BEA WebLogic Server

Programming WebLogic RMI

BEA WebLogic Server 6.0
Document Date: March 3, 2001

Programming WebLogic RMI

Document Edition	Date	Software Version
	March 3, 2001	BEA WebLogic Server Version 6.0

Contents

About This Document

Audience....v
e-docs Web Site.... vi
How to Print the Document.... vi
Related Information.... vi
Contact Us!.... vii
Documentation Conventions.... vii

1. Introduction to WebLogic RMI

Introducing WebLogic RMI.... 1-1
Advantages of WebLogic RMI.... 1-2
  Differences in WebLogic RMI's Implementation.... 1-3
WebLogic RMI Features.... 1-5
    WebLogic RMI Compiler.... 1-5
    Proxy Classes.... 1-6
    WebLogic RMI Registry and Server.... 1-6
WebLogic RMI Performance and Scalability.... 1-6

2. Programming WebLogic RMI

WebLogic RMI Compiler.... 2-1
  WebLogic RMI Compiler Options.... 2-2
  Additional WebLogic RMI Compiler Features.... 2-4
Proxies in WebLogic RMI.... 2-5
  Using the WebLogic RMI Compiler with Proxies.... 2-5
WebLogic RMI Registry.... 2-6
Implementation Features.... 2-6

3. WebLogic RMI API

Overview of the WebLogic RMI API 3-1
Implementing with WebLogic RMI 3-2
  Creating classes that can be invoked remotely........ 3-2
    Step 1. Write a Remote interface 3-3
    Step 2. Implement the Remote interface 3-4
    Step 3. Compile the java class........ 3-6
    Step 4. Compile the implementation class with RMI compiler 3-6
  Invoking methods on remote objects in your client code........ 3-7
  Full Code Examples 3-7
Using the WebLogic RMI Compiler for Clustered Services........ 3-9
  Cluster-specific RMI compiler options 3-9
  Non-Replicated Stubs........ 3-10

About This Document

This document introduces BEA WebLogic Server™ RMI features and describes the RMI implementation that run on the WebLogic Server platform.

The document is organized as follows:

- Chapter 1, “Introduction to WebLogic RMI,” is an overview of WebLogic Server RMI features and its architecture.
- Chapter 2, “Programming WebLogic RMI,” describes how to use the WebLogic RMI functions and features.
- Chapter 3, “WebLogic RMI API,” describes the packages shipped as part of WebLogic RMI. The public API includes the WebLogic implementation of the RMI base classes, the registry, and the server packages.

Audience

This document is written for application developers who want to build e-commerce applications using the Remote Method Invocation (RMI) features. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. In additon to this document you may want to review Programming RMI over IIOP. 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)

Contact Us!

Your feedback on the BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName`*`;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n name -a address`
	`     [-p portnumber] [-t timeout] [-s send]`

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`     password {application} {source}`

`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

1 Introduction to WebLogic RMI

The following sections introduce WebLogic RMI and describe the advantages and features.

- Introducing WebLogic RMI
- Advantages of WebLogic RMI
- WebLogic RMI Features
- WebLogic RMI Performance and Scalability

Introducing WebLogic RMI

Remote Method Invocation (RMI) is the standard for distributed object computing in Java. RMI allows an application to obtain a reference to an object that exists elsewhere in the network, and then to 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.

WebLogic implements the JavaSoft RMI specification. WebLogic RMI provides standards-based distributed object computing. WebLogic Server enables fast, reliable, large-scale network computing, and WebLogic RMI allows products, services, and resources to exist anywhere on the network but appear to the programmer and the end user as part of the local environment.

WebLogic RMI scales linearly under load, and execution requests can be partitioned into a configurable number of server threads. Multiple server threads allow WebLogic Server to take advantage of latency time and available processors.

There are differences in the JavaSoft reference implementation of RMI and WebLogic's RMI product; however, these differences are completely transparent to the developer. WebLogic RMI is WebLogic's implementaion of JavaSoft RMI.

In addition, WebLogic RMI is fully integrated with WebLogic JNDI. Applications can be partitioned into meaningful name spaces by using either the JNDI API or the Registry interfaces in WebLogic RMI.

