BEA WebLogic Enterprise

Using the WLE SPI Implementations for JNDI

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Using the WLE SPI Implementations for JNDI

Document Edition	Date	Software Version
5.0	December 1999	BEA WebLogic Enterprise 5.0

Contents

About This Document

What You Need to Know v
e-docs Web Site v
How to Print the Document vi
Related Information vi
Contact Us! vii
Documentation Conventions viii

1. Using the WLE SPI Implementations for JNDI

Overview of JNDI in WLE 1-2
The JNDI API and SPI 1-4
The Naming Interface — javax.naming 1-4
The Directory Interface — javax.naming.directory 1-4
The Service Provider Interface — javax.naming.spi 1-5
Additional WLE SPI Implementations 1-5
WLE JNDI Packaging 1-6
Location of the WLE JNDI Javadoc 1-6
Unified Naming and Directory Services 1-7
Using the Remote Naming Service for Client Connections and SSL Support .. 1-8
Step 1: Set Up JNDI Environment Properties for the Initial Context 1-8
WLEContext.INITIAL_CONTEXT_FACTORY Property 1-9
WLEContext.PROVIDER_URL Property 1-9
WLEContext.SECURITY_AUTHENTICATION Property 1-10
WLE Keys Required for BEA TUXEDO Style Authentication 1-11
Step 2: Establish an InitialContext with the WLE Domain 1-12
Step 3: Use the Context to Look Up a Named Server Object 1-13

Step 4: Use the Named Server Object to Get a Reference for the Desired Remote Object, and Invoke Operations on the Remote Object 1-13
Step 5: Complete the Session 1-14
Providing Remote Client Access to the UserTransaction Interface 1-14
Using the Application Naming Service to Access Local Objects 1-15
Using the Application Naming Service to Access Global Objects 1-16
Overview of Features 1-16
Accessing the Factories Subcontext 1-17
Binding Objects into the Factories Subcontext 1-17
Unbinding Objects from the Factories Subcontext 1-19
J2EE Requirements 1-19
Cross-Domain Support 1-20
The J2EE Naming Context 1-20
Overview of Requirements 1-21
Accessing Environment Entries 1-21
Using EJB References 1-22
Obtaining Resource Factory References 1-23
Obtaining a UserTransaction Object 1-24

Index

About This Document

This document explains how to use the BEA WebLogic Enterprise (WLE) Service Provider Interface (SPI) implementations for Java Naming and Directory Interface (JNDI) in WLE applications. The information in this document supplements the Sun Microsystems, Inc. JNDI 1.1 Specification for the SPI. The basic JNDI framework implementation in WLE is based on version 1.1.2 of the Sun Microsystems, Inc. standard extension classes.

What You Need to Know

This document is intended mainly for programmers and system administrators who need to create and maintain transactional, scalable WLE applications.

e-docs Web Site

The BEA WLE product documentation is available on the BEA corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the "e-docs" Product Documentation page at http://e-docs.beasys.com.

How to Print the Document

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

A PDF version of this document is available on the WLE 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 WLE documentation Home page, click the PDF Files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

For more information about topics covering Java 2 Enterprise Edition (J2EE), distributed object computing, transaction processing, and Java programming, see the WLE Bibliography in the WebLogic Enterprise online documentation. In addition, the following online and printed documents may be of interest to developers using JNDI.

Java 2 Platform Enterprise Edition home page, Sun Microsystems, Inc., at http://java.sun.com/j2ee/

JNDI Executive Summary, Sun Microsystems, Inc., at ftp://ftp.javasoft.com/docs/jndi/jndispi.pdf

JNDI: Java Naming and Directory Interface, Version 1.1, Sun Microsystems, Inc., at ftp://ftp.javasoft.com/docs/jndi/jndi.pdf

JNDI SPI: Java Naming and Directory Service Provider Interface, Version 1.1, Sun Microsystems, Inc., at ftp://ftp.javasoft.com/docs/jndi/jndispi.pdf

JNDI 1.1 API and SPI Specification, Sun Microsystems, Inc., in Javadoc format at http://java.sun.com/products/jndi/javadoc/index.html

Enterprise JavaBeans, Version 1.1, Sun Microsystems, Inc., at http://java.sun.com/products/ejb/docs.html

Java Transaction API (JTA) Version 1.0.1, Sun Microsystems, Inc., at http://java.sun.com/products/jta/

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA WLE 5.0 release.

