Oracle® Fusion Middleware

Developing RMI Applications for Oracle WebLogic Server

14c (14.1.1.0.0)
F18295-04
January 2023

Oracle Fusion Middleware Developing RMI Applications for Oracle WebLogic Server, 14c (14.1.1.0.0)

F18295-04

Contents

7 WebLogic RMI Integration with Load Balancers

8 Using RMI over IIOP

9 Configuring WebLogic Server for RMI-IIOP

Preface

This document is written for application developers who want to build e-commerce applications using Remote Method Invocation (RMI) and Internet Interop-Orb-Protocol (IIOP) features.

Audience

It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language. This document emphasizes the value-added features provided by WebLogic Server and key information about how to use WebLogic Server features when developing applications with RMI.

Documentation Accessibility

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

Accessible Access to Oracle Support

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

Diversity and Inclusion

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

Related Documentation

For information on topics related to WebLogic RMI, see the following documents:

- Java RemoteMethod Invocation (RMI) at http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/ is a link to basic tutorials on Remote Method Invocation.
- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.

- *Developing JNDI Applications for Oracle WebLogic Server* is a guide using the WebLogic Java Naming and Directory Interface.
- *Developing Standalone Clients for Oracle WebLogic Server* is a guide to developing common stand alone clients that interoperate with WebLogic Server.
- *Tuning Performance of Oracle WebLogic Server* contains information on monitoring and improving the performance of WebLogic Server applications.
- `https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138209.html` provides an overview of CORBA and Java platform.
- `http://docs.oracle.com/javase/8/docs/technotes/guides/idl/index.html` contains information using standard IDL (Object Management Group Interface Definition Language) and IIOP.
- `https://www.omg.org` is the Object Management Group home page.
- *CORBA Language Mapping Specification* at `http://www.omg.org/technology/documents/index.htm`

Samples and Tutorials

In addition to this document, Oracle 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.

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

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java Platform, Enterprise Edition (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 Oracle-recommended best practices. MedRec is optionally installed in the WebLogic Server installation. You can start MedRec from the *ORACLE_HOME*`\user_projects\domains\medrec` directory, where *ORACLE_HOME* is the directory you specified as Oracle Home when you installed Oracle WebLogic Server. For more information about the WebLogic Server samples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

MedRec includes a service tier comprised 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 the *ORACLE_HOME*`\wlserver\samples\server\examples\src\examples` directory, where *ORACLE_HOME* represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

New and Changed WebLogic Server Features

This section includes new and changed features for recent patch sets of WebLogic Server:

- New WebLogic RMI annotations that provide remote access to plain java objects. See Using WebLogic RMI Annotations.
- A new connection attribute, `WLContext.CONNECT_TIMEOUT`, to define the length of time a client waits for connections to the server to be bootstrapped or re-established. `WLContext.REQUEST_TIMEOUT` is deprecated. See Using a Connect Timeout.
- A new connection attribute, `WLContext.RESPONSE_READ_TIMEOUT`, to define the length of time that a client waits to receive a response from a server. `WLContext.RMI_TIMEOUT` is deprecated. See Using a Read Timeout.

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

Conventions

The following text conventions are used in this document:

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

1

Understanding WebLogic RMI

This chapter describes the features of WebLogic RMI.
This chapter includes the following sections:

What is WebLogic RMI?

WebLogic Remote Method Invocation (RMI) enables an application to obtain a reference to a remote object that exists within the network on a virtual machine. RMI provides remote communication between the applications using objects that are distributed over multiple virtual machines.

Remote Method Invocation (RMI) is the standard for distributed object computing in Java. RMI enables an application to obtain a reference to an object that exists elsewhere in the network, and then invoke methods on that object as though it existed locally in the client's virtual machine. RMI specifies how distributed Java applications should operate over multiple Java virtual machines.

This document contains information about using WebLogic RMI, but it is not a beginner's tutorial on remote objects or writing distributed applications. If you are just beginning to learn about RMI, visit `http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/` and review the *RMI Tutorial*.

Features of WebLogic RMI

RMI has several advantages over traditional remote procedure call systems because RMI is a part of Java's object-oriented approach. View WebLogic RMI Features to know how each of these features is implemented in the WebLogic framework.

The following table highlights important features of WebLogic implementation of RMI:

Table 1-1 WebLogic RMI Features

Feature	WebLogic RMI
Overall performance	Enhanced by WebLogic RMI integration into the WebLogic Server framework, which provides underlying support for communications, scalability, management of threads and sockets, efficient garbage collection, and server-related support.
Standards compliant	Compliance with the Java Platform Standard Edition 6.0 API Specification
Annotations	Provides annotation support that can be embedded inside remote java objects.

Table 1-1 (Cont.) WebLogic RMI Features

Feature	WebLogic RMI
Failover and Load balancing	WebLogic Server support for failover and load balancing of RMI objects.
Request Timeouts	You can specify a timeout period for a remote call to complete.
WebLogic RMI compiler	Stubs and skeletons dynamically generated by WebLogic RMI at run time, which obviates need to explicitly run `weblogic.rmic`, except for clusterable or Internet Inter-ORB Protocol (IIOP) clients.
Dynamic Proxies	A class used by the clients of a remote object. In the case of RMI, skeleton and a stub classes are used. The stub class is the instance that is invoked upon in the client's Java Virtual Machine (JVM). The skeleton class, which exists in the remote JVM, unmarshals the invoked method and arguments on the remote JVM, invokes the method on the instance of the remote object, and then marshals the results for return to the client.
Security Support	No Security Manager required. WebLogic Server implements authentication, authorization, and Java EE security services.
Transaction Support	WebLogic Server supports transactions in the Java Platform, Enterprise Edition (Java EE) programming model.
Internet Protocol version 6 (IPv6) Support	Support for 128 bit addressing space.

2

WebLogic RMI Features

This chapter describes the WebLogic RMI features and guidelines required to program RMI for use with WebLogic Server.
This chapter includes the following sections:

WebLogic RMI Clients

WebLogic RMI is divided between a client and server framework. The client run time does not have server sockets and therefore does not listen for connections. It obtains its connections through the server. Only the server knows about the client socket. Therefore if you plan to host a remote object on the client, you must connect the client to WebLogic Server. WebLogic Server processes requests for and passes information to the client. In other words, client-side RMI objects can only be reached through a single WebLogic Server, even in a cluster. If a client-side RMI object is bound into the JNDI naming service, it only be reachable as long as the server that carried out the bind is reachable.

WebLogic RMI Security Support

WebLogic Server implements authentication, authorization, and Java EE security services. See *Developing Applications with the WebLogic Security Service*.

WebLogic RMI Transaction Support

Oracle WebLogic Server supports transactions in the Java Platform, Enterprise Edition (Java EE) programming model. For detailed information on using transactions in WebLogic RMI applications, see the following:

- Transactions in WebLogic Server RMI Applications in *Developing JTA Applications for Oracle WebLogic Server* provides an overview on how transactions are implemented in WebLogic RMI applications.
- Transactions in RMI Applications in *Developing JTA Applications for Oracle WebLogic Server* provides general guidelines when implementing transactions in RMI applications for WebLogic Server.

Failover and Load Balancing RMI Objects

The following sections contain information on WebLogic Server support for failover and load balancing of RMI objects:

Clustered RMI Applications

For clustered RMI applications, failover is accomplished using the object's replica-aware stub. When a client makes a call through a replica-aware stub to a service that fails, the stub detects the failure and retries the call on another replica.

To make Java EE services available to a client, WebLogic binds an RMI stub for a particular service into its JNDI tree under a particular name. The RMI stub is updated with the location of other instances of the RMI object as the instances are deployed to other servers in the cluster. If a server within the cluster fails, the RMI stubs in the other server's JNDI tree are updated to reflect the server failure.

You specify the generation of replica-aware stubs for a specific RMI object using the `-clusterable` option of the WebLogic RMI compiler, as explained in Table 4-1. For example:

```
$ java weblogic.rmic -clusterable classes
```

See Replication and Failover for EJBs and RMIs in *Administering Clusters for Oracle WebLogic Server*.

Load Balancing RMI Objects

The load balancing algorithm for an RMI object is maintained in the replica-aware stub obtained for a clustered object. You specify the load balancing algorithm for a specific RMI object using the `-loadAlgorithm <algorithm>` option of the WebLogic RMI compiler. A load balancing algorithm that you configure for an object overrides the default load balancing algorithm for the cluster. The WebLogic Server RMI compiler supports the following load balancing algorithms:

- Round Robin Load Balancing
- Weight-Based Load Balancing
- Random Load Balancing
- Server Affinity Load Balancing Algorithms

For example, to set load balancing on an RMI object to round robin, use the following `rmic` options:

```
$ java weblogic.rmic -clusterable -loadAlgorithm round-robin classes
```

To set load balancing on an RMI object to weight-based server affinity, use `rmic` options:

```
   $ java weblogic.rmic -clusterable -loadAlgorithm weight-based -stickToFirstServer
classes
```

See Load Balancing for EJBs and RMI Objects in *Administering Clusters for Oracle WebLogic Server*.

