BEA WebLogic Server®

Programming Stand-alone Clients

Version 10.0
Revised: April 2008

Contents

10.Developing Clients for CORBA Objects

11.Developing a WebLogic C++ Client for a Tuxedo ORB

12.Developing Security-Aware Clients

13.Using EJBs with RMI-IIOP Clients

A. Client Application Deployment Descriptor Elements

B. Using the WebLogic JarBuilder Tool

C. Code Examples

CHAPTER 1

Introduction and Roadmap

The following sections describe the contents and organization of this guide—*Programming Stand-alone Clients*:

Document Scope and Audience

This document is a resource for developers who want to create stand-alone client applications that inter-operate with WebLogic Server®.

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 Java Platform, Enterprise Edition (Java EE) 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 stand-alone clients.

Guide to This Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the scope and organization of this guide.
- Chapter 2, "Overview of Stand-alone Clients," describes basic client- server functionality.
- Chapter 3, "Developing T3 Clients," describes how to create T3 clients.
- Chapter 4, "Developing a Java EE Application Client (Thin Client)," describes how to create a Java EE application client.
- Chapter 5, "WebLogic JMS Thin Client," describes how to a create WebLogic JAMS thin client.
- Chapter 6, "Reliably Sending Messages Using the JMS SAF Client," describes how to create a Store-and-Forward client.
- Chapter 7, "Developing a J2SE Client," describes how to create a J2SE client.
- Chapter 8, "Developing a WLS-IIOP Client," provides information on how to create a WebLogic Server-IIOP client.
- Chapter 9, "Developing a CORBA/IDL Client," describes how to create a CORBA/IDL client.
- Chapter 10, "Developing Clients for CORBA Objects," describes how to create a client that inter-operates with CORBA objects.
- Chapter 11, "Developing a WebLogic C++ Client for a Tuxedo ORB," describes how to create a C++ client for the Tuxedo ORB.
- Chapter 12, "Developing Security-Aware Clients," describes how to create a security-aware client.
- Chapter 13, "Using EJBs with RMI-IIOP Clients," describes how to use EJBs with an RMI-IIOP client.
- Appendix A, "Client Application Deployment Descriptor Elements," is a reference for the standard Java EE client application deployment descriptor, `application-client.xml`, and `weblogic-appclient.xml`.
- Appendix B, "Using the WebLogic JarBuilder Tool," provides information on creating the `wlfullclient.jar` using the JarBuilder tool

- Appendix C, "Code Examples," provides BEA examples that demonstrate connectivity between numerous clients and applications. This section includes examples that demonstrate using EJBs with RMI-IIOP, connecting to C++ clients, and setting up interoperability with a Tuxedo Server.

Related Documentation

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

- *Understanding WebLogic RMI* is a guide to using Remote Method Invocation (RMI) and Internet Interop-Orb-Protocol (IIOP) features.
- *Developing Applications with WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- *WebLogic Server Performance and Tuning* contains information on monitoring and improving the performance of WebLogic Server applications.

Samples and Tutorials

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

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

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the *WL_HOME*`\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

MedRec includes a service tier consisting primarily of Enterprise Java Beans (EJBs) that work together to process requests from Web applications, Web services, and workflow applications, and future client applications. The application includes message-driven, stateless session, stateful session, and entity EJBs.

Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in *WL_HOME*`\samples\server\examples\src\examples`, where *WL_HOME* is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server Start menu.

New and Changed Features for This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see "What's New in WebLogic Server 10" in *Release Notes*.

CHAPTER 2

Overview of Stand-alone Clients

In the context of this document, a *stand-alone 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.) Stand-alone 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. The following sections provide an overview:

RMI-IIOP Clients

IIOP can be a transport protocol for distributed applications with interfaces written in Java RMI. For more information, see:

- "WebLogic JMS Thin Client" on page 5-1
- "Reliably Sending Messages Using the JMS SAF Client" on page 6-1
- "Developing a J2SE Client" on page 7-1
- "Developing a WLS-IIOP Client" on page 8-1

For more information, see "Using RMI over IIOP" in *Programming WebLogic RMI*.

BEA T3 (RMI) Clients

A T3 client is a Java RMI client that uses BEA's proprietary T3 protocol to communicate with WebLogic Server. See:

- "Developing T3 Clients" on page 3-1
- Using WebLogic RMI with T3 Protocol in *Programming WebLogic RMI*.

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 execute 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" on page 9-1
- "Developing Clients for CORBA Objects" on page 10-1
- "Developing a WebLogic C++ Client for a Tuxedo ORB" on page 11-1

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 with JMX*.

WebServices Clients

A stand-alone WebServices client uses WebLogic client classes to invoke a Web Service hosted on WebLogic Server or on other application servers. See Invoking a Web Service from a Stand-alone Client in *Programming Web Services for WebLogic Server*.

Client Types and Features

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

Note: In this release, client applications should use the `wlfullclient.jar` file to provide the WebLogic Server specific functionary previously provided in the `weblogic.jar` file. You can generate the `wlfullclient.jar` file for client applications using the JarBuilder tool. See "Using the WebLogic JarBuilder Tool" on page B-1.

Table 2-1 WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements	Key Features
Java EE Application Client (Thin Client) (Introduced in WebLogic Server 8.1)	RMI	Java	IIOP	• `wlclient.jar` • JDK 1.4 and higher	• Supports WLS clustering. • Supports many Java EE features, including security and transactions. • Supports SSL. • Uses CORBA 2.4 ORB. • See Chapter 4, "Developing a Java EE Application Client (Thin Client)."
JMS Thin Client (Introduced in WebLogic Server 8.1)	RMI	Java	IIOP	• `wljmsclient.jar` • `wlclient.jar` • JDK 1.4 and higher	• Thin client functionality • WebLogic JMS, except for client-side XML selection for multicast sessions and JMSHelper class methods. • See Chapter 5, "WebLogic JMS Thin Client."

Table 2-1 WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements	Key Features
JMS SAF Client	RMI	• Java	IIOP	• `wlsafclient.jar` • `wljmsclient.jar` • `wlclient.jar` • JDK 1.4 and higher	• Locally stores messages on the client and forwards them to server-side JMS destinations when the client is connected. • See Chapter 6, "Reliably Sending Messages Using the JMS SAF Client."
T3	RMI	Java	T3	`wlfullclient.jar`	• Supports WLS-Specific features. • Fast, scalable. • No CORBA inter-operability. • See Chapter 3, "Developing T3 Clients."
J2SE	RMI	Java	IIOP	no WebLogic classes	• Provides connectivity to WLS environment. • Does not support WLS-specific features. Does not support many Java EE features. • Uses CORBA 2.3 ORB. • Requires use of `com.sun.jndi.cosnaming`. CNCtxFactory. • See Chapter 7, "Developing a J2SE Client."

