



Author: Jon Barnett  
Date: 31<sup>st</sup> October 2003

## **JBoss, JNDI and a little XDoclet: A starter**

This is a short tutorial on deployment descriptors used in JBoss for lookups and how to use XDoclet to help maintain your descriptors. The tutorial does assume a basic knowledge of Java 2 Platform Enterprise Edition (J2EE) coding, especially Enterprise Java Beans (EJB) and servlets and a knowledge of Ant build scripts. Along the way, we'll discuss the organisation of J2EE deployment descriptors, component name binding and the reasons for having the apparent complexity in J2EE component deployment. It is not intended to explain everything about XDoclet and its features nor is it intended to discuss other areas of the deployment descriptor.

### ***J2EE components***

There are two main component types in J2EE – EJBs and servlets. The important characteristics of EJB components are that they usually represent distinct units of work, can be widely distributed and are more loosely coupled with other components. Servlets form a presentation layer and help provide a session-oriented service that is based on HTTP. Servlets are more tightly-coupled as they usually chain sequences of actions together and are organised to interoperate in a specific manner. \*Stateful session beans as the name implies do offer capability for binding other EJB components into an activity sequence, however the session control lies above this layer. Any of the components mentioned can use resources available in the J2EE environment, including other components.

In line with theories of component re-use, EJB components should be less tightly bound with specific applications. This gives flexibility to a system by being able to substitute components within a specific implementation. This also prevents applications from being less monolithic and rigid.

J2EE components are also able to be deployed in any environment that complies with the J2EE environmental requirements. However, the environment may physically implement the location of the component or resources for the component in its own manner.

For reasons that will become apparent, the nature of EJB components should be kept in mind when we consider deployment descriptors.

### ***Component lifecycle***

In a J2EE environment, components have two distinct parts of the lifecycle - development of the component and the operational use of the component. The development engineer has the responsibility of building the component to operate within certain parameters. The deployment engineer has the responsibility of managing the binding of the component to its run-time environment, including defining the physical location of the resources the component will use. At the time of deployment, the deployment engineer can implement a specific combination of components, within the overall architectural dependencies defined by the application designer.

## Deployment descriptors

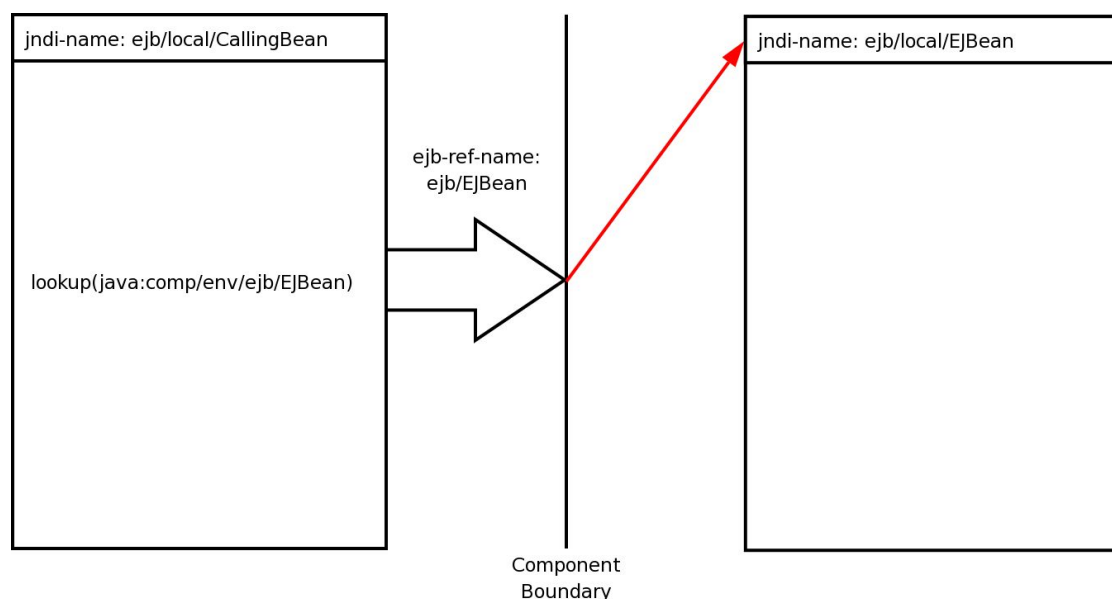
Deployment descriptors are an XML-based set of instructions that define all the necessary run-time parameters that controls what resources are available to a component, the directory entry for the component so it can be located by other components, and any security restrictions on accessing the component or any of its methods. The deployment engineer creates the deployment descriptor so the component can locate and make use of the resources available within the J2EE environment it resides.