Parameter-Based Routing for Clustered Objects

Parameter-based routing allows you to control load balancing behavior at a lower level. Any clustered object can be assigned a CallRouter using the `weblogic.rmi.cluster.CallRouter` interface. This is a class that is called before each invocation with the parameters of the call. The CallRouter is free to examine the parameters and return the name server to which the call should be routed.

```
   weblogic.rmi.cluster.CallRouter.

      Class java.lang.Object
        Interface weblogic.rmi.cluster.CallRouter
        (extends java.io.Serializable)
```

A class implementing this interface must be provided to the RMI compiler (`rmic`) to enable parameter-based routing. Run `rmic` on the service implementation using these options (to be entered on one line):

```
   $ java weblogic.rmic -clusterable -callRouter <callRouterClass> <remoteObjectClass>
```

The call router is called by the clusterable stub each time a remote method is invoked. The router is responsible for returning the name of the server to which the call should be routed.

Each server in the cluster is uniquely identified by its name as defined with the WebLogic Server Console. These are the names that the method router must use for identifying servers.

Consider the `ExampleImpl` class which implements a remote interface `Example`, with one method `foo`:

```
   public class ExampleImpl implements Example {
   public void foo(String arg) { return arg; }
   }
```

This `CallRouter` implementation `ExampleRouter` ensures that all *foo* calls with 'arg' < " n " go to server1 (or server3 if server1 is unreachable) and that all calls with 'arg' >= "n " go to server2 (or server3 if server2 is unreachable).

```
public class ExampleRouter implements CallRouter {
  private static final String[] aToM = { "server1", "server3" };
  private static final String[] nToZ = { "server2", "server3" };

  public String[] getServerList(Method m, Object[] params) {
    if (m.GetName().equals("foo")) {
      if (((String)params[0]).charAt(0) < 'n') {
        return aToM;
      } else {
        return nToZ;
      }
    } else {
      return null;
    }
```

```
  }
}
```

This `rmic` call associates the `ExampleRouter` with `ExampleImpl` to enable parameter-based routing:

```
$ rmic -clusterable -callRouter ExampleRouter ExampleImpl
```

Custom Call Routing and Collocation Optimization

If a replica is available on the same server instance as the object calling it, the call is not load-balanced as it is more efficient to use the local replica. See Optimization for Collocated Objects in *Administering Clusters for Oracle WebLogic Server*.

Request Timeouts

You can specify timeout period for the remote call to complete or the client receives a `weblogic.rmi.extensions.RequestTimeoutException`.

WebLogic Server provides the following connect and read timeouts:

- Using a Connect Timeout
- Using a Read Timeout

Using a Connect Timeout

Use a connect timeout to define the length of time a client waits for connections to the server to be bootstrapped or re-established. The following table describes how to set this timeout.

Table 2-1 Setting a Connect Timeout

Description	Scope
System property: `-Dweblogic.ConnectTimeout=`*`seconds`*	Server
Set `KernelMBean.ConnectTimeout` property	Server
Set `NetworkAccessPointMBean.connectTimeout` property. For non-default channels only and overrides a server scoped setting. Only connections established using this channel definition are subject to this timeout.	Channel
Set `WLContext.CONNECT_TIMEOUT` to establish the connection to a server. (Within the scope of the context, used for both bootstrapping a connection as well re-establishing a lost connection.)	Thread

Using a Read Timeout

Use a read timeout to define the length of time that a client waits to receive a response from a server. The following table describes various ways to set this timeout value.

Table 2-2 Setting a Read Timeout

Description	Scope
Set `WLContext.RESPONSE_READ_TIMEOUT` in the JNDI (InitialContext) environment used to lookup the remote stub.	Interface (stub)
Specify the method level annotation (`@RmiMethod(timeout=<value>)`) in the remote object implementation class.	Method
Set a timeout attribute in method definition in an `rtd.xml` file for non-annotated classes. See Example rtd.xml file with a Timeout.	Method
Specify the method level annotation `@TransactionTimeoutSeconds(<timeout>)` in the EJB bean implementation class.	Method
Specify a remote-client-timeout in EJB descriptor (`weblogic-ejb-jar.xml`). See Example weblogic-ejb-jar.xml file with a Timeout.	Method

Consider the following when implementing a read timeout:

- In the WebLogic Server EJB bean implementation, a transaction timeout (`@TransactionTimeoutSeconds`) takes precedence over `remote-client-timeout` if it has a greater value.
- The precedence of multiple read timeouts is determined using the following rules:
  - A timeout specified in `rtd.xml` on the client overrides any other read timeout.
  - A timeout specified using an `RmiMethod` annotation overrides a `WLContext.RESPONSE_READ_TIMEOUT`.

Example rtd.xml file with a Timeout

The following code provides an example of an `rtd.xml` file that includes an timeout:

```
<rmi Name="foo">
   <method
       name="methodname"
       timeout="timeoutinmilliseconds">
   </method>
</rmi>
```

To generate an `rtd.xml file` on the client, set `-Dweblogic.RefreshClientRuntimeDescriptor=true` on both the client and the server. When the flag is `true`, a check is made to see if the `rtd.xml` file is available on the classpath. If available, the file is read and the values specified are used.

Example weblogic-ejb-jar.xml file with a Timeout

The following code provides an example of how to specify a `remote-client-timeout` in the `weblogic-ejb-jar.xml` file:

```
<weblogic-enterprise-bean>
     <ejb-name>AccountBean</ejb-name>
     . . .
     <remote-client-timeout>5</remote-client-timeout>
</weblogic-enterprise-bean>
```

Creating Pinned Services

You can also use `weblogic.rmic` to generate stubs that are not replicated in the cluster. These stubs are known as "pinned " services, because after they are registered they are available only from the host with which they are registered and will not provide transparent failover or load balancing. Pinned services are available cluster-wide, because they are bound into the replicated cluster-wide JNDI tree. However, if the individual server that hosts the pinned services fails, the client cannot failover to another server.

You specify the generation of non-replicated stubs for a specific RMI object by not using the `-clusterable` option of the WebLogic RMI compiler, as explained in Table 4-1. For example:

```
$ java weblogic.rmic classes
```

Dynamic Proxies in RMI

A dynamic proxy or proxy is a class used by the clients of a remote object. This class implements a list of interfaces specified at runtime when the class is created. In the case of RMI, dynamically generated bytecode and proxy classes are used. The proxy class is the instance that is invoked upon in the client's Java Virtual Machine (JVM). The proxy class marshals the invoked method name and its arguments; forwards these to the remote JVM. After the remote invocation is completed and returns, the proxy class unmarshals the results on the client. The generated bytecode — which exists in the remote JVM — unmarshals the invoked method and arguments on the remote JVM, invokes the method on the instance of the remote object, and then marshals the results for return to the client.

3

Using WebLogic RMI Annotations

This chapter describes the WebLogic RMI annotations that provide remote access to plain java objects.
WebLogic RMI provides a rich descriptor framework to associate various security, transactions, clustering, and timeout attributes to a remote class and its methods. These attributes can be specified as annotations in plain java implementation classes with non-remote interfaces when the remote object implementation is bound to a WebLogic JNDI tree. See weblogic.rmi.annotation in *Java API Reference for Oracle WebLogic Server*.

This chapter includes the following sections:

Introduction to WebLogic RMI Annotations

WebLogic RMI provides annotation support that can be embedded inside a remote java object and simplifies development by allowing you to avoid running `weblogic.rmic` tool on the compiled class.

To make a plain java object remotely accessible, do the following:

1. Create an interface that you want to access on the client. This interface must extend `java.rmi.Remote`. See Example 3-1.
2. Create an implementation class that implements the interface in Step 1.
3. Add the desired annotation `@Rmi` or `@RmiMethod` to the implementation class added in Step 2. The annotations need to be provided on the implementation class and methods, not on the interfaces.
4. Compile and bundle the classes in an application.
5. Deploy the application.
6. Bind the annotated plain java object in the WebLogic JNDI tree.
7. A client looks up the plain java object as remote object from the WebLogic JNDI tree and narrows it to the plain interfaces annotated as remote interfaces. The corresponding stub is either generated on the client, downloaded, or pre-generated using the WebLogic RMI compiler and made available on the client.

> **Note:**
>
> Do not use the WebLogic RMIC option to generate stubs and skeletons based on the Sun RMI compiler.

Example 3-1 Example RMI Annotation

```
package myrmi.example;

import java.rmi.RemoteException;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;

import weblogic.rmi.annotation.Rmi;
import weblogic.rmi.annotation.RmiMethod;

@Rmi(remoteInterfaces={MyRemoteInterface.class})
public class RmiMethodAnnotations implements MyRemoteInterface{
  public RmiMethodAnnotations() {
  }

  public int getIndex() throws RemoteException {
    return 0;
  }

  @RmiMethod(asynchronousResult=true)
  public Future<String> ejbAsynchronousSayHello(String name) {
    return new FutureTask(new MyRunnable(), new Object());
  }

  class MyRunnable implements Runnable {

    public void run() {
    }
  }
}
```

Example 3-2 Example RMI without Annotations

```
package myrmi.example;

import java.rmi.Remote;
import java.rmi.RemoteException;

import java.util.concurrent.Future;

public interface MyRemoteInterface extends Remote {

  int getIndex() throws RemoteException;

  public Future<String> ejbAsynchronousSayHello(String name);

  public String sayBye();

}
```

This allows the WebLogic RMI layer to treat the `RmiMethodAnnotations` object as remote object when it is bound to the WLS JNDI tree.