The WebLogic RMI compiler generates stubs and skeltons that completely replace the stubs and skeltons produced by RMIC, the RMI compiler. Like RMIC, the WebLogic RMI compiler produces enough compile-time information to support runtime resolution of classes, methods, and instances to override Object methods, and to deliver exceptions raised in the server to the invoking clients.

If you are just beginning to learn about RMI, visit the JavaSoft website and take the RMI tutorial. This document contains information about using WebLogic RMI, but it is not a beginner's tutorial on remote objects or writing distributed applications.

Advantages of WebLogic RMI

As a service that operates within WebLogic Server, WebLogic RMI has some characteristics that differ from JavaSoft's reference implementation of RMI. These characteristics do not change how you uses WebLogic RMI, but they do affect performance and scalability.

WebLogic RMI is completely standards-compliant. If you are an RMI user, you can convert your programs by changing nothing more than the import statement and running your remote classes through the WebLogic RMI compiler..

Here is a brief comparison of WebLogic RMI and the JavaSoft RMI reference implementation. In general, like JavaSoft's reference implementation of RMI, WebLogic RMI provides transparent remote invocation in different JVMs. Remote interfaces and implementations that are written to the RMI specification can be compiled with the WebLogic RMI compiler and used without changes. But there are significant differences in the fundamental implementation of WebLogic RMI that enhance the many aspects of distributed computing.

Differences in WebLogic RMI's Implementation

The following sections identify some of the differences in WebLogic's implementaion of RMI compaired to the JavaSoft RMI implementation.

Performance-related issues

Management of threads and sockets. The reference implementation uses multiple sockets—an expensive, limited resource — to support communications among the server, clients, and the RMI registry. WebLogic RMI uses a single, multiplexed, asynchronous, bidirectional connection for WebLogic RMI client-to-network traffic. The same connection may also support WebLogic JDBC requests or other services. With WebLogic RMI, the complicated virtual circuit that is maintained by the reference implementation is replaced by a single connection to the WebLogic Server.

Serialization. WebLogic RMI uses serialization, which offers a significant performance gain, even for one-time use of a remote class.

Resolution of co-located objects. Unlike the reference implementation, with WebLogic RMI there is no performance penalty for co-located objects that are defined as remote. References to co-located "remote" objects are resolved as direct references to the actual implementation object, rather than to the generated proxies.

Processes for supporting services. The WebLogic RMI registry, a functional replacement for the RMI registry process, runs insides the WebLogic Server; no additional processes are needed for WebLogic RMI.

Garbage collection. JavaSoft's RMI implementation supports distributed garbage collection, which is an expensive task. With WebLogic RMI, garbage collection is handled more efficiently at the session level by the WebLogic Server.

Power and flexibility

SmartStub support. Stubs in WebLogic RMI can be "SmartStubs," which are custom stubs that allow for application-specific partitioning of logic and data *at the object level* between client and server. SmartStubs override the generated stubs and permit custom optimizations of WebLogic RMI infrastructure.

Ease of use

Fewer things to implement. WebLogic RMI provides ease-of-use extensions for the use of remote interfaces and code generation. For example, a remote method does not need to declare RemoteException, and WebLogic RMI does not require a separate stub and skeleton class for every remote class.

Security without a Security Manager. There is no requirement in WebLogic RMI to set a Security Manager. You may comment out the call to `setSecurityManager()` when converting RMI code to WebLogic RMI. Since all WebLogic RMI services are provided by WebLogic Server, which has other more sophisticated options for security SSL and ACLs, there is no need for separate functionality that applies only to the WebLogic RMI service.

Naming/Registry issues

Flexible naming and lookup. WebLogic RMI allows you to chose among several schemes for naming, binding, and looking up remote objects. In your URLs, you can use the standard **rmi://** scheme, or **http://** which tunnels WebLogic RMI requests over HTTP, thus making WebLogic RMI remote invocation available even through firewalls.

WebLogic RMI is fully integrated with WebLogic JNDI. Applications can be partitioned into meaningful name spaces by using either the Java Naming and Directory Interface (JNDI) API or the registry interfaces in WebLogic RMI. JNDI allows publication of RMI objects through enterprise naming services, such as LDAP or NDS.