In order to provide the flexibility to place J2EE components in differently implemented yet compliant J2EE environments, there is a separation between programming or logical references the developer uses for resources and the physical resources with which the deployment engineer works. An important role of the deployment descriptors is to map the logical references to the physical resources they represent.

The flexibility of mapping also avoids any naming clashes individual component builders may have with component calls. This allows vendors to use their own naming conventions without fear of creating problems when their components are deployed with another vendor's components in a target environment.

### A practical example

The following shows a J2EE component calling another J2EE component. The directory search uses an internally known name `ejb/EJBBean` – which is located in the component environment space `java:comp/env/ejb/EJBBean`.



The deployment descriptors provide a mapping between the resource name used by the component and the name used by the J2EE environment. The physical location of the EJB factory (home interface) is at `ejb/local/EJBBean`. We are using a notation of `ejb/local/EJBBean` to indicate this is a JNDI binding to the local home interface for the target EJB. The corresponding binding to the remote home interface would be `ejb/remote/EJBBean`. You can use any naming scheme you like but XDoclet imposes some predefined prefixing.

The `ejb-jar.xml` descriptor file has its content defined by the standard. The fragment that would be defined in the calling component for our example would look like this:

```
<ejb-local-ref >
  <ejb-ref-name>ejb/EJBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.amity.EJBeanLocalHome</local-home>
  <local>com.amity.EJBeanLocal</local>
</ejb-local-ref>
```

This defines the information the component will use internally. For your specific J2EE environment, you will usually require an additional deployment descriptor file to provide the other half of the remapping. In JBoss, this is the `jboss.xml` file. The corresponding `jboss.xml` fragment for the example is:

```
<ejb-local-ref>
  <ejb-ref-name>ejb/EJBean</ejb-ref-name>
  <local-jndi-name>ejb/local/EJBean</local-jndi-name>
</ejb-local-ref>
```

Note the linking is provided by the common reference, `ejb-ref-name`.

There are actually a few ways to provide the referencing but this provides the most explicit and non-packaging dependent manner. Some people prefer using the `link` element. However, this relies on the components being packed in the same deployment file. It removes some of the flexibility of deployment in that you have some assumptions about expected components being packed together. However, for some J2EE servers, look up and call processing may be optimized for this physical organization of components.

## ***JBoss descriptor generation via XDoclet***

XDoclet can help automate the creation of deployment descriptors and also the EJB interfaces. This is quite useful in cutting down the amount of work hand-building these for a JBoss deployment. However, there are also drawbacks with the XDoclet method. Since the descriptor configurations are tied to the source files, in essence this creates an early binding of references – descriptors are built at about the same time as the code, rather than at deployment time. For full automation, it means that much of the information on the packaging must be known at creation time of the compiled code. In order to help insulate yourself from these problems, you may find it better to use the `ejb-local-ref` and `ejb-ref` elements rather than the `link` element. It means that you can later cut and paste the descriptors later to suit your production packing.

XDoclet can also be used for creation of your `web.xml` for your web applications. This does not suffer from the same problems as the building of EJB components. Since web application components are more tightly bound together than EJB components, and EJB components and other resources are not packaged with web applications, there are not multiple methods for referencing components.

XDoclet can also generate simple descriptors for JMX services. However, since an MBean is often a datasource or connector related service the XDoclet limitation on generating multiple or repeated configurations to different underlying resources is a problem. For example, you would not be able to generate the following easily using XDoclet:

```
<mbean code="com.amity.flexcorp.objectcache.Loader"
      name="Amity:service=ObjectCache,name=TableCache">
  <attribute name="CacheName">TableCache</attribute>
  <attribute name="CacheClass">
    com.amity.flexcorp.objectcache.ObjectPool</attribute>
  <attribute name="Factory">
    com.amity.flexcorp.objectcache.Factory
  </attribute>
  <attribute name="CacheClass">
    com.amity.flexcorp.objectpool.EntityDefinition
  </attribute>
  <depends>
    jboss.jca:name=AmityPool,service=ManagedConnectionPool
  </depends>
</mbean>

<mbean code="com.amity.flexcorp.objectcache.Loader"
      name="Amity:service=ObjectCache,name=FormatCache">
  <attribute name="CacheName">FormatCache</attribute>
  <attribute name="CacheClass">
    com.amity.flexcorp.objectcache.FormatPool</attribute>
  <attribute name="Factory">
    com.amity.flexcorp.objectcache.Factory
  </attribute>
  <attribute name="CacheClass">
    com.amity.flexcorp.objectpool.FormatDefinition
  </attribute>
  <depends>
    jboss.jca:name=AmityPool,service=ManagedConnectionPool
  </depends>
</mbean>
```