If you have any questions about this version of BEA WLE, or if you have problems installing and running BEA WLE, contact BEA Customer Support through BEA WebSupport at **www.beasys.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	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

Convention	Item
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Using the WLE SPI Implementations for JNDI

This topic includes the following sections:

- Overview of JNDI in WLE
- The JNDI API and SPI
- Unified Naming and Directory Services
- Using the Remote Naming Service for Client Connections and SSL Support
  - Step 1: Set Up JNDI Environment Properties for the Initial Context
  - Step 2: Establish an InitialContext with the WLE Domain
  - Step 3: Use the Context to Look Up a Named Server Object
  - Step 4: Use the Named Server Object to Get a Reference for the Desired Remote Object, and Invoke Operations on the Remote Object
  - Step 5: Complete the Session
- Providing Remote Client Access to the UserTransaction Interface
- Using the Application Naming Service to Access Local Objects
- Using the Application Naming Service to Access Global Objects
  - Overview of Features
  - Accessing the Factories Subcontext

- Binding Objects into the Factories Subcontext
- Unbinding Objects from the Factories Subcontext
- J2EE Requirements
- Cross-Domain Support

- The J2EE Naming Context
  - Overview of Requirements
  - Accessing Environment Entries
  - Using EJB References
  - Obtaining Resource Factory References
  - Obtaining a UserTransaction Object

Overview of JNDI in WLE

In the enterprise, naming and directory services provide the means for your application to locate objects on the network. These services are key to building distributed applications. A **naming service** provides a mechanism to name objects and retrieve objects by name. A **directory service** is a naming service that also allows for attributes to be associated with each object, and provides a way to retrieve an object by its attributes instead of its name.

The Java Naming and Directory Interface (JNDI) is an API that provides directory and naming services to Java applications. JNDI is an integral component of the Sun Microsystems Java 2 Platform Enterprise Edition (J2EE) technology.

JNDI is defined to be independent of any specific naming or directory service implementation. A variety of services, new and existing ones, can be accessed in a common way. The JNDI Service Provider Interface (SPI) provides a means by which different naming and directory providers can develop and integrate their implementations so that the corresponding services are accessible from applications that use JNDI.

The JNDI support provided in the WLE software leverages from the standard Sun Microsystems, Inc., JNDI API classes. This support allows any service-provider implementation to be plugged into the JNDI framework using the standard SPI conventions. The support allows Java applications in WLE to access external directory services such as LDAP in a standardized fashion, by plugging in the appropriate service-provider.

In addition, BEA provides two WLE-specific SPI implementations to enable access to naming features in the WLE system. The WLE SPI provides the following features:

- A WLE **remote naming service** that provides a point of entry into the WLE system for remote Java clients using RMI on IIOP. The clients are Java EJB and RMI clients. This remote naming service also provide a global object naming service for finding server objects in the domain. The WLE FactoryFinder, NameManager, and EventBroker services are contained in separate system servers, and provide the underlying infrastructure for global namespace management and replication. The WLE JNDI SPI implementation utilizes these WLE system services for its global namespace management.
- A WLE **application naming service** for binding to and looking up:
  - Local objects in a given Java server
  - Named server objects throughout a WLE domain, and imported cross-domain objects.

  The local objects include deployment information and resources provided to applications within the containing environment of the server. The naming and access conventions for these local objects conforms to the J2EE specifications. There is also a specially named subcontext, `wle.factories`, in this namespace, which provides a mapping to the WLE NameManager/FactoryFinder for binding to and looking up global object references.

Note: WLE CORBA Java clients continue to use the existing WLE CosLifeCycle::FactoryFinder and CORBA naming services to access named factory objects. Both the CORBA and JNDI naming implementations share the same underlying namespace infrastructure and are functionally equivalent. Both programming styles are supported by a common naming infrastructure.

The JNDI API and SPI

The WLE SPI implementation for JNDI is based on the Sun Microsystems, Inc. JNDI 1.1 and SPI specifications. The basic JNDI framework implementation is based on version 1.1.2 of the Sun Microsystems Inc. standard extension classes.

The JNDI API is contained in two packages: `javax.naming` for the naming operations and `javax.naming.directory` for directory operations. The JNDI SPI is contained in the package `javax.naming.spi`.

The Naming Interface — javax.naming

The `javax.naming.Context` interface is the core interface that specifies a naming context. It defines basic operations such as adding a name-to-object binding, looking up the object bound to a specified name, listing the bindings, removing a name-to-object binding, and creating and destroying subcontexts of the same type.

`Context.lookup()` is the most commonly used operation. The context implementation can return an object of whatever class is required by the Java application.

The application is not exposed to any naming service implementation. In fact, a new type of naming service can be introduced without requiring the application to be modified or even disrupted if it is running.

The Directory Interface — javax.naming.directory

The WLE SPI supports the external interfaces in the `javax.naming` and `javax.naming.spi` packages. There is no WLE SPI support for the directory interfaces in `javax.naming.directory`. However, third-party directory provider services can be plugged-in, as shown in “Unified Naming and Directory Services” on page 1-7.