Example 3-2 provides an example of code that implements the same methods without using annotations.

Annotations for WebLogic RMI

The following topics provide reference information about WebLogic RMI annotations:

- Rmi
- RmiMethod

Rmi

The following sections describe the annotation in more detail.

Description

Provides class-level annotation support for remote objects that specify the remote implementation class.

See weblogic.rmi.annotation.Rmi.

Attributes

The following table summarizes the attributes.

Table 3-1 Attributes of the Rmi Annotation

Name	Description	Data Type	Default Value
`callRouterClassname`	The `CallRouter` class that is called before each invocation with the parameters of the call and it returns the name server to which the call should be routed. Parameter-based routing allows to provide a more fine-grained load balancing behavior.	`String`	""
`clusterable`	Indicates if the remote object is clusterable.	`boolean`	`false`
`defaultRMIMethodParams`	Default RMI Method annotation. Can be over-ridden with a method annotation.	`RmiMethod`	`@weblogic.rmi.annotation.RmiMethod`
`loadAlgorithm`	Load Algorithm for clustered remote object. Legal Values are: • `RANDOM` • `ROUND_ROBIN` • `WEIGHT_BASED` • `SERVER_AFFINITY` • `ROUND_ROBIN_AFFINITY` • `RANDOM_AFFINITY` • `WEIGHT_BASED_AFFINITY` • `DEFAULT` Default is `ROUND_ROBIN`.	`LoadAlgorithmType`	`weblogic.rmi.annotation.LoadAlgorithmType.DEFAULT`

Table 3-1 (Cont.) Attributes of the Rmi Annotation

Name	Description	Data Type	Default Value
`stickToFirstServer`	Enables sticky load balancing. The server chosen for servicing the first request is used for all subsequent requests. Only used for a clusterable remote object.	`boolean`	`false`
`remoteInterfaces`	A comma-separated list of Interface class names to be treated as remote interface	`Class`	""

RmiMethod

The following sections describe the annotation in more detail.

Description

Provides method-level annotation support for remote objects that specify the remote implementation class.

See weblogic.rmi.annotation.RmiMethod.

Attributes

The following table summarizes the attributes.

Table 3-2 Attributes of the RmiMethod Annotation

Name	Description	Data Type	Default Value
`asynchronousResult`	If `true`, marks a method for asynchronous processing. Typically when a method is invoked, the result is returned upon the completion of the method execution. When `asynchronousResult=true`, the return type of the method can be either `void` or a `Future` object. If the type is a `Future` object, it can be polled to see when the result is available. If the type is a `void`, the method is treated as an asynchronous one-way call.	`boolean`	`false`
`dispatchPolicy`	Specifies the Work Manager used to schedule remote object requests.	`String`	""
`idempotent`	Specifies an Idempotent method.	`boolean`	`false`
`oneway`	Specifies a one-way call.	`boolean`	`false`
`timeout`	Specifies a timeout for a remote call.	`int`	0
`transactional`	Specifies a transactional method. If not, suspend a transaction before making the RMI call and resume the transaction after the call completes.	`boolean`	`false`

Exception Handling

The following sections provide information on WebLogic RMI annotation exception handling:

Application Exceptions

Clients receive all checked application exceptions.

System Exceptions

Client receive all the errors and runtime exceptions encountered during remote method invocation.

Remote exceptions are handled as follows:

- Checked exceptions are thrown directly to a client.
- Unchecked exceptions are wrapped in a `RuntimeException` and then thrown to the client.
- Generated EJB 3.0 objects annotate the `remoteExceptionWrapper` to be `EJBException` for all EJB methods. Clients then receive all remote exceptions wrapped in `EJBException`.

You can specify the `remoteExceptionWrapper` annotation for an entire implementation class or for a particular method which wraps all remote exceptions in the specified runtime exception before throwing it back to the client. If the `remoteExceptionWrapper` annotation is not specified then the remote exceptions are wrapped as shown in Table 3-3.

Table 3-3 Exception Wrapping in WebLogic Clients

Client	Exception Wrapping
WL Full Client	`RemoteRuntimeException`[1] wraps `RemoteException`
WL Thin T3 Client	`RemoteRuntimeException`[2] wraps `RemoteException`
WLS-IIOP Client[3]	`RemoteRuntimeException` wraps `java.rmi.ServerException` wraps `RemoteException` or `RemoteRuntimeException` wraps RemoteException

Thin Client	`java.lang.RuntimeException` wraps `ServerException` wraps `RemoteException`
	or
	`RuntimeException` wraps `RemoteException`

Java SE Client	`java.lang.RuntimeException` wraps `ServerException` wraps `RemoteException`
	or
	`RuntimeException` wraps `RemoteException`

[1] `weblogic.rmi.extensions.RemoteRuntimeException` is a sub-class of `RuntimeException`

[2] `weblogic.rmi.extensions.RemoteRuntimeException` is a sub-class of `RuntimeException`

[3] The existing T3 protocol layer doesn't always wraps the `RemoteException` as `java.rmi.ServerException` but the IIOP protocol always does it on the Server.

Cluster Failover

Clustered stubs automatically handle the failover of a remote call to another node in the cluster based on the type of exception received. Wrapping remote exceptions, such as `RuntimeException`, in the stub does not change the failover behavior for a remote object.

RMI Callback Objects

Passing a callback object with an annotated remote object requires the callback remote object to `extend java.rmi.Remote` interface.

> **Note:**
>
> Some client types can not support callback objects because they do not have access to WebLogic classes. For example, the Java SE client.

Annotation and WebLogic RMI Descriptor Merging

Annotations specified in the implementation class cannot be over-ridden on the server. You must ensure that the right set of descriptor values are used by merging the application descriptors and deployment plans.

4

Using the WebLogic RMI Compiler

This chapter describes how to use the features and options of the WebLogic RMI compiler. This chapter includes the following sections:

- Overview of the WebLogic RMI Compiler
- WebLogic RMI Compiler Features
- WebLogic RMI Compiler Options
- Java SE Enhancements

Overview of the WebLogic RMI Compiler

The WebLogic RMI compiler (`weblogic.rmic`) is a command-line utility for generating and compiling remote objects. Use `weblogic.rmic` to generate dynamic proxies on the client-side for custom remote object interfaces in your application and provide hot code generation for server-side objects.

You only need to explicitly run `weblogic.rmic` for clusterable or IIOP clients. WebLogic RMI over IIOP extends the RMI programming model by providing the ability for clients to access RMI remote objects using the Internet Inter-ORB Protocol (IIOP). See Using RMI over IIOP .

WebLogic RMI Compiler Features

The following sections provide information on WebLogic RMI Compiler features for this release:

- Hot Code Generation
- Proxy Generation
- Additional WebLogic RMI Compiler Features

Hot Code Generation

When you run `rmic`, you use WebLogic Server's hot code generation feature to automatically generate bytecode in memory for server classes. This bytecode is generated on the fly as needed for the remote object. WebLogic Server no longer generates the skeleton class for the object when `weblogic.rmic` is run.

Hot code generation produces the bytecode for a server-side class that processes requests from the dynamic proxy on the client. The dynamically created bytecode de-serializes client requests and executes them against the implementation classes, serializing results and sending them back to the proxy on the client. The implementation for the class is bound to a name in the JNDI tree in WebLogic Server.

Proxy Generation

The default behavior of the WebLogic RMI compiler is to produce proxies for the remote interface and for the remote classes to share the proxies. A proxy is a class used by the clients of a remote object. In the case of RMI, dynamically generated bytecode and proxy classes are used.

For example, `example.hello.HelloImpl` and `counter.example.CiaoImpl` are represented by a single proxy class and bytecode—the proxy that matches the remote interface implemented by the remote object, in this case, `example.hello.Hello`.

When a remote object implements more than one interface, the proxy names and packages are determined by encoding the set of interfaces. You can override this default behavior with the WebLogic RMI compiler option `-nomanglednames`, which causes the compiler to produce proxies specific to the remote class. When a class-specific proxy is found, it takes precedence over the interface-specific proxy.

In addition, with WebLogic RMI proxy classes, the proxies are not final. References to collocated remote objects are references to the objects themselves, not to the proxies.

The dynamic proxy class is the serializable class that is passed to the client. A client acquires the proxy for the class by looking up the class in the WebLogic JNDI. The client calls methods on the proxy just as if it were a local class and the proxy serializes the requests and sends them to WebLogic Server.

Additional WebLogic RMI Compiler Features

Other features of the WebLogic RMI compiler include the following:

- Signatures of remote methods do not need to throw `RemoteException`.
- Remote exceptions can be mapped to `RuntimeException`.
- Remote classes can also implement non-remote interfaces.

WebLogic RMI Compiler Options

The WebLogic RMI compiler accepts any option supported by the Java compiler; for example, you could add `-d \classes examples.hello.HelloImpl` to the compiler option at the command line. All other options supported by the Java compiler can be used and are passed directly to the Java compiler.

The following table lists `java weblogic.rmic` options. Enter these options after `java weblogic.rmic` and before the name of the remote class.

```
$java weblogic.rmic [options] <classes>...
```

Table 4-1 WebLogic RMI Compiler Options

Option	Description
`-help`	Prints a description of the options.
`-version`	Prints version information.
`-d <dir>`	Specifies the target (top level) directory for compilation.