Client-side invocation

Client-to-server, client-to-client, or server-to-client invocations. Since WebLogic RMI operates within a well-defined server environment, where clients and servers are connected with optimized, multiplexed, asynchronous, bidirectional connections, WebLogic RMI can support client-side callbacks into applications. This architecture allows clients to host remote objects with levels of service and resource access that are similar to those supported by the WebLogic Server.

This capability enables a client application to publish its objects through the registry. Other clients or servers can use the client-resident objects just like any server-resident objects. No performance penalty is imposed on the publishing client for using its own objects, even though stubs and skeletons will exist for accessing the objects remotely. When a second client looks up the object and then invokes on it, the request will be routed through the

WebLogic Server to the first client, which will broker the request and invoke the remote object within its local JVM.

Inheritance

Preservation of a logical object hierarchy. There is no requirement in WebLogic RMI to extend UnicastRemoteObject.This functionality is built into WebLogic Server and preserves your object hierarchy. There is no artificial need for every remote class to inherit from UnicastRemoteObject in order to inherit implementation from the `rmi.server` package; rather, your remote classes can inherit from classes within your application hierarchy and yet retain the behavior of the `rmi.server` package.

Instrumentation and management

The reference implementation RMI server obscures RMI operations; it has few tools for debugging or monitoring your RMI classes. WebLogic Server, which hosts the RMI registry, provides a well-instrumented environment for development and deployment of distributed applications.

WebLogic RMI Features

Like the JavaSoft reference implementation of RMI, WebLogic RMI has a code generator, and a registry and server hosted by a WebLogic Server. But WebLogic RMI provides features that are different or missing from the reference implementation.

WebLogic RMI Compiler

The WebLogic RMI code generator (compiler) produces a stub and a skeleton that support the interfaces implemented by the remote object. The object’s implementation is bound to a name in the RMI registry, and any client can acquire a remote stub of the object upon which it can invoke by looking up the object in the registry.

With WebLogic RMI it is possible to specify that platform-specific compilers should be used. In addition, the WebLogic RMI compiler accepts and passes on any additional Java compiler options to the Java compiler.

The WebLogic RMI compiler allows a flexible inheritance structure; remote classes can also implement non-remote interfaces, and code generation can be done on the descendants of a class, or on an abstract class. Your distributed application can exist in a meaningful object hierarchy, rather than the artificial hierarchy that results under the reference RMI where every remote class must inherit from a single interface.

Proxy Classes

The WebLogic RMI compiler increases efficiency in its use of proxies. Proxy classes are the resulting skeleton and stub classes that any RMI compiler produces when run against a remote class (that is, one that implements a remote interface). The WebLogic RMI compiler by default produces proxies for the *remote interface*, and the remote classes share the proxies. This is a much more efficient system than the reference implementation's model of producing proxies for each remote class. When a remote object implements more than one interface, the proxy names and packages are determined by encoding the set of interfaces, unless you choose to produce class-specific proxies. When a class-specific proxy is found, it takes precedence over the interface-specific proxy.

WebLogic RMI Registry and Server

With WebLogic RMI, the RMI registry is hosted by WebLogic Server, which provides the necessary networking infrastructure and execution model for using RMI in a production environment.

WebLogic Server, and likewise the registry, can be accessed by a variety of client protocols, includinga secure (SSL) mode of WebLogic's protocol, HTTP tunneling, HTTPS, and IIOP. A call to look up an object in the WebLogic registry may use various URL schemes, including the default **rmi://** scheme, **http://**, **https://**, and **iiop://**.

WebLogic RMI Performance and Scalability

WebLogic RMI performance is enhanced by its integration into the WebLogic Server framework, which provides the underlying support for communications, management of threads and sockets, efficient garbage collection, and server-related support.

WebLogic RMI scales dramatically better than the reference implementation, and provides outstanding performance and scalability. Even relatively small, single-processor, PC-class servers can support well over a thousand simultaneous RMI clients, depending on the total workload of the server and the complexity of the method calls.

WebLogic RMI, while offering the flexibility and universality of the JavaSoft standard, provides the power and performance necessary for a production environment. Because WebLogic RMI depends upon and takes advantage of the sophisticated infrastructure of WebLogic Server, it is fast and scalable, with many additional features that support real-world use of RMI for building complex, distributed systems.

