The role of JNDI in J2EE

Decoupling yourself from trouble

Kirk Pepperdine
Joseph Ottinger

January 13, 2005

Mastering J2EE can be daunting, with an ever-growing list of technologies and acronyms. The Java Naming and Directory Interface (JNDI) has been at the core of the Java 2 Platform, Enterprise Edition (J2EE) from its inception, but it is often underutilized by novice J2EE developers. This article will help demystify the role of JNDI in J2EE applications and show how it can help decouple your application from the details of deployment.

Inasmuch as the J2EE platform has improved the life of the average enterprise developer, that improvement has come at the cost of having to learn about the many specifications and technologies that J2EE has incorporated in its bid to become an all-inclusive distributed computing platform. Dolly Developer is but one of many developers that has yet to discover one feature that helps ease the burden that accompanies the deployment of any enterprisewide application: JNDI, the Java Naming and Directory Interface. Let's see how Dolly manages without JNDI and how proper use of JNDI improves her situation.

An all-too-familiar journey

Dolly Developer is coding a Web application that uses JDBC data sources. She knows that she's using MySQL, so she encodes a reference to the MySQL JDBC driver class and connects to the database in her Web application by using the appropriate JDBC URL. She recognizes that database connection pooling is important, so she includes a connection pooling package and configures it to use no more than 64 connections; she knows the database server has been set up to allocate 128 client connections.

Dolly heads for disaster

Everything goes smoothly during the development phase. On deployment, however, things start to unravel. Her network administrator tells her that she doesn't have access to the production or staging servers from her desktop, so she has to create a different version of the code for each stage of deployment. Because of this situation, she needs a new JDBC URL and, hence, a separate deployable for testing, staging, and production. (People familiar with configuration
management are cringing at the notion of deploying separate builds into each environment, but since this seems to be a fairly common situation, maybe they're cringing less than they should be.)

Just when Dolly thought she had "solved" her configuration problems by spinning separate deployables with different URLs, she finds that her database administrator doesn't want to run a MySQL instance in production. It's fine for development, he says, but the business standard for mission-critical data is DB2®. Now her builds differ not only in their database URLs but will need to have different drivers, too.

It gets worse. Her application is so useful and becomes so critical, that it gets failover capability from the application server and is replicated into a four-server cluster. But the database administrator raises a red flag, as every instance of her application uses 64 connections, and the database server as a whole has only 200 available connections -- all being contended for by Dolly's application. Furthermore, the DBA has determined that Dolly's application requires only 32 connections, and only during a single one-hour period during the day. As her application scales up, the application ends up with contention issues at the database layer, and her only option is to change the number of clustered connections and prepare to do so again if the cluster grows or her application gets replicated in another cluster. It would seem that she has made decisions about the application's configuration that would have been best left to the system and database administrators.

J2EE roles

Dolly could have avoided this dilemma if she had developed her application with knowledge of the J2EE roles. The J2EE specification delegates responsibilities into numerous development roles: Component Provider, Application Assembler, Deployer, and System Administrator. (In many organizations, the Component Provider and Assembler roles are merged, as are the Deployer and System Administrator roles.) Before JNDI's role in J2EE can be truly understood, it's important to grasp what the J2EE roles do.

**Component Provider**

This role is responsible for creating J2EE components, which can be Web applications, Enterprise JavaBean (EJB) components, or application clients (such as a Swing-based GUI client application). Component Providers include HTML content designers, document programmers, and other developer roles. Most J2EE developers spend quite a bit of time in the Component Provider role.

**Application Assembler**

This role ties multiple J2EE modules into a cohesive, deployable whole: the enterprise archive (EAR) file. The Application Assembler selects components, identifies how they will interact, configures their security and transactional attributes, and packages the application into an EAR file. Many IDEs, such as WebSphere® Studio, IDEA, JBuilder, WebLogic Workshop, and others, have features to assist the Application Assembler with interactive configuration of EAR files.

**Deployer**

This role is responsible for deployment, which means installing an EAR into a J2EE container (an application server), configuring resources such as database connection pools, binding
resources required by the application to specific resources in the application server, and starting up the application.

System Administrator
This role is responsible for making sure the resources the container needs are available to the container.

Roles in action
Imagine an enterprise application that contains a single Web application and a single EJB component for business logic and persistence. Developing this application might involve a number of Component Providers, although in many cases, the same person fulfills all of the duties. The components could include data transfer objects (a JAR file), the EJB interface (another JAR file), the EJB implementation itself (yet another JAR file), and the user interface components -- servlets, JSPs, HTML pages, and other static Web content. The user interface components are further packaged into a Web application, which contains servlet classes, JSP files, static content, and the JARs containing other required components, including the EJB interfaces.

