BEA WebLogic Server

Programming WebLogic JNDI

BEA WebLogic Server Version 6.1
Document Date: November 17, 2003

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic JNDI

Document Edition	Document Date	Software Version
N/A	November 17, 2003	BEA WebLogic Server Version 6.1

About This Document

This document explains how to program with the JNDI feature provided with the BEA WebLogic Server™ product.

This document is organized as follows:

- Chapter 1, “Introduction to WebLogic JNDI,” provides an overview of the JNDI capabilities in WebLogic Server.
- Chapter 2, “Programming with WebLogic JNDI,” explains how to program with the WebLogic JNDI functionality in Java client applications.

Audience

This document is intended for programmers who are developing applications with WebLogic Server and want to use the JNDI feature.

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.

Contact Us!

Your feedback on 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.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and filenames 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.

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

CHAPTER

1 Introduction to WebLogic JNDI

This section presents an overview of the JNDI implementation in WebLogic Server.

Overview of JNDI in WebLogic Server

In an enterprise, naming services provide a means for your application to locate objects on the network. A naming service associates names with objects and finds objects based on their given names. (The RMI registry is a good example of a naming service.)

The Java Naming and Directory Interface (JNDI) is an application programming interface (API) that provides naming services to Java applications. JNDI is an integral component of Sun Microsystems Inc.'s Java 2 Enterprise Edition (J2EE) technology.

JNDI is defined to be independent of any specific naming or directory service implementation. It supports the use of a single method for accessing various new and existing services.

The JNDI support provided by WebLogic Server is based on the standard JNDI API classes defined by Sun Microsystems, Inc. This support allows any service-provider implementation to be plugged into the JNDI framework using the standard service provider interface (SPI) conventions. In addition, JNDI allows Java applications in WebLogic Server to access external directory services such as LDAP in a standardized fashion, by plugging in the appropriate service provider. WebLogic Server supports version 1.2.1 of the JNDI API.

The WebLogic Server implementation of JNDI supplies methods that:

- Give clients access to the WebLogic name services
- Make objects available in the WebLogic namespace
- Retrieve objects from the WebLogic namespace

Each WebLogic Server cluster is supported by a replicated cluster-wide JNDI tree that provides access to both replicated and pinned RMI and EJB objects. While the JNDI tree representing the cluster appears to the client as a single global tree, the tree containing the cluster-wide services is actually replicated across each WebLogic Server in the cluster. For more information, see Using WebLogic JNDI in a Clustered Environment.

The integrated naming service provided by WebLogic Server JNDI may be used by many other WebLogic services. For example, WebLogic RMI can bind and access remote objects by both standard RMI methods and JNDI methods.

In addition to the standard Sun Microsystems Inc. interfaces for JNDI, WebLogic Server provides its own implementation, `weblogic.jndi.WLInitialContextFactory`, that uses the standard JNDI interfaces.

In your application code, you need not instantiate this class directly. Instead, you can use the standard `javax.naming.InitialContext` class and set the appropriate hash table properties, as documented in the section Setting Up JNDI Environment Properties for the InitialContext. All interaction is done through the `javax.naming.Context` interface, as described in the JNDI Javadoc.

For instructions on using the WebLogic JNDI API for client connections, see Programming with WebLogic JNDI.

CHAPTER

2 Programming with WebLogic JNDI

The following sections describe programming with WebLogic JNDI including:

Using WebLogic JNDI from a Java Client

The WebLogic Server JNDI Service Provider Interface (SPI) provides an `InitialContext` implementation that allows remote Java clients to connect to WebLogic Server. The client can specify standard JNDI environment properties that identify a particular WebLogic Server deployment and related connection properties for logging in to WebLogic Server.

To participate in a session with WebLogic Server, a Java client must be able to get an object reference for a remote object and invoke operations on the object. To accomplish this, the client application code must perform the following procedure:

1. Set up JNDI environment properties for the `InitialContext`.
2. Establish an `InitialContext` with WebLogic Server.
3. Use the Context to look up a named object in the WebLogic Server namespace.
4. Use the named object to get a reference for the remote object and invoke operations on the remote object.
5. Complete the session.