2 Programming WebLogic RMI

The following sections describe the WebLogic RMI features used to program RMI for use with WebLogic Server.

- WebLogic RMI Compiler
- Proxies in WebLogic RMI
- WebLogic RMI Registry
- Implementation Features

WebLogic RMI Compiler

Run the WebLogic RMI compiler (rmic) by executing the `weblogic.rmic` class on the Java class file you want to execute via RMI. The compiler produces a stub and skeleton.

The stub class is the serializable class that is passed to the client. The skeleton is a server-side class that processes requests from the stub on the client. The implementation for the class is bound to a name in the RMI registry in the WebLogic Server.

A client acquires the stub for the class by looking up the class in the registry. WebLogic Server serializes the stub and sends it to the client. The client calls methods on the stub just as if it were a local class and the stub serializes the requests and sends them to the skeleton on the WebLogic Server. The skeleton deserializes client requests and executes them against the implementation classes, serializing results and sending them back to the stub on the client.

With the WebLogic RMI compiler, you can specify that more platform-specific compilers should be used. By default, `weblogic.rmic` invokes the compiler from the JDK distribution, `javac`, but you may call any compiler with the `-compiler` option to `weblogic.rmic`, as shown in this example which calls the Symantec Java compiler:

```
 $ java weblogic.rmic -compiler \Cafe\bin\sj.exe
```

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 command above. All other options supported by the compiler can be used, and are passed directly to the Java compiler. The following options for `java weblogic.rmic` are available for you to use, and should be entered in one line, after `java weblogic.rmic` and before the name of the remote class.

Table 1: WebLogic RMI Compiler Options

-help	Prints a description of the options
-version	Prints version information
-d <dir>	Target (top-level) directory for complation
-verbosemethods	Instruments proxies to print debug info for std err
-descriptor	Associates or creates a descriptor for each remote class
-nomanglednames	Does not mangle the names of stubs and skeletons
-idl	Generates IDLs for remote interfaces
-idlOverwrite	Overwrites existing IDL files
-idlVerbose	Displays verbose information for IDL information
-idlStrict	Generate IDL according to OMG standard

Table 1: WebLogic RMI Compiler Options

-idlNoFactories	Do not generate factory methods for valuetypes
-idlDirectory <idlDirectory>	Specifies the directory where IDL files will be created (Default = current directory)
-clusterable	Marks the service as clusterable (can be hosted by multiple servers in a WebLogic 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 failover (Default = weblogic.cluster.loadAlgorithm). Must be one of the following: round-robin, random, or weight-based.
-callRouter <callRouterClass>	Only for use in conjunction with -clusterable. Specifies the class to be used for routing method calls. This class must implement weblogic.rmi.extensions.CallRouter. If specified, an instance of the class will be called before each method call and be given the opportunity to choose a server to route to based on the method parameters. It either returns a server name or null--indicating that the current load algorithm should be used.
-stickToFirstServer	Only for use in conjunction with -clusterable. Enables 'sticky' load balancing. The server chosen for servicing the first request will be used for all subsequent requests.
-methodsAreIdempotent	Only for use in conjuction with -clusterable. Indicates that the methods on this class are idempotent. This allows the stub to attempt recovery form any communication failure, even if it can not ensure that failure occurred before the remote methode was invoked. By default (if this option is not used) the stub will only retry on failures that are guaranteed to have occured before the remote method was invoked.
-replicaListRefreshInterval <seconds>	Only for use in conjunction with -clusterable. Specifies the minimum time to wait between attempts to refresh the replica list from the cluster (Default = 180 seconds).
-iiop	Generate IIOP stubs from servers
-iiopDirectory	Directory where IIOP proxy classes are written

Table 1: WebLogic RMI Compiler Options

-keepgenerated	Keeps the generated .java files
-commentary	Emits commentary
-compiler <compiler>	Specifies the java compiler (Default = javac)
-comilerclass <null>	Loads the compiler as a class instead of an executable
-g	Compiles debugging information into a class file
-O	Compiles with optimization enabled
-debug	Compiles with debugging enabled
-nowarn	Compiles without warnings
-verbose	Compiles with verbose output
-nowrite	Does not generate .class files
-deprecation	Warns of deprecated calls
-normi	Passes through to Symantec’s sj
-J <option>	Flags passed through to java runtime
-classpath <path>	Classpath to use during compilation

Additional WebLogic RMI Compiler Features

Other features of the WebLogic RMI compiler include:

- 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. For example, `java.io.Input/OutputStream` classes are not serializable and do not implement an interface that conforms to the specification for a remote interface.
- Code generation can be done on the descendants of a class.
- Code generation can be done from an abstract class.