XDoclet does not currently provide support JBoss WSR deployment generation. JBoss 3.2.x provides that support. The packaged module is located in JBOSS\_HOME/client and is called xdoclet-module-jboss-net.jar. As this document is intended to illustrate the JNDI and descriptor mapping we do not cover WSR deployments here.

XDoclet can help minimise the maintenance work for synchronizing the deployment descriptors with your component code for the majority of the JBoss deployments.

## Installing XDoclet

XDoclet is intended to work with your Ant build process and most Java-based IDEs do incorporate Ant. We also use Ant as we can rebuild components if necessary at customer-site, without the need for a full IDE. So you will need an installation of Ant and the ANT\_HOME environment variable set to your Ant installation directory prior to installing XDoclet.

You can then obtain the XDoclet source or the pre-built library. The current XDoclet libraries (1.2b4 and earlier) have a problem currently with supporting ejb-local-ref for JBoss. Until the patch has been accepted, or an alternative adopted, you will need to make the changes yourself. Read about the reason for the changes and how to make the changes at <http://www.amitysolutions.com.au/documents/XDocletChange-technote.pdf>

Either having downloaded the pre-built XDoclet distribution, or built your own from the source files, you can then take the XDoclet bin distribution and unpack it to your destination directory. On my Linux development laptop, this was /usr/local/xdoclet-1.2.b4. The bin sub-directory has some important build examples. The lib sub-directory has the important XDoclet libraries for generating the interface code and the deployment descriptors. The XDoclet project and documentation can be found at <http://xdoclet.sourceforge.net/>.

We use very standard build scripts for our components so for all EJBs, there is one sub-build script; for all web applications there is one sub-build script; and for all libraries there is one sub-build script. This reduces script maintenance substantially. These are all kept in a single configuration directory – denoted by \${config.dir}. We also took the XDoclet script, xdoclet.xml and placed it in here. It might be known as build.xml in your samples directory for the distribution. We modified the script to fit within our build process, using only specific Ant calls. Appendix A provides a sample of the XDoclet-related build script. The ejbdoclet task would be called from your main build script in the following ways. For EJBs:

```
<!-- Create the deployment descriptors from the embedded xDoclet -->
<target name="ejbdoclet" depends="init">
  <mkdir dir="${build.dir}/META-INF"/>
  <ant antfile="${config.dir}/xdoclet.xml" target="ejbdoclet"/>
  <mkdir dir="${dist.dir}/lib"/>
</target>
```

For WARs:

```
<!-- Create the deployment descriptors from the embedded xDoclet -->
<target name="webdoclet" depends="init">
  <mkdir dir="${build.dir}/WEB-INF"/>
  <ant antfile="${config.dir}/xdoclet.xml" target="webdoclet"/>
</target>
```

In this instance, the creation of the descriptors and the generated source files for any interfaces (for the EJBs) are created before the compilation phase of the build.

This covers the basics of the XDoclet installation and configuration. Read through the samples and the hints we provide in Appendix A to get an understanding on how to modify things for your build environment.

## Generating descriptors and source files

In order to create a simple EJB descriptor set for JBoss deployment, we can put the following in the Bean source CallingBean.java – this example creates no useful interface source as we specify existing interface source code.

```
/**
 * @ejb.bean                name="CallingBean"
 *                          description="A bean that uses some resources"
 *                          jndi-name="ejb/amity/remote/CallingBean"
 *                          local-jndi-name="ejb/amity/local/CallingBean"
 *                          type="Stateless"
 *                          transaction-type="Bean"
 *                          schema="EJB 2.0"
 *                          remote-business-interface="Calling"
 *                          local-business-interface="CallingLocal"
 * @ejb.transaction        type="Required"
 * @ejb.transaction-type   type="Container"
 *
 * @ejb.resource-ref       res-auth="Application"
 *                          res-name="jdbc/amity/AmityPool"
 *                          res-type="javax.sql.DataSource"
 * @jboss.resource-ref     res-ref-name="jdbc/amity/AmityPool"
 *                          jndi-name="java:/AmityPool"
 *
 * @ejb.resource-ref       res-auth="Application"
 *                          res-name="objc/amity/TableCache"
 *                          res-type="com.amity.objectcache.TableCache"
 * @jboss.resource-ref     res-ref-name="objc/amity/TableCache"
 *                          jndi-name="java:/TableCache"
 *
 * @ejb.ejb-external-ref   ref-name="ejb/amity/EJBean"
 *                          type="Session"
 *                          view-type="local"
 *                          home="com.amity.EJBLocalHome"
 *                          business="com.amity.EJBLocal"
 * @jboss.ejb-local-ref    ref-name="amity/EJBean"
 *                          jndi-name="ejb/amity/local/EJBean"
 *
 * @author    jbarnett
 * @version 1.0
 */
```