The Service Provider Interface — javax.naming.spi

The JNDI SPI allows different naming and directory service providers to develop and integrate their implementations so that the corresponding services are accessible from applications that use JNDI. Also, JNDI allows specification of names that span multiple namespaces. If one service provider implementation needs to interact with another in order to complete an operation, the SPI provides methods that allow different provider implementations to cooperate to complete client JNDI operations.

Additional WLE SPI Implementations

In addition to the standard Sun Microsystems Inc. interfaces for the JNDI API, WLE provides its own implementation that uses the standard JNDI SPI interfaces. The two `InitialContextFactory` class implementations that are provided in WLE are:

`com.beasys.jndi.WLEInitialContextFactory`

`weblogic.jndi.WLInitialContextFactory`

In your application code, you do not instantiate either of these classes directly. Instead you use the standard `javax.naming.InitialContext` class and set the appropriate Hashtable keys, as documented in the section "Using the Remote Naming Service for Client Connections and SSL Support" on page 1-8. All interaction is done through the `javax.naming.Context` interface, as described in the JNDI Javadoc.

As a convenience to application programmers, WLE also provides a `com.beasys.jndi.WLEContext` interface. This interface extends `javax.naming.Context`. There are some JNDI constants defined that you can use with the JNDI environment. These constants are used for authentication on the `InitialContext`, as explained in the section "WLE Keys Required for BEA TUXEDO Style Authentication" on page 1-11.

WLE JNDI Packaging

The server classes for WLE JNDI are set automatically when the JavaServer is booted.

The client classes for WLE JNDI are in:

```
<drive>:\wledir\java\jdk\m3envobj.jar
```

```
<drive>:\wledir\java\jdk\wleclient.jar
```

```
<drive>:\wledir\java\jdk\wlej2eecl.jar
```

```
<drive>:\wledir\java\jdk\weblogicaux.jar
```

Verify that your CLASSPATH is set correctly. If necessary, set your CLASSPATH to include the following JAR files:

Client on Windows NT:

```
set JDIR=%TUXDIR%\java\jdk
```

```
set CLASSPATH=.;%JDIR%\m3envobj.jar;%JDIR%\wleclient.jar;
%JDIR%\wlej2eecl.jar;%JDIR%\weblogicaux.jar
```

```
set JDIR=
```

Client on UNIX:

```
export JDIR=${TUXDIR}/java/jdk
```

```
export CLASSPATH=.:${JDIR}/m3envobj.jar:${JDIR}/wleclient.jar:
${JDIR}/wlej2eecl.jar:${JDIR}/weblogicaux.jar
```

```
export JDIR=
```

Location of the WLE JNDI Javadoc

A copy of the JNDI Javadoc from Sun Microsystems are provided as a convenience to developers in the WLE documentation and in a WLE installation folder. The default location for the top-level Javadoc page in the WLE installation folder is:

```
<drive>:\wledir\docs\index.html
```

Unified Naming and Directory Services

The WLE JNDI SPI implements the standard unified interface to multiple naming and directory services. The implementation allows a WLE J2EE Java application to connect to any naming and directory service, such as LDAP, NDS, or NIS, if the appropriate third-party JNDI service provider is used.

WLE provides this support by utilizing and redistributing the standard JNDI extension classes, provided by Sun Microsystems with JNDI version 1.1.2. BEA and any third-party vendors conforming to the JNDI Service Provide Interface (SPI) may provide access to vendor-specific naming and directory services.

The following code fragment uses JNDI to access a Sun Microsystems Inc. LDAP service provider.

```
Hashtable env = new Hashtable();
/*
* Specify the initial context implementation to use.
* The service provider supplies the factory class.
*/
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");

/* Specify host and port of LDAP server */
env.put(Context.PROVIDER_URL, "ldap://ldap.bigfoot.com:389");

/* Connect to the server and establish the initial context */
DirContext ctx = new InitialDirContext(env);

/* Access the LDAP directory using the context */
   .
   .
   .
```

Using the Remote Naming Service for Client Connections and SSL Support

The WLE remote naming SPI provides an `InitialContext` implementation that allows remote Java clients to connect into a WLE system. The client can specify standard JNDI context environment properties to identify the WLE system and other related connection parameters for logging into a WLE system.

To participate in a session with a WLE server application, 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:

- Step 1: Set Up JNDI Environment Properties for the Initial Context
- Step 2: Establish an InitialContext with the WLE Domain
- Step 3: Use the Context to Look Up a Named Server Object
- Step 4: Use the Named Server Object to Get a Reference for the Desired Remote Object, and Invoke Operations on the Remote Object
- Step 5: Complete the Session

Step 1: Set Up JNDI Environment Properties for the Initial Context

All Java remote client applications must first create environment properties. The `InitialContext` factory uses various properties to customize the `InitialContext` for a specific environment. You can set these properties by using a **Hashtable**. These properties, which are name-to-value pairs, determine how the `WLEInitialContextFactory` creates the `WLEContext`:

WLEContext.INITIAL_CONTEXT_FACTORY Property

Set this property to the WLE initial context factory, `com.beasys.jndi.WLEInitialContextFactory`, to access the WLE domain and remote naming services.