Table 4-1 (Cont.) WebLogic RMI Compiler Options

Option	Description
`-dispatchPolicy <queueName>`	Specifies a configured execute queue that the service should use to obtain execute threads in WebLogic Server.
`-oneway`	Specifies all calls are one-way calls.
`-idl`	Generates IDLs for remote interfaces.
`-idlOverwrite`	Overwrites existing IDL files.
`-idlVerbose`	Displays verbose information for IDL information.
`-idlDirectory <idlDirectory>`	Specifies the directory where IDL files will be created (Default is the current directory).
`-idlFactories`	Generates factory methods for valuetypes.
`-idlNoValueTypes`	Prevents the generation of valuetypes and the methods/attributes that contain them.
`-idlNoAbstractInterfaces`	Prevents the generation of abstract interfaces and the methods/attributes that contain them.
`-idlStrict`	Generates IDL according to OMG standard.
`-idlVisibroker`	Generate IDL compatible with Visibroker 4.5 C++.
`-idlOrbix`	Generate IDL compatible with Orbix 2000 2.0 C++.
`-iiopTie`	Generate CORBA skeletons using Sun's version of rmic.
`-iiopSun`	Generate CORBA stubs using Sun's version of rmic.
`-nontransactional`	Suspends the transaction before making the RMI call and resumes after the call completes.
`-compiler <javac>`	Specifies the Java compiler. If not specified, the -compilerclass option will be used.
`-compilerclass <com.sun.tools.javac.Main>`	Compiler class to invoke.
`-clusterable`	This cluster-specific options marks the service as clusterable (can be hosted by multiple servers in a WebLogic Server cluster). Each hosting object, or replica, is bound into the naming service under a common name. When the service stub is retrieved from the naming service, it contains a replica-aware reference that maintains the list of replicas and performs load-balancing and fail-over between them.
`-loadAlgorithm <algorithm>`	Only for use in conjunction with -clusterable. Specifies a service-specific algorithm to use for load-balancing and fail-over (Default is weblogic.cluster.loadAlgorithm). Must be one of the following: round-robin, random, or weight-based.
`-callRouter <callRouterClass>`	This cluster-specific option used in conjunction with -clusterable specifies the class to be used for routing method calls. This class must implement weblogic.rmi.cluster.CallRouter. If specified, an instance of the class is called before each method call and can designate a server to route to based on the method parameters. This option either returns a server name or null. Null means that you use the current load algorithm.

Table 4-1 (Cont.) WebLogic RMI Compiler Options

Option	Description
`-stickToFirstServer`	This cluster-specific option used in conjunction with -clusterable enables "sticky " load balancing. The server chosen for servicing the first request is used for all subsequent requests.
`-methodsAreIdempotent`	This cluster-specific option used in conjunction with -clusterable indicates that the methods on this class are idempotent. This allows the stub to attempt recovery from any communication failure, even if it can not ensure that failure occurred before the remote method was invoked. By default (if this option is not used), the stub only retries on failures that are guaranteed to have occurred before the remote method was invoked.
`-iiop`	Generates IIOP stubs from servers.
`-iiopDirectory`	Specifies the directory where IIOP proxy classes are written.
`-timeout`	Used in conjunction with remote-client-timeout .
`-commentary`	Emits commentary.
`-nomanglednames`	Causes the compiler to produce proxies specific to the remote class.
`-g`	Compile debugging information into the class.
`-O`	Compile with optimization.
`-nowarn`	Compile without warnings.
`-verbose`	Compile with verbose output.
`-verboseJavac`	Enable Java compiler verbose output.
`-nowrite`	Prevent the generation of `.class` files.
`-deprecation`	Provides warnings for deprecated calls.
`-classpath <path>`	Specifies the classpath to use.
`-J<option>`	Use to pass flags through to the Java runtime.
`-keepgenerated`	Allows you to keep the source of generated stub and skeleton class files when you run the WebLogic RMI compiler.
`-disableHotCodeGen`	Causes the compiler to create stubs at skeleton classes when compiled.

Non-Replicated Stub Generation

You can also use `weblogic.rmic` to generate stubs that are not replicated in the cluster. These stubs are known as "pinned " services, because after they are registered they are available only from the host with which they are registered and will not provide transparent failover or load balancing. Pinned services are available cluster-wide, because they are bound into the replicated cluster-wide JNDI tree. However, if the individual server that hosts the pinned services fails, the client cannot failover to another server.

Using Persistent Compiler Options

During deployment, appc and ejbc run each EJB container class through the RMI compiler to create RMI descriptors necessary to dynamically generate stubs and skeletons. Use the `weblogic-ejb-jar.xml` file to persist `iiop-security-descriptor` elements. See weblogic-ejb-jar.xml Elements in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

Java SE Enhancements

You can find additional information on Java SE enhancements for Java RMI at http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/.

5

Using WebLogic RMI with T3 Protocol

This chapter provides information on using WebLogic RMI with T3 protocol.
This chapter includes the following sections:

- RMI Communication in WebLogic Server
- Determining Connection Availability
- Using a WebLogic T3/T3s Client Proxy

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 *Oracle WebLogic Server 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.

Using a WebLogic T3/T3s Client Proxy

The WebLogic T3/T3s Client Proxy provides the ability to route outbound client requests to a proxy WebLogic T3 server. In this situation, each client routes all outbound requests to the proxy server. The proxy server then directs the request to the WebLogic Server instance that services the request. On both of client and server side, the configuration affects all

applications using a T3 connection as client. For example, if an application creates T3 connection to access a WebLogic T3 server, such as calling methods on remote objects using WebLogic RMI, the proxy configuration is applied to the connection logic.

To enable a client proxy, set the following properties:

T3:

```
-Dhttp.proxyHost=<proxy hostname>
-Dhttp.proxyPort=<proxy port>
-Dhttp.nonProxyHosts=<hostnames>
```

T3s:

```
-Dhttps.proxyHost=<proxy hostname>
-Dhttps.proxyPort=<proxy port>
-Dhttps.nonProxyHosts=<hostnames>
```

where:

- *proxy hostname* is the network address of the user's proxy server.
- *proxy port* is the port number. If not explicitly set, the value of the port number is set to 80.
- *hostnames* is a "|" separated list of one or more host names that WebLogic Server excludes from a proxy configuration. You can use the wildcard character "*" for matching. For example: `-Dhttp.nonProxyHosts="*.oracle.com|localhost"`.

6

How to Implement WebLogic RMI

This chapter describes the `java.rmi.Remote` interface which is the basic building block for all remote objects even though it contains no methods. You extend this "tagging" interface—that is, it functions as a tag to identify remote classes—to create your own remote interface, with method stubs that create a structure for your remote object. Then you implement your own remote interface with a remote class. This implementation is bound to a name in the registry, where a client or server can look up the object and use it remotely.
If you have written RMI classes, you can drop them in WebLogic RMI by changing the import statement on a remote interface and the classes that extend it. To add remote invocation to your client applications, look up the object by name in the registry. WebLogic RMI exceptions are identical to and extend `java.rmi` exceptions so that existing interfaces and implementations do not have to change exception handling.

This chapter includes the following sections:

Creating Classes That Can Be Invoked Remotely

You can write the RMI classes and drop them in WebLogic RMI and change the import statement on a remote interface and the classes that extend the remote interface. WebLogic RMI generates code that has flexible runtime, and creates dynamic bytecode that is independent of the class that implements the interface.

You can write your own WebLogic RMI classes in just a few steps.

Step 1. Write a Remote Interface

Every class that can be remotely invoked implements a remote interface. Write the remote interface in adherence with the following guidelines.

- A remote interface must extend the interface *java.rmi.Remote*, which contains no method signatures. Include method signatures that will be implemented in every remote class that implements the interface.
- The remote interface must be public. Otherwise a client gets an error when attempting to load a remote object that implements it.
- It is not necessary for each method in the interface to declare `java.rmi.RemoteException` in its throws block. The exceptions that your application throws can be specific to your application, and can extend `RuntimeException`. WebLogic

RMI subclasses `java.rmi.RemoteException`, so if you already have existing RMI classes, you will not have to change your exception handling.

- Your Remote interface may not contain much code. All you need are the method signatures for methods you want to implement in remote classes.

  Here is an example of a remote interface with the method signature `sayHello()`.

  WebLogic RMI supports more flexible runtime code generation; WebLogic RMI supports dynamic proxies and dynamically created bytecode that are type-correct but are otherwise independent of the class that implements the interface. If a class implements a single remote interface, the proxy and bytecode that is generated by the compiler will have the same name as the remote interface. If a class implements more than one remote interface, the name of the proxy and bytecode that result from the compilation depend on the name mangling used by the compiler.

Example 6-1 Hello.java Remote Interface

```
package examples.rmi.hello;

import java.rmi.RemoteException;
/**
 * This interface is the remote interface.
 *
 * Copyright (c) 1999,2012, Oracle and/or its affiliates. All Rights Reserved.
 */
public interface Hello extends java.rmi.Remote {
  String sayHello() throws RemoteException;
}
```

Step 2. Implement the Remote Interface

Write the class be invoked remotely. The class should implement the remote interface that you wrote in Step 1, which means that you implement the method signatures that are contained in the interface. All the code generation that takes place in WebLogic RMI is dependent on this class file.