The first eleven lines relate to the generation of binding information for the EJB. Also note that this example makes no use of the generated classes as we define specific local and remote business interfaces in the tag set.

Refer to tutorials and EJB development books for more information on the contents and requirements for `ejb-jar.xml`. The next three groupings define the external resources this EJB will use and how the EJB will locate them. They define in order a JDBC datasource, a special object cache resource and another EJB.

The first resource is an object of type `javax.sql.DataSource` and this EJB can look up the DataSource (a connection factory) using the name `java:comp/env/jdbc/amity/AmityPool`, formed from the `res-name` element. The `java:comp/env` contains references available only in the components local definitions space. The `jboss.resource-ref` tags indicate the JBoss mappings for the resource and shows that the global JNDI name by which JBoss knows the resource is `java:/AmityPool`. This corresponds to a JBoss datasource definition, `amity-pool.xml` containing:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>AmityPool</jndi-name>
    <connection-url>jdbc:postgresql://poseidon:5432/amity</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>dbadmin</user-name>
    <password>password</password>
  </local-tx-datasource>
</datasources>
```

Note the JNDI name specified, as this corresponds to the reference we use. Similarly, a custom MBean resource, `TableCache` might be defined:

```
<server>
  <mbean code="com.amity.flexcorp.objectcache.Loader"
        name="Amity:service=ObjectCache,name=TableCache">
    <attribute name="CacheName">TableCache</attribute>
    <attribute name="CacheClass">
      com.amity.objectcache.ObjectPool</attribute>
    <attribute name="Factory">
      com.amity.flexcorp.objectcache.Factory
    </attribute>
    <attribute name="CacheClass">
      com.amity.objectpool.EntityDefinition
    </attribute>
    <depends>
      jboss:jca:name=AmityPool,service=ManagedConnectionPool
    </depends>
  </mbean>
</server>
```

The resource-ref tags show the JBoss mappings for TableCache and the global JNDI name by which JBoss knows the resource is java:/TableCache. The EJB coding can refer to the resource binding by the reference java:comp/env/objc/amity/TableCache.

Finally, for the EJB local interface reference we can use the lookup reference java:comp/env/ejb/amity/EJBean. This reference maps to ejb/amity/local/EJBean. You will notice that the EJB XDoclet notation is more complex than a resource reference. This is because we also need information on the type of reference we are obtaining – in this case, a reference to a local home object but it could have been to a (remote) interface. We also need to know the type of EJB and the classes for the home and EJB interfaces. Also note that XDoclet adds an ejb/ prefix to the JBoss ref-name so do not add this in the XDoclet tag. The ejb-ref-name for ejb-local-ref will be correctly created with ejb/amity/EJBean.

The ejb-jar.xml created from this XDoclet definition via the ejbdoclet task is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar >
  <description><![CDATA[Binary Retriever EJB]]></description>
  <display-name>Generated by XDoclet</display-name>
  <enterprise-beans>
    <!-- Session Beans -->
    <session >
      <description><![CDATA[A bean that uses some resources]]></description>
      <ejb-name>CallingBean</ejb-name>
      <home>com.amity.CallingHome</home>
      <remote>com.amity.Calling</remote>
      <local-home>com.amity.CallingLocalHome</local-home>
      <local>com.amity.CallingLocal</local>
      <ejb-class>com.amity.CallingBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <ejb-local-ref >
        <ejb-ref-name>ejb/amity/EJBean</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local-home>com.amity.EJBLocalHome</local-home>
        <local>com.amity.EJBLocal</local>
      </ejb-local-ref>
      <resource-ref >
        <res-ref-name>jdbc/amity/AmityPool</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Application</res-auth>
      </resource-ref>
    </session >
  </enterprise-beans>
</ejb-jar >
```

```

        <resource-ref >
            <res-ref-name>objc/amity/TableCache</res-ref-name>
            <res-type>com.amity.TableCache</res-type>
            <res-auth>Application</res-auth>
        </resource-ref>
    </session>
</enterprise-beans>
<!-- Assembly Descriptor -->
<assembly-descriptor >
<!-- transactions -->
<container-transaction >
    <method >
        <ejb-name>CallingBean</ejb-name>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
<ejb-client-jar>callingclient.jar</ejb-client-jar>
</ejb-jar>