The class `com.beasys.jndi.WLEInitialContextFactory` provides the implementation for delegating JNDI methods to the WLE JNDI implementation. The class `com.beasys.jndi.WLEInitialContextFactory` provides an entry point for a client into the WLE domain namespace.

For example:

```
Hashtable env = new Hashtable();
/*
* Specify the initial context implementation to use.
* The service provider supplies the factory class.
*/
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");
   .
   .
   .
```

WLEContext.PROVIDER_URL Property

Set the URL of the service provider with the property name `java.naming.provider.url`. This property value should specify an IIOP Listener/Handler for the desired WLE target domain.

For example:

```
   .
   .
   .
env.put(WLEContext.PROVIDER_URL,
        "corbaloc://myhost:1000");
   .
   .
   .
```

The URL is interpreted directly by the Tobj_Bootstrap. The same syntax rules apply to indicate the search order or equivalence of handler addresses. You can also use the URL to specify an SSL connection using the `corbalocs:` URL scheme. The acceptable syntaxes for the URL are described in the Javadoc file for Tobj_Bootstrap.

The host and port combination specified in the URL must match the ISL parameter in the WLE application's UBBCONFIG file. The format of the host and port combination, as well as the capitalization, must match. If the addresses do not match, the communication with the WLE domain fails.

A WLE server that acts as a client should not specify this Context.PROVIDER_URL property because the server is already connected to the application in which it is booted. If this property is specified on the server it must be set to an empty string or null values.

WLEContext.SECURITY_AUTHENTICATION Property

The WLE system supports different levels of authentication. The SECURITY_AUTHENTICATION value determines whether certificate-based SSL authentication is attempted or BEA TUXEDO style authentication is used.

Valid values for this property key are "none", "simple", or "strong", as recommended by the Sun Microsystems Inc. JNDI specification.

For example:

```
   .
   .
   .
env.put(WLEContext.SECURITY_AUTHENTICATION,
        "strong");
   .
   .
   .
```

If you explicitly specify WLEContext.SECURITY_AUTHENTICATION="none", this setting causes WLE client software to bypass all security setup on the initial connection. If you do not include the WLEContext.SECURITY_AUTHENTICATION property at all, the default behavior is for the client code to check the current security level on the target domain and check for any required authentication parameters. You can explicitly set WLEContext.SECURITY_AUTHENTICATION="none" if your intention is to bypass the security level checking and authentication setup on the client. This might be done, for example, in a case where you know there is no authentication required on the target domain and you want to optimize performance. However, if authentication is required on the target domain and the WLEContext.SECURITY_AUTHENTICATION="none", a security exception might be generated until the client later attempts to access an object on the server.

If you specify the "strong" value, certificate-based authentication is attempted using SSL protocols. This is implemented using the CORBA PrincipalAuthenticator.authenticate method with an AuthenticationMethod value of CertificateBased.

If the SECURITY_AUTHENTICATION value is "simple" or is not specified, BEA TUXEDO style authentication is used. See the next section for information about the WLE specific keys used to support BEA TUXEDO style authentication.

WLE Keys Required for BEA TUXEDO Style Authentication

The WLE specific property keys in this section provide the additional parameters needed for BEA TUXEDO style authentication. For example:

```
Hashtable env = new Hashtable();
env.put(WLEContext.PROVIDER_URL, "corbaloc://myhost:1000");
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");

// Add Authentication parameters
env.put(WLEContext.SECURITY_AUTHENTICATION, "simple");
env.put(WLEContext.SYSTEM_PASSWORD, "RMI");
env.put(WLEContext.SECURITY_PRINCIPAL, "user");
env.put(WLEContext.CLIENT_NAME, "client");
env.put(WLEContext.SECURITY_CREDENTIALS, "userpw");

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

The property keys are as follows:

WLEContext.SECURITY_PRINCIPAL

Specifies the identity of the principal for authenticating the caller to the WLE domain.

WLEContext.SECURITY_CREDENTIALS

Specifies the credentials of the principal for authenticating the caller to the WLE domain. For SSL certificate-based authentication, enabled via SECURITY_AUTHENTICATION="strong", it specifies a pass phrase for accessing the private key and client certificate. For BEA TUXEDO style authentication, it identifies a String user password or an arbitrary Object user_data for the BEA TUXEDO AUTHSVC.

WLEContext.CLIENT_NAME

Specifies the client name for BEA TUXEDO style authentication.

WLEContext.SYSTEM_PASSWORD

Specifies the system password for BEA TUXEDO style authentication.

WLEContext.CODEBASE

Specifies the URL for network class loading.

Step 2: Establish an InitialContext with the WLE Domain

To access a service provider using JNDI, you create an `InitialContext` and identify a context factory, which creates the context for a certain provider.

To access a WLE domain, use the WLE context factory, "`com.beasys.jndi.WLEInitialContextFactory`" as an argument. After the context is created, it provides client access to factory names in the WLE domain using WLE as the name service provider.

To create a WLE remote domain context from a remote Java client, you must minimally specify this factory as the initial context factory, and specify the JNDI environment as properties passed to the constructor of the `InitialContext`.

The following example shows how to setup the initial environment properties and create the initial context:

```
Hashtable env = new Hashtable();
/*
* Specify the initial context implementation to use.
* This example will access the remote WLE domain context.
*/
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
"com.beasys.jndi.WLEInitialContextFactory");

/*
* Specify host and port of ISL/ISH gateway – this is only
* specified for remote clients.
*/
env.put(WLEContext.PROVIDER_URL, "corbaloc://myhost:1000");

/* Connect to the domain and establish the initial context */
Context ctx = new InitialContext(env);
   .
   .
   .
```

Step 3: Use the Context to Look Up a Named Server Object

The client application must obtain an initial object that provides services for the application. This object is named by the server and is bound to the domain namespace.

For EJB applications this object is usually an EJB Home object and provides references to other remote objects in the application. The `lookup` method on the `Context` object is used to obtain this named object. The argument passed to the `lookup` method is a string that contains the name of the desired server object.

The following code fragment shows how to get access to a named server object:

```
CartHome cartHome = (CartHome) ctx.lookup("johns-carts");
```

Step 4: Use the Named Server Object to Get a Reference for the Desired Remote Object, and Invoke Operations on the Remote Object

CORBA client applications get object references to other CORBA remote objects from factories.

EJB client applications get object references to EJB remote objects from EJB Homes.

RMI client applications can also get object references to other RMI remote objects from an initial named object.

All of these initial named remote objects are known to the WLE system generically as a "factory". A **factory** is any server object that can return a reference to another remote object and is bound into the WLE domain namespace.

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

The following code fragment obtains the desired remote object, and then invokes a method on the remote object:

```
Cart cart = cartHome.create("John", "7506");
```

```
cart.addItem(66);
```

Step 5: Complete the Session

After a client is finished working with a context, the client should close the context in order to release resources for the session and implicitly logoff. For example:

```
try {
  ctx.close();

} catch (Exception e) {
// a failure occurred
}
```

Providing Remote Client Access to the UserTransaction Interface

The Java Transaction API (JTA) defines the UserTransaction interface that is used by applications to start and commit or abort transactions. An application client gets a UserTransaction object through a JNDI lookup by using the name "java:comp/UserTransaction".

For example:

```
Context initCtx = new InitialContext(env);
UserTransaction utx = (UserTransaction)initCtx.lookup(
"java:comp/UserTransaction");
utx.begin();
   .
   .
   .

utx.commit();
```

Using the Application Naming Service to Access Local Objects

The WLE application naming SPI provides an application naming service for both:

- Local objects on a given Java server
- Global object references known throughout the domain

By default the root context and any application-created subcontexts are mapped into a local namestore on the server. This local namestore allows Java server applications to access local objects using standard JNDI conventions. This application naming service supports a hierarchical namespace and the creation of subcontexts.

Note: There is also a predefined subcontext named `wle.factories` that is mapped to the FactoryFinder/NameManager services in the WLE domain. This global context is discussed in the section "Accessing the Factories Subcontext" on page 1-17.

If an application running on a WLE server, such as an EJB or RMI object, needs access to local objects in the current JavaServer, it can create a JNDI Context. Because the object making the call is already logically in the environment of the server, there is no need to specify environment properties to access the default application namespace.

To create a context from within a server-side object, all you need to do is construct a new `InitialContext`. For example:

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

You do not need to specify a factory or a provider URL. By default, the context will be created as a WLE application context and will connect to the default naming service.

Using the Application Naming Service to Access Global Objects

This section describes the following topics about using the WLE application naming service to access global objects:

- Overview of Features
- Accessing the Factories Subcontext
- Binding Objects into the Factories Subcontext
- Unbinding Objects from the Factories Subcontext
- J2EE Requirements
- Cross-Domain Support

Overview of Features

The WLE application naming SPI provides an application naming service for both:

- Local objects on a given Java server, as described in the previous section, "Using the Application Naming Service to Access Local Objects" on page 1-15.
- Global object references known throughout the domain, as described in this section.

The WLE application naming SPI provides a specially identified named subcontext, `wle.factories`, that provides naming services for server objects identified throughout a WLE domain.

The WLE application naming SPI also provides support for named objects imported from other domains via the FactoryFinder cross-domain feature. This subcontext uses the WLE FactoryFinder and NameManager services to manage its bindings. To increase availability and reliability, you can create multiple FactoryFinders and NameManagers in the event one FactoryFinder or NameManager fails.

The `wle.factories` subcontext is mapped directly into the namespace managed by the FactoryFinder and NameManager. This subcontext is a flat namespace:

- It is a single-level JNDI subcontext, without any imbedded subcontexts
- All remote objects bound into the namespace are known throughout the WLE domain

The underlying namespace is also used by CORBA clients, and is extended via JNDI to support EJB and RMI clients. The CORBA client uses the CORBA FactoryFinder to access remote CORBA factory objects. An EJB or RMI client uses JNDI to access registered remote objects. An application may choose to implement naming conventions if it is desirable to prevent name collisions or to partition the namespace using its own syntax conventions.

Accessing the Factories Subcontext

Remote Java clients access the WLE domain namespace using the procedures described in the section "Using the Remote Naming Service for Client Connections and SSL Support" on page 1-8. Remote clients may perform `lookup` methods on the remote context, but cannot perform binding methods on the namespace.

Server applications access the WLE domain namespace by using the `wle.factories` subcontext under the default `InitialContext` on the server. The server initializes the `wle.factories` subcontext to the domain namespace in which it is booted. Server applications have full access to the subcontext and may bind objects into it. In the following example, a server object connects to the factories subcontext:

```
/* Get a default context and retrieve the factories subcontext */

Context rootCtx = new InitialContext();
Context factoryCtx = (Context) rootCtx.lookup("wle.factories");
```

Binding Objects into the Factories Subcontext

Server applications can create named objects in the domain namespace, which then allows client applications to easily locate the objects managed by the server. These objects are usually factories so that client applications can obtain references to other

remote objects in the application from the initial named objects. The binding of named remote objects into the domain namespace is typically the final step of the server application initialization process.

Note: For EJB applications the named factories are called Home objects. The EJB bean provider code does not need to explicitly bind the Home into the Context. The binding operation is done automatically by the EJB Container during server deployment, based on the bean-name in the deployment descriptor.

To use an object in the WLE domain namespace, WLE requires that objects returned by lookup must be remotely accessed from the client in a form that preserves its functionality. Thus it must be known to WLE as a remote object. What is actually stored in the underlying WLE NameManager is a mapping from the registered name to the WLE remote object reference.

The sample program in Listing 1-1 shows how to bind an RMI remote object into the WLE factories context during server startup. This makes the object available to clients who wish to lookup the object by name. The sample also shows how to remove the object from the JNDI context during server shutdown.

Listing 1-1 Sample bind.java Program

```
import javax.naming.*;    // Use standard JNDI 1.1 interfaces

public class ServerImpl extends com.beasys.Tobj.Server {
  static final String factoryName = "SimpFactory";
  SimpFactoryImpl factory;
  Context factoryCtx;

  // The initialize method is called during JavaServer startup
  public boolean initialize(String[] argv) {
    try {
    // Get the "wle.factories" subcontext from the default InitialContext

factoryCtx = (Context) new InitialContext().lookup("wle.factories");

      // Create the factory and make it available to clients via JNDI
      factoryCtx.bind(factoryName, new SimpFactoryImpl());
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
    return true;
  }
```

```
 // The release method is called during JavaServer shutdown
  public void release() {
    try {
      // Remove the binding from the "wle.factories" subcontext
      factoryCtx.unbind(factoryName);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Unbinding Objects from the Factories Subcontext

Once a server application receives a request to shut down, the server application can no longer receive requests from clients. During server shutdown it should unbind each of the objects previously bound into the namespace.

```
// Remove this factory reference from the WLE namespace
factoryCtx.unbind(tellerFname);
```

J2EE Requirements

The enterprise bean's **Home interface** defines the methods for the client to create, remove, and find EJB objects of the same type. That is, they are implemented by the same enterprise bean. The Home interface is specified by the Bean Provider. The Container creates a class that implements the Home interface. The Home interface extends the `javax.ejb.EJBHome` interface.

The container is responsible for making the Home interfaces of the deployed enterprise beans available to the client through JNDI. A client can locate an enterprise Bean Home interface through the WLE JNDI remote namespace provider. When an EJB-JAR module is deployed into a WLE JavaServer, the runtime system will automatically read the deployment descriptor and bind the remote Home into the JNDI factories subcontext.

Cross-Domain Support

For multidomain configurations the FactoryFinder supports access to named object references in another domain. You or your administrator define domain parameters in the application's `DMCONFIG` configuration file. The WLE application naming service is built on top of the FactoryFinder. When the FactoryFinder is configured for multidomain naming support, cross-domain names are available through the WLE application namespace.