The following sections discuss JNDI client operations for connecting to a specific WebLogic Server. For information about using JNDI in a cluster of WebLogic Servers, see "Using WebLogic JNDI from a Client in a Clustered Environment" on page 2-16.

Before you can use JNDI to access an object in a WebLogic Server environment, you must load the object into the WebLogic Server JNDI tree. For instructions on loading objects in the JNDI tree, see Managing JNDI.

Setting Up JNDI Environment Properties for the InitialContext

The first task that must be performed by any Java client application is to create environment properties. The `InitialContext` factory uses various properties to customize the `InitialContext` for a specific environment. You can set these properties either by using a hash table or the `set()` method of a WebLogic Environment object. These properties, which are specified name-to-value pairs, determine how the `WLInitialContextFactory` creates the Context.

The following properties are used to customize the `InitialContext`:

- `Context.PROVIDER_URL`— specifies the URL of the WebLogic Server that provides the name service. The default is `t3://localhost:7001`.

- `Context.SECURITY_PRINCIPAL`—specifies the identity of the User (that is, a User defined in a WebLogic Server security realm) for authentication purposes. The property defaults to the `guest` User unless the thread has already been associated with a WebLogic Server User. For more information, see Managing Security at http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html.
- `Context.SECURITY_CREDENTIALS`—specifies either the password for the User defined in the `Context.SECURITY_PRINCIPAL` property or an object that implements the `weblogic.security.acl.UserInfo` interface with the `Context.SECURITY_CREDENTIALS` property defined. If you pass a UserInfo object in this property, the `Context.PROVIDER_URL` property is ignored. The property defaults to the `guest` User unless the thread has already been associated with a User. For more information, see Managing Security at http://e-docs.bea.com/wls/docs61/adminguide/cnfgsec.html.

You can use the same properties on either a client or a server. If you define the properties on a server-side object, a local Context is used. If you define the properties on a client or another WebLogic Server, the Context delegates to a remote Context running on the WebLogic Server specified by the `Context.PROVIDER_URL` property.

Listing 2-1 shows how to obtain a Context using the properties `Context.INITIAL_CONTEXT_FACTORY` and `Context.PROVIDER_URL`.

Listing 2-1 Obtaining a Context

```
Context ctx = null;
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL,
       "t3://localhost:7001");

try {
  ctx = new InitialContext(ht);
  // Use the context in your program
}
catch (NamingException e) {
  // a failure occurred
}
finally {
  try {ctx.close();}
  catch (Exception e) {
    // a failure occurred
  }
```

```
}
```

Additional WebLogic-specific properties are also available for configuring security parameters and controlling how objects are bound into the cluster-wide JNDI tree. Note that bindings may or may not be replicated across the JNDI tree of each server within the cluster. Properties such as these are identified by constants in the `weblogic.jndi.WLContext` class. For more information about JNDI-related clustering issues, see "Using WebLogic JNDI from a Client in a Clustered Environment" on page 2-16.

Creating a Context Using a Hash Table

You can create a Context with a hash table in which you have specified the properties described in "Setting Up JNDI Environment Properties for the InitialContext" on page 2-2.

To do so, pass the hash table to the constructor for `InitialContext`. The property `java.naming.factory.initial` is used to specify how the `InitialContext` is created. To use WebLogic JNDI, you must always set the `java.naming.factory.initial` property to `weblogic.jndi.WLInitialContextFactory.` This setting identifies the factory that actually creates the Context.

Creating a Context Using a WebLogic Environment Object

You can also create a Context by using a WebLogic environment object implemented by the `weblogic.jndi.environment` interface. Although the environment object is WebLogic-specific, it offers the following advantages:

- A set of defaults which reduces the amount of code you need to write.
- Convenience `set()` methods that provide compile-type type-safety. The type-safety `set()` methods can save you time both writing and debugging code.