Proxies in WebLogic RMI

A *proxy* is a class used by the clients of a remote object, in the case of RMI, a *skeleton* and a *stub*. The stub class is the instance that is invoked upon in the client's Java Virtual Machine (JVM); the stub marshals the invoked method name and its arguments, forwards these to the remote JVM, and -- after the remote invocation is completed and returns -- unmarshals the results on the client. 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.

In the JavaSoft RMI reference implementation, there is a one-to-one correspondence between the proxy classes and the remote objects. For example, running the JavaSoft RMI compiler against `example.hello.HelloImpl` -- which implements the `remote class example.hello.Hello` -- will produce two classes, `example.hello.HelloImpl_Skel` and `example.hello.HelloImpl_Stub`. If another class -- for example, `counter.example.CiaoImpl` also implements the same remote interface (`example.hello.Hello`), a virtually identical pair of proxy classes will be produced with the JavaSoft RMI compiler (`counter.example.CiaoImpl_Skel` and `counter.example.CiaoImpl_Stub`).

Using the WebLogic RMI Compiler with Proxies

The WebLogic RMI compiler works differently. 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. For example, `example.hello.HelloImpl` and `counter.example.CiaoImpl` are represented by a single stub and skeleton, 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 will cause 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 stubs are not final. References to colocated remote objects are references to the objects themselves, not to the stubs.

WebLogic RMI Registry

WebLogic Server hosts the RMI registry and provides server infrastructure for RMI clients. The overhead for RMI registry and server communications is minimal, since registry traffic is multiplexed over the same connection as JDBC and other kinds of traffic. Clients use a single socket for RMI; scaling for RMI clients is linear in the WebLogic Server environment.

The WebLogic RMI registry is created when WebLogic Server starts up, and calls to create new registries simply locate the existing registry. Objects that have been bound in the registry can be accessed with a variety of client protocols, including the standard **rmi:**//, as well as **http:**//, or **https:**//. In fact, all of the naming services use JNDI.

Implementation Features

In general, functional equivalents of all methods in the `java.rmi` package are provided in WebLogic RMI, except for those methods in the `RMIClassLoader` and the method `java.rmi.server.RemoteServer.getClientHost()`.

All other interfaces, exceptions, and classes are supported in WebLogic RMI. Here are notes on particular implementations that may be of interest:

`rmi.Naming` is implemented as a final class in WebLogic RMI with all public methods supported by JNDI, which is the preferred mechanism for naming objects in WebLogic RMI.

`rmi.RMISecurityManager` is implemented as a non-final class with all public methods in WebLogic RMI and, unlike the restrictive JavaSoft reference implementation, is entirely permissive. Security in WebLogic RMI is an integrated part of the larger WebLogic environment, for which there is support for SSL (Secure Socket Layer) and ACLs (Access Control Lists).

`rmi.registry.LocateRegistry` is implemented as a final class with all public methods, but a call to `LocateRegistry.createRegistry(int port)` will not create a colocated registry, but rather will attempt to connect to the server-side instance that implements JNDI, for which host and port are designated by attributes. In WebLogic RMI, a call to this method allows the client to find the JNDI tree on the WebLogic Server.

Note: You can use protocols other than the default (rmi) as well, and provide the scheme, host, and port as a URL, as shown here:

Note: `LocateRegistry.getRegistry(https://localhost:7002);`

Note: which will locate a WebLogic Server registry on the local host at port 7002, using a standard SSL protocol.

`rmi.server.LogStream` diverges from the JavaSoft reference implementation in that the `write(byte[])` method logs messages through the WebLogic SErver log file.

`rmi.server.RemoteObject` is implemented in WebLogic RMI to preserve the type equivalence of UnicastRemoteObject, but the functionality is provided by the WebLogic RMI base class `Stub`.

