolve(new NameComponent[] { new
NameComponent("where", "")}));
.
.
.
```

Direct ORB creation

The following code listing provides information on how to create a WebLogic ORB.

Example 5-2 Direct ORB Creation

```
.
.
.
ORB orb = ORB.init();
MySvc svc =
MySvcHelper.narrow(orb.string_to_object("corbaname:iiop:exthost:extport#where"));
.
.
.
```

Using JNDI

The following code listing provides information on how to access the WebLogic ORB using JNDI.

Example 5-3 Accessing the WebLogic ORB Using JNDI

```
.
.
.
MySvc svc = MySvcHelper.narrow(new
InitialContext().lookup("corbaname:iiop:exthost:extport#where"));
.
.
.
```

The WebLogic ORB supports most client ORB functions, including DII (Dynamic Invocation Interface). To use this support, you must not instantiate a foreign ORB inside the server. This will not yield any of the integration benefits of using the WebLogic ORB.

Supporting Inbound CORBA Calls

WebLogic Server also provides basic support for inbound CORBA calls as an alternative to host an ORB inside the server. To do this, you use `ORB.connect()` to publish a CORBA server inside WebLogic Server by writing an RMI-object that implements a CORBA interface.

Given the MySVC examples above:

Example 5-4 Supporting Inbound CORBA Calls

```
.
.
.
class MySvcImpl implements MvSvcOperations, Remote
{
public void do_something_remote() {}