The WebLogic Environment object provides the following defaults:

- If you do not specify an `InitialContext` factory, `WLInitialContextFactory` is used.
- If you do not specify a user and password in the `Context.SECURITY_PRINCIPAL` and `Context.CREDENTIALS` properties, the `guest` User and password are used unless the thread has already been associated with a user.
- If you do not specify a `Context.PROVIDER_URL` property, `t3://localhost:7001` is used.

If you want to create `InitialContext` with these defaults, write the following code:

```
Environment env = new Environment();
Context ctx = env.getInitialContext();
```

If you want to set only a WebLogic Server to a Distributed Name Service (DNS) name for client cluster access, write the following code:

```
Environment env = new Environment();
env.setProviderURL("t3://myweblogiccluster.com:7001");
Context ctx = env.getInitialContext();
```

Note: Every time you create a new JNDI environment object, you are creating a new security scope. This security scope ends with a `context.close()` method.

The `environment.getInitialContext()` method does not work correctly with the IIOP protocol.

Listing 2-2 illustrates using a JNDI Environment object to create a security context.

Listing 2-2 Creating a Security Context with a JNDI Environment Object

```
weblogic.jndi.Environment environment = new
weblogic.jndi.Environment();
environment.setInitialContextFactory(
  weblogic.jndi.Environment.DEFAULT_INITIAL_CONTEXT_FACTORY);
environment.setProviderURL(“t3://bross:4441”);
environment.setSecurityPrincipal(“guest”);
environment.setSecurityCrendentials(“guest”);
Hashtable props = environment.getProperties();
InitialContext ctx = new InitialContext(props);
```

Creating a Context from a Server-Side Object

You may also need to create a Context from an object (an Enterprise JavaBean (EJB) or Remote Method Invocation (RMI) object) that is instantiated in the Java Virtual Machine (JVM) of WebLogic Server. When using a server-side object, you do not need to specify the `Context.PROVIDER_URL` property. Usernames and passwords are required only if you want to sign in as a specific User. Server-side contexts run in the context of WebLogic Server.

To create a Context from within a server-side object, you first must create a new `InitialContext`, as follows:

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

You do not need to specify a factory or a provider URL. By default, the context is created as a Context and is connected to the local naming service unless the object you are accessing is on a different WebLogic domain and the user credentials are different in the other WebLogic domain. If the user credentials are different on the other WebLogic domain, you must create a JNDI Context with a username and password. For more information, see "Using WebLogic JNDI Between WebLogic Domains" on page 2-18.

JNDI Contexts and Threads

When you create a JNDI Context with a username and password, you associate a user with a thread. When the Context is created, the user is pushed onto the context stack associated with the thread. Before starting a new Context on the thread, you must close the first Context so that the first user is no longer associated with the thread. Otherwise, users are pushed down in the stack each time a new context created. This is *not* an efficient use of resources and may result in the incorrect user being returned by `ctx.lookup()` calls. This scenario is illustrated by the following steps:

1. Create a Context (with username and credential) called `ctx1` for `user1`. In the process of creating the context, `user1` is associated with the thread and pushed onto the stack associated with the thread. The current user is now `user1`.

2. Create a second Context (with username and credential) called `ctx2` for `user2`. At this point, the thread has a stack of users associated with it. `User2` is at the top of the stack and `user1` is below it in the stack, so `user2` is used is the current user.

3. If you do a `ctx1.lookup("abc")` call, `user2` is used as the identity rather than `user1`, because `user2` is at the top of the stack. To get the expected result, which is to have `ctx1.lookup("abc")` call performed as `user1`, you need to do a `ctx2.close()` call. The `ctx2.close()` call removes `user2` from the stack associated with the thread and so that a `ctx1.lookup("abc")` call now uses user1 as expected.

   Note: There are two situations where a `close()` call does not remove the current user from the stack and this can cause JNDI context problems. For information on how to avoid JNDI context problems, see "How to Avoid Potential JNDI Context Problems" on page 2-7.

