

JNDI

Java Naming and Directory Interface

See also:

<http://java.sun.com/products/jndi/tutorial/trailmap.html>

Naming service

A naming service is an entity that
•**associates names with objects.** We call this **binding names to objects**. *This is similar to a telephone company's associating a person's name with a specific residence's telephone number*

•**provides a facility to find an object based on a name.** We call this **looking up or searching for an object**. *This is similar to a telephone operator finding a person's telephone number based on that person's name and connecting the two people.*

In general, a naming service can be used to find any kind of generic object, like a file handle on your hard drive or a printer located across the network.

Directory service

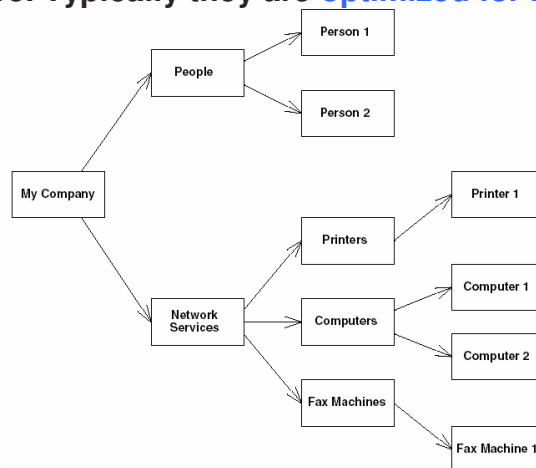
A **directory object** differs from a generic object because you can store **attributes** with directory objects. For example, you can use a directory object to represent a user in your company. You can store information about that user, like the user's password, as attributes in the directory object.

A **directory service** is a naming service that has been extended and enhanced to provide directory object operations for manipulating attributes.

A **directory** is a system of directory objects that are all connected. Some examples of directory products are Netscape Directory Server and Microsoft's Active Directory.

Directory service

Directories are similar to DataBases, except that they typically are organized in a **hierarchical tree-like** structure. Typically they are **optimized for reading**.



Examples of Directory services

Netscape Directory Server

Microsoft 's Active Directory

Lotus Notes (IBM)

NIS (Network Information System) by Sun

NDS (Network Directory Service) by Novell

LDAP (Lightweight Directory Access Protocol)

JNDI concepts

JNDI is a system for Java-based clients to interact with naming and directory systems. JNDI is a bridge over naming and directory services, that provides one common interface to disparate directories.

Users who need to access an LDAP directory use the same API as users who want to access an NIS directory or Novell's directory. All directory operations are done through the JNDI interface, providing a common framework.

JNDI advantages

-You only need to learn a single API to access all sorts of directory service information, such as security credentials, phone numbers, electronic and postal mail addresses, application preferences, network addresses, machine configurations, and more.

-JNDI insulates the application from protocol and implementation details.

-You can use JNDI to read and write whole Java objects from directories.

- You can link different types of directories, such as an LDAP directory with an NDS directory, and have the combination appear to be one large, federated directory.

JNDI advantages

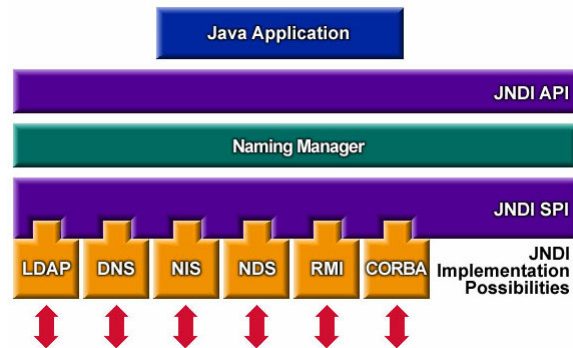
Applications can store factory objects and configuration variables in a global naming tree using the JNDI API.

JNDI, the Java Naming and Directory Interface, provides a global memory tree to store and lookup configuration objects. JNDI will typically contain configured Factory objects.

JNDI lets applications cleanly separate configuration from the implementation. The application will grab the configured factory object using JNDI and use the factory to find and create the resource objects.

In a typical example, the application will grab a database DataSource to create JDBC Connections. Because the configuration is left to the configuration files, it's easy for the application to change databases for different customers.

JNDI Architecture



The JNDI homepage
<http://java.sun.com/products/jndi>
has a list of service providers.

JNDI concepts

An **atomic name** is a simple, basic, indivisible component of a name. For example, in the string `/etc/fstab`, `etc` and `fstab` are atomic names.

A **binding** is an association of a name with an object.