The domain of an object reference is unknown to the application, and invocations on an object reference for a remote domain are transparent to the application. This transparency allows administrators to configure services in individual domains and to spread resources across multiple domains. Administrators are required to identify any named remote objects that can be used in the current (local) domain, but that are resident in a different (remote) domain. You identify these objects in a FactoryFinder domain configuration file named `factory_finder.ini`. This is an ASCII file that can be created and updated using a text editor.

For related information, see Configuring Multiple Domains (WLE System) in the WebLogic Enterprise online documentation.

The J2EE Naming Context

This section describes the following topics about the J2EE naming context:

- Overview of Requirements
- Accessing Environment Entries
- Using EJB References
- Obtaining Resource Factory References
- Obtaining a UserTransaction Object

Overview of Requirements

The Sun Microsystems Inc. J2EE specification at `http://java.sun.com/j2ee/` identifies specific JNDI requirements for an enterprise component to access named objects and external resources in a uniform manner. These requirements include:

1. An application component naming environment, for generic customization of the application component's business logic
2. Interfaces for obtaining the Home interface of an enterprise bean, using an EJB reference

   An **EJB reference** is a special entry in the application component's environment.
3. Interfaces for obtaining a resource factory, using a resource factory reference

   A **resource factory reference** is a special entry in the application component's environment.
4. Interfaces for obtaining the JTA `UserTransaction` interface to start, commit, and abort transactions

Accessing Environment Entries

An EJB component instance locates the environment naming context using the JNDI interfaces. An instance creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the naming environment via the `InitialContext` under the name `java:comp/env`.

The EJB's environment entries are stored directly in the environment naming context, or in any of its direct or indirect subcontexts. The value of an environment entry is of the Java type declared by the Bean Provider in the deployment descriptor. The environment entries are declared using the `<env-entry>` elements in the deployment descriptor.

The following code fragment shows how an EJB accesses its environment entries:

```
public void setTaxInfo(int numberOfExemptions, ...)
throws InvalidNumberOfExemptionsException {
   .
```

```
   .
   .

// Obtain the application component's environment naming context.
Context initCtx = new InitialContext();
Context myEnv = (Context)initCtx.lookup("java:comp/env");

// Obtain the maximum number of tax exemptions
// configured by the Deployer.
Integer max = (Integer)myEnv.lookup("maxExemptions");

// Obtain the minimum number of tax exemptions
// configured by the Deployer.
Integer min = (Integer)myEnv.lookup("minExemptions");

// Use the environment entries to customize business logic.
if (numberOfExeptions > max.intValue() ||
      numberOfExemptions < min.intValue())
      throw new InvalidNumberOfExemptionsException();

// Get some more environment entries. These environment
// entries are stored in subcontexts.
String val1 = (String)myEnv.lookup("foo/name1");
Boolean val2 = (Boolean)myEnv.lookup("foo/bar/name2");

// The application component can also lookup using full pathnames.
Integer val3 = (Integer)
initCtx.lookup("java:comp/env/name3");
Integer val4 = (Integer)
initCtx.lookup("java:comp/env/foo/name4");
   .
   .
   .

}
```

Using EJB References

This section describes the programming and deployment descriptor interfaces that allow the Bean Provider to refer to the Homes of enterprise beans using logical names called EJB references. The **EJB references** are special entries in the EJB's environment. The Deployer binds the EJB references to the enterprise bean's Home interfaces in the WLE operational environment.

The deployment descriptor also allows the Application Assembler to link an EJB reference declared in one application component to an enterprise bean contained in an EJB-JAR file in the same J2EE application. The link is an instruction to the tools used by the Deployer that the EJB reference should be bound to the home of the specified target enterprise bean.

The following example shows how an application component uses an EJB reference to locate the Home interface of an enterprise bean:

```
public void changePhoneNumber(...) {
   .
   .
   .
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();

// Look up the home interface of the EmployeeRecord
// enterprise bean in the environment.
Object result = initCtx.lookup(
"java:comp/env/ejb/EmplRecord");

// Convert the result to the proper type.
EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
   javax.rmi.PortableRemoteObject.narrow(result,
   EmployeeRecordHome.class");
   .
   .
   .

}
```

In the previous example, the Bean Provider assigned the environment entry `ejb/EmplRecord` as the EJB reference name to refer to the Home of an enterprise bean. The Bean Provider must declare all the EJB references using the `<ejb-ref>` elements of the deployment descriptor.

Obtaining Resource Factory References

A **resource** is an object that encapsulates access to a resource manager. A **resource factory** is an object that is used to create resources. For example, an object that implements the `java.sql.Connection` interface is a resource that provides access to a database management system, and an object that implements the `javax.sql.DataSource` interface is a resource factory.