Table 2-1 WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements	Key Features
WLS-IIOP (Introduced in WebLogic Server 7.0)	RMI	Java	IIOP	`wlfullclient.jar`	• Supports WLS-Specific features. • Supports SSL. • Fast, scalable. • Not ORB-based. • See Chapter 8, "Developing a WLS-IIOP Client."
JMX	RMI	Java	IIOP	`wljmxclient.jar`	See Accessing WebLogic Server MBeans with JMX.
WebServices	SOAP	Java	HTTP/S	`wseeclient.jar`	See Invoking a Web Service from a Stand-alone 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 WLS-specific features. • Does not support Java. • See Chapter 9, "Developing a CORBA/IDL Client."

Table 2-1 WebLogic Server Client Types and Features

Client	Type	Language	Protocol	Client Class Requirements	Key Features
C++ Client	CORBA	C++	IIOP	Tuxedo libraries	• Interoperability between WLS applications and Tuxedo clients/services. • Supports SSL. • Uses CORBA 2.3 ORB. • See Chapter 11, "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 WLS applications and Tuxedo clients/services. • Supports SSL and transactions. • Uses CORBA 2.3 ORB. • See Chapter 10, "Developing Clients for CORBA Objects."

WebLogic JarBuilder Tool

For WebLogic Server 10.0 and higher releases, client applications need to use the `wlfullclient.jar` file to provide the WebLogic Server specific functionary previously provided in the `weblogic.jar` file. You can generate the `wlfullclient.jar` file for client applications using the JarBuilder tool. See "Using the WebLogic JarBuilder Tool" on page B-1.

CHAPTER 3

Developing T3 Clients

A T3 client is an RMI client that uses BEA's proprietary T3 protocol to communicate with WebLogic server instance.

The following sections provide information on developing T3 clients:

T3 Client Basics

A T3 client:

- Is an RMI client that uses the Java-to-Java model of distributed computing. For information on developing RMI applications, see Understanding WebLogic RMI. You cannot integrate clients written in languages other than Java.
- Uses BEA's proprietary T3 protocol to communicate with Java programs. The URL used for the initial context takes the form `t3://`*`ip address`*`:`*`port`*.
- Requires the `wlfullclient.jar` in your classpath.
- Supports WebLogic Server specific features. See Table 2-1.

Developing a T3 Client

Creating a basic T3 client consists of the following

1. Obtain a reference to the remote object.
   a. Get the initial context of the server that hosts the service using a T3 URL.
   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.

Sample code to for a simple T3 client is provided in Listing 3-1.

Listing 3-1 Simple T3 hello Client

```
package examples.rmi.hello;

import java.io.PrintStream;
import weblogic.utils.Debug;
import javax.naming.*;
import java.util.Hashtable;

/**
* This client uses the remote HelloServer methods.
*
* @author Copyright (c) 1999-2004 by BEA Systems, Inc. All Rights Reserved.
*/
public class HelloClient {

private final static boolean debug = true;

/**
* Defines the JNDI context factory.
*/
public final static String
JNDI_FACTORY="weblogic.jndi.WLInitialContextFactory";

int port;
```

```
String host;

   private static void usage() {
   System.err.println("Usage: java examples.rmi.hello.HelloClient " +
      "<hostname> <port number>");
   System.exit(-1);
   }

   public HelloClient() {}

   public static void main(String[] argv) throws Exception {
      if (argv.length < 2) {
      usage();
     }
   String host = argv[0];
   int port = 0;
   try {
      port = Integer.parseInt(argv[1]);
      }
   catch (NumberFormatException nfe) {
      usage();
      }
   try {

      InitialContext ic = getInitialContext("t3://" + host + ":" + port);

      Hello obj =
         (Hello) ic.lookup("HelloServer");
      System.out.println("Successfully connected to HelloServer on " +
         host + " at port " +
         port + ": " + obj.sayHello() );
      }
   catch (Throwable t) {
      t.printStackTrace();
      System.exit(-1);
      }
```

```
  }

   private static InitialContext getInitialContext(String url)
      throws NamingException
   {
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
   env.put(Context.PROVIDER_URL, url);
   return new InitialContext(env);
   }
}
```

RMI Communication in WebLogic Server

RMI communications in WebLogic Server use the T3 protocol to transport data between WebLogic Server and other Java programs, including clients and other WebLogic Server instances. A server instance keeps track of each Java Virtual Machine (JVM) with which it connects, and creates a single T3 connection to carry all traffic for a JVM. See Configure T3 protocol in *Administration Console Online Help*.

For example, if a Java client accesses an enterprise bean and a JDBC connection pool on WebLogic Server, a single network connection is established between the WebLogic Server JVM and the client JVM. The EJB and JDBC services can be written as if they had sole use of a dedicated network connection because the T3 protocol invisibly multiplexes packets on the single connection.

Determining Connection Availability

Any two Java programs with a valid T3 connection—such as two server instances, or a server instance and a Java client—use periodic point-to-point "heartbeats" to announce and determine continued availability. Each end point periodically issues a heartbeat to the peer, and similarly, determines that the peer is still available based on continued receipt of heartbeats from the peer.

- The frequency with which a server instance issues heartbeats is determined by the *heartbeat interval*, which by default is 60 seconds.

- The number of missed heartbeats from a peer that a server instance waits before deciding the peer is unavailable is determined by the *heartbeat period*, which by default, is 4. Hence, each server instance waits up to 240 seconds, or 4 minutes, with no messages—either heartbeats or other communication—from a peer before deciding that the peer is unreachable.
- Changing timeout defaults is not recommended.

Communicating with a Server in Admin Mode

To communicate with a server instance that is in `admin` mode, you need to configure a communication channel by setting the following flag on your client:

```
-Dweblogic.AdministrationProtocol=t3
```

CHAPTER 4

Developing a Java EE Application Client (Thin Client)

A Java EE application client runs on a client machine and can provide a richer user interface than can be provided by a markup language. Application clients directly access enterprise beans running in the business tier, and may, as appropriate, communicate via HTTP with servlets running in the Web tier. An application client is typically downloaded from the server, but can be installed on a client machine.

The following sections provide information on developing Java EE clients:

Overview of the Java EE Application Client

Although a Java EE application client (thin client) is a Java application, it differs from a stand-alone Java application client because it is a Java EE component, hence it offers the advantages of portability to other Java EE-compliant servers, and can access Java EE services.

BEA provides the following application client JAR files:

- A standard client JAR (`wlclient.jar`) that provides Java EE functionality. See "How to Develop a Thin Client" on page 4-3.

- A JMS client JAR (`wljmsclient.jar`), which when deployed with the `wlclient.jar`, provides Java EE and WebLogic JMS functionality. See "WebLogic JMS Thin Client" on page 5-1.
- A JMS SAF client JAR (`wlsafclient.jar`), which when deployed with the `wljmsclient.jar` and `wlclient.jar` enables standalone JMS clients to reliably send messages to server-side JMS destinations, even when a destination is temporarily unreachable. Sent messages are stored locally on the client and are forwarded to the destination when it becomes available. See "Reliably Sending Messages Using the JMS SAF Client" on page 6-1.

These application client JAR files reside in the `WL_HOME/server/lib` subdirectory of the WebLogic Server installation directory.