A **context** is an object that contains zero or more bindings. Each binding has a distinct atomic name. Each of the `mtab` and `exports` atomic names is bound to a file on the hard disk.

A **compound name** is zero or more atomic names put together. e.g. the entire string `/etc/fstab` is a compound name. Note that a compound name consists of multiple bindings.

JNDI names

JNDI names look like URLs.

A typical name for a database pool is `java:comp/env/jdbc/test`. The `java:` scheme is a memory-based tree. `comp/env` is the standard location for Java configuration objects and `jdbc` is the standard location for database pools.

Other URL schemes are allowed as well, including RMI (`rmi://localhost:1099`) and LDAP. Many applications, though will stick to the `java:comp/env` tree.

Examples

<code>java:comp/env</code>	Configuration environment
<code>java:comp/env/jdbc</code>	JDBC DataSource pools
<code>java:comp/env/ejb</code>	EJB remote home interfaces
<code>java:comp/env/cmp</code>	EJB local home interfaces (non-standard)
<code>java:comp/env/jms</code>	JMS connection factories
<code>java:comp/env/mail</code>	JavaMail connection factories
<code>java:comp/env/url</code>	URL connection factories
<code>java:comp/UserTransaction</code>	UserTransaction interface

JNDI names

There are three commonly used levels of naming scope in JBoss:
names under `java:comp`,
names under `java:`,
any other name.

`java:comp` context and its subcontexts are **only available to the application component associated with that particular context**.

Subcontexts and object bindings directly under `java:` are **only visible within the JBoss server virtual machine and not to remote clients**.

Any other context or object binding is **available to remote clients, provided the context or object supports serialization**.

An example of where the restricting a binding to the `java:` context is useful would be a `javax.sql.DataSource` connection factory that can only be used inside of the JBoss server where the associated database pool resides. On the other hand, an EJB home interface would be bound to a globally visible name that should be accessible by remote client.

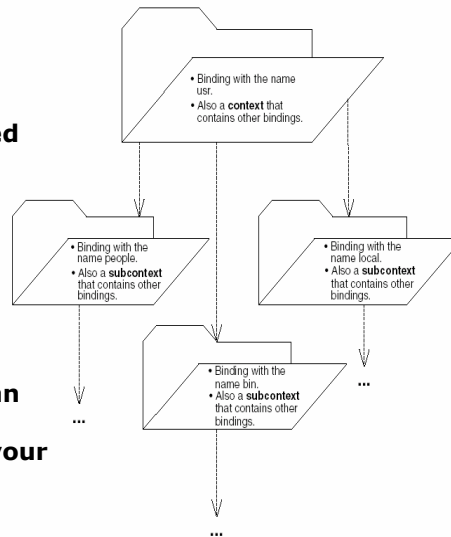
Contexts and Subcontexts

A naming system is a connected set of contexts.

A namespace is all the names contained within naming system.

The starting point of exploring a namespace is called an *initial context*. An initial context is the first context you happen to use.

To acquire an initial context, you use an *initial context factory*. An initial context factory basically is your JNDI driver.



Acquiring an initial context

When you acquire an initial context, you must supply the necessary information for JNDI to acquire that initial context.

For example, if you're trying to access a JNDI implementation that runs within a given server, you might supply:

- The IP address of the server
- The port number that the server accepts
- The starting location within the JNDI tree
- Any username/password necessary to use the server

Acquiring an initial context

package examples;

```
public class InitCtx {
    public static void main(String args[]) throws Exception {
        // Form an Initial Context
        javax.naming.Context ctx =
            new javax.naming.InitialContext();
        System.err.println("Success!");
        Object result = ctx.lookup("PermissionManager");
    }
}
```

java

```
-Djava.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
-Djava.naming.provider.url=jnp://193.205.194.162:1099
-Djava.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
examples.InitCtx
```

Acquiring an initial context

java.naming.factory.initial: The name of the environment property for specifying the initial context factory to use. The value of the property should be the fully qualified class name of the factory class that will create an initial context.

java.naming.provider.url: The name of the environment property for specifying the location of the JBoss JNDI service provider the client will use. The NamingContextFactory class uses this information to know which JBossNS server to connect to. The value of the property should be a URL string. For JBossNS the URL format is

`jnp://host:port/[jndi_path]`.

Everything but the host component is optional. The following examples are equivalent because the default port value is 1099.

