

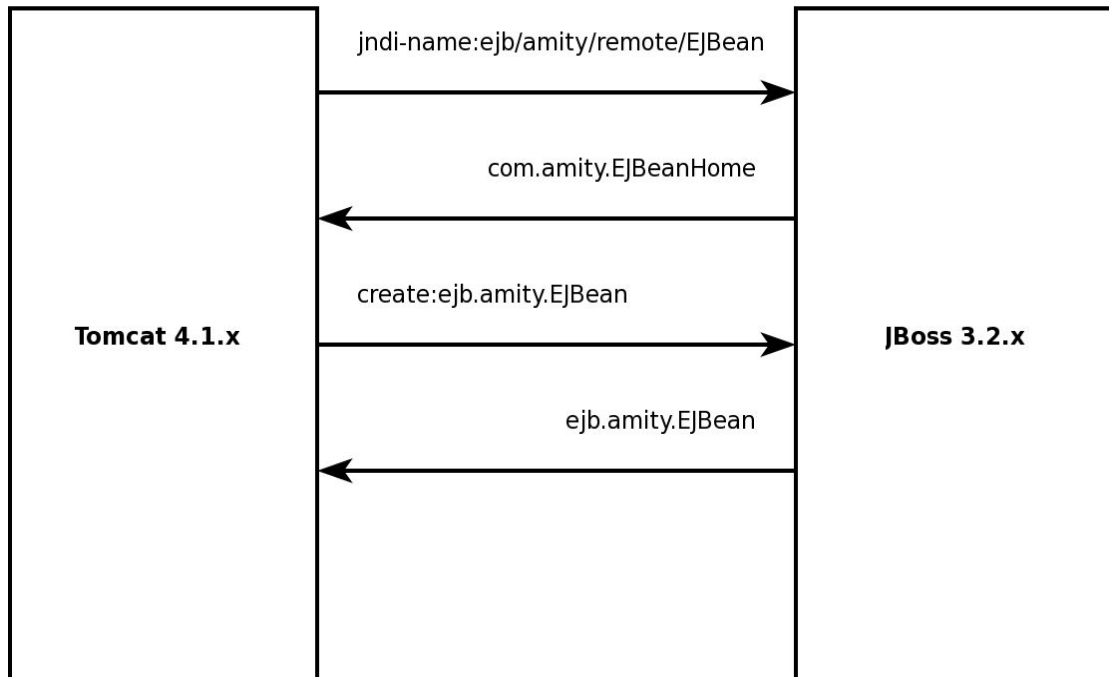
## JBoss, Tomcat and JNDI

This technote is a quick guide to adding JNDI lookups for JBoss components to a standalone Tomcat server. It discusses how to configure Tomcat to achieve this and what components are required to support the configuration.

### The JNDI lookup problem

You have a web application that uses EJB components and it works in a JBoss environment where the web application server is embedded in the JBoss server. However, you now need to deliver the same functionality for a separated JBoss-Tomcat environment. This configuration may be required for quite a few client installations.

Tomcat provides its own JNDI service but it cannot be used to directly locate JBoss components as there are no locations of JBoss stored in the Tomcat JNDI service. You need to get your web application to perform the lookup using the JBoss JNDI service as the location of the home objects for your EJBs are stored in that directory.



The preceding diagram gives the sequence of events required for your web application to obtain and use EJB components located on the JBoss application server. We will use this as our reference in the rest of the examples. Based on this broad outline of the events, there are several ways to achieve the lookup for your application.

The first method for achieving your goal is to hardcode the lookup into your application.

```
try
{
    Properties jndiProps = new Properties();
    jndiProps.setProperty(Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");
    jndiProps.setProperty(Context.URL_PKG_PREFIXES,
        "org.jboss.naming:org.jnp.interface");
    jndiProps.setProperty(Context.PROVIDER_URL, "jnp://jboss-server:1099")
    Object reference = (new InitialContext(jndiProps)).lookup
        ("ejb/amity/remote/EJBean");
    EJBeanHome home = (EJBeanHome)PortableRemoteObject.narrow(reference,
        EJBeanHome.class);
    EJBean bean = home.create();
    ...
}
catch(Exception e)
{
    e.printStackTrace();
}
```

However, the drawback with this method is that the code has to be changed if the server location changes. This requires the involvement of a programmer and for large deployments can involve a reasonable amount of changes and testing.