```

Note that XDoclet also generates the DOCTYPE tags necessary. Similarly, the jboss.xml generated by XDoclet appears as:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.2//EN"
"http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
    <unauthenticated-principal>nobody</unauthenticated-principal>
    <enterprise-beans>
        <session>
            <ejb-name>CallingBean</ejb-name>
            <jndi-name>ejb/amity/remote/CallingBean</jndi-name>
            <local-jndi-name>ejb/amity/local/CallingBean</local-jndi-name>
            <ejb-local-ref>
                <ejb-ref-name>ejb/amity/EJBean</ejb-ref-name>
                <local-jndi-name>ejb/amity/local/EJBean</local-jndi-name>
            </ejb-local-ref>
            <resource-ref>
                <res-ref-name>jdbc/amity/AmityPool</res-ref-name>
                <jndi-name>java:/AmityPool</jndi-name>
            </resource-ref>
            <resource-ref>

```

```

        <res-ref-name>objc/amity/TableCache</res-ref-name>
        <jndi-name>java:/TableCache</jndi-name>
    </resource-ref>
    <method-attributes>
    </method-attributes>
</session>
</enterprise-beans>
<resource-managers>
</resource-managers>
</jboss>

```

These files have been edited for clarity as XDoclet generates a lot of comments and spacing lines.

The lookup code within CallingBean for the datasource, object cache and EJB resources respectively would be:

```

Context naming = new InitialContext();
DataSource amityDS = (DataSource)naming.lookup
("java:comp/env/jdbc/amity/AmityPool");
ObjectCache tableCache = (ObjectCache)naming.lookup
("java:comp/env/objc/amity/TableCache");
EJBLocalHome ejbHome = (EJBLocalHome)naming.lookup
("java:comp/env/ejb/amity/EJBean");

```

For EJB components, you can check whether the JNDI resolutions are complete by using the JNDIView list function within the JBoss JMX-Console web application.

With your component coding you could use the direct JNDI names in your calls.

```

Context naming = new InitialContext();
DataSource amityDS = (DataSource)naming.lookup("java:/AmityPool");
ObjectCache tableCache = (ObjectCache)naming.lookup("java:/TableCache");
EJBLocalHome ejbHome = (EJBLocalHome)naming.lookup
("ejb/amity/local/EJBean");

```

However, this allows for no flexibility should you have a JNDI naming clash or should the J2EE deployment environment otherwise prevent you from binding the resources to the required physical JNDI location. Using the internal component reference allows the deployment engineer to move the physical JNDI location of a resource to suit the J2EE environment without compromising the operation of applications and components residing in the environment.

For web applications, you place your XDoclet tags into the relevant servlet source code. For each servlet, insert the necessary servlet configuration information for that servlet alone, and the resources that the servlet may use. Where the resource may be used more than once, only add the resource declaration in one of the servlets – probably the major servlet for ease of location.

For example, in one of our applications, the help topic system is used by many servlets so we declare the XDoclet tags in the MainServlet source code.

The webdoclet task generates the necessary deployment descriptors for your web application.