`jnp://www.jboss.org:1099/`

`www.jboss.org:1099`

`www.jboss.org`

Acquiring an initial context

java.naming.factory.url.pkgs:

The name of the environment property for specifying the list of package prefixes to use when loading in URL context factories. The value of the property should be a colon-separated list of package prefixes for the class name of the factory class that will create a URL context factory. For the JBoss JNDI provider this must be

[org.jboss.naming:org.jnp.interfaces](#).

This property is essential for locating the jnp: and java: URL context factories of the JBoss JNDI provider.

Another example

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;
class Lookup {
    public static void main(String[] args) {
        // Check that user has supplied name of file to lookup
        if (args.length != 1) {
            System.err.println("usage: java Lookup <filename>");
            System.exit(-1);
        }
        String name = args[0];
        // Identify service provider to use
        Hashtable env = new Hashtable(11);
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
```

```
    try {
        // Create the initial context
        Context ctx = new InitialContext(env);
        // Look up an object
        Object obj = ctx.lookup(name);
        // Print it out
        System.out.println(name +
            " is bound to: " + obj);
        // Close the context when we're done
        ctx.close();
    } catch (NamingException e) {
        System.err.println("Problem looking up "
            + name + ": " + e);
    }
}
```

LDAP example

```
try {
    // Create the initial directory context
    DirContext ctx = new InitialDirContext(env);

    // Ask for all attributes of the object
    Attributes attrs = ctx.getAttributes("cn=Ronchetti
Marco");

    // Find the surname ("sn") and print it
    System.out.println("sn: " + attrs.get("sn").get());

    // Close the context when we're done
    ctx.close();
} catch (NamingException e) {
    System.err.println("Problem getting attribute: " + e);
}}
```

```
package jndiaccessstoldap;
import javax.naming.Context;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.DirContext;
import javax.naming.directory.Attributes;
import javax.naming.NamingException;
import java.util.Hashtable;
public class Getattr {
    public static void main(String[] args) {
        // Identify service provider to use
        Hashtable env = new Hashtable(11);
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
        //env.put(Context.PROVIDER_URL, "ldap://ldap.unitn.it:389/o=JNDITutorial");
        env.put(Context.PROVIDER_URL, "ldap://ldap.unitn.it:389/o=personale");
    }
}
```

Operations on a JNDI context

list() retrieves a list of contents available at the current context. This typically includes names of objects bound to the JNDI tree, as well as subcontexts.

lookup() moves from one context to another context, such as going from `c:\` to `c:\windows`. You can also use `lookup()` to look up objects bound to the JNDI tree. The return type of `lookup()` is JNDI driver specific.

rename() gives a context a new name

Operations on a JNDI context

createSubcontext() creates a subcontext from the current context, such as creating `c:\foo \bar` from the folder `c:\foo`.

destroySubcontext() destroys a subcontext from the current context, such as destroying `c:\foo \bar` from the folder `c:\foo`.

bind() writes something to the JNDI tree at the current context. As with `lookup()`, JNDI drivers accept different parameters to `bind()`.

rebind() is the same operation as `bind`, except it forces a bind even if there is already something in the JNDI tree with the same name.

JNDI in JBoss

The `JNDIView` MBean allows the user to view the JNDI namespace tree as it exists in the JBoss server using the JMX agent view interface.

The screenshot shows the JBoss JMX Management Console interface. The browser address bar indicates the URL `http://localhost:8080/jmx-console/index.jsp`. The page title is "JMX Agent View" and the domain is "toki.local". There is an "ObjectName Filter" field with the value `(e.g. "jboss:*", "":service=invoker,*")` and buttons for "Apply Filter" and "Clear Filter". The "JMXImplementation" section lists several services, including `name=PropertyEditorManager,type=Service`, `name=SystemProperties,type=Service`, `readonly=true,service=invoker,target=Naming,type=http`, `service=ClientUserTransaction`, `service=JNDIView`, `service=Mali`, `service=Naming`, `service=TransactionManager`, `service=UUIDKeyGeneratorFactory`, `service=WebService`, `service=XidFactory`, `service=invoker,target=Naming,type=http`, and `service=invoker,type=http`. The JBoss logo is visible in the top right corner.

JNDI in JBoss

MBean Inspector

http://localhost:8080/jmx-console/HtmlAdaptor?action=ir

Attribute Name	Attribute Value
Name (R) java.lang.String <i>The class name of the MBean</i>	JNDIView
State (R) int <i>The status of the MBean</i>	3
StateString (R) java.lang.String <i>The status of the MBean in text form</i>	Started

Operation Name	Parameters
list java.lang.String <i>Output JNDI info as text</i>	verbose boolean <i>If true, list the class of each object in addition to its name</i> <input checked="" type="radio"/> True <input type="radio"/> False <input type="button" value="Invoke"/>
listXML java.lang.String <i>Output JNDI info in XML format</i>	<input type="button" value="Invoke"/>

JNDI in JBoss

Operation Results

http://localhost:8080/jmx-console/HtmlAdaptor

java: Namespace