- Your class can implement more than one remote interface. Your class can also define methods that are not in the remote interface, but you cannot invoke those methods remotely.
- Example 6-2 implements a class that creates a `HelloImpl` and binds it to the unique name, `HelloServer`, in the registry. The method `sayHello()` provides a greeting.
- The `main()` method creates an instance of the remote object and registers it in the WebLogic JNDI tree, by binding it to a name (a URL that points to the implementation of the object). A client that needs to obtain a proxy to use the object remotely will be able to look up the object by name.

WebLogic RMI does not require that you set a Security Manager in order to integrate security into your application. Security is handled by WebLogic Server support for SSL and ACLs.

Example 6-2 HelloImpl.java Remote Interface Implementation

```
package examples.rmi.hello;

import javax.naming.*;
import java.rmi.RemoteException;
```

```
/**
 * Copyright (c) 1999,2012, Oracle and/or its affiliates. All Rights Reserved.
 */
public class HelloImpl implements Hello{
  private String name;

  /**
   * Constructs a HelloImpl with the specified string.
   *
   * @param s                 String message
   */
  public HelloImpl(String s) throws RemoteException {
    super();
    name = s;
  }

  /**
   * Returns a string.
   *
   * @return                  String message
   * @exception               java.rmi.RemoteException
   */
  public String sayHello() throws java.rmi.RemoteException {
    return  "Hello World!";
  }

  /**
   * Allows the WebLogic Server to instantiate this implementation
   * and bind it in the registry.
   */
  public static void main(String args[]) throws Exception {


    try {
      HelloImpl obj = new HelloImpl("HelloServer");
      Context ctx = new InitialContext();
      ctx.bind("HelloServer", obj);
      System.out.println("HelloImpl created and bound in the registry " +
                         "to the name HelloServer");

    }
    catch (Exception e) {
      System.err.println("HelloImpl.main: an exception occurred:");
      System.err.println(e.getMessage());
      throw e;
    }
  }
}
```

Step 3: Create a Client that Invokes Remote Methods

In general, once you create an initial context, it takes just a single line of code to get a reference to the remote object. Do this with the `Naming.lookup()` method. The following sections provide additional information on creating clients:

- Setting Client Timeouts
- Example HelloClient.java Client

Setting Client Timeouts

You can set client side timeouts while configuring your initial context:

- To set the amount of time a request waits for a connection response, use the `weblogic.jndi.connectTimeout`.
- To set the amount of time a request waits for a response from the remote server after a connection has been established, use the `weblogic.jndi.responseReadTimeout`.
- See Request Timeouts.

For example:

```
. . .
// Get an InitialContext
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, url);
env.put("weblogic.jndi.connectTimeout", new Long(15000));
env.put("weblogic.jndi.responseReadTimeout", new Long(15000));
return new InitialContext(env);
. . .
```

Example HelloClient.java Client

Here is a short WebLogic client application that uses an object created in Example 6-2.

Example 6-3 Example HelloClient.java Client

```
package examples.rmi.hello;

import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * This client uses the remote HelloServer methods.
 *
 * @author Copyright (c) 1999,2012, Oracle and/or its affiliates. All Rights
Reserved.
 */
public class HelloClient
{
  // 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>");
  }

  public HelloClient() {}

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

```
    if (argv.length < 2) {
      usage();
      return;
    }
    String host = argv[0];
    int port = 0;
    try {
      port = Integer.parseInt(argv[1]);
    }
    catch (NumberFormatException nfe) {
      usage();
      throw nfe;
    }

    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 (Exception ex) {
      System.err.println("An exception occurred: "+ex.getMessage());
      throw ex;
    }
  }

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


}
```

Step 4. Compile the Java Classes

Use `javac` or some other Java compiler to compile the `.java` files to produce `.class` files for the remote interface and the class that implements it.

Example 6-4 provides an Ant script that can be used in the WebLogic Server examples environment to compile the `.java` files and install the `.class` files into the `serverclasses` and `clientclasses` directories configured for the WebLogic Server examplesServer.

Example 6-4 Example build.xml file to Compile Java Classes

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="rmi.hello" default="all" basedir=".">
  <property environment="env"/>
  <property file="../../../examples.properties"/>
  <property name="build.compiler" value="${compiler}"/>
  <!-- set global properties for this build -->
  <property name="source" value="${basedir}"/>
  <target name="all" depends="build"/>
  <target name="build" depends="compile.server, compile.client"/>
  <!-- Compile server classes into the serverclasses directory -->
```

```
  <target name="compile.server">
    <javac srcdir="${source}"
      destdir="${server.classes.dir}"
      includes="Hello.java, HelloImpl.java"
      classpath="${ex.classpath};${server.classes.dir}"
      deprecation="${deprecation}" debug="${debug}" debugLevel="${debugLevel}"
    />
  </target>
  <!-- Compile client classes into the clientclasses directory -->
  <target name="compile.client">
    <javac srcdir="${source}"
      destdir="${client.classes.dir}"
      includes="HelloClient.java"
      classpath="${ex.classpath};${server.classes.dir}"
      deprecation="${deprecation}" debug="${debug}" debugLevel="${debugLevel}"
    />
   </target>
</project>
```

Run the RMI Hello Code Sample

Use the following instructions to run the WebLogic RMI Hello example:

- Prerequisites
- Setup the RMI Hello Example
- Configure a Startup Class
- Restart the examplesServer
- Run the Example

Prerequisites

Install WebLogic server, including the examples. It is assumed that you know how to start the examplesServer and how to set an environment in a shell to run examples.

Setup the RMI Hello Example

Use the following steps to setup the Hello example:

1. Open a shell and set the samples environment.
2. Change to the *ORACLE_HOME*\wlserver\samples\server\examples\src\examples directory, where *ORACLE_HOME* refers to the directory in which you installed WebLogic Server. For more information on the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.
3. Create an `rmi` directory with a subdirectory named `hello`.
4. Copy and save the contents of Example 6-1 as a file named `Hello.java` in the `hello` directory.
5. Copy and save the contents of Example 6-2 as a file named `HelloImpl.java` in the `hello` directory.
6. Copy and save the contents of Example 6-3 as a file named `HelloClient.java` in the `hello` directory.

7. Copy and save the contents of Example 6-4 as a file named `build.xml` in the `hello` directory.
8. Execute the following command from the shell where you copied the example files:

```
ant build
```

Configure a Startup Class

Start an instance of the exampleServer. Create a startup class with the following information:

- Name: `MyHello`
- Class Name: `examples.rmi.hello.HelloImpl`
- Targets: examplesServer

See Configure startup classes in *Oracle WebLogic Server Administration Console Online Help*.

> **Note:**
>
> In this example, the build script makes sure that the startup class is in a location on the server's classpath.

Restart the examplesServer

Restart the examplesServer. As the server boots, you should see the following in the server log:

```
HelloImpl created and bound in the registry to the name HelloServer
```

Once the server is running, you can verify that `HelloServer` is registered by viewing the JNDI tree. See View objects in the JNDI tree in *Oracle WebLogic Server Administration Console Online Help*.

Run the Example

Execute the following command from the shell where you copied the example files:

```
java examples.rmi.hello.HelloClient localhost 7001
```

The results are:

```
Successfully connected to HelloServer on localhost at port 7001: Hello World!
```

7

WebLogic RMI Integration with Load Balancers

This chapter describes WebLogic RMI support for load balancers, including hardware load balancers and web servers with a web server plug-in.
This chapter includes the following sections:

How WebLogic Server Supports Load Balancers

WebLogic Server clients that use RMI can interoperate with a load balancer using the following mechanisms:

- When tunneling T3 over HTTP/HTTPS, WebLogic Server supports routing through a hardware load balancer or a web server with a web server plug-in provided that request forwarding mechanism to the WebLogic Cluster is configured to use sticky session routing. See HTTP Tunneled T3 Load Balancing.
- When using T3 directly, WebLogic Server supports using a hardware load balancer to bootstrap the initial T3 connections to the cluster by specifying a `PROVIDER_URL` that points to the load balancer when creating the JNDI `InitialContext`. See Native T3 Load Balancing.

> **Note:**
>
> All other uses of a hardware load balancer with WebLogic RMI are unsupported—regardless of whether or not they work.

HTTP Tunneled T3 Load Balancing

When tunneling T3 over HTTP (or HTTPS), the WebLogic Server runtime creates an `HttpSession` for each RMI session and passes the session ID back and forth between the client and the server using the normal HTTP mechanisms. This allows the web server plug-in or hardware load balancer to route all RMI requests from a particular client back to the same server in the cluster for the duration of that session. Oracle does not recommend enabling tunneling on channels that are available external to the firewall.

> **Note:**
>
> External load balancers distribute initial context requests that come from Java clients over T3 and the default channel. However, do not route client requests, following the initial context request, through the load balancers. When using the T3 protocol with external load balancer, you must ensure that only the initial context request is routed through the load balancer and that subsequent requests are routed and controlled using WebLogic Server load balancing.

