Client JNDI Names

What's My Bean's JNDI Name?

There are a few things to keep in mind before you start reading:

1. A default JNDI name is provided to your EJB.
2. You can customize the JNDI name.
3. If the name is taken, an error will be logged and deployment will continue.

Set the **openejb.jndiname.failoncollision** system property to 'true' if you'd like a strict guarantee the jndi names you want are created.

Default JNDI name

The default JNDI name is in the following format:

```
{ejbName}{interfaceType.annotationName}
```

Lets try and understand the above format. Both **ejbName** and **interfaceType.annotationName** are pre-defined variables. There are other pre-defined variables available which you could use to customize the JNDI name format.

JNDI Name Formatting

The **openejb.jndiname.format** property allows you to supply a template for the global JNDI names of all your EJBs. With it, you have complete control over the structure of the JNDI layout can institute a design pattern just right for your client apps. See the Service Locator doc for clever ways to use the JNDI name formatting functionality in client code.

variable	description
moduleId	Typically the name of the ejb-jar file or the <ejb-jar id=""> id value if specified
ejbType	STATEFUL, STATELESS, BMP_ENTITY, CMP_ENTITY, or MESSAGE_DRIVEN
ejbClass	for a class named *org.acme.superfun.WidgetBean* results in **org.acme.superfun.WidgetBean**
ejbClass.simpleName	for a class named *org.acme.superfun.WidgetBean* results in **WidgetBean**
ejbClass.packageName	for a class named *org.acme.superfun.WidgetBean* results in **org.acme.superfun**
ejbName	The ejb-name as specified in xml or via the 'name' attribute in an @Stateful, @Stateless, or @MessageDriven annotation
deploymentId	The unique system id for the ejb. Typically the ejbName unless specified in the openejb-jar.xml or via changing the openejb.deploymentId.format
interfaceType	see interfaceType.annotationName
interfaceType.annotationName	Following the EJB 3 annotations @RemoteHome, @LocalHome, @Remote and @Local • **RemoteHome** *(EJB 2 EJBHome)* • **LocalHome** *(EJB 2 EJBLocalHome)* • **Remote** *(EJB 3 Business Remote)* • **Local** *(EJB 3 Business Local)* • **Endpoint** *(EJB webservice endpoint)*
interfaceType.xmlName	Following the ejb-jar.xml descriptor elements <home>, <local-home>, <business-remote>, <business-local>, and <service-endpoint>: • **home** *(EJB 2 EJBHome)* • **local-home** *(EJB 2 EJBLocalHome)* • **business-remote** *(EJB 3 Business Remote)* • **business-local** *(EJB 3 Business Local)* • **service-endpoint** *(EJB webservice endpoint)*
interfaceType.xmlNameCc	Camel-case version of interfaceType.xmlName: • **Home** *(EJB 2 EJBHome)* • **LocalHome** *(EJB 2 EJBLocalHome)* • **BusinessRemote** *(EJB 3 Business Remote)* • **BusinessLocal** *(EJB 3 Business Local)* • **ServiceEndpoint** *(EJB webservice endpoint)*

interfaceType. openejbLegacyName	Following the OpenEJB 1.0 hard-coded format: • *(empty string) (EJB 2 EJBHome)* • **Local** *(EJB 2 EJBLocalHome)* • **BusinessRemote** *(EJB 3 Business Remote)* • **BusinessLocal** *(EJB 3 Business Local)* • **ServiceEndpoint** *(EJB webservice endpoint)*
interfaceClass	• *(business)* for a class named *org.acme.superfun.WidgetRemote* results in **org.acme.superfun.WidgetRemote** • *(home)* for a class named *org.acme.superfun.WidgetHome* results in **org.acme.superfun.WidgetHome**
interfaceClass.simpleName	• *(business)* for a class named *org.acme.superfun.WidgetRemote* results in **WidgetRemote** • *(home)* for a class named *org.acme.superfun.WidgetHome* results in **WidgetHome**
interfaceClass.packageName	for a class named *org.acme.superfun.WidgetRemote* results in **org.acme.superfun**

This can be set on a server level via a *system property*, for example:

TODO

Show the recommended way to set system properties or this property (if a gbean attribute is added)

Setting the JNDI name

It's possible to set the desired jndi name format for the whole server level, an ejb, an ejb's "local" interface (local/remote/local-home/home), and for an individual interface the ejb implements. More specific jndi name formats act as an override to any more general formats. The most specific format dictates the jndi name that will be used for any given interface of an ejb. It's possible to specify a general format for your server, override it at an ejb level and override that further for a specific interface of that ejb.