How to Avoid Potential JNDI Context Problems

While issuing a `close()` call usually behaves as described in "JNDI Contexts and Threads" on page 2-6. However, there are two exceptions to expected behavior:

- First Login
- Last Used

First Login

When using protocols other than IIOP, the first user is "sticky" in the sense that it becomes the default user when no other user is present. This scenario is described in the following steps:

1. Create a Context (with username and credential) called `ctx1` for user1. In the process of creating the context, `user1` is associated with the thread and stored in the stack, that is, the current identity is set to `user1`.

2. Do a `ctx1.close()` call.

3. Do a `ctx1.lookup()`call. *The current identity is user1*.

4. Create a Context (with username and credential) called `ctx2` for user2. In the process of creating the context, `user2` is associated with the thread and stored in the stack, that is, the current identity is set to `user2`.

5. Do a `ctx2.close()` call.
6. Do a `ctx2.lookup()`call. *The current identity is user1*.

Since `ctx1` *was the first user, the current identity stays set to* `user1` after step 4. Note that not only is `user1` the current user on this thread, it is the current user on all threads that do not have another identity defined. Thus, `user1` becomes the default user when no other user identity is present. This is not good practice as any subsequent logins that do not have a username and credential will be granted the identify of `user1` by default.

To work around this problem, implement one of the following options:

- ***Option 1*:** If the client has control of `main()`, implement the wrapper code shown in Listing 2-3 in the client code.

Listing 2-3 JNDI Context and Threads Wrapper Code

```
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import weblogic.security.Security;

public class client
{
  public static void main(String[] args)
  {
  Security.runAs(new Subject(),
    new PrivilegedAction() {
     public Object run() {
      //
      //If implementing in client code, main() goes here.
      //
     return null;
     }
   });
  }
}
```

- ***Option 2:*** If the client does not have control of `main()`, implement the wrapper code shown in Listing 2-3 on each thread's `run()` method.
- ***Option 3:*** Create a Context that logs in as a non-privileged user (a non-privileged user is a user that is not a member of any group). Be sure this is the first logon on the client. Immediately execute `ctx.close()` call to remove

the non-privileged user from the thread's user stack. Because the non-privileged user is the first user to logon, it becomes the default user. Subsequently, any thread that has an empty user stack will have the identity of non-privileged user.

Note: If you choose to use Option 3, be advised of how non-privileged users relate to the `users` and `everyone` groups, which are configured by default in the security realm in this release of WebLogic Server. The `users` and `everyone` groups are convenience groups that allow you to apply global roles and security policies. By default, all WebLogic Server users, including non-privileged users, are members of the `everyone` group, but non-privileged users are not members of the `users` group.

Last Used

When using IIOP, an exception to expected behavior arises when there is one Context on the stack and that Context is removed by a `close()`. The identity of the last context removed from the stack determines the current identity of the user. This scenario is described in the following steps:

1. Create a Context (with username and credential) called `ctx1` for user1. In the process of creating the context, `user1` is associated with the thread and stored in the stack, that is, the current identity is set to `user1`.
2. Do a `ctx1.close()` call.
3. Do a `ctx1.lookup()`call. *The current identity is user1*.
4. Create a Context (with username and credential) called `ctx2` for user2. In the process of creating the context, `user2` is associated with the thread and stored in the stack, that is, the current identity is set to `user2`.
5. Do a `ctx2.close()` call.
6. Do a `ctx2.lookup()`call. *The current identity is user2*.

Using the Context to Look Up a Named Object

Every Java client application must obtain an initial object that provides services for the application. An object cannot not be looked up, however, unless it is loaded in the WebLogic Server namespace. For more information, see Managing JNDI in the Administration Guide.

The `lookup()` method on the Context is used to obtain named objects. The argument passed to the `lookup()` method is a string that contains the name of the desired object. Listing 2-4 shows how to retrieve an EJB named `ServiceBean`.