`rmi.server.RemoteServer` is implemented as the abstract superclass of `rmi.server.UnicastRemoteObject` and all public methods are supported in WebLogic RMI with the exception of `getClientHost()`.

`rmi.server.UnicastRemoteObject` is implemented as the base class for remote objects, and all the methods in this class are implemented in terms of the WebLogic RMI base class `Stub`. This allows the stub to override non-final Object methods and equate these to the implementation without making any requirements on the implementation.

In WebLogic RMI, all method parameters are pass-by-value, *unless* the invoking object resides in the same Java Virtual Machine (JVM) as the RMI object. In this scenerio, method parameters are pass-by-reference.

Note: WebLogic RMI does not support uploading classes from the client. In other words, any classes passed to a remote object must be available within the server's CLASSPATH.

The `setSecurityManager()` method is provided in WebLogic RMI for compilation compatibility only. No security is associated with it, since WebLogic RMI depends on the more general security model within WebLogic Server. If, however, you *do* set a SecurityManager, you can set only one. Before setting a SecurityManager, you should test to see if one has already been set; if you try to set another, your program will throw an exception. Here is an example:

```
if (System.getSecurityManager() == null)
  System.setSecurityManager(new RMISecurityManager());
```

The following classes are implemented but unused in WebLogic RMI:

- `rmi.dgc.Lease`
- `rmi.dgc.VMID`
- `rmi.server.ObjID`
- `rmi.server.Operation`
- `rmi.server.RMIClassLoader`
- `rmi.server.RMISocketFactory`
- `rmi.server.RemoteStub`
- `rmi.server.UID`

3 WebLogic RMI API

The following sections describe the WebLogic RMI API.

- Overview of the WebLogic RMI API
- Implementing with WebLogic RMI
- Using the WebLogic RMI Compiler for Clustered Services

Overview of the WebLogic RMI API

There are several packages shipped as part of WebLogic RMI. The public API includes the WebLogic implementation of the RMI base classes, the registry, and the server packages. It also includes the WebLogic RMI compiler and supporting classes that are not part of the public API.

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.

The basic building block for all Remote objects is the interface `weblogic.rmi.Remote`, which 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, from whence a client or server may look up the object and use it remotely.

As in the JavaSoft reference implementation of RMI, the `weblogic.rmi.Naming` class is an important one. It includes methods for binding, unbinding, and rebinding names to remote objects in the registry. It also includes a `lookup()` method to give a client access to a named remote object in the registry.

In addition, WebLogic JNDI provides naming and lookup services. WebLogic RMI supports naming and lookup in JNDI.

WebLogic RMI Exceptions are identical to and extend `java.rmi` exceptions so that existing interfaces and implementations do not have to changeexception handling.

Implementing with WebLogic RMI

There are two parts to using WebLogic RMI. First you create the interfaces and classes that you will invoke remotely. Then you add code to your client application that carries out the remote invocations. The following sections detail these implementation phases.

- Creating classes that can be invoked remotely
  - Step 1. Write a Remote interface
  - Step 2. Implement the Remote interface
  - Step 3. Compile the java class
  - Step 4. Compile the implementation class with RMI compiler
  - Step 5. Write a client that invokes on remote objects
  - Full code examples
- Setting up WebLogic Server for RMI
- WebLogic RMI for clustered services
  - Cluster-specific RMI compiler options
  - Non-replicated stubs

Creating classes that can be invoked remotely

You can write your own WebLogic RMI classes in just a few steps. Here is a simple example.

Step 1. Write a Remote interface

Every class that can be remotely invoked implements a remote interface. A remote interface must extend the interface `weblogic.rmi.Remote`, which contains no method signatures.

The interface that you write should include method signatures that will be implemented in every remote class that implements it. Interfaces in Java are a powerful concept, and allow great flexibility at both design time and runtime. If you need more information on how to write an interface, see the JavaSoft tutorial Creating Interfaces.

Your Remote interface should follow guidelines similar to those followed if you are using JavaSoft's RMI:

- It must be public. Otherwise a client will get an error when attempting to load a remote object that implements it.
- It must *extend* the interface `weblogic.rmi.Remote`.
- Unlike JavaSoft's RMI, it is not necessary that each method in the interface declares `weblogic.rmi.RemoteException` in its `throws` block. The Exceptions that your application throws can be specific to your application, and may extend RuntimeException. WebLogic's RMI subclasses java.rmi.RemoteException so if you already have existing RMI classes, you will not have to change your exception handling.