Via the <jndi> tag for a specific ejb

The following sets the name specifically for the interface org.superbiz.Foo.

```
<openejb-jar>
  ...

  <session>
    ...

    <jndi name="foo" interface="org.superbiz.Foo"/>
  </session>
</openejb-jar>
```

Or more generally...

```
<openejb-jar>
  ...

  <session>
    ...

    <jndi name="foo" interface="Remote"/>
  </session>
</openejb-jar>
```

Or more generally still...

```
<openejb-jar>
  ...

  <session>
    ...

    <jndi name="foo"/>
  </session>
</openejb-jar>
```

The 'name' attribute can still use templates if it likes, such as:

```
<openejb-jar>
  ...

  <session>
    ...

    <jndi name="ejb/{interfaceClass.simpleName}" interface="org.superbiz.Foo"/>
  </session>
</openejb-jar>
```

Multiple <jndi> tags

Multiple <jndi> tags are allowed making it possible for you to be as specific as you need about the jndi name of each interface or each logical group of iterfaces (Local, Remote, LocalHome, RemoteHome).

Given an ejb, FooBean, with the following interfaces:

- business-local: org.superbiz.LocalOne
- business-local: org.superbiz.LocalTwo
- business-remote: org.superbiz.RemoteOne
- business-remote: org.superbiz.RemoteTwo
- home: org.superbiz.FooHome
- local-home: org.superbiz.FooLocalHome

The following four examples would yield the same jndi names. The intention with these examples is to show the various ways you can isolate specific interfaces or types of interfaces to gain more specific control on how they are named.

#1

```
<openejb-jar>
  ...

  <session>
    ...

    <jndi name="LocalOne" interface="org.superbiz.LocalOne"/>
    <jndi name="LocalTwo" interface="org.superbiz.LocalTwo"/>
    <jndi name="RemoteOne" interface="org.superbiz.RemoteOne"/>
    <jndi name="RemoteTwo" interface="org.superbiz.RemoteTwo"/>
    <jndi name="FooHome" interface="org.superbiz.FooHome"/>
    <jndi name="FooLocalHome" interface="org.superbiz.FooLocalHome"/>
  </session>
</openejb-jar>
```

#2

```
<openejb-jar>
  ...

  <session>
    ...

    <!-- applies to LocalOne and LocalTwo -->
    <jndi name="{interfaceClass.simpleName}" interface="Local"/>

    <!-- applies to RemoteOne and RemoteTwo -->
    <jndi name="{interfaceClass.simpleName}" interface="Remote"/>

    <!-- applies to FooHome -->
    <jndi name="{interfaceClass.simpleName}" interface="RemoteHome"/>

    <!-- applies to FooLocalHome -->
    <jndi name="{interfaceClass.simpleName}" interface="LocalHome"/>
  </session>
</openejb-jar>
```

#3

```
<openejb-jar>
  ...

  <session>
    ...

    <!-- applies to RemoteOne, RemoteTwo, FooHome, and FooLocalHome -->
    <jndi name="{interfaceClass.simpleName}"/>

    <!-- these two would count as an override on the above format -->
    <jndi name="LocalOne" interface="org.superbiz.LocalOne"/>
    <jndi name="LocalTwo" interface="org.superbiz.LocalTwo"/>
  </session>
</openejb-jar>
```

#4

```
<openejb-jar>
  ...

  <session>
    ...

    <!-- applies to LocalOne, LocalTwo, RemoteOne, RemoteTwo, FooHome, and FooLocalHome -->
    <jndi name="{interfaceClass.simpleName}"/>
  </session>
</openejb-jar>
```

Global Settings

You can change the JNDI setttings by specifying the GERONIMO_OPTS before lanuching the command `geronimo start`. For example, to change the JNDI format into *{ejbName}/{interfaceClass}*, use this syntax:

```
set GERONIMO_OPTS="-Dopenejb.jndiname.format={ejbName}/{interfaceClass}
```

You are responsible for ensuring the names don't conflict.

Conservative settings

A very conservative setting such as "*{deploymentId}/{interfaceClass}*" would guarantee that each and every single interface is bound to JNDI. If your bean had a legacy EJBObject interface, three business remote interfaces, and two business local interfaces, this pattern would result in **six** proxies bound into JNDI.