Listing 2-4 Looking Up a Named Object

```
try {
  ServiceBean bean = (ServiceBean)ctx.lookup("ejb.serviceBean");
}
catch (NameNotFoundException e) {
  // binding does not exist
}
catch (NamingException e) {
  // a failure occurred
}
```

Using a Named Object to Get an Object Reference

EJB client applications get object references to EJB remote objects from EJB Homes. RMI client applications get object references to other RMI objects from an initial named object. Both initial named remote objects are known to WebLogic Server as factories. A factory is any object that can return a reference to another object that is in the WebLogic namespace.

The client application invokes a method on a factory to obtain a reference to a remote object of a specific class. The client application then invokes methods on the remote object, passing any required arguments.

Listing 2-5 contains a code fragment that obtains a remote object and then invokes a method on it.

Listing 2-5 Using a Named Object to Get an Object Reference

```
ServiceBean bean = ServiceBean.Home.create("ejb.ServiceBean")
Servicebean.additem(66);
```

Closing the Context

After clients finish working with a Context, BEA Systems recommends that the client close the Context in order to release resources and avoid memory leaks. BEA recommends that you use a `finally{}` block and wrap the `close()` method in a `try{}` block. If you attempt to close a context that was never instantiated because of an error, the Java client application throws an exception.

In Listing 2-6, the client closes the context, releasing the resource being used.

Listing 2-6 Closing the Context

```
try {
       ctx.close();
} catch (Exception e) {
//a failure occurred
}
```

Using WebLogic JNDI in a Clustered Environment

The intent of WebLogic JNDI is to provide a naming service for J2EE services, specifically EJB, RMI, and Java Messaging Service (JMS). Therefore, it is important to understand the implications of binding an object to the JNDI tree in a clustered environment.

The following sections discuss how WebLogic JNDI is implemented in a clustered environment and offer some approaches you can take to make your own objects available to JNDI clients.

Clustering J2EE Services

WebLogic RMI is the enabling technology that allows clients in one JVM to access EJBs and JMS services from a client in another JVM. RMI stubs marshal incoming calls from the client to the RMI object. To make J2EE 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.

When a client connects to a cluster, it is actually connecting to one of the WebLogic Servers already in the cluster. Because the JNDI tree for this WebLogic Server contains the RMI stubs for all services offered by the other WebLogic Servers in the cluster in addition to its own services, the cluster appears to the client as one WebLogic Server hosting all of the cluster-wide services. When a new WebLogic Server joins a cluster, each WebLogic Server already in the cluster is responsible for sharing information about its own services to the new WebLogic Server. With the information collected from all the other servers in the cluster, the new server may create its own copy of the cluster-wide JNDI tree.

RMI stubs significantly affect how WebLogic JNDI is implemented in a clustered environment:

- RMI stubs are relatively small. This allows WebLogic JNDI to replicate stubs across all WebLogic Servers in a cluster with little overhead in terms of server-to-server cross-talk.
- RMI stubs serve as the mechanism for replication across a cluster. An instance of a RMI object is deployed to a single WebLogic Server, however, the stub is replicated across the cluster.

Making Custom Objects Available to a WebLogic Server Cluster

When you bind a custom object (a non-RMI object) into a JNDI tree in a WebLogic Server cluster, the object is replicated across all the servers in the cluster. However, if the host server goes down, the custom object is removed from the cluster's JNDI tree. Custom objects are not replicated unless the custom object is bound again. You need to unbind and rebind a custom object every time you want to propagate changes made to the custom object. Therefore, WebLogic JNDI should not be used as a distributed object cache. You can use a third-party solution with WebLogic Server to provide distributed caches.

Suppose the custom object needs to be accessed only by EJBs that are deployed on only one WebLogic Server. Obviously it is unnecessary to replicate this custom object throughout all the WebLogic Servers in the cluster. In fact, you should avoid replicating the custom object in order to avoid any performance degradation due to unnecessary server-to-server communication. To create a binding that is not replicated

across WebLogic Servers in a cluster, you must specify the `REPLICATE_BINDINGS` property when creating the context that binds the custom object to the namespace. Listing 2-7 illustrates the use of the `REPLICATE_BINDINGS` property.