The thin client uses the RMI-IIOP protocol stack and leverages features of J2SE 1.4 It also requires the support of the JDK ORB. The basics of making RMI requests are handled by the JDK, which makes possible a significantly smaller client. Client-side development is performed using standard Java EE APIs, rather than WebLogic Server APIs.

The development process for a thin client application is the same as it is for other Java EE applications. The client can leverage standard Java EE artifacts such as InitialContext, UserTransaction, and EJBs. The WebLogic Server thin client supports these values in the protocol portion of the URL—IIOP, IIOPS, HTTP, HTTPS, T3, and T3S—each of which can be selected by using a different URL in InitialContext. Regardless of the URL, IIOP is used. URLs with T3 or T3S use IIOP and IIOPS respectively. HTTP is tunnelled IIOP, HTTPS is IIOP tunnelled over HTTPS.

Server-side components are deployed in the usual fashion. Client stubs can be generated at either deployment time or runtime.To generate stubs when deploying, run `appc` with the `-iiop` and `-basicClientJar` options to produce a client jar suitable for use with the thin client. Otherwise, WebLogic Server generates stubs on demand at runtime and serves them to the client. Downloading of stubs by the client requires that a suitable security manager be installed. The thin client provides a default light-weight security manager. For rigorous security requirements, a different security manager can be installed with the command line options `-Djava.security.manager -Djava.security.policy==policyfile`. Applets use a different security manager which already allows the downloading of stubs.

The thin client JAR replaces some classes in `wlfullclient.jar`. If both the full JAR and the thin client JAR are in the CLASSPATH, the thin client JAR should be first in the path. Note, however, that `wlfullclient.jar` is not required to support the thin client. If desired, you can use this syntax to run with an explicit CLASSPATH:

```
java -classpath "<WL_HOME>/lib/wlclient.jar;<CLIENT_CLASSES>"
your.app.Main
```

Note: `wljmsclient.jar` has a reference to `wlclient.jar` so it is only necessary to put one or the other Jar in the client CLASSPATH. Do not put the `wljmsclient.jar` in the server-side CLASSPATH.

The thin client jar contains the necessary Java EE interface classes, such as `javax.ejb`, no other jar files are necessary on the client.

How to Develop a Thin Client

To develop a thin client:

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. Here is an excerpt from the implementation class developed from the previous Ping 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 from doing so in normal RMI. For more information on developing RMI objects, see "Understanding WebLogic RMI".

4. Run the WebLogic RMI or EJB compiler against the implementation class to generate the necessary IIOP stub.

   Note: If you plan on downloading stubs, it is not necessary to run `rmic`.

   ```
   $ java weblogic.rmic -iiop nameOfImplementationClass
   ```

   To generate stubs when deploying, run `appc` with the `-iiop` and `-clientJar` options to produce a client JAR suitable for use with the thin client. Otherwise, WebLogic Server will generate stubs on demand at runtime and serve them to the client.

   A stub is the client-side proxy for a remote object that forwards each WebLogic RMI call to its matching server-side skeleton, which in turn forwards the call to the actual remote object implementation.

5. Make sure that the files you have created—the remote interface, the class that implements it, and the stub—are in the CLASSPATH of WebLogic Server.

6. Obtain an initial context.

   RMI clients access remote objects by creating an initial context and performing a lookup (see next step) on the object. The object is then cast to the appropriate type.

   In obtaining an initial context, you must use `weblogic.jndi.WLInitialContextFactory` when defining your JNDI context factory. Use this class when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`.

7. Modify the client code to perform the lookup in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

   RMI over IIOP RMI clients differ from regular RMI clients in that IIOP is defined as the protocol when obtaining an initial context. Because of this, lookups and casts must be performed in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method. For example, an RMI client creates an initial context, performs a lookup on the EJBean home, obtains a reference to an EJBean, and calls methods on the EJBean.

   You must use the `javax.rmi.PortableRemoteObject.narrow()` method in any situation where you would normally cast an object to a specific class type. A CORBA client may return an object that does not implement your remote interface; the narrow method is provided by your ORB to convert the object so that it implements your remote interface. For example, the client code responsible for looking up the EJBean home and

casting the result to the Home object must be modified to use the javax.rmi.PortableRemoteObject.narrow() as shown below:

Performing a lookup:

```
.
.
.
/**
 * RMI/IIOP clients should use this narrow function
 */
private Object narrow(Object ref, Class c) {
  return PortableRemoteObject.narrow(ref, c);
}

/**
 * Lookup the EJBs home in the JNDI tree
 */
private TraderHome lookupHome()
  throws NamingException
{
  // Lookup the beans home using JNDI
  Context ctx = getInitialContext();

  try {
Object home = ctx.lookup(JNDI_NAME);
return (TraderHome) narrow(home, TraderHome.class);
} catch (NamingException ne) {
log("The client was unable to lookup the EJBHome.  Please
make sure ");
log("that you have deployed the ejb with the JNDI name
"+JNDI_NAME+" on the WebLogic server at "+url);
throw ne;
  }
}

/**
 * Using a Properties object will work on JDK130
 * and higher clients
 */
private Context getInitialContext() throws NamingException {

  try {
// Get an InitialContext
Properties h = new Properties();
h.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
h.put(Context.PROVIDER_URL, url);
return new InitialContext(h);
  } catch (NamingException ne) {
log("We were unable to get a connection to the WebLogic
```

```
server at "+url);
log("Please make sure that the server is running.");
throw ne;
  }
}
.
.
.
```

The `url` defines the protocol, hostname, and listen port for the WebLogic Server instance and is passed in as a command-line argument.

```
public static void main(String[] args) throws Exception {

  log("\nBeginning statelessSession.Client...\n");

  String url = "iiop://localhost:7001";
```

8. Connect the client to the server over IIOP by running the client with a command such as:

```
$ java -Djava.security.manager -Djava.security.policy=java.policy
  examples.iiop.ejb.stateless.rmiclient.Client iiop://localhost:7001
```

Using Java EE Client Application Modules

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 with 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" on page A-1.

Extracting 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 execution.
- `weblogic.j2eeclient.Main`—Executes the client code.

You use the `weblogic.ClientDeployer` utility to extract the client-side JAR file from a Java EE EAR file, creating a deployable JAR file. Execute 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 "weblogic-appclient.xml Descriptor Elements" on page A-5.

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.

Executing a Client Application

Once 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 clientjar URL [application 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.

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`
- `javax.jms.QueueConnectionFactory`
- `javax.jms.TopicConnectionFactory`
- `javax.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`. BEA 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.

Protocol Compatibility

Interoperability between WebLogic Server thin clients is supported in the following scenarios.