The second method is to make use of the JNDI InitialContext implementation and a custom Object Factory. This separates the programming references to the JNDI resource from the physical location of the JBoss JNDI server. This method applies the same mapping principles used for the JBoss web.xml – jboss-web.xml implementation. We make use of web.xml for the web application and instead of jboss-web.xml, we use the Tomcat server.xml to achieve the mapping to the physical resources. Note that this allows us to still use the web.xml – jboss-web.xml when we deploy the web application on JBoss so there is no programming changes necessary to the web application. However, in order to achieve this deployment flexibility, the web application must make use of the remote EJB interfaces and cannot take advantage of the local EJB interface efficiencies.

The XDoclet tags for the web.xml generation in our example would be:

```
* @web.ejb-local-ref      name="ejb/amity/EJBean"
*                          type="Session"
*                          home="com.amity.EJBeanHome"
*                          local="com.amity.EJBean"
```

This generates the following EJB reference for our web application in web.xml.

```
<ejb-ref >
  <ejb-ref-name>ejb/amity/EJBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.amity.EJBeanHome</home>
  <remote>com.amity.EJBean</remote>
</ejb-ref>
```

The servlet coding to access the resource would be:

```
try
{
    Object reference = (new InitialContext()).lookup
        ("java:comp/env/ejb/amity/EJBean");
    EJBeanHome home = (EJBeanHome)PortableRemoteObject.narrow(reference,
        EJBeanHome.class);
    EJBean bean = home.create();
    ...
}
catch(Exception e)
{
    e.printStackTrace();
}
```

Take note that in particular we create an InitialContext using the JNDI properties already defined for the environment. Both JBoss and Tomcat define these to refer to their local JNDI services. This is much less set up work than required for the previous example. Also note that we will not need to change the lookup code when we deploy the application to the standalone Tomcat server or to the JBoss server.

In order to accomplish the mapping for Tomcat, we need to modify the CATALINA\_HOME/conf/server.xml in order to define destination for the reference used by the web application. This requires defining a context for the web application and then defining the resources for that context.

Additionally, we need Tomcat to be able to generate the actual home object that will be used by the application. Tomcat cannot natively do this but provides for plug-ins to perform the task. This is achieved by building a class that implements the Naming Object Factory and defining it as the factory for the resource parameters. We will get to the implementation later. For now we will look at the configuration and assume we have an Object Factory implementation called com.amity.utils.tomcat.EJBRefFactory.

The required declarations for our application to reference the JBoss located component is shown below. We define an EJB whose name matches that of the `ejb-ref-name` defined in the application's `web.xml` file. In this example, the value providing the mapping link is `ejb/amity/EJBean`. Check the `Ejb` tag and compare it with the `web.xml` entry for `ejb-ref` shown earlier. We also define resource parameters tagged by `ResourceParams` that are linked to `Ejb` by the name. Again, this value is `ejb/amity/EJBean`. The following example `server.xml` fragment shows the important features of the configuration.

```
<Context path="/webapplication" docBase="webapplication" debug="0"
    reloadable="true" crossContext="true">
  <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_webapplication_log." suffix=".txt"
    timestamp="true"/>
  <Ejb name="ejb/amity/EJBean" type="Session"
    home="com.amity.EJBeanHome"
    remote="com.amity.EJBean"/>
  <ResourceParams name="ejb/amity/EJBean">
    <parameter>
      <name>factory</name>
      <value>com.amity.utils.tomcat.EJBRefFactory</value>
    </parameter>
    <parameter>
      <name>java.naming.provider.url</name>
      <value>jnp://jboss-server:1099</value>
    </parameter>
    <parameter>
      <name>jndi-name</name>
      <value>ejb/amity/remote/EJBean</value>
    </parameter>
  </ResourceParams>
</Context>
```

The class that will actually do the work of finding the home object for the `EJBean` component is **`com.amity.utils.tomcat.EJBRefFactory`**. The parameters needed for the class to find the resource is at a minimum the URL of the JNDI service, **`jnp://jboss-server:1099`** and the global JNDI name for the home object, **`ejb/amity/remote/EJBean`**. These are all declared in the `ResourceParams` section.