How to Configure the External Listen Address

WebLogic Server provides an `External Listen Address` to provide an IP address to use in RMI stubs to allow clients to connect to the server through a Network Address Translating (NAT) Firewall. As long as the NAT firewall maps a unique external IP address to the unique internal IP address of the server, each stub delivered to the client uniquely identifies the cluster member holding the object that the stub is a proxy for. The `External Listen Address` is set differently for default and custom network channels:

- For the default channel, use the `ExternalDNSName` attribute on the ServerMBean. See ExternalDNSName in *MBean Reference for Oracle WebLogic Server*.
- For a custom channel, use the `PublicAddress` and `PublicPort` on the NetworkAccessMBean. See NetworkAccessPointMBean in *MBean Reference for Oracle WebLogic Server*.

Example Custom Channel Configuration for a Load Balancer

Configure a T3 network channel on all WebLogic Server instances in the cluster. The network channel accepts tunneled traffic from the load balancer. To ensure all client requests are routed through the load balancer, set `External Listen Address` to the end point where loadbalancer, or the web server, accepts traffic from the client. Enable `HTTP` protocol and set `tunneling-enabled=true`. Configure the load balancer or web server to route http traffic to WebLogic Server. If using Oracle HTTP Server (OHS) as a webserver, this can be achieved by changing the `httpd.conf` configuration file. For example:

The WebLogic Server `config.xml`:

```
<network-access-point>
      <name>tunnelChannel</name>
      <protocol>t3</protocol>
      <listen-address>example.com</listen-address>
      <listen-port>11001</listen-port>
      <http-enabled-for-this-protocol>true</http-enabled-for-this-protocol>
      <tunneling-enabled>true</tunneling-enabled>
      <outbound-enabled>false</outbound-enabled>
      <enabled>true</enabled>
      <two-way-ssl-enabled>false</two-way-ssl-enabled>
      <client-certificate-enforced>false</client-certificate-enforced>
</network-access-point><network-access-point>
. . .
```

OHS/Webtier's `httpd.conf` file

```
<LocationMatch ^/bea_wls_internal/>
SetHandler weblogic-handler
WeblogicCluster example.com:11001
</LocationMatch>
. . .
```

Session Failover

Session failover is transparent to the client. When a server shuts down the client RJVM receives a `PeerGone` exception. This causes the `HTTPClientJVMConnection` to be closed. When the next request comes from the same client, the request is failed over to the next member in the cluster for both stateless and stateful beans. If an exception occurs during request processing, that request is not failed over and the exception is propagated to the client.

Cookie Persistence

The tunneling client caches the cookie it receives after initial request and sends it back in every subsequent request.

Pinned Objects

In a cluster, even if an object is pinned and the `replicate_bindings!= false`, the stub is replicated to all the members of the cluster. Tunneling does not affect the normal pinned object behavior.

Stateful Session EJBs

If `External Listen Address` is not set, the stub that the client gets back has the list of available hosts to route to and the behavior is similar to sending direct t3 requests.

If `External Listen Address` is set then failover does not work because the primary and secondary hosts get set to the `externalDNSName` and load balancer hangs trying to route to itself.

Native T3 Load Balancing

If the cluster member fails, the client invocation on a *non-cluster-aware stub* also fails since the firewall does not attempt to redirect the request to another cluster member. For a cluster-aware stub invocation, the request should be transparently routed around the failure and the invocation delivered to a different cluster member using the `External Listen Address` contained in the cluster-aware stub. See How to Configure the External Listen Address.

Use the hardware load balancer to load balance the initial T3 connection request when creating the JNDI `InitialContext` by specifying a `PROVIDER_URL` that points to the load balancer provided that the `External Listen Address` is not set to point to the hardware load balancer. This configuration works because the hardware load balancer is only involved in routing the initial TCP connection request to one of the managed servers. Once the connection is established, all RMI stubs contain the server's `ListenAddress` (or `External Listen Address` in the case of a NAT firewall) that uniquely identifies the server for which the stub is acting as a proxy.

Failover Support

WebLogic RMI does not support failover when used with a hardware loadbalancer.

For information on how WebLogic Server RMI handles failover, see Failover and Load Balancing RMI Objects.

8

Using RMI over IIOP

This chapter provides a high-level view of RMI over IIOP (RMI-IIOP) and RMI-IIOP interoperability between this release and prior WebLogic Server releases.
This chapter includes the following sections:

What is RMI over IIOP?

RMI over IIOP extends RMI to work across the IIOP protocol. This has two benefits that you can leverage. In a Java to Java paradigm, this allows you to program against the standardized Internet Interop-Orb-Protocol (IIOP). If you are not working in a Java-only environment, it allows your Java programs to interact with Common Object Request Broker Architecture (CORBA) clients and execute CORBA objects. CORBA clients can be written in a variety of languages (including C++) and use the Interface-Definition-Language (IDL) to interact with a remote object.

Overview of WebLogic RMI-IIOP

WebLogic Server provides its own ORB implementation which is instantiated by default when programs call `ORB.init()`, or when `"java:comp/ORB"` is looked up in JNDI. See CORBA Support for WebLogic Server for information how WebLogic Server complies with specifications for CORBA support in Java SE.

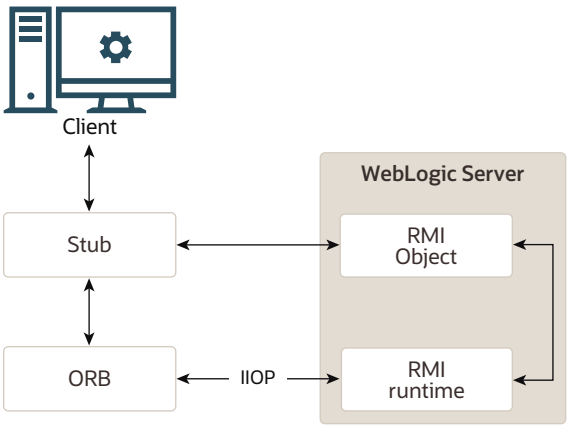
The WebLogic Server implementation of RMI-IIOP allows you to:

- Connect Java RMI clients to WebLogic Server using the standardized IIOP protocol
- Connect CORBA/IDL clients, including those written in C++, to WebLogic Server
- Interoperate between WebLogic Server and Tuxedo clients
- Connect a variety of clients to EJBs hosted on WebLogic Server

How you develop your RMI-IIOP applications depends on what services and clients you are trying to integrate. See *Developing Standalone Clients for Oracle WebLogic Server* for more information on how to create applications for various clients types that use RMI and RMI-IIOP.

The following diagram shows RMI Object Relationships for objects that use IIOP.

Figure 8-1 RMI Object Relationships



Support for RMI-IIOP with RMI (Java) Clients

You can use RMI-IIOP with Java/RMI clients, taking advantage of the standard IIOP protocol. WebLogic Server provides multiple options for using RMI-IIOP in a Java-to-Java environment, including the new Java EE Application Client (thin client), which is based on the new small footprint client jar. To use the new thin client, you need to have the `wlclient.jar` (located in `WL_HOME/server/lib`) on the client side's CLASSPATH. For more information on RMI-IIOP client options, see *Developing Standalone Clients for Oracle WebLogic Server*.

Support for RMI-IIOP with Tuxedo Client

WebLogic Server contains an implementation of the WebLogic Tuxedo Connector, an underlying technology that enables you to interoperate with Tuxedo servers. Using WebLogic Tuxedo Connector, you can leverage Tuxedo as an ORB, or integrate legacy Tuxedo systems with applications you have developed on WebLogic Server. See the *Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*.

Support for RMI-IIOP with CORBA/IDL Clients

The developer community requires the ability to access Java EE services from CORBA/IDL clients. However, Java and CORBA are based on very different object models. Because of this, sharing data between objects created in the two programming paradigms was, until recently, limited to Remote and CORBA primitive data types. Neither CORBA structures nor Java objects could be readily passed between disparate objects. To address this limitation, the Object Management Group (OMG) created the *Objects-by-Value Specification* at `http://www.omg.org/technology/documents/index.htm`. This specification defines the enabling technology for exporting the Java object model into the CORBA/IDL programming model--allowing for the interchange of complex data types between the two models. WebLogic Server can support Objects-by-Value with any CORBA ORB that correctly implements the specification.

9

Configuring WebLogic Server for RMI-IIOP

This chapter describes the concepts and procedures necessary to configure WebLogic Server to interoperate using RMI over IIOP (RMI-IIOP).
This chapter includes the following sections:

Set the Listening Address

To facilitate the use of IIOP, always specify a valid IP address or DNS name for the Listen Address attribute in the configuration file (`config.xml`) to listen for connections.

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 *Administering Server Environments for Oracle WebLogic Server*.

Setting Network Channel Addresses

The following sections provide information to consider when implementing IIOP network channel addresses for thin clients.

Considerations for Proxys and Firewalls

Many typical environments use firewalls, proxys, or other devices that hide the application server's true IP address. Because IIOP relies on a per-object addressing scheme where every object contains a host and port, anything that masks the true IP address of the server will prevent the external client from maintaining a connection. To prevent this situation, set the `PublicAddress` on the server IIOP network channel to the virtual IP that the client sees.

Considerations for Clients with Multiple Connections

IIOP clients publish addressing information that is used by the application server to establish a connection. In some situations, such as running a VPN where clients have more than one

connection, the server cannot see the IP address published by the client. In this situation, you have two options:

- Use a bi-directional form of IIOP. Use the following WebLogic flag:

  ```
  -Dweblogic.corba.client.bidir=true
  ```

  In this instance, the server does not need the IP address published by the client because the server uses the inbound connection for outbound requests.
- Use the following JDK property to set the address the server uses for outbound connections:

  ```
  -Dcom.sun.CORBA.ORBServerHost=client_ipaddress
  ```

  where `client_ipaddress` is an address published by the client.