Table 4-1 Thin Client Inter-operability

From \ To	WebLogic Server 8.1 (JDK 1.4)	WebLogic Server 9.x (JDK 1.5)	WebLogic Server 10.0 (JDK 1.5)
WebLogic Server 8.1 `wlclient.jar` (JDK 1.4)	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS
WebLogic Server 8.1 `wljmsclient.jar` (JDK 1.4)	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS
WebLogic Server 9.x `wlclient.jar` (JDK 1.5)	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS
WebLogic Server 9.x `wljmsclient.jar` (JDK 1.5)	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS

Table 4-1 Thin Client Inter-operability

From / To	WebLogic Server 8.1 (JDK 1.4)	WebLogic Server 9.x (JDK 1.5)	WebLogic Server 10.0 (JDK 1.5)
WebLogic Server 10.0 `wlclient.jar` (JDK 1.5)	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS
WebLogic Server 10.0 `wljmsclient.jar` (JDK 1.5)	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS	IIOP, IIOPS, HTTP, HTTPS

CHAPTER 5

WebLogic JMS Thin Client

The following sections describe how to deploy and use the WebLogic JMS thin client:

Overview of the JMS Thin Client

The JMS thin client (the `wljmsclient.jar` deployed with the `wlclient.jar`), provides Java EE and WebLogic JMS functionality using a much smaller client footprint than the full WebLogic JAR. The smaller footprint is obtained by using:

- A client-side library that contains only the set of supporting files required by client-side programs.
- The RMI-IIOP protocol stack available in the JRE. RMI requests are handled by the JRE, enabling a significantly smaller client.
- Standard Java EE APIs, rather than WebLogic Server APIs.

For more information on developing WebLogic Server thin client applications, see "Developing a Java EE Application Client (Thin Client)" on page 4-1.

JMS Thin Client Functionality

Although much smaller in size than the full WebLogic JAR, the JMS thin client (the `wljmsclient.jar` and `wlclient.jar`) provide the following functionality to client applications and applets:

- Full WebLogic JMS functionality—both standard JMS and WebLogic extensions—except for client-side XML selection for multicast sessions and the JMSHelper class methods
- EJB (Enterprise Java Bean) access
- JNDI access
- RMI access (indirectly used by JMS)
- SSL access (using JSSE in the JRE)
- Transaction capability
- Clustering capability
- HTTP/HTTPS tunneling
- Fully internationalized

Limitations of Using the JMS Thin Client

The following limitations apply to the JMS thin client:

- It does not provide the JDBC or JMX functionality of the normal `wlfullclient.jar` file.
- It does not inter-operate with WebLogic Server 7.0 or earlier.
- It is only supported by the JDK ORB.
- It has lower performance than the thick client, especially with non-persistent messaging.

Deploying the JMS Thin Client

The `wljmsclient.jar` and `wlclient.jar` 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 Platform (for example, `c:\bea\wlserver_10.0\server\lib`).

Deployment of the JMS thin client depends on the following requirements:

- The JMS thin client requires the standard thin client, which contains the base client support for clustering, security, and transactions. Therefore, the `wljmsclient.jar` and the `wlclient.jar` must be installed somewhere on the client's file system. However, `wljmsclient.jar` has a reference to `wlclient.jar` so it is only necessary to put one or the other Jar in the client's `CLASSPATH`.
- RMI-IIOP is required for client-server communication.
  - URLs using `t3` or `t3s` will transparently use `iiop` or `iiops`
  - URLs using `http` or `https` will transparently use `iiop` tunneling.
- To facilitate the use of IIOP, always specify a valid IP address or DNS name for the Listen Address attribute to listen for connections.

  Note: The Listen Address default value of `null` allows it to "listen on all configured network interfaces". However, this feature only works with the T3 protocol. If you need to configure multiple listen addresses for use with the IIOP protocol, then use the Network Channel feature, as described in Configuring Network Resources in *Configuring WebLogic Server Environments*.
- Each client must have the JRE 1.4.*n* or higher installed.
- Applications must adhere to Java EE programming guidelines, in particular the use of `PortableRemoteObject.narrow()` rather than using casts.

For more information on developing thin client applications for WebLogic Server, see "Developing a Java EE Application Client (Thin Client)" on page 4-1.

CHAPTER 6

Reliably Sending Messages Using the JMS SAF Client

The following sections describe how to configure and use the JMS SAF Client feature to reliably send JMS messages from standalone JMS clients to server-side JMS destinations:

Overview of Using Store-and-Forward with JMS Clients

The JMS SAF Client feature extends the JMS store-and-forward service introduced in WebLogic Server 9.0 to standalone JMS clients. Now 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 feature consists of two main parts: the JMS SAF client implementation that writes messages directly to a client-side persistent store on the local file system and a SAF forwarder that takes the messages written to the store and sends them to a WebLogic Server instance. There is also an optional `SAFClient` initialization API in `weblogic.jms.extensions` that allows JMS SAF clients to turn the SAF forwarder mechanism on and off whenever necessary. For more information, see "The JMS SAF Client Initialization API" on page 6-14.

Note: For information on the server-side WebLogic JMS SAF for reliably sending JMS messages to potentially unavailable destinations, see Configuring SAF for JMS Messages in *Configuring and Managing WebLogic Store-and-Forward*.

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. 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 installation.

The `ClientSAFGenerate` utility creates entries for all connection factories, stand-alone 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" on page 6-3. 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" on page 6-12. 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-jmsmd.xsd` schema for JMS modules and contains the root element `weblogic-client-jms`. The `weblogic-jmsmd.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" on page 6-6.

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 fowarder.

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 domain. You can also generate a configuration file from an existing JMS SAF client configuration file, as described in "ClientSAFGenerate Utility Syntax" on page 6-5.

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 `wlfullclient.jar` in the `CLASSPATH` instead of the thin JMS and JMS SAF clients. See "Installing the JMS SAF Client JAR Files on Client Machines" on page 6-11.

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 domain containing the JMS module file that you want to use as the basis for the JMS SAF client configuration file:

   ```
   c:\bea\wlserver_10.0\samples\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 weblogic -moduleFile examples-jms.xml
   -outputFile d:\temp\ClientSAF-jms.xml
   ```

   Table 6-1 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" on page 6-5.

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" on page 6-9.
5. Copy the generated configuration can file to the client machine(s) where you will run your JMS SAF client applications. See "Installing the JMS SAF Client JAR Files on Client Machines" on page 6-11.

   Note: `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, as described in "Modify Your JMS Client Applications To Use the JMS SAF Client's Initial JNDI Provider" on page 6-12.

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 6-1 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.

Table 6-1 ClientSAFGenerate Arguments

Argument	Definition
`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`	The path to the generated output file. If a path is not specified, the utility sends its output to `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-jmsmd.xsd` schema contains several top-level elements that correspond to server-side WebLogic JMS SAF features. Table 6-2 makes clear what the relationship between the top-level element in the schema and the corresponding management MBean.

Table 6-2 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`

Table 6-2 weblogic-client-saf Elements

weblogic-client-jms Element	WebLogic Server Management Bean
`saf-error-handling`	`SAFErrorHandlingBean`
`persistent-store`	For more information, see "Default Store Options for JMS SAF Clients" on page 6-9.

Caution: 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 6-3 describes the differences between the standard `SAFAgentMBean` fields and the fields in the JMS SAF client configuration file.

Table 6-3 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.

Table 6-3 Modified SAFAgentMBean Fields

Server-side SAF Fields	Difference in JMS SAF Client Configuration File
`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.