Though it may sound like a lot of components to break out, it's hardly out of the scope of reason, especially when you consider how many JAR files are used to build a typical Web application. It's important to realize that dependencies must be carefully managed here. The interfaces and transfer objects are allowable dependencies for the Web application and the EJB implementation, but the lines of dependencies should all run in the same direction; cyclic dependencies are to be avoided. J2EE components, such as WAR files and EJB JAR files, must declare their dependence on resources outside their deployment units.

The Application Assembler is responsible for the inclusion of dependencies in the Web application and the packaging of the whole into a single enterprise application. Tools help a lot here. IDEs can help create a project structure that reflects the dependencies of modules and JARs, and allow you to specify inclusion or exclusion of modules at will.

The Deployer is responsible for ensuring that resources required by the components exist in the deployment environment and binding them to the platform's available resources. For example, an external EJB reference (an ejb-ref in the deployment descriptor) in a Web application is tied to an actual deployed EJB component at this point -- and no sooner.

Late binding to external resources
Any nontrivial J2EE application is going to require access to information that describes the environment in which it is expected to function. This means that developing and testing components will require that the developer take on some of the deployment duties, if only temporarily for the purposes of testing the code. It's important to understand that by doing this, you are stepping outside of the developer domain. Otherwise, the temptation is to put in reliance on a JDBC driver, or a URL, a JMS queue name, or other machine resources with unintended, and occasionally disastrous, implications.

JNDI to the rescue
The solution to Dolly's problem is to remove all direct references to the data store from her application code. No references to JDBC drivers, no server names, no user names or passwords
not even database pooling or connection management. Dolly needs to write her code to be ignorant of what specific external resources it is going to access, with the understanding that others will provide the links it needs to utilize those external resources. This will allow the deployer (whoever is in that role) to allocate database connections to her application, without Dolly having to be involved. (There are good business reasons for this, too, ranging from data security to Sarbanes-Oxley compliance.)

Many developers know that tight coupling between code and external resources is potentially a problem, but often ignore the separation of roles in practice. In small development efforts (in terms of team size or deployments), ignorance of the separation of roles can be successful enough. (After all, it's fine to lock your application into a specific PostgreSQL instance when it's only your personal recipe application, and you're not planning on relying on it.)

The J2EE specification requires that all J2EE containers provide an implementation of the JNDI specification. The role of JNDI in J2EE is to be the "switchboard" -- a generic mechanism for J2EE components to find other components, resources, or services indirectly at run time. In most cases, the container-supplied JNDI provider can serve as a limited data store so an administrator can set up execution properties in one application, and have other applications reference them (Java Management Extensions (JMX) can also be used for this purpose). The primary role of JNDI in a J2EE application is to provide the indirection layer so that components can find required resources without being much aware of the indirection.

Dolly moves to calmer waters

Let's revisit Dolly's situation. In her simple Web application, she used a JDBC connection directly from her application code. With an examination of Listing 1, we can see that she has explicitly coded the name of the JDBC driver, the database URL, and her user name and password into her servlet:

Listing 1. Typical (but not good) JDBC usage

```
Connection conn=null;
try {
    Class.forName("com.mysql.jdbc.Driver",
        true, Thread.currentThread().getContextClassLoader());
    conn=DriverManager.getConnection("jdbc:mysql://dbserver?user=dolly&password=dagger");
    /* use the connection here */
    conn.close();
} catch(Exception e) {
    e.printStackTrace();
}
finally {
    if(conn!=null) {
        try {
            conn.close();
        } catch(SQLException e) {}
    }
}
```

Instead of specifying the configuration information in this manner, Dolly (and her co-workers) would be better served by using JNDI to find a JDBC dataSource, as shown in Listing 2:
To be able to acquire a JDBC connection, we first need to perform some minor deployment configuration so we can look up a JDBC DataSource in the local component's JNDI context. This can be a bit of a chore, but it's easy to learn. Unfortunately, it means that to even test a component, the developer has to wade into the Deployer's realm, and be prepared to configure the application server.

**Configuring the JNDI reference**

In order for JNDI to resolve the java:comp/env/jdbc/mydatasource reference, the Deployer is required to insert a `<resource-ref>` tag into the web.xml file (the deployment descriptor for the Web application.) The `<resource-ref>` tag is a way of saying, "This component has a dependency on an external resource." Listing 3 shows an example:

### Listing 3. A resource-ref entry

```xml
<resource-ref>
  <description>Dolly's DataSource</description>
  <res-ref-name>jdbc/mydatasource</res-ref-name>
  <res-ref-type>javax.sql.DataSource</res-ref-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

The `<resource-ref>` entry informs the servlet container that a resource in the component naming context, called jdbc/mydatasource, will be set up by the Deployer. The component naming context is indicated by the prefix java:comp/env/, so the fully qualified local resource name is java:comp/env/jdbc/mydatasource.

That defines only the local reference to the external resource, and doesn't create the actual resource that this reference will point to. (A Java language analogue might be that the `<resource-ref>` declares a reference, such as Object foo, but doesn't set `foo` to actually reference any Object.)

The Deployer's job is to create a DataSource (or, in our Java language example, create an Object for `foo` to point to). Each container has its own mechanism for setting up data sources. In
JBoss, for example, a data source is defined with a service (see $JBOSS/server/default/deploy/hsqldb-ds.xml for an example), which specifies that it is a global JNDI name for the `DataSource` (by default, `DefaultDS`). Once the resource has been created, there is still a critical third step: to connect, or bind, the resource to the local name(s) used by the application components. In the case of a Web application, a vendor-specific deployment descriptor extension is used to specify this binding, an example of which is shown in Listing 4. (JBoss uses a file called `jboss-web.xml` for the vendor-specific Web application deployment descriptor.)

**Listing 4. Binding a resource to a JNDI name in the vendor-specific deployment descriptor**

```xml
<resource-ref>
  <res-ref-name>jdbc/mydatasource</res-ref-name>
  <jndi-name>java:DefaultDS</jndi-name>
</resource-ref>
```

This says that the local resource ref name (`jdbc/mydatasource`) should be mapped to the global resource named `java:DefaultDS`. If the global resource name changes for any reason, the application code doesn't change -- only this mapping. There are two levels of indirection here -- one to define and name the resource (`java:DefaultDS`), the other to bind the local component-specific name (`jdbc/mydatasource`) to the named resource. (In fact, there is the possibility for a third level of indirection as you can map resources at the EAR level, as well.)

**Moving past DataSource**

Of course, resources in J2EE aren't limited to JDBC data sources. There are several types of references, including resource references (discussed already), environment entries, and EJB references. EJB references, in particular, expose another key role for JNDI in J2EE: finding other application components.

Consider what happens when a company purchases a deployable EJB component to process customer orders, from Order Ontology Processing Services (OOPS). For the sake of example, we'll call it ProcessOrders V1.0. ProcessOrders 1.0 comes in two pieces: a set of interfaces and support classes (the home and remote interfaces, and supporting transfer objects), and the actual EJB components, themselves. OOPS was chosen because of its expertise in this arena.

The company follows the J2EE specification and writes a Web application that uses an EJB reference. Its Deployer binds ProcessOrders 1.0 into the JNDI tree as `ejb/ProcessOrders/1.0`, and resolves the Web application's resource name to point to that global JNDI name. This is all very normal use of an EJB component so far. However, it gets more complicated when we consider the interaction between the company's development cycle and that of its suppliers. Again, JNDI can help us here, too.

Let's assume OOPS releases a new version, ProcessOrders V1.1. This new version has some functionality that a new application internal to the company needs and, naturally, extends the business interface for its EJB components.
The company has a few choices here: It can update all of its applications to use the new version, it can write its own, or it can use JNDI's reference resolution to allow each application to use its own version of the EJB component, without affecting any other application. Updating all applications at once would be a maintenance nightmare, requiring a full regression test of all components, which is often a huge burden, and another round of debugging if any functional tests fail.

Writing in-house components is often an unnecessary duplication of work. If the component is written by a company that has expertise in that business area, it's not likely that a given IT shop will manage to master the business functionality, as well as the specialized component provider.

As you could probably have guessed, the best solution is to use JNDI resolution. EJB JNDI references are very much like the JDBC resource references. For each reference, the Deployer will bind the new component into the global tree at a specific name (say, `ejb/ProcessOrders/1.1`), and for every other component that requires the EJB component, resolve the EJB reference in the deployment descriptor for that component. The older applications, which rely on V1.0, require no change and no retesting, lowering the implementation time, cost, and complexity. In environments where services tend to be versioned, this is a powerful approach. This sort of configuration management can be done for all acquirable components in the application architecture, from EJB components to JMS queues and topics to simple configuration strings or other objects, lowering maintenance costs as services change over time, and easing deployment and integration efforts.

**Conclusion**

There is an old computer science joke that says that every programming problem can be resolved with just one more layer of abstraction (or indirection). In J2EE, JNDI is the glue that holds J2EE applications together, but not so tightly that they can't easily be taken apart and reassembled. It is the indirection provided by JNDI that enables the delivery of scalable, capable, yet flexible applications across the enterprise. That's the promise of J2EE, and it is fully realizable with some planning and forethought. In fact, it's easier than many people think.
Related topics

- Learn more about JNDI from the The JNDI Tutorial from Sun Microsystems.
- "Using JNDI Outside J2EE," an excerpt from Teach Yourself J2EE in 21 Days (Sams Publishing, 2004), shows you how to define the type of name service you're using, and how to define the host name and port number of the server on your network.
- developerWorks columnist Brian Goetz discusses using JNDI with J2SE for application configuration in his article "J2EE or J2SE? JNDI works with both" (JavaWorld, April 2002).
- Using JNDI with the WebSphere Application Server J2EE Thin Application Client shows how JNDI can be used in J2EE application clients to communicate with a remote JNDI provider.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)