public static main() {
MySvc svc = new MySvcTie(this);
InitialContext ic = new InitialContext();
((ORB)ic.lookup("java:comp/ORB")).connect(svc);
ic.bind("where", svc);
}
}
.
.
.
```

When registered as a startup class, the CORBA service will be available inside the WebLogic Server CosNaming service at the location "where".

6

Developing a WebLogic C++ Client for a Tuxedo ORB

Learn how a WebLogic C++ client uses the Tuxedo 8.1 or higher C++ Client ORB to generate IIOP requests for EJBs running on WebLogic Server. This client supports object-by-value and the CORBA Interoperable Naming Service (INS).
This chapter includes the following sections:

WebLogic C++ Client Advantages and Limitations

Learn about the advantages and limitations offered by WebLogic C++ client.

A WebLogic C++ client offers these advantages:

- Simplifies your development process by avoiding third-party products
- Provides a client-side solution that allows you to develop or modify existing C++ clients
- The Tuxedo C++ Client ORB is packaged with Tuxedo 8.1 and higher.

The WebLogic C++ client has the following limitations:

- Provides security through the WebLogic Server Security service.
- Provides only server-side transaction demarcation.

How the WebLogic C++ Client Works

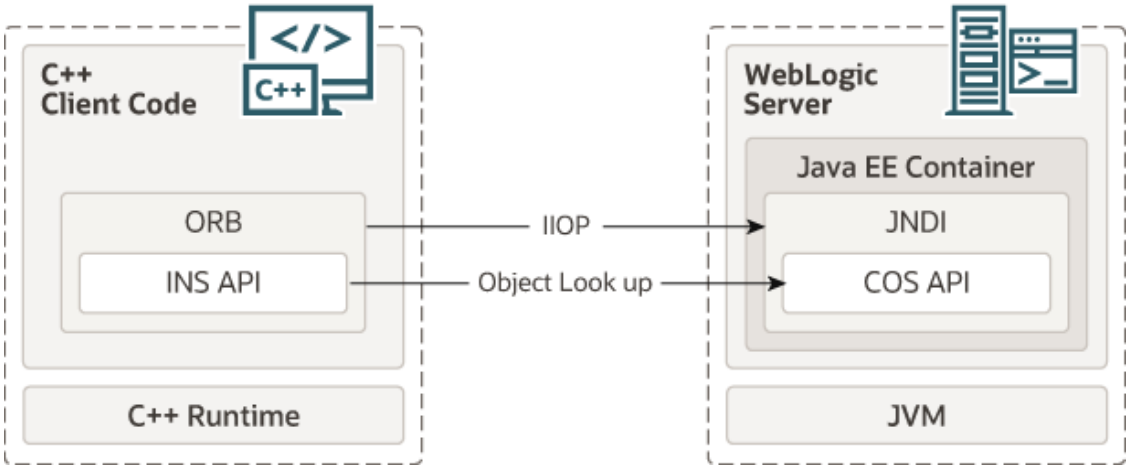
Learn how a WebLogic C++ client processes requests using the CORBA Interoperable Name Service (INS).

The WebLogic C++ client processes requests as follows:

- The WebLogic C++ client code requests a WebLogic Server service.
  - The Tuxedo ORB generates an IIOP request.
  - The ORB object is initially instantiated and supports Object-by-Value data types.

The client uses the CORBA Interoperable Name Service (INS) to look up the EJB object bound to the JNDI naming service. For more information on how to use the Interoperable Naming Service to get object references to initial objects such as NameService, see Interoperable Naming Service Bootstrapping Mechanism in *CORBA Programming Reference* for Oracle Tuxedo 8.0 at http://docs.oracle.com/cd/E13203_01/tuxedo/tux80/interm/corbaprog.htm#client.

Example 6-1 WebLogic C++ Client to WebLogic Server Interoperability



Developing WebLogic C++ Clients

Learn how to develop a C++ client.

Use the following steps to develop a C++ client:

1. Use the ejbc compiler with the `-idl` option to compile the EJB with which your C++ client will interoperate. This action generates an IDL script for the EJB.
2. Use the C++ IDL compiler to compile the IDL script and generate the CORBA client stubs, server skeletons, and header files. For information on the use of the C++ IDL Compiler, see OMG IDL Syntax and the C++ IDL Compiler in *CORBA Programming Reference* for Oracle Tuxedo 8.0 at http://docs.oracle.com/cd/E13203_01/tuxedo/tux80/interm/corbaprog.htm#client
3. Discard the server skeletons; the EJB represents the server side implementation.
4. Create a C++ client that implements an EJB as a CORBA object. For general information on how to create CORBA client applications, see *Creating CORBA Client Applications* for Oracle Tuxedo 8.0 at http://docs.oracle.com/cd/E13203_01/tuxedo/tux80/interm/corbaprog.htm#client
5. Use the Tuxedo `buildobjclient` command to build the client.

7

Using Java EE Client Application Modules

Learn how Java EE specifies a standard for including client application code (a client module) in an EAR file. This allows the client side of an application to be packaged along with the other modules that make up the application.
The client module is declared in the `META-INF/application.xml` file of the EAR using a `<java>` tag. See Enterprise Application Deployment Descriptor Elements in *Developing Applications for Oracle WebLogic Server*.

> **Note**
>
> The `<java>` tag is often confused to be a declaration of Java code that can be used by the server-side modules. This is not its purpose, it is used to declare client-side code that runs outside of the server-side container.

A client module is basically a JAR file containing a special deployment descriptor named `META-INF/application-client.xml`. This client JAR file also contains a Main-Class entry in its `META-INF/MANIFEST.MF` file to specify the entry point for the program. For more information on the `application-client.xml` file, see Client Application Deployment Descriptor Elements.

> **Note**
>
> When you use the Java Web Start to connect to JMS queues and topics deployed in WebLogic Server, you may get `java.security.AccessControlException.` To avoid security failures, you must set the system property `-Dweblogic.j2ee.client.isWebStart` to `true` in the client side.

This chapter includes the following sections:

Extracting a Client Application

Learn how to use `weblogic.ClientDeployer` and `weblogic.j2eeclient.Main` utilities to extract a client application.

WebLogic Server includes two utilities that facilitate the use of client modules. They are:

- `weblogic.ClientDeployer`—Extracts the client module from the EAR and prepares it for being run.
- `weblogic.j2eeclient.Main`—Runs the client code.

See ClientDeployer in *Command Reference for Oracle WebLogic Server*.

You use the `weblogic.ClientDeployer` utility to extract the client-side JAR file from a Java EE EAR file, creating a deployable JAR file. Run the `weblogic.ClientDeployer` class on the Java command line using the following syntax:

```
java weblogic.ClientDeployer ear-file client1 [client2 client3 ...]
```

The `ear-file` argument is a Java archive file with an `.ear` extension or an expanded directory that contains one or more client application JAR files.

The client arguments specify the clients you want to extract. For each client you name, the `weblogic.ClientDeployer` utility searches for a JAR file within the EAR file that has the specified name containing the .jar extension.

For example, consider the following command:

```
java weblogic.ClientDeployer app.ear myclient
```

This command extracts myclient.jar from app.ear. As it extracts, the `weblogic.ClientDeployer` utility performs two other operations.

- It ensures that the JAR file includes a META-INF/application-client.xml file. If it does not, an exception is thrown.
- It reads from a file named myclient.runtime.xml and creates a weblogic-application-client.xml file in the extracted JAR file. This is used by the `weblogic.j2eeclient.Main` utility to initialize the client application's component environment (`java:comp/env`). For information on the format of the runtime.xml file, see Client Application Deployment Descriptor Elements.

> **Note**
>
> You create the <client>.runtime.xml descriptor for the client program to define bindings for entries in the module's META-INF/application-client.xml deployment descriptor.

Running a Client Application

Learn how to use the `weblogic.j2eeclient.Main` utility to run a client application after the extraction of the client-side JAR file.

After the client-side JAR file is extracted from the EAR file, use the `weblogic.j2eeclient.Main` utility to bootstrap the client-side application and point it to a WebLogic Server instance using the following command:

```
java weblogic.j2eeclient.Main [options] [clientjar] URL [client-args]
```

For example:

```
java weblogic.j2eeclient.Main myclient.jar t3://localhost:7001
```

The `weblogic.j2eeclient.Main` utility creates a component environment that is accessible from `java:comp/env` in the client code. See ClientDeployer in *Command Reference for Oracle WebLogic Server*.

If a resource mentioned by the `application-client.xml` descriptor is one of the following types, the `weblogic.j2eeclient.Main` class attempts to bind it from the global JNDI tree on the server to `java:comp/env` using the information specified earlier in the `myclient.runtime.xml` file.

- `ejb-ref`
- `jakarta.jms.QueueConnectionFactory`

- `jakarta.jms.TopicConnectionFactory`
- `jakarta.mail.Session`
- `javax.sql.DataSource`

The user transaction is bound into `java:comp/UserTransaction`.

The `<res-auth>` tag in the `application.xml` deployment descriptor is currently ignored and should be entered as `application`. Oracle does not currently support form-based authentication.

The rest of the client environment is bound from the `weblogic-application-client.xml` file created by the `weblogic.ClientDeployer` utility.

The `weblogic.j2eeclient.Main` class emits error messages for missing or incomplete bindings.

Once the environment is initialized, the `weblogic.j2eeclient.Main` utility searches the JAR manifest of the client JAR for a `Main-Class` entry. The main method on this class is invoked to start the client program. Any arguments passed to the `weblogic.j2eeclient.Main` utility after the URL argument is passed on to the client application.

The client JVM must be able to locate the Java classes you create for your application and any Java classes your application depends upon, including WebLogic Server classes. You stage a client application by copying all of the required files on the client into a directory and bundling the directory in a JAR file. The top level of the client application directory can have a batch file or script to start the application. Create a `classes/` subdirectory to hold Java classes and JAR files, and add them to the client `Class-Path` in the startup script.

You may also want to package a Java Runtime Environment (JRE) with a Java client application.

> **Note**
>
> The use of the Class-Path manifest entries in client module JARs is not portable, as it has not yet been addressed by the Java EE standard.

8

Developing Security-Aware Clients

Learn how to develop WebLogic clients that use the Java Authentication and Authorization Service (JAAS) and Secure Sockets Layer (SSL) to create security-aware clients.
This chapter includes the following sections:

Developing Clients that use JAAS

JAAS enforces access controls based on user identity and is the preferred method of authentication for most WebLogic Server clients. A typical use case is providing authentication to read or write to a file.

For more information about how to implement JAAS authentication, see Using JAAS Authentication in Java Clients in *Developing Applications with the WebLogic Security Service*.

> **Note**
>
> The WLS-IIOP client does not support JAAS. See Developing Clients that use JNDI Authentication.

Developing Clients that use JNDI Authentication

Learn how to develop certificate authentication (also referred to as two-way SSL authentication) using JNDI authentication.

See Using JNDI Authentication in *Developing Applications with the WebLogic Security Service*.

Developing Clients that use SSL

WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between WebLogic Server clients and servers, Java clients, Web browsers, and other servers. All SSL clients need to specify trust. Trust is a set of CA certificates that specify which trusted certificate authorities are trusted by the client.

In order to establish an SSL connection, RMI clients need to trust the certificate authorities that issued the server's digital certificates. The location of the server's trusted CA certificate is specified when starting the RMI client.

> **Note**
>
> WebLogic Server's integration with Java Secure Socket Extension (JSSE) does not use the default `javax.net.ssl.SSLContext` instance or any of the following JVM system properties that define keystore settings:
>
> - `javax.net.ssl.keyStore`
> - `javax.net.ssl.keyStorePassword`
> - `javax.net.ssl.keyStoreType`
> - `javax.net.ssl.trustStore`
> - `javax.net.ssl.trustStorePassword`
> - `javax.net.ssl.trustStoreType`

By default, all trusted certificate authorities available from the JDK (`...\jre\lib\security\cacerts`) are trusted by RMI clients. However, if the server's trusted CA certificate is stored in one of the following trust keystores, you need to specify certain command line arguments in order to use the keystore:

- `Demo Trust`—The trusted CA certificates in the demonstration Trust keystore (`DemoTrust.jks`) are located in the *`WL_HOME`*`\server\lib` directory. In addition, the trusted CAs in the JDK cacerts keystore are trusted. To use the Demo Trust, specify the following command-line argument:

  ```
  -Dweblogic.security.TrustKeyStore=DemoTrust
  ```

  Optionally, use the following command-line argument to specify a password for the JDK cacerts trust keystore:

  ```
  -Dweblogic.security.JavaStandardTrustKeyStorePassPhrase=password
  ```

  where *`password`* is the password for the Java Standard Trust keystore. This password is defined when the keystore is created.

- `Custom Trust`—A trust keystore you create. To use Custom Trust, specify the following command-line arguments.

  Specify the fully qualified path to the trust keystore:

  ```
  -Dweblogic.security.CustomTrustKeyStoreFileName=filename
  ```

  Specify the type of the keystore:

  ```
  -Dweblogic.security.CustomTrustKeyStoreType=jks
  ```

  Optionally, specify the password defined when creating the keystore:

  ```
  -Dweblogic.security.CustomTrustKeyStorePassPhrase=password
  ```

- Oracle's keytool utility can also be used to generate a private key, a self-signed digital certificate for WebLogic Server, and a Certificate Signing Request (CSR). For more information about Oracle's keytool utility, see the keytool-Key and Certificate Management Tool description at https://docs.oracle.com/en/java/javase/17/docs/specs/man/keytool.html.

  For a tutorial on using keytool to create a client certificate, see section "Creating a Client Certificate for Mutual Authentication" in *The Jakarta EE Tutorial*, at https://jakarta.ee/

learn/docs/jakartaee-tutorial/9.1/security/security-advanced/security-advanced.html#_creating_a_client_certificate_for_mutual_authentication.

> **Note**
>
> When using the keytool utility, the default key pair generation algorithm is DSA. WebLogic Server does not support the use of the Digital Signature Algorithm (DSA). Specify another key pair generation and signature algorithm when using WebLogic Server.

You can find more information about how to implement SSL in Configuring SSL and Configuring Keystores in *Administering Security for Oracle WebLogic Server*.

> **Note**
>
> Although JSSE supports Server Name Indication (SNI) in its SSL implementation, WebLogic Server does not support SNI.

Thin Client Restrictions for JAAS and SSL

WebLogic Thin clients only support two-way SSL by requiring the `SSLContext` to be provided by the `SECURITY_CREDENTIALS` property.

WebLogic Thin client applications only support JAAS authentication through the following methods:

- `weblogic.security.auth.login.UsernamePasswordLoginModule.login`
- `weblogic.security.Security.runAs`

To understand how thin clients support two-way SSL using `SSLContext`, see the sample client code below:

Example 8-1 Client Code with sslcontext

```
.
.
.
System.out.println("Getting initial context");
Hashtable props = new Hashtable();
props.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
props.put(Context.PROVIDER_URL,"t3s:/" + host + ":" + port);

props.put(Context.SECURITY_PRINCIPAL,"weblogic");
props.put(Context.SECURITY_CREDENTIALS, "password");

//Set the ssl properties through system property
//set the path to the keystore file (one key inside the store)
System.setProperty("javax.net.ssl.keyStore", YOUR-KEY_STORE_FILE_PATH);
//set the keystore pass phrase
System.setProperty("javax.net.ssl.keyStorePassword",YOUR_KEY_STORE_PASS_PHRASE);

//Set the trust store
//set the path to the trust store file
System.setProperty("javax.net.ssl.trustStore",YOUR-TRUST_STORE_FILE_PATH);
```

```
//set the trust store pass phrase
System.setProperty("javax.net.ssl.trustStorePassword",YOUR_TRUST_STORE_PASS_PHRASE);

Context ctx = new InitialContext(props);
.
.
.
```

Install Client Restrictions for SSL

The WebLogic Install client does not support two-way SSL if your server's trusted CA certificate is stored in a Custom Trust. The client cannot load the Identity keystore using the `weblogic.security.CustomIdentityKeyStoreFileName` property at the command line.

To use two-way SSL with this client, you need to do one of the following: specify a trusted certificate authority that is available from the JDK, use the demonstration Trust keystore, or use Oracle's keytool utility to generate a private key, a self-signed digital certificate for WebLogic Server, and a Certificate Signing Request (CSR).

Security Code Examples

Security samples are optionally provided with the WebLogic Server product. A description of each sample and instructions on how to build, configure, and run a sample, are provided in the `package-summary.html` file.

The samples are located in the *`ORACLE_HOME`*`\wlserver\samples\server\examples\src\examples\security` directory. You can modify these code examples and reuse them. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

9

Using EJBs with RMI-IIOP Clients

Learn how to implement Enterprise JavaBeans that use RMI-IIOP to provide EJB interoperability in heterogeneous server environments.
This chapter includes the following sections:

Accessing EJBs with a Java Client

A Java RMI client uses an ORB and IIOP to access Enterprise beans residing on a WebLogic Server instance.

See Understanding Enterprise JavaBeans in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Accessing EJBs with a CORBA/IDL Client

A non-Java platform CORBA/IDL client can access any Enterprise bean object on WebLogic Server. The sources of the mapping information are the EJB classes as defined in the Java source files. WebLogic Server provides the `weblogic.appc` utility for generating required IDL files.

These files represent the CORBA view into the state and behavior of the target EJB. Use the `weblogic.appc` utility to:

- Place the EJB classes, interfaces, and deployment descriptor files into a JAR file.
- Generate WebLogic Server container classes for the EJBs.
- Run each EJB container class through the RMI compiler to create stubs and skeletons.
- Generate a directory tree of CORBA IDL files describing the CORBA interface to these classes.

The `weblogic.appc` utility supports a number of command qualifiers. See Developing a CORBA/IDL Client.

Resulting files are processed using the compiler, reading source files from the `idlSources` directory and generating CORBA C++ stub and skeleton files. These generated files are sufficient for all CORBA data types with the exception of value types (see Limitations of WebLogic RMI-IIOP in *Developing RMI Applications for Oracle WebLogic Server*.) Generated IDL files are placed in the `idlSources` directory. The Java-to-IDL process is full of pitfalls. Refer to the *Java Language Mapping to OMG IDL* specification at `http://www.omg.org/technology/documents/index.htm`.

Example IDL Generation

The following is an example of how to generate the IDL from a bean you have already created:

1. Generate the IDL files