Caution: You can only specify one `saf-agent` element in a JMS SAF client configuration file.

Table 6-4 describes the differences between the standard `JMSConnectionFactoryBean` fields and the fields in the JMS SAF client configuration file.

Table 6-4 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.
`FlowControlParamsBean`	All fields are ignored. JMS SAF client cannot receive messages.
`LoadBalancingParamsBean`	All fields are ignored. JMS SAF client cannot load balance since it is not connected to a server.

Table 6-5 describes the differences between the standard `SAFImportedDestinationsBean` fields and the fields in the JMS SAF client configuration file.

Table 6-5 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, and which 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 6-6 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.

Caution: You can only specify one `persistent-store` element in a JMS SAF client configuration file.

Here's 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 *Configuring WebLogic Server Environments*.

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 installation, which encrypts cleartext 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 the `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. Here's an example of obtaining an encrypted password:

   ```
   Password Key ("quit" to end):
   Password ("quit" to end):
   <password-encrypted>{Algorithm}PBEWithMD5AndDES{Salt}9IsTPAuZdcQ={Data}
   d6SSPp3GwPAfEXn8izyZA0IRCV/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 cleartext 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 6-7 ClientSAFEncrypt Arguments

Argument	Definition
`weblogic.management.allow PasswordEcho`	Optional. Allows echoing characters entered on the command line. `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`	Cleartext 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

How you install the JMS SAF client depends on whether your client machines require smaller `JAR` files (thin clients) or whether they can accommodate using the single, higher-performing `wlfullclient.jar` file, which contains all the necessary functionality and is also the recommended best practice.

The required WebLogic 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 product installation (for example, c:\bea\weblogic92\server\lib).

When smaller JAR sizes are required for thin clients, the JMS SAF client requires installing the following JAR files to a directory on the client machine's file system and added to its CLASSPATH:

- wlsafclient.jar
- wljmsclient.jar
- wlclient.jar

The wljmsclient.jar has a reference to the wlclient.jar so it is only necessary to put one or the other JAR in the client machine's CLASSPATH.

Again, the recommended best practice is to use the larger, higher-performing wlfullclient.jar, which must be installed to a directory on the client machine's file system and added to its CLASSPATH. Using the wlfullclient.jar file also allows you to run the ClientSAFGenerate utility on a client machine to generate a configuration file from an existing JMS SAF client configuration file, as described in "Steps to Generate a JMS SAF Client Configuration File from a JMS Module" on page 6-3.

For more information on deploying thin clients, see Chapter 4, "Developing a Java EE Application Client (Thin Client)."

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" on page 6-3.

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 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" on page 6-15.

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" on page 6-14.

Optional JNDI Properties for JMS SAF Clients

There are also two optional JMS SAF client JNDI properties:

- `Context.PROVIDER_URL` – This must be an `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

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. For more information, see Administering a Persistent Store in *Configuring WebLogic Server Environments*.

JMS Programming Considerations with JMS SAF Clients

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 *Programming WebLogic JMS*.

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.

Java Runtime

Each client machine must have J2SE 1.4 runtime or higher 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" on page 6-11.

Tuning JMS SAF Clients

JMS SAF clients can take advantage of the tuning parameters available with the server-side SAF service. For more information, see Tuning WebLogic JMS Store-and-Forward in the *WebLogic Performance and Tuning Guide*.

Limitations of Using the JMS SAF Client

In addition to the field-level limitations discussed in "Valid SAF Elements for JMS SAF Client Configurations" on page 6-6, 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" on page 6-15.
- 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.

CHAPTER 7

Developing a J2SE Client

A J2SE client is oriented towards the Java EE programming model; it combines the capabilities of RMI with the IIOP protocol without requiring WebLogic Server classes. The following sections provide information on developing a J2SE Client:

J2SE Client Basics

A J2SE client runs an RMI-IIOP-enabled ORB hosted by a Java EE or J2SE container, in most cases a 1.3 or higher JDK. A J2SE client has the following characteristics:

- It provides a light-weight connectivity client that uses the IIOP protocol, an industry standard.
- It is a J2SE-compliant model, rather than a Java EE-compliant model—it does not support many of the features provided for enterprise-strength applications. It does not support security, transactions, or JMS.

How to Develop a J2SE Client

To develop an application using RMI-IIOP with an RMI client:

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 "Understand ing WebLogic RMI".

4. Run the WebLogic RMI or EJB compiler against the implementation class to generate the necessary IIOP stub. Note that it is no longer necessary to use the `-iiop` option to generate the IIOP stubs:

   ```
   $ java weblogic.rmic nameOfImplementationClass
   ```

   A stub is the client-side proxy for a remote object that forwards each WebLogic RMI call to its matching server-side skeleton, which in turn forwards the call to the actual remote object implementation. Note that the IIOP stubs created by the WebLogic RMI compiler are intended to be used with the JDK 1.3.1_01 or higher ORB. If you are using another ORB, consult the ORB vendor's documentation to determine whether these stubs are appropriate.

5. Make sure that the files you have now created -- the remote interface, the class that implements it, and the stub -- are in the CLASSPATH of WebLogic Server.

6. Obtain an initial context.

   RMI clients access remote objects by creating an initial context and performing a lookup (see next step) on the object. The object is then cast to the appropriate type.

   In obtaining an initial context, you must use `com.sun.jndi.cosnaming.CNCtxFactory` when defining your JNDI context factory. (WLInitialContextFactory is deprecated for this client in WebLogic Server 8.1) Use `com.sun.jndi.cosnaming.CNCtxFactory` when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`.

   Note: The Sun JNDI client supports the capability to read remote object references from the namespace, but not generic Java serialized objects. This means that you can read items such as EJBHome out of the namespace but not DataSource objects. There is also no support for client-initiated transactions (the JTA API) in this configuration, and no support for security. In the stateless session bean RMI Client example, the client obtains an initial context as is done below:

   `Obtaining an InitialContext:`

```
.
.
.
* Using a Properties object as follows will work on JDK13
* and higher clients.

   */

  private Context getInitialContext() throws NamingException {

try {
  // Get an InitialContext
  Properties h = new Properties();
  h.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.cosnaming.CNCtxFactory");
  h.put(Context.PROVIDER_URL, url);
  return new InitialContext(h);
} catch (NamingException ne) {
  log("We were unable to get a connection to the WebLogic server at
  "+url);
  log("Please make sure that the server is running.");
  throw ne;
  }

/**

* This is another option, using the Java2 version to get an
* InitialContext.
* This version relies on the existence of a jndi.properties file in
```