With JavaSoft's RMI, every class that implements a remote interface must have accompanying, precompiled stubs and skeletons. WebLogic RMI supports more flexible runtime code generation; WebLogic RMI supports stubs and skeletons that are type-correct but are otherwise independent of the class that implements the interface. If a class implements a single remote interface, the stub and skeleton 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 stub and skeleton that result from compilation will depend on the name mangling used by the compiler.

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. It has only one method signature.

```
package examples.rmi.multihello;

import weblogic.rmi.*;

public interface Hello extends weblogic.rmi.Remote {
  String sayHello() throws RemoteException;

}
```

Step 2. Implement the Remote interface

Now write the class that will 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. Currently, all the code generation that takes place in WebLogic RMI is dependent on this class file, but this will change in future releases.

With WebLogic's RMI, there is no need for your class to extend UnicastRemoteObject, which is required by JavaSoft's RMI. (If you extend UnicastRemoteObject, WebLogic RMI will not care, but it isn't necessary.) This allows you to retain a class hierarchy that makes sense for your application.

Your class may implement more than one Remote interface. Your class may also define methods that are not in the Remote interface, but you will not be able to invoke those methods remotely. You should also define at least a default constructor.

In this example, we implement a class that creates multiple HelloImpls and binds each to a unique name in the Registry. The method `sayHello()` greets the user and identifies the object which was remotely invoked.

```
package examples.rmi.multihello;

import weblogic.rmi.*;

public class HelloImpl implements Hello {

  private String name;

  public HelloImpl(String s) throws RemoteException {

    name = s;

  }

  public String sayHello() throws RemoteException {

    return "Hello! From " + name;

  }
```

Finally, write a main that creates an instance of the remote object and registers it in the WebLogic RMI registry, by binding it to a name (a URL that points to the implementation of the object). A client that wants to obtain a stub to use the object remotely will be able to look up the object by name.

Here is an example of a `main()` for the HelloImpl class. This registers the HelloImpl object under the name HelloRemoteWorld in a WebLogic Server registry.

```
public static void main(String[] argv) {
  // Not needed with WebLogic RMI
  // System.setSecurityManager(new RmiSecurityManager());
  // But if you include this line of code, you should make
  // it conditional, as shown here:
  // if (System.getSecurityManager() == null)
  //   System.setSecurityManager(new RmiSecurityManager());
  int i = 0;
  try {
    for (i = 0; i < 10; i++) {
      HelloImpl obj = new HelloImpl("MultiHelloServer" + i);
      Naming.rebind("//localhost/MultiHelloServer" + i, obj);
     System.out.println("MultiHelloServer" + i + " created.");
    }
    System.out.println("Created and registered " + i +
                       " MultiHelloImpls.");
  }
  catch (Exception e) {
    System.out.println("HelloImpl error: " + e.getMessage());
    e.printStackTrace();
  }
}
```

WebLogic RMI does not require that you set a SecurityManager in order to integrate security into your application. Security is handled by WebLogic Server support for SSL and ACLs. If you must, you may use your own security manager, but do not install it in the WebLogic Server.

Step 3. Compile the java class

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

Step 4. Compile the implementation class with RMI compiler

Run the WebLogic RMI compiler against the remote class to generate the stub and skeleton. A stub is the client-side proxy for a remote object that forwards each WebLogic RMI call to its matching serverside skeleton, which in turn forwards the call to the actual remote object implementation. To run the WebLogic RM compiler, use the command pattern:

```
$ java weblogic.rmic nameOfRemoteClass
```

where *nameOfRemoteClass* is the full package name of the class that implements your Remote interface. With the examples we have used previously, the command would be:

```
$ java weblogic.rmic examples.rmi.hello.HelloImpl
```

You should set the flag `-keepgenerated` when you run the WebLogic RMI compiler if you want to keep the generated stub and skeleton files. For a listing of the available WebLogic RMI compiler options, see Chapter , “WebLogic RMI Compiler.”.