Listing 2-7 Using the REPLICATE_BINDINGS Property

```
Hashtable ht = new Hashtable();
//turn off binding replication
ht.put(WLContext.REPLICATE_BINDINGS, "false");
try {
  Context ctx = new InitialContext(ht);
  //bind the object
  ctx.bind("my_object", MyObect);
} catch (NamingException ne) {
  //failure occured
}
```

When you are using this technique and you need to use the custom object, you must explicitly obtain an `InitialContext` for the WebLogic Server. If you connect to any other WebLogic Server in the cluster, the binding does not appear in the JNDI tree.

If you need a custom object accessible from any WebLogic Server in the cluster, deploy the custom object on each WebLogic Server in the cluster without replicating the JNDI bindings.

When using WebLogic JNDI to replicate bindings, the bound object will be handled as if it is owned by the host WebLogic Server. If the host WebLogic Server fails, the custom object is removed from all the JNDI trees of all WebLogic Servers in the cluster. This behavior can have an adverse effect on the availability of the custom object.

Note: You can not use a `/` or `-` character in a JNDI `Context.bind(String Name)`. If the Binding name string contains a `/` or `-` character, a `javax.naming.NameNotFoundException` is raised.

Data Caching Design Pattern

A common task in Web applications is to cache data used by multiple objects for a period of time to avoid the overhead associated with data computation or connecting to another service.

Suppose you have designed a custom data caching object that performs well on a single WebLogic Server and you would like to use this same object within a WebLogic cluster. If you bind the data caching object in the JNDI tree of one of the WebLogic Servers, WebLogic JNDI will, by default, copy the object to each of the other WebLogic Servers in the cluster. It is important to note that since this is not an RMI object, what you are binding into the JNDI tree (and copying to the other WebLogic Servers) is the object itself, not a stub that refers to a single instance of the object hosted on one of the WebLogic Servers. Do not assume from the fact that WebLogic Server copies a custom object between servers that custom objects can be used as a distributed cache. Remember the custom object is removed from the cluster if the WebLogic Server to which it was bound fails and changes to the customer object are not propagated through the cluster unless the object is unbound and rebound to the JNDI tree.

For the sake of performance and availability, it is often desirable to avoid using WebLogic JNDI's binding replication to copy large custom objects with high availability requirements to all of the WebLogic Servers in a cluster. As an alternative, you can deploy a separate instance of the custom object on each of the WebLogic Servers in the cluster. When binding the object to each WebLogic Server's JNDI tree, you should make sure to turn off binding replication as described in "Making Custom Objects Available to a WebLogic Server Cluster" on page 2-13. In this design pattern, each WebLogic Server has a copy of the custom object but you will avoid copying large amounts of data from server to server.

Regardless of which approach you use, each instance of the object should maintain its own logic for when it needs to refresh its cache independently of the other data cache objects in the cluster. For example, suppose a client accesses the data cache on one WebLogic Server. It is the first time the caching object has been accessed, so it computes or obtains the information and saves a copy of the information for future requests. Now suppose another client connects to the cluster to perform the same task as the first client only this time the connection is made to a different WebLogic Server in the cluster. If this the first time this particular data caching object has been accessed, it will need to compute the information regardless of whether other data caching

objects in the cluster already have the information cached. Of course, for any future requests, this instance of the data cache object will be able to refer to the information it has saved.

Exactly-Once-Per-Cluster Design Pattern

In some cases, it is desirable to have a service that appears only once in the cluster. This is accomplished by deploying the service on one machine only. For RMI objects, you can use the default behavior of WebLogic JNDI to replicate the binding (the RMI stub) and the single instance of your object will be accessible from all WebLogic Servers in the cluster. This is referred to as a pinned service. For non-RMI objects, make sure that you use the `REPLICATE_BINDINGS` property when binding the object to the namespace. In this case, you will need to explicitly connect to the host WebLogic Server to access the object. Alternatively, you can create an RMI object that is deployed on the same host WebLogic Server that can act as a proxy for your non-RMI object. The stub for the proxy can be replicated (using the default WebLogic JNDI behavior) allowing clients connected to any WebLogic Server in the cluster to access the non-RMI object via the RMI proxy.