```
> java weblogic.appc -compiler javac -keepgenerated -idl -idlDirectory idlSources
build\std_ejb_iiop.jar %APPLICATIONS%\ejb_iiop.jar
```

2. Compile the EJB interfaces and client application (the example here uses a CLIENT_CLASSES and APPLICATIONS target variable):

```
> javac -d %CLIENT_CLASSES% Trader.java TraderHome.java TradeResult.java Client.java
```

3. Run the IDL compiler against the IDL files built in Step 1:

```
>%IDL2CPP% idlSources\examples\rmi_iiop\ejb\Trader.idl
. . .

>%IDL2CPP% idlSources\javax\ejb\RemoveException.idl
```

4. Compile your C++ client.

A

Client Application Deployment Descriptor Elements

Learn how to deploy descriptors for Java EE client applications supported by WebLogic Server. This appendix includes the following sections:

Overview of Client Application Deployment Descriptor Elements

Learn how to configure server-side modules by using `application.xml` deployment descriptor and client module using `application-client.xml` deployment descriptor and a WebLogic-specific run time deployment descriptor.

When it comes to Java EE applications, often users are only concerned with the server-side modules (Web applications, EJBs, and connectors). You configure these server-side modules using the `application.xml` deployment descriptor, discussed in Enterprise Application Deployment Descriptor Elements in *Developing Applications for Oracle WebLogic Server*.

However, it is also possible to include a client module (a JAR file) in an EAR file. This JAR file is only used on the client side; you configure this client module using the `application-client.xml` deployment descriptor. This scheme makes it possible to package both client and server side modules together. The server looks only at the parts it is interested in (based on the `application.xml` file) and the client looks only at the parts it is interested in (based on the `application-client.xml` file).

For client-side modules, two deployment descriptors are required: a Java EE standard deployment descriptor, `application-client.xml`, and a WebLogic-specific run time deployment descriptor with a name derived from the client application JAR file.

application-client.xml Deployment Descriptor Elements

The `application-client.xml` file is the deployment descriptor for Java EE client applications.

The `application-client.xml` file must begin with the following DOCTYPE declaration:

```
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD Java EE Application Client 1.2//EN"
"http://java.sun.com/j2ee/dtds/application-client_1_2.dtd">
```

The following sections describe each of the elements that can appear in the file.

application-client

`application-client` is the root element of the application client deployment descriptor. The application client deployment descriptor describes the EJB modules and other resources used by the client application.

The following table describes the elements you can define within an `application-client` element.

Table A-1 application-client Elements

Element	Description
`<icon>`	Optional. Locations of small and large images that represent the application in a GUI tool. This element is not currently used by WebLogic Server.
`<display-name>`	Application display name, a short name that is intended to be displayed by GUI tools.
`<description>`	Optional. Description of the client application.
`<env-entry>`	Contains the declaration of a client application's environment entries. Elements you can define within a `env-entry` element are: • `description`—Optional. Contains a description of the particular environment entry. • `env-entry-name`—Contains the name of a client application's environment entry. • `env-entry-type`—Contains the fully qualified Java type of the environment entry. The possible values are: `java.lang.Boolean`, `java.lang.String`, `java.lang.Integer`, `java.lang.Double`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Long`, and `java.lang.Float`. • `env-entry-value`—Optional. Contains the value of a client application's environment entry. The value must be a String that is valid for the constructor of the specified `env-entry-type`.
`<ejb-ref>`	Used for the declaration of a reference to an EJB referenced in the client application. Elements you can define within an `ejb-ref` element are: • `description`—Optional. Provides a description of the referenced EJB. • `ejb-ref-name`—Contains the name of the referenced EJB. Typically the name is prefixed by `ejb/`, such as `ejb/Deposit`. • `ejb-ref-type`—Contains the expected type of the referenced EJB, either `Session` or `Entity`. • `home`—Contains the fully-qualified name of the referenced EJB's home interface. • `remote`—Contains the fully-qualified name of the referenced EJB's remote interface. • `ejb-link`—Specifies that an EJB reference is linked to an Enterprise Java Bean in the Java EE application package. The value of the`ejb-link` element must be the name of the `ejb-name` of an EJB in the same Java EE application.

Table A-1 (Cont.) application-client Elements

Element	Description
`<resource-ref>`	Contains a declaration of the client application's reference to an external resource. Elements you can define within a `resource-ref` element are: • `description`—Optional. Contains a description of the referenced external resource. • `res-ref-name`—Specifies the name of the resource factory reference name. The resource factory reference name is the name of the client application's environment entry whose value contains the JNDI name of the data source. • `res-type`—Specifies the type of the data source. The type is specified by the Java interface or class expected to be implemented by the data source. • `res-auth`—Specifies whether the EJB code signs on programmatically to the resource manager, or whether the container will sign on to the resource manager on behalf of the EJB. In the latter case, the container uses information that is supplied by the deployer. The res-auth element can have one of two values: `Application` or `Container`.

weblogic-appclient.xml Descriptor Elements

This XML-formatted deployment descriptor is not stored inside of the client application JAR file like other deployment descriptors, but must be in the same directory as the client application JAR file.

The file name for the deployment descriptor is the base name of the JAR file, with the extension `.runtime.xml`. For example, if the client application is packaged in a file named c:/applications/ClientMain.jar, the run-time deployment descriptor is in the file named c:/applications/ClientMain.runtime.xml.

application-client

The `application-client` element is the root element of a WebLogic-specific run-time client deployment descriptor. The following table describes the elements you can define within an `application-client` element.

Table A-2 application-client Elements

Element	Description
`<env-entry>`	Specifies values for environment entries declared in the deployment descriptor. Elements you can define within a `env-entry` element are: • `env-entry-name`—Name of an application client's environment entry. Example: `<env-entry-name>EmployeeAppDB</env-entry-name>` • `env-entry-value`—Value of an application client's environment entry. The value must be a valid String for the constructor of the specified type, which takes a single String parameter.

Table A-2 (Cont.) application-client Elements

Element	Description