All that remains is to build an implementation of Object Factory that performs the lookup and return the located home object if it exists. The Object Factory has only one method of importance that must be provided in the implementation. This is `getObjectInstance`. We use NetBeans as the development platform and it generates the code skeleton including methods required to be implemented when we create a class that implements an interface. Our implementation of the method is shown next.

```

private final static String JNDINAME = "jndi-name";
private final static String URL = "jnp://localhost:1099";
private final static String FACTORY =
    "org.jnp.interfaces.NamingContextFactory";
private final static String PKG_PREFIXES =
    "org.jboss.naming:org.jnp.interface";
public Object getObjectInstance(Object obj, javax.naming.Name name,
    javax.naming.Context context, java.util.Hashtable hashtable)
    throws java.lang.Exception
{
    String jndiName = null;
    String factory = null;
    String url = null;
    String pkgPrefixes = null;
    Object home = null;
    // Make sure that the reference object was the Tomcat EjbRef
    Class referenceClass = Class.forName("org.apache.naming.EjbRef");
    if (referenceClass.isAssignableFrom(obj.getClass()))
    {
        String parameterName;
        String parameterValue;
        RefAddr address = null;
        Properties env = new Properties();
        // Obtain the reference to the parameters defined in ResourceParams
        Reference reference = (Reference)obj;
        // Process all the parameters
        Enumeration parameters = reference.getAll();
        while (parameters.hasMoreElements())
        {
            address = (RefAddr)parameters.nextElement();
            parameterValue = address.getContent().toString();
            parameterName = address.getType();
            if (parameterName.equals(Context.INITIAL_CONTEXT_FACTORY))
                factory = parameterValue;
            else if (parameterName.equals(Context.PROVIDER_URL))
                url = parameterValue;
            else if (parameterName.equals(Context.URL_PKG_PREFIXES))
                pkgPrefixes = parameterValue;
            else if (parameterName.equals(JNDINAME))
                jndiName = parameterValue;
        }
    }
}

```

```

// Only do the lookup when there is a defined jndi-name parameter
if (jndiName != null)
{
    // Use the default JNDI URL
    if (url == null)
        url = URL;
    // Use the default JBoss JNDI factory
    if (factory == null)
        factory = FACTORY;
    // Use the default JBoss JNDI package prefixes
    if (pkgPrefixes == null)
        pkgPrefixes = PKG_PREFIXES;
    env.put(Context.INITIAL_CONTEXT_FACTORY, factory);
    env.put(Context.PROVIDER_URL, url);
    env.put(Context.URL_PKG_PREFIXES, pkgPrefixes);
    // Perform the lookup
    home = (new InitialContext(env)).lookup(jndiName);
}
}
return home;
}

```

You may notice that the implementation does a bit more processing but essentially performs the same work as the hard-coded example. We add some refinements so that if parameter values are not provided for the JBoss JNDI factory, the provider URL or the URL package prefixes, some defaults for JBoss are substituted.

When the InitialContext lookup is performed in Tomcat, the work is delegated to the defined resource factory for the resource.

Our implementation allows the following definitions.

1. Default JBoss JNDI factory, default JBoss package prefix and provider URL is `jnp://localhost:1099`

```

<ResourceParams name="ejb/amity/EJBean">
    <parameter>
        <name>factory</name>
        <value>com.amity.utils.tomcat.EJBRefFactory</value>
    </parameter>
    <parameter>
        <name>jndi-name</name>
        <value>ejb/amity/remote/EJBean</value>
    </parameter>
</ResourceParams>

```

## 2. No defaults

```
<ResourceParams name="ejb/amity/EJBean">
  <parameter>
    <name>factory</name>
    <value>com.amity.utils.tomcat.EJBRefFactory</value>
  </parameter>
  <parameter>
    <name>java.naming.provider.url</name>
    <value>jnp://jboss-server:1099</value>
  </parameter>
  <parameter>
    <name>java.naming.factory.initial</name>
    <value>org.jnp.interfaces.NamingContextFactory</value>
  </parameter>
  <parameter>
    <name>java.naming.factory.url.pkgs</name>
    <value>org.jboss.naming:org.jnp.interfaces</value>
  </parameter>
  <parameter>
    <name>jndi-name</name>
    <value>ejb/amity/remote/EJBean</value>
  </parameter>
</ResourceParams>
```

The only mandatory parameters for this implementation are the factory and the jndi-name.

Finally, the only thing necessary to get the configuration operational is to copy jbossall-client.jar and the JAR containing your custom class into CATALINA\_HOME/shared/lib for Tomcat 4.1.24 or earlier; or CATALINA\_HOME/common/lib for Tomcat 4.1.27 or later.

There are two things to be aware of when using the implementation in Tomcat. The lookups are less efficient than in an integrated JBoss environment as there are two InitialContext and two lookup operations performed in the Tomcat implementation. The deployment for mapping is a bit more cumbersome in Tomcat as the configuration is conducted in server.xml and requires a restart of the Tomcat server for the configuration changes to be implemented.

Further information on related topics can be found at:

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jndi-resources-howto.html>

<http://www.amitysolutions.com.au/documents/JBossJNDI-technote.pdf>

The package containing this document, the source implementation and the JAR can be found at

<http://www.amitysolutions.com.au/downloads/jbosstomcatjndi.tar.gz>