```
* the application's classpath. See
* Programming WebLogic JNDI for more information

private static Context getInitialContext()
  throws NamingException

{
  return new InitialContext();
}
.
.
.
```

7. Modify the client code to perform the lookup in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

   RMI-IIOP clients differ from regular RMI clients in that IIOP is defined as the protocol when the client is obtaining an initial context. Because of this, lookups and casts must be performed in conjunction with the `javax.rmi.PortableRemoteObject.narrow()` method.

   For example, an RMI client creates an initial context, performs a lookup on the EJBean home, obtains a reference to an EJBean, and calls methods on the EJBean.

   You must use the `javax.rmi.PortableRemoteObject.narrow()` method in any situation where you would normally cast an object to a specific class type. A CORBA client may return an object that does not implement your remote interface; the narrow method is provided by your orb to convert the object so that it implements your remote interface. For example, the client code responsible for looking up the EJBean home and casting the result to the `Home` object must be modified to use the `javax.rmi.PortableRemoteObject.narrow()` as shown below:

   `Performing a lookup:`

```
.
.
.
/**
 * RMI/IIOP clients should use this narrow function
 */
private Object narrow(Object ref, Class c) {
  return PortableRemoteObject.narrow(ref, c);
}

/**
 * Lookup the EJBs home in the JNDI tree
 */
private TraderHome lookupHome()
  throws NamingException
```

```
{
  // Lookup the beans home using JNDI
  Context ctx = getInitialContext();

  try {
Object home = ctx.lookup(JNDI_NAME);
return (TraderHome) narrow(home, TraderHome.class);
} catch (NamingException ne) {
log("The client was unable to lookup the EJBHome.  Please
make sure ");
log("that you have deployed the ejb with the JNDI name
"+JNDI_NAME+" on the WebLogic server at "+url);
throw ne;
  }
}

/**
 * Using a Properties object will work on JDK130
 * and higher clients
 */
private Context getInitialContext() throws NamingException {

  try {
// Get an InitialContext
Properties h = new Properties();
h.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.cosnaming.CNCtxFactory");
h.put(Context.PROVIDER_URL, url);
return new InitialContext(h);
  } catch (NamingException ne) {
log("We were unable to get a connection to the WebLogic
server at "+url);
log("Please make sure that the server is running.");
throw ne;
  }
}
.
.
.
```

The `url` defines the protocol, hostname, and listen port for the WebLogic Server and is passed in as a command-line argument.

```
public static void main(String[] args) throws Exception {

  log("\nBeginning statelessSession.Client...\n");

  String url       = "iiop://localhost:7001";
```

8. Connect the client to the server over IIOP by running the client with a command such as:

```
$ java -Djava.security.manager -Djava.security.policy=java.policy
  examples.iiop.ejb.stateless.rmiclient.Client iiop://localhost:7001
```

9. Set the security manager on the client:

   ```
   java -Djava.security.manager -Djava.security.policy==java.policy
   myclient
   ```

   To narrow an RMI interface on a client, the server needs to serve the appropriate stub for that interface. The loading of this class is predicated on the use of the JDK network classloader and this is **not** enabled by default. To enable it you set a security manager in the client with an appropriate java policy file. For more information on Java security, see Sun's site at `http://java.sun.com/security/index.html`. The following is an example of a `java.policy` file:

   ```
   grant {
   // Allow everything for now
   permission java.security.AllPermission;
   }
   ```

CHAPTER 8

Developing a WLS-IIOP Client

The WebLogic Server-IIOP client is a non-ORB based JS2E client that provides WebLogic Server-specific features. The following sections provide information on developing WLS-IIOP clients:

WLS-IIOP Client Features

The WLS-IIOP client supports WebLogic Server specific features, including

- Clustering
- .SSL
- Scalability

For more information, see "Client Types and Features" on page 2-3.

How to Develop a WLS-IIOP Client

The procedure for developing a WLS-IIOP Client is the same as the procedure described in "Developing a J2SE Client" on page 7-1 with the following additions:

- Include the full `wlfullclient.jar` (located in *`WL_HOME`*`/server/lib`) in the client's CLASSPATH.

- Use `weblogic.jndi.WLInitialContextFactory` when defining your JNDI context factory. Use this class when setting the value for the "`Context.INITIAL_CONTEXT_FACTORY`" property that you supply as a parameter to new `InitialContext()`.
- You do not need to use the `-D weblogic.system.iiop.enableClient=true` command line option to enable client access when starting the client. By default, if you use `wlfullclient.jar`, `enableClient` is set to true.

CHAPTER 9

Developing a CORBA/IDL Client

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. The following sections provide information on how to develop clients for heterogeneous distributed applications:

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 Mapping to OMG IDL Specification at http://www.omg.org/cgi-bin/doc?formal/01-06-07

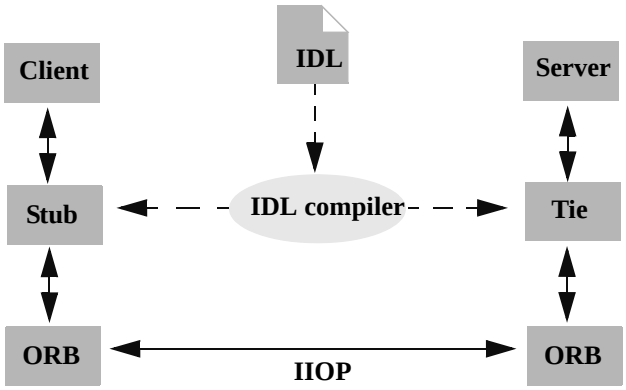
- CORBA/IIOP 2.4.2 Specification at http://www.omg.org/cgi-bin/doc?formal/01-02-33

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 BEA does not recommend it.

The following figure shows how IDL takes part in a RMI-IIOP model.

Figure 9-1 IDL Client (Corba object) relationships



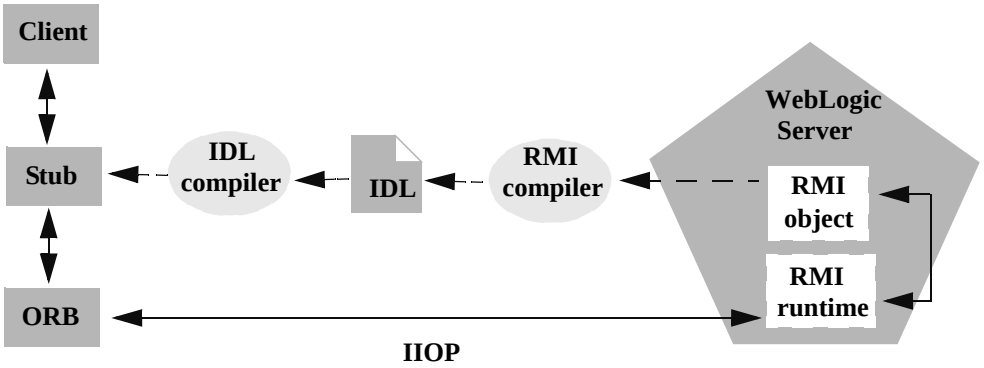
Java to IDL Mapping

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 runtime environment.

The following figure shows this process.

Figure 9-2 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 BEA Systems has tested with respect to Objects-by-Value support:

Table 9-1 ORBs Tested with Respect to Objects-by-Value Support

Vendor	Versions	Objects-by-Value
BEA	Tuxedo 8.x C++ Client ORB	Supported
Borland	VisiBroker 3.3, 3.4	Not supported

Table 9-1 ORBs Tested with Respect to Objects-by-Value Support

Vendor	Versions	Objects-by-Value
Borland	VisiBroker 4.x, 5.x	Supported
Iona	Orbix 2000	Supported (BEA has encountered problems with this implementation)

For more information on Objects-by-Value, see "Limitations of Passing Objects by Value" in *Programming WebLogic RMI*.

Procedure for Developing a CORBA/IDL Client

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

1. Follow steps 1 through 3 in "Developing a J2SE Client" on page 7-1.
2. 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 Programming WebLogic Enterprise JavaBeans. Also reference the Java IDL specification at Java Language Mapping to OMG IDL Specification at http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm.

   The following compiler options are specific to RMI over IIOP:

Table 9-2 RMI-IIOP Compiler Options

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

Table 9-2 RMI-IIOP Compiler Options

Option	Function
`-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.

3. 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.

   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 that comes with WebLogic Server.

4. 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 (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.

5. 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.
   ```