`<ejb-ref>`	Specifies the JNDI name for a declared EJB reference in the deployment descriptor. Elements you can define within an `ejb-ref` element are: • `ejb-ref-name`—Name of an EJB reference. The EJB reference is an entry in the application client's environment. Oracle recommends that name is prefixed with`ejb/`. Example: `<ejb-ref-name>ejb/Payroll</ejb-ref-name>.` • `jndi-name`—JNDI name for the EJB.

`<resource-ref>`	Declares an application client's reference to an external resource. It contains the resource factory reference name, an indication of the resource factory type expected by the application client's code, and the type of authentication (bean or container).
	Example:
	`<resource-ref>`
	`<res-ref-name>EmployeeAppDB</res-ref-name>`
	`<jndi-name>enterprise/databases/HR1984</jndi-name>`
	`</resource-ref>`
	Elements you can define within a `resource-ref` element are:
	• `res-ref-name`—Name of the resource factory reference name. The resource factory reference name is the name of the application client's environment entry whose value contains the JNDI name of the data source.
	• `jndi-name`—JNDI name for the resource.

`<resource-description>`	Maps the JNDI name of a server resource to an EJB resource reference in WebLogic Server. Elements you can define within a `resource-description` element are: • `res-ref-name`—Specifies the name of a resource reference. • `jndi-name`—Specifies a JNDI name for the resource.
`<resource-env-description>`	Maps a `resource-env-ref`, declared in the `ejb-jar.xml` deployment descriptor, to the JNDI name of the server resource it represents. Elements you can define within a `resource-env-description` element are: • `res-env-ref-name`—Specifies the name of a resource environment reference. • `jndi-name`—Specifies a JNDI name for the resource environment reference.
`<ejb-reference-description>`	Elements you can define within an `ejb-reference-description` element are: • `ejb-ref-name`—Specifies the name of an EJB reference used in your Web application. • `jndi-name`—Specifies a JNDI name for the reference.

Table A-2 (Cont.) application-client Elements

Element	Description
`<service-reference-description>`	Elements you can define within an `ejb-reference-description` element are: • `service-ref-name` • `wsdl-url` • `call-property`—The `call-property` element has the following sub-elements: – `name` – `value` • `port-info`—The `port-info` element has the following sub-elements: – `port-name` – `stub-property` – `call-property`

B

Accessing WebLogic Server MBeans from JConsole Using WebLogic Install Client JARs

Learn how to access WebLogic Server MBeans from JConsole with the WebLogic thin T3 client (`wlthint3client.jar`) or the install client (`weblogic.jar`).

This appendix includes the following section:

Using JConsole with WebLogic Install Client JARs to Access WebLogic Server MBeans

Use this procedure to access WebLogic Server MBeans from JConsole with WebLogic install client JARs, `wlthint3client.jar` or `weblogic.jar`.

1. From a command prompt, make sure that the JDK is on the path.
2. Invoke JConsole with either the thin T3 client (`wlthint3client.jar`) or the install client (`weblogic.jar`) in the class path:

   - To invoke `wlthint3client.jar`:
     - For UNIX:

       ```
       $ jconsole -J-Djava.class.path=$JAVA_HOME/lib/
       jconsole.jar:$JAVA_HOME/lib/tools.jar:$WL_HOME/server/lib/
       wlthint3client.jar
       ```

     - For Windows:

       ```
       c:> jconsole -J-Djava.class.path=%JAVA_HOME%
       \lib\jconsole.jar;%JAVA_HOME%\lib\tools.jar;%WL_HOME%
       \server\lib\wlthint3client.jar
       ```

   - To invoke `weblogic.jar`:
     - For UNIX:

       ```
       $ jconsole -J-Djava.class.path=$JAVA_HOME/lib/
       jconsole.jar:$JAVA_HOME/lib/tools.jar:$WL_HOME/server/lib/
       weblogic.jar
       ```

     - For Windows:

       ```
       c:> jconsole -J-Djava.class.path=%JAVA_HOME%
       \lib\jconsole.jar;%JAVA_HOME%\lib\tools.jar;%WL_HOME%
       \server\lib\weblogic.jar
       ```

> **Note**
>
> You must explicitly set the classpath using `-J-Djava.class.path=option`. The current classpath is not taken by JConsole.

3. Set up remote connections with an MBean server:

```
$ jconsole -J-Djmx.remote.protocol.provider.pkgs=weblogic.management.remote
```

> **Note**
>
> If you are running JConsole on the same machine as your WebLogic Server instance, then start JConsole simply by running the command `jconsole` at the command line.

4. In the JConsole window, select **Remote Process**.
5. In the **Remote Process** text box, enter the following URL:

```
service:jmx:t3://[host address]:[wls server port]/jndi/
weblogic.management.mbeanservers.domainruntime
```

   where `host:port` represents the host name and port of the WebLogic Server instance that hosts your MBeans. For example, `localhost:7001`.
6. Enter the administrator role credentials for **Username** and **Password** fields.
7. Click **Connect**.
8. Click **Insecure Connection**.