```
/**
 * Main servlet that generates a list of information
 *
 * @web.servlet          name="MainServlet"
 *                      display-name="Important list servlet"
 *                      description="The main servlet for lists"
 * @web.servlet-mapping url-pattern="/Main"
 *
 * @web.ejb-local-ref   name="ejb/amity/CallingBean"
 *                      type="Session"
 *                      home="com.amity.CallingLocalHome"
 *                      local="com.amity.CallingLocal"
 * @jboss.ejb-local-ref ref-name="amity/CallingBean"
 *                      jndi-name="ejb/amity/local/CallingBean"
 *
 * @web.resource-ref    res-auth="Application"
 *                      res-name="jdbc/amity/AmityPool"
 *                      res-type="javax.sql.DataSource"
 * @jboss.resource-ref  res-ref-name="jdbc/amity/AmityPool"
 *                      jndi-name="java:/AmityPool"
 *
 * @author  jbarnett
 * @version 1.0
 */
```

XDoclet generates the following web.xml from this single servlet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app >
  <display-name>Main</display-name>
  <description><![CDATA[App WAR]]></description>
  <servlet>
    <servlet-name>MainServlet</servlet-name>
    <display-name>Important list servlet </display-name>
    <description><![CDATA[TThe main servlet for lists ]]></description>
    <servlet-class>com.amity.MainServlet</servlet-class>
```

```

</servlet>
<servlet-mapping>
  <servlet-name>MainServlet</servlet-name>
  <url-pattern>/Main</url-pattern>
</servlet-mapping>
<resource-ref >
  <res-ref-name>jdbc/amity/AmityPool</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
<ejb-local-ref>
  <ejb-ref-name>ejb/amity/CallingBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.amity.CallingLocalHome</local-home>
  <local>com.amity.CallingLocal</local>
</ejb-local-ref>
</web-app>

```

Similarly, XDoclet creates a jboss.xml deployment descriptor.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.3//EN"
"http://www.jboss.org/j2ee/dtd/jboss-web_3_2.dtd">
<jboss-web>
  <!-- Resource Environment References -->
  <!-- Resource references -->
  <resource-ref>
    <res-ref-name>jdbc/amity/AmityPool</res-ref-name>
    <jndi-name>java:/AmityPool</jndi-name>
  </resource-ref>
  <!-- EJB References -->
  <!-- EJB Local References -->
  <ejb-local-ref>
    <ejb-ref-name>ejb/amity/CallingBean</ejb-ref-name>
    <local-jndi-name>ejb/amity/local/CallingBean</local-jndi-name>
  </ejb-local-ref>
</jboss-web>

```

The webdoclet task also generates a struts-config file and a taglib file but we omit these as they have no bearing on our example.

## Remote interfaces

XDoclet also supports calls to remote interfaces of EJBs. For EJB to EJB calls, the necessary XDoclet fragment would consist of:

```
* @ejb.ejb-external-ref    ref-name="ejb/amity/EJBean"
*                           type="Session"
*                           view-type="remote"
*                           home="com.amity.EJBHome"
*                           business="com.amity.EJBLocal"
* @jboss.ejb-ref-jndi      ref-name="amity/EJBean"
*                           jndi-name="ejb/amity/remote/EJBean"
```

The ejb-jar.xml fragment that is generated looks like this.

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

The fragment from jboss.xml is:

```
<ejb-ref>
  <ejb-ref-name>ejb/amity/EJBean</ejb-ref-name>
  <jndi-name>ejb/amity/remote/EJBean</jndi-name>
</ejb-ref>
```

Note that your lookup code will change as you are doing a remote call even if the EJB is located in the same application server instance, running in the same Java Virtual Machine (JVM). This does have a performance penalty although JBoss performs some optimizations if it determines the target EJB resides in the same JVM.

```
Context naming = new InitialContext();
Object ref = naming.lookup("java:comp/env/ejb/amity/EJBean");
EJBHome ejbHome = (EJBHome)PortableRemoteObject.narrow(ref, EJBHome.class);
```

Similarly, for a servlet calling an EJB, the following XDoclet fragment is required:

```
* @web.ejb-ref            name="ejb/amity/CallingBean"
*                           type="Session"
*                           home="com.amity.CallingHome"
*                           remote="com.amity.Calling"
* @jboss.ejb-ref-jndi      ref-name="ejb/amity/CallingBean"
*                           jndi-name="ejb/amity/remote/CallingBean"
```

The ejb-ref-jndi for web.xml requires the ejb/ prefix and departs from the standard convention.

The generated web.xml fragment is:

```
<ejb-ref >
    <ejb-ref-name>ejb/amity/CallingBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.amity.CallingHome</home>
    <remote>com.amity.Calling</remote>
</ejb-ref>
```

The corresponding jboss-web.xml fragment is:

```
<ejb-ref>
    <ejb-ref-name>ejb/amity/CallingBean</ejb-ref-name>
    <jndi-name>ejb/amity/local/CallingBean</jndi-name>
</ejb-ref>
```

## **Monolithic development**

The current line of Java Integrated Development Environments (IDE) can induce developers to think of J2EE applications in a monolithic and inflexible way. Although the J2EE development process has been designed to allow late physical binding of components to the J2EE environment, the IDEs try to make development, deployment and testing of applications easier by automating packaging and this leads to early binding. This tends to encourage consideration of the tight coupling of components, rather than thinking of components as independent units, and considering the re-use or sharing of components in multiple applications within an operational environment.

We also feel that the extended use of Enterprise Archives (EAR), especially during the development phase is detrimental to component-based thinking and is counter-productive for rapid deployment and test cycles. We have encountered many developers who build very large EARs and complain about the time to deploy a system for testing. Although the tools may provide deployment options that take out the thinking required for packaging, we believe that developers should understand the deployment build process and the trade-off for the choices made in the process.

XDoclet, combined with a less IDE-automated packaging process can encourage a more component focused approach to J2EE development, while still automating the deployment descriptor generation. This walk through of the process is intended to demonstrate the mechanics of the J2EE deployment process, how component discovery is accomplished and how XDoclet tags are composed to generate the discovery information. The resultant deployment descriptors can be used by the deployment engineer to create the production deployment descriptors and will be less affected by how the production code is grouped and packaged.

There are other ways to achieve component discovery, notably through the use of the link element. This element usually implies a physical proximity between components and we believe it should be used sparingly. However, there may be reasons of performance, or implementation of designs with tight coupling of very specialised components that may render the use of the link element acceptable. Such considerations are left as the responsibility of the J2EE design engineer.

## Appendix A

An example of xdoclet.xml for EJB and WAR deployments. Much of the non-JBoss tasks have been removed.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="XDoclet" default="jar" basedir=".">
  <!-- Properties to run xdoclet -->
  <property file="${config.dir}/build.properties"/>
  <property file="${config.dir}/xdoclet.properties"/>
  <!-- Linux environment configuration -->
  <property environment="env"/>
  <!-- ===== -->
  <!-- Define the class path -->
  <!-- ===== -->
  <path id="xdoclet.class.path">
    <fileset dir="${jboss-client.dir}">
      <include name="jboss-j2ee.jar"/>
      <include name="servlet.jar"/>
    </fileset>
    <fileset dir="${xdoclet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
  </path>
  <!-- ===== -->
  <!-- Initialise -->
  <!-- ===== -->
  <target name="init">
    <tstamp>
      <format property="TODAY" pattern="d-MM-yy"/>
    </tstamp>
    <taskdef
      name="xdoclet"
      classname="xdoclet.DocletTask"
      classpathref="xdoclet.class.path"
    />
    <taskdef
      name="ejbdoclet"
      classname="xdoclet.modules.ejb.EjbDocletTask"
      classpathref="xdoclet.class.path"
    />
    <taskdef
      name="webdoclet"
```

```

        classname="xdoclet.modules.web.WebDocletTask"
        classpathref="xdoclet.class.path"
    />
</target>
<!-- ===== -->
<!-- Prepares the directory structure -->
<!-- ===== -->
<target name="prepare" depends="init">
    <mkdir dir="${generated-src.dir}"/>
</target>
<!-- ===== -->
<!-- Invoke XDoclet's ejbdoclet -->
<!-- ===== -->
<target name="ejbdoclet" depends="prepare">
    <echo>+-----+</echo>
    <echo>|                                     |</echo>
    <echo>| R U N N I N G   E J B D O C L E T   |</echo>
    <echo>|                                     |</echo>
    <echo>+-----+</echo>
    <mkdir dir="${meta-inf.dir}"/>
    <ejbdoclet
        destdir="${generated-src.dir}"
        mergedir="parent-fake-to-debug"
        excludedtags="@version,@author,@todo"
        addedtags="@xdoclet-generated at ${TODAY},@copyright The XDoclet
Team,@author XDoclet and ${env.USER},@version ${version}"
        ejbspec="2.0"
        force="${samples.xdoclet.force}"
        verbose="false"
    >
        <fileset dir="${src.dir}">
            <include name="${ejb.dir}/*Bean.java"/>
        </fileset>
        <packageSubstitution packages="ejb"
substituteWith="interfaces"/>
        <remoteinterface/>
        <localinterface/>
        <homeinterface/>
        <localhomeinterface/>
        <dataobject/>
        <valueobject/>
        <entitypk/>