CHAPTER 10

Developing Clients for CORBA Objects

The following sections provide information on how to use the CORBA API:

Enhancements to and Limitations of CORBA Object Types

The RMI-IIOP runtime 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

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

This section provides examples of several mechanisms to access the WebLogic ORB. 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.

Listing 10-1 Accessing the WebLogic ORB from JNDI

```
.
.
.
```

```
ORB orb = (ORB)new InitialContext().lookup("java:comp/ORB");
NamingContext nc = NamingContextHelper.narrow(orb.string_to_object("corbal
oc: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.

Listing 10-2 Direct ORB Creation

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

Using JNDI

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

Listing 10-3 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 hosting 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:

Listing 10-4 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".

CHAPTER 11

Developing a WebLogic C++ Client for a Tuxedo ORB

The 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). The following sections provides information on developing WebLogic C++ clients for the Tuxedo ORB:

WebLogic C++ Client Advantages and Limitations

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 WebLogic C++ client has the following limitations:

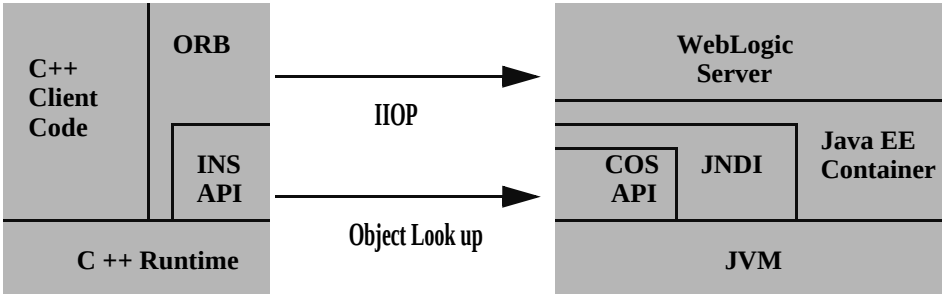
- Provides security through the WebLogic Server Security service.
- Provides only server-side transaction demarcation.

How the WebLogic C++ Client Works

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 initally 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.

Figure 11-1 WebLogic C++ Client to WebLogic Server Interoperability



Developing WebLogic C++ Clients

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.
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.
5. Use the Tuxedo `buildobjclient` command to build the client.

CHAPTER 12

Developing Security-Aware Clients

You can develop Weblogic clients that use the Java Authentication and Authorization Service (JAAS) and Secure Sockets Layer (SSL). The following sections provide information on security-aware clients:

Developing Clients That Use JAAS

JAAS enforces access controls based on user identity and is the preferred method of authentication for WebLogic Server clients. A typical use case is providing authentication to read or write to a file. Users requiring client certificate authentication (also referred to as two-way SSL authentication) should use JNDI authentication. For more information on how to implement JAAS authentication, see Using JAAS Authentication in Java Clients.

Developing Clients That Use SSL

BEA 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 digitial certificates. The location of the server's trusted CA certificate is specified when starting the RMI client.

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 types of 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.TrustKeystoreType=CustomTrust
  ```

  Optionally, specify the password defined when creating the keystore:

  ```
  -Dweblogic.security.CustomTrustKeystorePassPhrase=password
  ```

- Sun Microsystem'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). The keytool utility is a product of Sun Microsystems. Therefore, BEA Systems does not provide complete documentation on the utility. For more information about Sun's keytool utility, see the keytool-Key and Certificate Management Tool description at http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/keytool.html. Sun Microsystems

provides a tutorial Installing and Configuring SSL Support which includes a section "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 on how to implement SSL in "Configuring SSL" and "Configuring Identity and Trust" in *Securing WebLogic Server*.

Thin-Client Restrictions for JAAS and SSL

WebLogic thin-client applications only support JAAS authentication through the following classes:

- UsernamePasswordLoginModule
- Security.runAs

WebLogic thin-clients only support two-way SSL by requiring the `SSLContext` to be provided by the `SECURITY_CREDENTIALS` property. For example, see the client code below:

Listing 12-1 Client Code with sslcontext

```
.
.
.
// Get a KeyManagerFactory for KeyManagers
System.out.println("Retrieving KeyManagerFactory & initializing");
    KeyManagerFactory kmf =
    KeyManagerFactory.getInstance("SunX509","SunJSSE");
    kmf.init(ks,keyStorePassword);

// Get and initialize an SSLContext
System.out.println("Initializing the SSLContext");
    SSLContext sslCtx = SSLContext.getInstance("SSL");
    sslCtx.init(kmf.getKeyManagers(),null,null);

// Pass the SSLContext to the initial context factory and get an
```

```
// InitialContext
System.out.println("Getting initial context");
    Hashtable props = new Hashtable();
    props.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
    props.put(Context.PROVIDER_URL,
        "corbaloc:iiops:" +
         host + ":" + port +
        "/NameService");
    props.put(Context.SECURITY_PRINCIPAL,"weblogic");
    props.put(Context.SECURITY_CREDENTIALS, sslCtx);
    Context ctx = new InitialContext(props);
.
.
.
```

Security Code Examples

Security samples are provided with the WebLogic Server product. The samples are located in the *SAMPLES_HOME*\server\examples\src\examples\security directory. A description of each sample and instructions on how to build, configure, and run a sample, are provided in the package-summary.html file. You can modify these code examples and reuse them.

CHAPTER 13

Using EJBs with RMI-IIOP Clients

You can implement Enterprise JavaBeans that use RMI-IIOP to provide EJB interoperability in heterogeneous server environments:

Accessing EJBs with a Java Client

A Java RMI client can use an ORB and IIOP to access Enterprise beans residing on a WebLogic Server instance. See Understanding Enterprise JavaBeans in *Programming WebLogic Enterprise JavaBeans*.

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" on page 9-1.

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 *Programming WebLogic RMI*). 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/formal/java_language_mapping_to_omg_idl.htm`.

Also, Sun has an excellent guide, Enterprise JavaBeansTM Components and CORBA Clients: A Developer Guide at `http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/interop.html`.

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.

APPENDIX A

Client Application Deployment Descriptor Elements

The following sections describe deployment descriptors for Java EE client applications on WebLogic Server:

Overview of Client Application Deployment Descriptor Elements

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 with 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 runtime 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. It 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	Required Optional	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.