Using a IIOPS Thin Client Proxy

The IIOPs Thin Client Proxy provides a WebLogic thin client the ability to proxy outbound requests to a server. In this situation, each user routes all outbound requests through their proxy. The user's proxy then directs the request to the WebLogic Server. You should use this method when it is not practical to implement a Network Channel. To enable a proxy, set the following properties:

```
-Diiops.proxyHost=<host>
-Diiops.proxyPort=<port>
```

where:

- *hostname* is the network address of the user's proxy server.
- *port* is the port number. If not explicitly set, the value of the port number is set to 80.
- *hostname* and *port* support symbolic names, such as:

  ```
  -Diiops.proxyHost=https.proxyHost
  -Diiops.proxyPort=https.proxyPort
  ```

You should consider the following security implications:

- This feature does not change the behavior of WebLogic Server. However, using this feature does expose IP addresses though the client's firewall. As both ends of the connection are trusted and the linking information is encrypted, this is an acceptable security level for many environments.
- Some production environments do not allow enabling the `CONNECT` attribute on the proxy server. These environments should use HTTPS tunneling. See Setting Up WebLogic Server for HTTP Tunneling in *Administering Server Environments for Oracle WebLogic Server*.

Using RMI-IIOP with SSL and a Java Client

The Java clients that support SSL are the thin client and the WLS-IIOP client. To use SSL with these clients, simply specify an `ssl` URL.

Accessing WebLogic Server Objects from a CORBA Client through Delegation

WebLogic Server provides services that allow CORBA clients to access RMI remote objects. As an alternative method, you can also host a CORBA ORB (Object Request Broker) in WebLogic Server and delegate incoming and outgoing messages to allow CORBA clients to indirectly invoke any object that can be bound in the server.

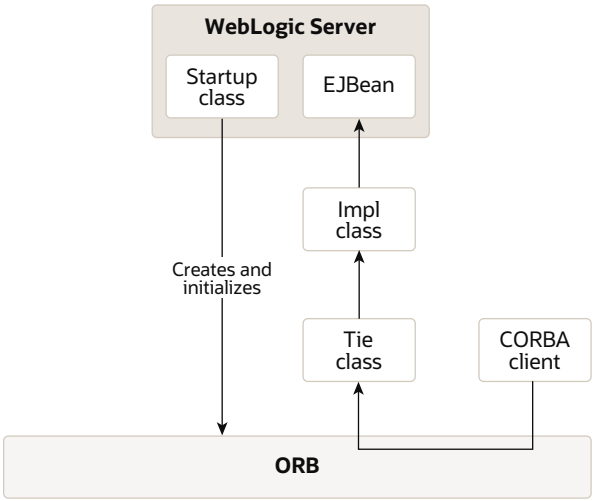
Overview of Delegation

Here are the main steps to create the objects that work together to delegate CORBA calls to an object hosted by WebLogic Server.

1. Create a startup class that creates and initializes an ORB so that the ORB is co-located with the JVM that is running WebLogic Server.
2. Create an IDL (Interface Definition Language) that will create an object to accept incoming messages from the ORB.
3. Compile the IDL. This will generate a number of classes, one of which will be the Tie class. Tie classes are used on the server side to process incoming calls, and dispatch the calls to the proper implementation class. The implementation class is responsible for connecting to the server, looking up the appropriate object, and invoking methods on the object on behalf of the CORBA client.

The following figure is a diagram of a CORBA client invoking an EJB by delegating the call to an implementation class that connects to the server and operates upon the EJB. Using a similar architecture, the reverse situation will also work. You can have a startup class that brings up an ORB and obtains a reference to the CORBA implementation object of interest. This class can make itself available to other WebLogic objects throughout the JNDI tree and delegate the appropriate calls to the CORBA object.

Figure 9-1 CORBA Client Invoking an EJB with a Delegated Call



Example of Delegation

The following code example creates an implementation class that connects to the server, looks up the `Foo` object in the JNDI tree, and calls the `bar` method. This object is also a startup class that is responsible for initializing the CORBA environment by:

- Creating the ORB
- Creating the Tie object
- Associating the implementation class with the Tie object
- Registering the Tie object with the ORB
- Binding the Tie object within the ORB's naming service

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.rmi.*;
import javax.naming.*;
import weblogic.jndi.Environment;

public class FooImpl implements Foo
{
  public FooImpl() throws RemoteException {
    super();
  }
  public void bar() throws RemoteException, NamingException {
    // look up and call the instance to delegate the call to...
    weblogic.jndi.Environment env = new Environment();
    Context ctx = env.getInitialContext();
    Foo delegate = (Foo)ctx.lookup("Foo");
    delegate.bar();
    System.out.println("delegate Foo.bar called!");
  }
  public static void main(String args[]) {
    try {
      FooImpl foo = new FooImpl();

      // Create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // Create and register the tie with the ORB
      _FooImpl_Tie fooTie = new _FooImpl_Tie();
      fooTie.setTarget(foo);
      orb.connect(fooTie);

      // Get the naming context
      org.omg.CORBA.Object o = \
      orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(o);

      // Bind the object reference in naming

      NameComponent nc = new NameComponent("Foo", "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, fooTie);

      System.out.println("FooImpl created and bound in the ORB registry.");
    }
```

```
    catch (Exception e) {
      System.out.println("FooImpl.main: an exception occurred:");
      e.printStackTrace();
    }
  }
}
```

Configuring CSIv2 authentication

The Common Secure Interoperability Specification, Version 2 (CSIv2) is an Open Management Group (OMG) specification that addresses the requirements of Common Object Request Broker Architecture (CORBA) security for interoperable authentication, delegation, and privileges. See Common Secure Interoperability Version 2 (CSIv2) in *Understanding Security for Oracle WebLogic Server*.

Use the following steps to use CSIv2 to authenticate an inbound call from a remote domain:

1. Update the Identity Asserter. See Configuring Identity Assertion Providers in *Administering Security for Oracle WebLogic Server*.
2. Update the User Name Mapper. See Configuring a User Name Mapper in *Administering Security for Oracle WebLogic Server*.
3. Add all users required by the application in the remote domain to the WebLogic AuthenticationProvider. See Create User in *Oracle WebLogic Server Administration Console Online Help*.

Using RMI over IIOP with a Hardware Load Balancer

> **Note:**
>
> This feature works correctly only when the bootstrap is through a hardware load-balancer.

An optional enhancement for WebLogic Server Oracle ORB and higher, supports hardware load balancing by forcing reconnection when bootstrapping. This allows hardware load-balancers to balance connection attempts

In most situations, once a connection has been established, the next `NameService` lookup is performed using the original connection. However, since this feature forces re-negotiation of the end point to the hardware load balancer, all in-flight requests on any existing connection are lost.

Use the `-Dweblogic.system.iiop.reconnectOnBootstrap` system property to set the connection behavior of the Oracle ORB. Valid values are:

- true—Forces re-negotiation of the end point.
- false—Default value.

Environments requiring a hardware load balancer should set this property to true.

Limitations of WebLogic RMI-IIOP

The following sections outline various issues relating to WebLogic RMI-IIOP.

Limitations Using RMI-IIOP on the Client

Use WebLogic Server with JDK 1.3.1_01 or higher. Earlier versions are not RMI-IIOP compliant. Note the following about these earlier JDKs:

- Send GIOP 1.0 messages and GIOP 1.1 profiles in IORs.
- Do not support the necessary pieces for EJB 2.0 interoperation (GIOP 1.2, codeset negotiation, UTF-16).
- Have bugs in its treatment of mangled method names.
- Do not correctly unmarshal unchecked exceptions.
- Have subtle bugs relating to the encoding of valuetypes.

Many of these items are impossible to support both ways. Where there was a choice, WebLogic supports the spec-compliant option.

Limitations Developing Java IDL Clients

Oracle strongly recommends developing Java clients with the RMI client model if you are going to use RMI-IIOP. Developing a Java IDL client can cause naming conflicts and classpath problems, and you are required to keep the server-side and client-side classes separate. Because the RMI object and the IDL client have different type systems, the class that defines the interface for the server-side will be very different from the class that defines the interface on the client-side.

Limitations of Passing Objects by Value

To pass objects by value, you need to use value types, (see `http://www.omg.org/cgi-bin/doc?formal/01-02-33`, You implement value types on each platform on which they are defined or referenced. This section describes the difficulties of passing complex value types, referencing the particular case of a C++ client accessing an Entity bean on WebLogic Server.

One problem encountered by Java programmers is the use of derived datatypes that are not usually visible. For example, when accessing an EJB finder the Java programmer will see a Collection or Enumeration, but does not pay attention to the underlying implementation because the JDK run-time will classload it over the network. However, the C++, CORBA programmer must know the type that comes across the wire so that he can register a value type factory for it and the ORB can unmarshal it.

Simply running `ejbc` on the defined EJB interfaces will *not* generate these definitions because they do not appear in the interface. For this reason `ejbc` will also accept Java classes that are not remote interfaces—specifically for the purpose of generating IDL for these interfaces. Review the `/iiop/ejb/entity/cppclient` example to see how to register a value type factory.

Java types that are serializable but that define `writeObject()` are mapped to custom value types in IDL. You must write C++ code to unmarshal the value type manually.

> **Note:**
>
> When using Tuxedo, you can specify the `-i` qualifier to direct the IDL compiler to create implementation files named `FileName_i.h` and `FileName_i.cpp`. For example, this syntax creates the `TradeResult_i.h` and `TradeResult_i.cpp` implementation files:
>
> ```
> idl -IidlSources -i   idlSources\examples\iiop\ejb\iiop\TradeResult.idl
> ```

The resulting source files provide implementations for application-defined operations on a value type. Implementation files are included in a CORBA client application.

Propagating Client Identity

Until recently insufficient standards existed for propagating client identity from a CORBA client. If you have problems with client identity from foreign ORBs, you may need to implement one of the following methods:

- The identity of any client connecting over IIOP to WebLogic Server will default to `<anonymous>`. You can set the user and password in the `config.xml` file to establish a single identity for all clients connecting over IIOP to a particular instance of WebLogic Server, as shown in the example below:

  ```
  <Server
  Name="myserver"
  NativeIOEnabled="true"
  DefaultIIOPUser="Bob"
  DefaultIIOPPassword="Gumby1234"
  ListenPort="7001">
  ```

- You can also set the `IIOPEnabled` attribute in the `config.xml`. The default value is `"true"`; set this to `"false"` only if you want to disable IIOP support. No additional server configuration is required to use RMI over IIOP beyond ensuring that all remote objects are bound to the JNDI tree to be made available to clients. RMI objects are typically bound to the JNDI tree by a startup class. EJB homes are bound to the JNDI tree at the time of deployment. WebLogic Server implements a `CosNaming Service` by delegating all lookup calls to the JNDI tree.
- This release supports RMI-IIOP `corbaname` and `corbaloc` JNDI references. See `http://www.omg.org/cgi-bin/doc?formal/01-02-33`. One feature of these references is that you can make an EJB or other object hosted on one WebLogic Server available over IIOP to other Application Servers. So, for instance, you could add the following to your `ejb-jar.xml`:

  ```
  <ejb-reference-description>
  <ejb-ref-name>WLS</ejb-ref-name>
  <jndi-name>corbaname:iiop:1.2@localhost:7001#ejb/javaee/interop/foo</jndi-name>
  </ejb-reference-description>
  ```

The reference-description stanza maps a resource reference defined in `ejb-jar.xml` to the JNDI name of an actual resource available in WebLogic Server. The `ejb-ref-name` specifies a resource reference name. This is the reference that the EJB provider places within the `ejb-jar.xml` deployment file. The `jndi-name` specifies the JNDI name of an actual resource factory available in WebLogic Server.

> **Note:**
>
> The `iiop:1.2` contained in the `<jndi-name>` section. This release contains an implementation of GIOP (General-Inter-Orb-Protocol) 1.2. The GIOP specifies formats for messages that are exchanged between inter-operating ORBs. This allows interoperability with many other ORBs and application servers. The GIOP version can be controlled by the version number in a `corbaname` or `corbaloc` reference.

These methods are not required when using `WLInitialContextFactory` in RMI clients or can be avoided by using the WebLogic C++ client.

10

Best Practices for Application Design

This chapter describes recommended design patterns when programming with RMI and RMI over IIOP.
This chapter includes the following sections:

Use java.rmi

Oracle recommends RMI users use `java.rmi`, see `http://docs.oracle.com/javase/8/docs/api/java/rmi/package-summary.html`. Although the WebLogic API contains the weblogic.rmi API, it is deprecated and is only provided as a compatibility API. Other WebLogic APIs provided for compatibility are:

- `weblogic.rmi.registry`
- `weblogic.rmi.server`
- `weblogic.rmi.extensions`

Use PortableRemoteObject

To maintain code portability, always use `PortableRemoteObject` when casting the home interfaces. For example:

```
Propshome home = (PropsHome)
PortableRemoteObject.narrow(
ctx.lookup( "Props" ),
PropsHome.class );
```

Use WebLogic Work Areas

A best practice is to use Work Areas:

- Work Contexts allow Java EE developers to define properties as application context which implicitly flow across remote requests and allow downstream components to work in the context of the invoking client. Work Contexts allow developers to pass properties without including them in a remote call. A Work Context is propagated with each remote call-allowing the called component to add or modify properties defined in the Work Context; similarly, the calling component can access the Work Context to obtain new or updated properties.
- Work Contexts ease the processing of implementing and maintaining functionality that requires that information to be passed to remote components, such as diagnostics

monitoring, application transactions, and application load-balancing. Work Contexts are also a useful mechanism for providing information to third-party components.

- Work Contexts can propagate user-defined properties across all request scopes supported by WebLogic Server-a Work Context is available to all of the objects that can exist within the request scope, including RMI calls. See *Developing Applications for Oracle WebLogic Server*.

How to Handle Changes in Security Context

WLS RMI does not carry forward the security context in the stub. The thread that establishes the stub has the right subject in its thread context. If the stub is later used in a different thread or the stub is used after the current thread context has changed as a result of some operations, subsequent calls using the stub may fail with `SecurityException`. Operations that can change the context of a thread include establishing a new initial context and running WLST programmatically. Thread context changes often surface as cross-domain security issues when using JMS, JTA, and MDBs in multi-domain configurations.

If an RMI stub is going to be used in a different thread, the application can use a JSR-237 work manager to schedule the new thread in the thread context that the stub is created so that the thread context is propagated to the new thread. For cases where this is not possible, or cases where the context of the original thread changes somehow, the application should reestablish the context under which the stub should be invoked with JAAS. The following public APIs can be used to reestablish the security context:

- `weblogic.security.Security.getCurrentSubject()`—obtain the current object on the thread.
- `weblogic.security.Security.runAs()`—resume the subject.

A

CORBA Support for WebLogic Server

This appendix provides the official specifications for CORBA support for this release of WebLogic Server.
This appendix includes the following sections:

Specification References

In general, this release of WebLogic Server adheres to the OMG specifications required by Java EE. For this release, the WebLogic ORB is compliant with following specification references:

- *CORBA 2.6: formal/01-12-01* at `http://www.omg.org/cgi-bin/doc?formal/01-12-01`
- *CORBA 2.3.1: formal/99-10-07* at `http://www.omg.org/cgi-bin/doc?formal/99-10-07`
- *IDL to Java language mapping: ptc/03-09-04* at `http://www.omg.org/cgi-bin/doc?ptc/03-09-04`
- *Revised IDL to Java language mapping 1.3: formal/00-11-03* at `http://www.omg.org/cgi-bin/doc?formal/00-11-03`
- *Java to IDL language mapping: ptc/00-01-06* at `http://www.omg.org/cgi-bin/doc?ptc/00-01-06`
- *Interoperable Naming Service: ptc/00-08-07* at `http://www.omg.org/cgi-bin/doc?ptc/00-08-07`
- *Transaction Service 1.2.1: formal/2001-11-03* at `http://www.omg.org/cgi-bin/doc?formal/2001-11-03`

> **Note:**
>
> If the above links do not take you to the referenced specification, the OMG may have changed the URL. You can search `http://www.omg.org` for the correct specification.

Supported Specification Details

Not all of the above specifications are implemented in the WebLogic ORB in this release. The following section provides a precise list of the supported specifications by chapter or section:

- CORBA 2.6, chapters 1-3, 6-7, 13 and 15.

- Revised IDL to Java language mapping, section 1.21.8.2, the orb.properties file.
- CORBA 2.6, chapter 4 and 5, excepting details relevant to excluded features from other chapters, such as PortableInterceptors.
- CORBA 2.6, sections 10.6.1 and 10.6.2 are supported for repository IDs.
- CORBA 2.6, section 10.7 for `TypeCode` APIs.
- CORBA 2.6, chapter 11, Portable Object Adapter (POA) excepting details relevant to excluded features from other chapters, such as PortableInterceptors.
- CORBA 2.6, chapter 26, conformance level 0 plus stateful.
- The Interoperable Naming Service.
- Section 1.21.8 of the Revised IDL to Java Language Mapping Specification (ptc/00-11-03) has been changed from the version in the IDL to Java Language Mapping Specification (ptc/00-01-08).
- Transaction Service 1.2.1, as defined by the EJB 2.1 specification.

Tools

For this release, the WebLogic ORB is compliant with the following tools:

- The IDL to Java compiler (`idlj`) is the one that comes bundled with Java SE and is compliant with following specification references:
  - CORBA 2.3.1, chapter 3 (IDL definition).
  - CORBA 2.3.1, chapters 5 and 6 (semantics of Value types).
  - CORBA 2.3.1, section 10.6.5 (pragmas).
  - The IDL to Java mapping specification.
  - The Revised IDL to Java language mapping specification section 1.12.1 (local interfaces).
- The Java to IDL compiler (the IIOP backend for *rmic*) complies with:
  - CORBA 2.6, chapters 5 and 6 (value types).
  - The Java to IDL language mapping. Note that this implicitly references section 1.21 of the IDL to Java language mapping.
  - IDL generated by the `-idl` flag complies with CORBA 2.6 chapter 3.

Index