```

```

<entitycmp/>
<entitybmp/>
<!--<session/>-->
<dao>
    <packageSubstitution packages="ejb" substituteWith="dao"/>
</dao>
<utilobject cacheHomes="true" includeGUID="true"/>
<deploymentdescriptor
    destdir="${meta-inf.dir}"
    validatexml="true"
    mergedir="fake-to-debug"
    description="${local.descriptor}"
    >
    <configParam name="clientjar" value="${project.name}
client.jar"/>
</deploymentdescriptor>
<jboss
    version="3.2"
    unauthenticatedPrincipal="nobody"
    xmlencoding="UTF-8"
    destdir="${meta-inf.dir}"
    validatexml="true"
    preferredrelationmapping="relation-table"
/>
</ejbdoclet>
</target>
<!-- =====>
<!-- Invoke XDoclet's webdoclet -->
<!-- =====>
<target name="webdoclet" depends="prepare">
    <echo>+-----+</echo>
    <echo>| |</echo>
    <echo>| R U N N I N G   W E B D O C L E T |</echo>
    <echo>| |</echo>
    <echo>+-----+</echo>
    <mkdir dir="${web-inf.dir}"/>
    <webdoclet
        destdir="${generated-src.lib.dir}"
        mergedir="parent-fake-to-debug"
        excludedtags="@version,@author,@todo"
        addedtags="@xdoclet-generated at ${TODAY},@copyright The XDoclet
Team,@author XDoclet and ${env.USER},@version ${version}"

```

```

force="${samples.xdoclet.force}"
verbose="false"
>
<fileset dir="${src.dir}">
    <include name="${web.dir}/*Servlet.java"/>
    <include name="${web.dir}/*Filter.java"/>
    <include name="${web.dir}/*Tag.java"/>
    <include name="${web.dir}/*Action.java"/>
</fileset>
<deploymentdescriptor
    servletspec="2.3"
    destdir="${web-inf.dir}"
    >
    <configParam name="Displayname" value="${project.name}"/>
    <configParam name="Description" value="${local.
descriptor}"/>
    <configParam name="Distributable" value="false"/>
</deploymentdescriptor>

<strutsconfigxml
    destdir="${web-inf.dir}"
    />
<jsptaglib
    jspversion="1.2"
    destdir="${web-inf.dir}"
    shortname="j2ee"
    validateXml="false"
    />
<jbosswebxml version="3.2" destdir="${web-inf.dir}"/>
</webdoclet>
</target>
<!-- ===== -->
<!-- Clean -->
<!-- ===== -->
<target name="clean">
    <delete dir="${samples.dist.dir}"/>
</target>
</project>

```

The xdoclet.properties file is:

```
# xDoclet Configuration
# NAMING PROPERTIES
Name=XDoclet
name=xdoclet
version=1.2b4
packages=${name}.*
# xDoclet location
xdoclet.root.dir = /usr/local/xdoclet-${version}
xdoclet.lib.dir = ${xdoclet.root.dir}/lib
# JAR PROPERTIES
xdoclet.jar = ${xdoclet.lib.dir}/${name}-${version}.jar
commons-logging.jar = ${xdoclet.lib.dir}/commons-logging.jar
commons-collections.jar = ${xdoclet.lib.dir}/commons-collections-2.0.jar
log4j.jar = ${xdoclet.lib.dir}/log4j.jar
ant.jar = ${ant.home}/lib/ant.jar
mockobjects-core.jar = ${xdoclet.lib.dir}/mockobjects-core-0.07.jar
junit.jar = ${xdoclet.lib.dir}/junit.jar
```

You will find it necessary to modify various property definitions so that the generation works properly. `{generated-src}` is the most important as this is where the interface files for an EJB deployment are created. You can have these generated into your primary source directory for compilation with your main Bean source. Should you use the generated interfaces, you will need to remove the references to local and remote business interfaces in the `@ejb.bean` tag otherwise the generation will try to extend these interfaces and will ignore any tags related to `@ejb.interface-method`.

The ejbdoclet task has some expectations for its class and source naming scheme.

The source file with the Bean code is `/fullpath/{Name}Bean.java`.

The remote and home interfaces will be:

`/fullpath/{Name}.java` and `/fullpath/{Name}Home.java` respectively.

The local and local-home interfaces will be:

`/fullpath/{Name}Local.java` and `/fullpath/{Name}LocalHome.java` respectively.

It will generate the descriptors accordingly. This was fine for our systems as we already build using these naming conventions.

For the webdoclet, the expectation is that the servlet descriptor instructions are contained in source code `{Name}Servlet.java`, `{Name}Filter.java`, `{Name}Tag.java` and `{Name}Action.java`. However, you can modify this and the EJB processing to your own requirements.

The deployment descriptors are generated directly into the correct location in the build directory – META-INF and WEB-INF respectively for EJB and WAR builds.