Table A-1 application-client Elements

Element	Required Optional	Description
`<env-entry>`		Contains the declaration of a client application's environment entries. Elements that can be defined within the `env-entry` element are: • `description`—Optional. The `description` element contains a description of the particular environment entry. • `env-entry-name`—The `env-entry-name` element contains the name of a client application's environment entry. • `env-entry-type`—The `env-entry-type` element 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. The `env-entry-value` element 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`.

Table A-1 application-client Elements

Element	Required Optional	Description
`<ejb-ref>`		Used for the declaration of a reference to an EJB referenced in the client application. Elements that can be defined within the `ejb-ref` element are: • `description`—Optional. The `description` element 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 JavaBean 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 application-client Elements

Element	Required Optional	Description
`<resource-ref>`		Contains a declaration of the client application's reference to an external resource. Elements that can be defined within the `resource-ref` element are: • `description`—Optional. The `description` element 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	Required Optional	Description
`<env-entry>`		Specifies values for environment entries declared in the deployment descriptor. Elements that can be defined within the `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 application-client Elements

Element	Required Optional	Description
`<ejb-ref>`		Specifies the JNDI name for a declared EJB reference in the deployment descriptor. Elements that can be defined within the `ejb-ref` element are: • `ejb-ref-name`—Name of an EJB reference. The EJB reference is an entry in the application client's environment. It is recommended 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 that can be defined within the `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.

APPENDIX B

Using the WebLogic JarBuilder Tool

The following sections provide information on creating the `wlfullclient.jar` using the JarBuilder tool:

Overview

Prior to this release, the `weblogic.jar` file could be bundled with a client application to provide WebLogic Server specific value added features, such as:

- Enhanced JDBC and WLS specific JMX interfaces
- WLS T3 Client
- WLS-IIOP Client

See "Client Types and Features" on page 2-3.

For WebLogic Server 10.0 and higher releases, client applications should use the `wlfullclient.jar` file to provide the WebLogic Server specific functionary previously provided in the `weblogic.jar` file. You can generate the `wlfullclient.jar` file for client applications using the JarBuilder tool. See "Creating a wlfullclient.jar File for a Client Application" on page B-2.

Note: Continuing to use `weblogic.jar` may result in a `ClassNotFoundException`.

Creating a wlfullclient.jar File for a Client Application

Use the following steps to create a `wlfullclient.jar` file for a client application:

1. Change directories to the `server/lib` directory.

   `cd` *`WL_HOME`*`/server/lib`

2. Use the following command to create `wlfullclient.jar` in the `server/lib` directory:

   `java -jar ../../../modules/com.bea.core.jarbuilder_`*`X.X.X.X`*`.jar`

   where *`X.X.X.X`* is the version number of the jarbuilder module in the *`WL_HOME`*`/server/lib` directory. For example:

   `java -jar ../../../modules/com.bea.core.jarbuilder_1.0.1.0.jar`

3. You can now copy and bundle the `wlfullclient.jar` with client applications.

4. Add the `wlfullclient.jar` to the client application's `classpath`.

APPENDIX C

Code Examples

The BEA developer web site dev2dev.com provides examples that demonstrate how to use EJBs with RMI-IIOP, connect to C++ clients, and set up interoperability with a Tuxedo Server.

The following table describes the examples.

Table C-1 WebLogic Server IIOP Examples

Example	ORB/Protocol	Requirements
`iiop.ejb.entity.tuxclient` Provides a Tuxedo client that uses complex valuetypes to call an entity session bean in WebLogic Server.	BEA IIOP	Tuxedo 8.x and higher. Requires custom marshalling of vector classes.
`iiop.ejb.entity.server.wls` Demonstrates connectivity between a C++ client or a Tuxedo client and an entity bean.	Not Applicable	
`iiop.ejb.stateless.rmiclient` Provides an RMI Java client that calls a stateless session bean in WebLogic Server. The example also demonstrates how to make an outbound RMI-IIOP call to a Tuxedo server using WebLogic Tuxedo Connector.	JDK 1.4	JDK 1.4 requires a security policy file to access server.

Table C-1 WebLogic Server IIOP Examples

Example	ORB/Protocol	Requirements
`iiop.ejb.stateless.sectuxclient` Provides a secure Tuxedo client that calls a stateless session bean from WebLogic Server.	BEA IIOP	Tuxedo 8.x and higher.
`iiop.ejb.stateless.server.tux` Illustrates how to call a stateless session bean from a variety of client applications through a Tuxedo Server. In conjunction with the Tuxedo Client, it also demonstrates server-to-server connectivity using WebLogic Tuxedo Connector.	Tuxedo TGIOP	Tuxedo 8.x and higher. WebLogic Tuxedo Connector to provide server-to-server connectivity. See Using WebLogic Tuxedo Connector for RMI/IIOP and Corba Interoperability.
`iiop.ejb.stateless.server.wls` Demonstrates how to use a variety of clients to call a stateless EJB directly in WebLogic Server or indirectly through a Tuxedo Server.	Not Applicable	
`iiop.ejb.stateless.tuxclient` Provides a Tuxedo client that calls a stateless session bean directly in WebLogic Server or to call the same stateless session bean in WebLogic through a Tuxedo server. The example also demonstrates how to make an outbound RMI-IIOP call from a Tuxedo server to WebLogic Server using WebLogic Tuxedo Connector.	BEA IIOP	Tuxedo 8.x and higher.
`iiop.ejb.stateless.txtuxclient` Provides a Tuxedo client that uses a transaction to call a stateless session bean.	BEA IIOP	Tuxedo 8.x and higher.

Table C-1 WebLogic Server IIOP Examples

Example	ORB/Protocol	Requirements
`iiop.rmi.corbaclient` Provides a CORBA client that demonstrates connectivity to a WebLogic Server.	BEA IIOP	Tuxedo 8.0 RP56 and higher. JDK 1.4 requires a security policy file to access server.
`iiop.rmi.rmiclient` Provides an RMI client that demonstrates connectivity to a WebLogic Server. The example also demonstrates how to make an outbound call from WebLogic Server to a Tuxedo server using WebLogic Tuxedo Connector.	Not Applicable	Requires a security policy file to access server.
`iiop.rmi.server.tux` Illustrates connectivity from a variety of client applications through a Tuxedo Server. In conjunction with the Tuxedo Client, it also domesticates server-to-server connectivity using WebLogic Tuxedo Connector.	Tuxedo TGIOP	Tuxedo 8.x and higher. WebLogic Tuxedo Connector to provide server-to-server connectivity. See Using WebLogic Tuxedo Connector for RMI/IIOP and Corba Interoperability.
`iiop.rmi.server.wls` Example illustrates connectivity between a variety of clients, Tuxedo, and WebLogic Server using a simple Ping application.	Not Applicable	
`iiop.rmi.tuxclient` Example provides a Tuxedo client which demonstrates connectivity to a Tuxedo Server.	BEA IIOP	Tuxedo 8.x and higher.