Running the WebLogic RMI compiler creates two new classes, a stub and a skeleton. These appear as *nameOfInterface*_Stub.class and *nameOfInterface*_Skel.class. The four files created -- the remote interface, the class that implements it, and the stub and skeleton created by the WebLogic RMI compiler -- should be placed in the appropriate directory in the `CLASSPATH` of the WebLogic Server whose URL you used in the naming scheme of the object's `main()`.

Invoking methods on remote objects in your client code

Once you compile and install the remote class, the interface it implements, and its stub and skeleton on the WebLogic Server, you can add code to a WebLogic client application to invoke methods in the remote class.

In general, it takes just a single line of code: you need to get a reference to the remote object. Do this with the `Naming.lookup()` method. Here is a short WebLogic client application that uses an object created in a previous example.

```
package mypackage.myclient;

import weblogic.rmi.*;

import weblogic.common.*;


public class HelloWorld throws Exception {


  // Look up the remote object in the

  // WebLogic’s registry

  Hello hi = (Hello)Naming.lookup("HelloRemoteWorld");

  // Invoke a method remotely

  String message = hi.sayHello();

  System.out.println(message);

}
```

This example demonstrates using a Java application as the client.

Full Code Examples

Here is the full code for the Hello interface.

```
package examples.rmi.hello;

import weblogic.rmi.*;
```

```
public interface Hello extends weblogic.rmi.Remote {

  String sayHello() throws RemoteException;

}
```

Here is the full code for the HelloImpl class that implements it.

```
package examples.rmi.hello;

import weblogic.rmi.*;

public class HelloImpl
    // Don’t need this in WebLogic RMI:
    // extends UnicastRemoteObject
    implements Hello {

  public HelloImpl() throws RemoteException {
    super();
  }

  public String sayHello() throws RemoteException {
    return "Hello Remote World!!";
  }

  public static void main(String[] argv) {
    try {
      HelloImpl obj = new HelloImpl();
      Naming.bind("HelloRemoteWorld", obj);
    }
    catch (Exception e) {
      System.out.println("HelloImpl error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Using the WebLogic RMI Compiler for Clustered Services

The following sections identifies the cluster-specific RMI options used with WebLogic's implementation of RMI and describes the non-replicated stubs.

Cluster-specific RMI compiler options

The RMI compiler (RMIC) has several flags that relate to clusters. These argument tags are *not* case sensitive; inner caps are used in these descriptions for ease of reading.

`-clusterable`

Generates a stub that can failover and load balance. By default, the generated stub will be capable of failover and will load balance between replicas using a round-robin scheduling algorithm.

`-methodsAreIdempotent`

May only be used in conjunction with "`-clusterable`". Indicates to the stub that it can attempt retries after failover even if it might result in executing the same method multiple times. If this flag isn't present, methods for this stub are not considered idempotent. The exceptions that are handled by this are described in Exceptions used for failover.

`-loadAlgorithm` load algorithm name

May only be used in conjunction with "`-clusterable`". Specifies a service-specific algorithm that will be used by the stub to handle failover and load balancing. If this argument is unspecified, the default load balancing algorithm specified in the Administration Console. For example, to specify weight-based load balancing:

```
 $ java weblogic.rmic -clusterable -loadAlgorithm=weight-based
```

`-stickToFirstServer`

May only be used in conjunction with "`-clusterable`". Enables 'sticky' load balancing. The server chosen for servicing the first request will be used for all subsequent requests.

`-replicaListRefreshInterval` *seconds*

May only be used in conjunction with "`-clusterable`". Specifies the minimum time to wait between attempts to refresh the replica list from the Cluster. Default is 180 seconds (3 minutes).

`-callRouter` *callRouterClass*

May only be used in conjunction with "`-clusterable`". Specifies the class to be used for routing method calls. This class must implement `weblogic.rmi.extensions.CallRouter`. If specified, an instance of this class will be called before each method call and be given the opportunity to choose the server given the method parameters. It either returns a server name or `null` indicating that the current load algorithm should be used to pick the server.

Non-Replicated Stubs

You can also generate stubs that are *not* replicated; these are known as "pinned" services, because after they are registered they will be 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, since they are bound into the replicated cluster-wide JNDI tree; but if the individual server that hosts them fails, the client cannot failover to another server.

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 will only be reachable as long as the Server that carried out the bind is reachable.