Similarly conservative settings would be:

- *{deploymentId}/{interfaceClass.simpleName}*
- *{moduleId}/{ejbName}/{interfaceClass}*
- *{ejbName}/{interfaceClass}*
- *{moduleId}/{ejbClass}/{interfaceClass}*
- *{ejbClass}/{interfaceClass}*
- *{ejbClass}/{interfaceClass.simpleName}*
- *{ejbClass.simpleName}/{interfaceClass.simpleName}*
- *{interfaceClass}/{ejbName}*

Bordeline optimistic:

- *{moduleId}/{interfaceClass}*
- *{interfaceClass}*

The above two settings would work if the interface wasn't shared by other beans.

Pragmatic settings

A more middle ground setting such as "*{deploymentId}{interfaceType.annotationName}*" would guarantee that at least one proxy of each interface type is bound to JNDI. If your bean had a legacy EJBObject interface, three business remote interfaces, and two business local interfaces, this pattern would result in **three** proxies bound into JNDI: one proxy dedicated to your EJBObject interface; one proxy implementing all three business remote interfaces; one proxy implementing the two business local interfaces.

Similarly pragmatic settings would be:

- *{moduleId}/{ejbClass}/{interfaceType.annotationName}*
- *{ejbClass}/{interfaceType.xmlName}*
- *{ejbClass.simpleName}/{interfaceType.xmlNameCc}*
- *{interfaceType}/{ejbName}*
- *{interfaceType}/{ejbClass}*

Optimistic settings

A very optimistic setting such as "*{deploymentId}*" would guarantee only one proxy for the bean will be bound to JNDI. This would be fine if you knew you only had one type of interface in your beans. For example, only business remote interfaces, or only business local interfaces, or only an EJBObject interface, or only an EJBLocalObject interface.

If a bean in the app did have more than one interface type, one business local and one business remote for example, by default OpenEJB will reject the app when it detects that it cannot bind the second interface. This strict behavior can be disabled by setting the **openejb.jndiname.failoncollision** system property to *false*. When this property is set to false, we will simply log an error that the second proxy cannot be bound to JNDI, tell you which ejb is using that name, and continue loading your app.

Similarly optimistic settings would be:

- *{ejbName}*
- *{ejbClass}*
- *{ejbClass.simpleName}*
- *{moduleId}/{ejbClass.simpleName}*
- *{moduleId}/{ejbName}*

Advanced Details on EJB 3.0 Business Proxies (the simple part)

If you implement your business interfaces, your life is simple as your proxies will also implement your business interfaces of the same type. Meaning any proxy OpenEJB creates for a business local interface will also implement your other business local interfaces. Similarly, any proxy OpenEJB creates for a business remote interface will also implement your other business remote interfaces.

Advanced Details on EJB 3.0 Business Proxies (the complicated part)

Read this section of either of these two apply to you:

- You do not implement your business interfaces in your bean class
- One or more of your business remote interfaces extend from javax.rmi.Remote

If neither of these two items describe your apps, then there is no need to read further. Go have fun.

Not implementing business interfaces

If you do not implement your business interfaces it may not be possible for us to implement all your business interfaces in a single interface. Conflicts in the throws clauses and the return values can occur as detailed here. When creating a proxy for an interface we will detect and remove any other business interfaces that would conflict with the main interface.

Business interfaces extending javax.rmi.Remote

Per spec rules many runtime exceptions (container or connection related) are thrown from javax.rmi.Remote proxies as java.rmi.RemoteException which is not a runtime exception and must be throwable via the proxy as a checked exception. The issue is that conflicting throws clauses are actually removed for two interfaces sharing the same method signature. For example two methods such as these:

- InterfaceA: void doIt() throws Foo;
- InterfaceB: void doIt() throws RemoteException;

can be implemented by trimming out the conflicting throws clauses as follows:

- Implementation: void doIt(){}

This is fine for a bean class as it does not need to throw the RMI required javax.rmi.RemoteException. However if we create a proxy from these two interfaces it will also wind up with a 'doIt(){}' method that cannot throw javax.rmi.RemoteException. This is very bad as the container does need to throw RemoteException to any business interfaces extending java.rmi.Remote for any container related issues or connection issues. If the container attempts to throw a RemoteException from the proxies 'doIt(){}' method, it will result in an UndeclaredThrowableException thrown by the VM.

The only way to guarantee the proxy has the 'doIt() throws RemoteException {}' method of InterfaceB is to cut out InterfaceA when we create the proxy dedicated to InterfaceB.