```
+-- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
+-- DefaultDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
+-- SecurityProxyFactory (class: org.jboss.security.SubjectSecurityProxyFactory)
+-- DefaultJMSProvider (class: org.jboss.jms.jndi.JBossMQProvider)
+-- comp (class: javax.naming.Context)
+-- JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
+-- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
+-- jaas (class: javax.naming.Context)
|   +-- JmsXARealm (class: org.jboss.security.plugins.SecurityDomainContext)
|   +-- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
|   +-- HsqlDbRealm (class: org.jboss.security.plugins.SecurityDomainContext)
+-- timedCacheFactory (class: javax.naming.Context)
Failed to lookup: timedCacheFactory, errmsg=null
+-- TransactionPropagationContextExporter (class: org.jboss.tm.TransactionPropagationContextExport
+-- Mail (class: javax.mail.Session)
+-- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)
+-- TransactionPropagationContextImporter (class: org.jboss.tm.TransactionPropagationContextImpo
+-- TransactionManager (class: org.jboss.tm.TxManager)
```


Global JNDI Namespace

```
+-- jmx (class: org.jnp.interfaces.NamingContext)
|   +-- invoker (class: org.jnp.interfaces.NamingContext)
|   |   +-- RMIA adaptor (proxy: $Proxy22 implements interface org.jboss.jmx.adaptor.rmi.RMIAdapt
|   |   |   +-- xml (class: org.jnp.interfaces.NamingContext)
|   |   |   |   +-- RMIA adaptor[link -> jmx/invoker/RMIAaptor] (class: javax.naming.LinkRef)
|   |   |   |   +-- OIL2XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
|   |   |   |   +-- HTTPXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
|   |   |   |   +-- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
|   |   |   |   +-- UserTransactionSessionFactory (proxy: $Proxy10 implements interface org.jboss.tm.usertx.int
|   |   |   |   +-- HTTPConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
```

JNDI e EJB: definizione di proprietà in configuration

An example ejb-jar.xml env-entry fragment

```
<!-- ... -->
<session>
<ejb-name>ASessionBean</ejb-name>
<!-- ... -->
<env-entry>
<description>The maximum number of tax exemptions allowed </description>
<env-entry-name>maxExemptions</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>15</env-entry-value>
</env-entry>
<env-entry>
<description>The tax rate </description>
<env-entry-name>taxRate</env-entry-name>
<env-entry-type>java.lang.Float</env-entry-type>
<env-entry-value>0.23</env-entry-value>
</env-entry>
</session>
<!-- ... -->
```

JNDI e EJB: accesso alle proprietà in configuration

env-entry access code fragment

```
InitialContext iniCtx = new InitialContext();
Context envCtx = (Context) iniCtx.lookup("java:comp/env");
Integer maxExemptions = (Integer) envCtx.lookup("maxExemptions");
Float taxRate = (Float) envCtx.lookup("taxRate");
```

JNDI e EJB: definizione di proprietà in configuration

An example ejb-jar.xml ejb-ref descriptor fragment

```
<session>
<ejb-ref>
<ejb-name>ShoppingCartUser</ejb-name>
<!--...-->
<ejb-ref-name>ejb/ShoppingCartHome</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>org.jboss.store.ejb.ShoppingCartHome</home>
<remote> org.jboss.store.ejb.ShoppingCart</remote>
<ejb-link>ShoppingCartBean</ejb-link>
</ejb-ref>
</session>
```

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ShoppingCartHome home =
    (ShoppingCartHome) ejbCtx.lookup("ShoppingCartHome");
```

JNDI e Servlets: definizione di proprietà in configuration

```
<web>
<!-- ... -->
<servlet> <servlet-name>AServlet</servlet-name> <!-- ... --> </servlet>
<!-- ... -->
<!-- JavaMail Connection Factories (java:comp/env/mail) -->
<resource-ref>
<description>Default Mail</description>
<res-ref-name>mail/DefaultMail</res-ref-name>
<res-type>javax.mail.Session</res-type>
<res-auth>Container</res-auth>
```

```
Context initCtx = new InitialContext();
javax.mail.Session s = (javax.mail.Session)
    initCtx.lookup("java:comp/env/mail/DefaultMail");
```