This section describes the application component programming and deployment descriptor interfaces that allow the application component code to refer to resource factories using logical names called resource factory references. The **resource factory references** are special entries in the EJB's environment. The Deployer binds the resource factory references to the actual resource factories in the WLE operational environment.

The following code fragment illustrates obtaining a resource:

```
public void changePhoneNumber(...) {
   .
   .
   .

// obtain the initial JNDI context
Context initCtx = new InitialContext();

// perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds = (javax.sql.DataSource)
      initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");

// Invoke factory to obtain a resource.
java.sql.Connection con = ds.getConnection();
   .
   .
   .

}
```

The Bean Provider must declare all the resource factory references in the deployment descriptor using the `<resource-ref>` elements.

Obtaining a UserTransaction Object

Many J2EE application component types are allowed to use the JTA `UserTransaction` interface to start, commit, and abort transactions. Such application components can find an appropriate object that implements the `UserTransaction` interface by looking up the JNDI name `java:comp/UserTransaction`.

The container is only required to provide `java:comp/UserTransaction` for those components that can make valid use of it. Any such `UserTransaction` object is only valid within the component instance that performed the lookup. Only some application component types are required to have access to a `UserTransaction` object. For details, refer to the Sun Microsystems Inc. EJB 1.1 specification.

Note: For your convenience, a PDF copy of the EJB 1.1 specification, Public Release 2, is included with the WLE online documentation. To access the HTML page that includes a copy of the EJB 1.1 specification, click the PDF Files button at the top of a WLE online documentation HTML page.

The following example illustrates how an application component acquires and uses a UserTransaction object:

```
public void updateData(...) {
   .
   .
   .

// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();

// Look up the UserTransaction object.
UserTransaction tx = (UserTransaction)initCtx.lookup(
    "java:comp/UserTransaction");

// Start a transaction.
tx.begin();
   .
   .
   .

// Perform transactional operations on data.
   .
   .
   .

// Commit the transaction.
tx.commit();
   .
   .
   .
}
```

Index

A

accessing environment entries 1-21
application naming service
    global objects 1-16
    local objects 1-15
authentication
    TUXEDO style 1-11

B

BEA TUXEDO
    authentication properties 1-11

C

CLASSPATH
    setting for WLE JAR files 1-6
client access
    to UserTransaction 1-14
CLIENT_NAME property key 1-11
completing a session 1-14
connecting remote Java clients 1-8
cross-domain support 1-20
customer support contact information vii

D

directory service
    definition 1-2
directory services 1-7
documentation, where to find it v
domains
    interdomain support 1-20

E

EJB
    using references 1-22
environment naming context 1-21
establishing initial context 1-12

F

factories
    resource references 1-23
    using to get remote objects 1-13

G

global objects
    using application naming service 1-16

H

hashtables
    setting environment properties 1-8

I

initial context
    establishing 1-12
INITIAL_CONTEXT_FACTORY property 1-9
InitialContextFactory class 1-5

Introduction
WLE JNDI SPI implementation 1-2

J

JAR packages
for WLE 1-6
java
comp/env initial context 1-21
Java archive files 1-6
Javadoc
location 1-6
javax.ejb.EJBHome interface 1-19
javax.naming interface 1-4
javax.naming.spi interface 1-5
JNDI
looking up server objects 1-13

M

m3.jar file 1-6
m3envobj.jar file 1-6

N

named server objects
looking up 1-13
naming context
environment 1-21
naming interface 1-4
naming service
definition 1-2
naming services 1-7
none security property 1-10

O

objects
looking up via JNDI 1-13
obtaining UserTransaction 1-24
obtaining resource factory references 1-23

P

packages
JAR files 1-6
passwords
SYSTEM_PASSWORD property key 1-11
printing product documentation vi
PROVIDER_URL property 1-9

R

references
EJB 1-22
related information vi
remote clients
access to UserTransaction 1-14
remote Java clients
connecting into WLE 1-8
remote naming service 1-8
remote objects
accessing via a factory 1-13
resource factory references 1-23

S

SECURITY_AUTHENTICATION property 1-10
SECURITY_CREDENTIALS property key 1-11
SECURITY_PRINCIPAL property key 1-11
server objects
looking up 1-13
Service Provider Interface (SPI)
overview 1-3
sessions
completing 1-14
simple security property 1-10
SPI implementation
overview 1-3
strong security property 1-10
support

technical vii
SYSTEM_PASSWORD property key 1-11

T

TUXEDO
authentication properties 1-11

U

UserTransaction interface
providing remote client access 1-14
UserTransaction objects
obtaining 1-24

W

weblogicaux.jar file 1-6
wleclient.jar file 1-6
WLEContext
setting environment properties 1-9
WLEInitialContextFactory
setting 1-12
WLEInitialContextFactory class 1-5
WLInitialContextFactory class 1-5