This design pattern for an exactly-once-per-cluster service presents an additional challenge for services with high availability requirements. Since the failover feature of WebLogic Clusters relies on having multiple deployments of each clustered service, failover for an exactly-once-per-cluster service will not be available. For services that require high availability, it is suggested that you implement a hardware, High-Availability (HA) framework for the host WebLogic Server. The framework allows WebLogic Server to be restarted in the event of a failure with a minimal amount of disruption to availability of the service.

Using WebLogic JNDI from a Client in a Clustered Environment

The JNDI binding for an object can appear in the JNDI tree for one WebLogic Server in the cluster, or it can be replicated to all the WebLogic Servers in the cluster. If the object of interest is bound in only one WebLogic Server, you must explicitly connect

to the host WebLogic Server by setting the `Context.PROVIDER_URL` property to the host WebLogic Server's URL when creating the Initial Context, as described in "Using WebLogic JNDI from a Java Client" on page 2-1.

In most cases, however, the object of interest is either a clustered service or a pinned service. As a result, a stub for the service is displayed in the JNDI tree for each WebLogic Server in the cluster. In this case, the client does not need to name a specific WebLogic Server to provide its naming service. In fact, it is best for the client to simply request that a WebLogic Cluster provide a naming service, in which case the context factory in WebLogic Server can choose whichever WebLogic Server in the cluster seems most appropriate for the client. Currently, a naming service provider is chosen within WebLogic using the DNS round-robin feature.

The context that is returned to a client of clustered services is, in general, implemented as a failover stub that can transparently change the naming service provider if a failure (such as a communication failure) with the selected WebLogic Server occurs.

Listing 2-8 shows how a client uses the cluster's naming service.

Listing 2-8 Using the Naming Service in a WebLogic Cluster

```
Hashtable ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL, "t3://acmeCluster:7001");
try {
  Context ctx = new InitialContext(ht);
  // Do the client's work
}
catch (NamingException ne) {
  // A failure occurred
}
finally {
  try {ctx.close();}
  catch (Exception e) {
    // a failure occurred
  }
}
```

The *hostname* specified as part of the provider URL is the DNS name for the cluster that can be defined by the `ClusterAddress` setting in a Cluster stanza of the `config.xml` file. `ClusterAddress` maps to the list of hosts providing naming service in this cluster. For more information, see Configuring WebLogic Servers and Clusters.

In Listing 2-8, the cluster name `acmeCluster` is used to connect to any of the WebLogic Servers in the cluster. The resulting Context is replicated so that it can fail over transparently to any WebLogic Server in the cluster.

An alternative method of specifying the initial point of contact with the WebLogic Cluster is to supply a comma-delimited list of DNS Server names or IP addresses, as shown in the following sample code:

```
ht.put(Context.PROVIDER_URL, "t3://acme1,acme2,acme3:7001");
```

Notice that all the WebLogic Servers must listen on the same port, as specified at the end of the URL.

Using WebLogic JNDI Between WebLogic Domains

Since the context stack associated with the current thread contains the user credentials, if a cached EJB Object (that is, the Handle is cached) is being referenced, BEA Systems recommends that you get an initial context before accessing methods on that EJB. For example, you can cache the EJB's Handle and subsequently used it without incurring the overhead of the `lookup()` and `create()` methods. However, if the current thread's context is associated with a different user than the one that was used initially to lookup the EJB, and if the user credentials (username/password) are not the same in both WebLogic domains, a security exception is thrown if the EJB’s cached handle is used. Therefore, in this scenario, the client is required to obtain a new context before using the EJB's cached Handle.