Lotus Domino Rnext/5.x

COMPONENTS

Implement and Call EJBs with Domino and JBoss

By John Duggan

What are EJBs? Does Domino support them? Why should I need to know about them? If you want the answers, read on!

Rnext includes Tomcat, which means Domino will support JSP (JavaServer Pages) and servlets. However, Domino doesn't provide support for EJBs. There are no plans to implement EJBs within Domino, and the recommendation from Lotus is to use WebSphere Application Server (WAS).

Subscriber Download

The complete listing for the EJB container.
http://Advisor.com/Article/DUGGJ05

Implementing WAS has many advantages, but it's a complex and expensive solution if you're only going to use it for EJBs. Fortunately, there is a viable alternative: A product

called JBoss. This article shows you how to use JBoss so you can use EJBs with Domino. First, though, a little terminology.

J2EE

Java 2 Platform, Enterprise Edition (J2EE) is a collection of APIs that includes:

Enterprise JavaBeans (EJB)

Servlets—Server-side components to dynamically generate Web pages.

JavaServer Pages (JSP)—HTML pages containing Java scripts.

Java Database Connectivity (JDBC)—Connections to relational databases.

Java Naming and Directory Interface (JNDI)—Directory service for finding objects.

JavaMail—Mail API.

Java Transaction API (JTA)—Lets components manage their own transactions.

Java Messaging Service (JMS)—A mechanism for components to send messages asynchronously.

The purpose of J2EE is to define a standard for developing multi-tier enterprise applications (figure 1).

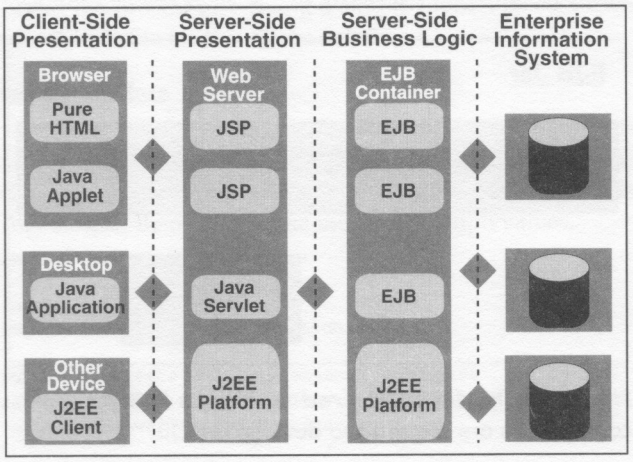

***Figure 1:* The multi-tier structure of J2EE**—A component-based architecture for enterprise development. (Source: www.java.sun.com)

Sun owns J2EE, currently on version 2. A number of application servers from a range of vendors are J2EE-compliant, such as BEA, Oracle, Sun, and IBM. You can find a list of these servers (along with reviews) at: http://www.theserverside.com/reviews/index.jsp. At the core of J2EE is the EJB technology.

What is an EJB?

As defined by Sun, an Enterprise JavaBean is a server-side component that encapsulates the business logic of an application. EJB 2.0 is a specification, not a product; thus any server that complies with the specification can run compiled EJBs.

Notice in figure 1 there is a business logic layer and a presentation layer. EJBs let you write multi-platform server-side business logic for the J2EE platform.

EJB container

An EJB container is an EJB server. It's responsible for loading, activating, and maintaining the lifecycle of an EJB. An EJB can't function outside an EJB container. Figure 2 shows the traditional setup of a J2EE server and the role of an EJB container.

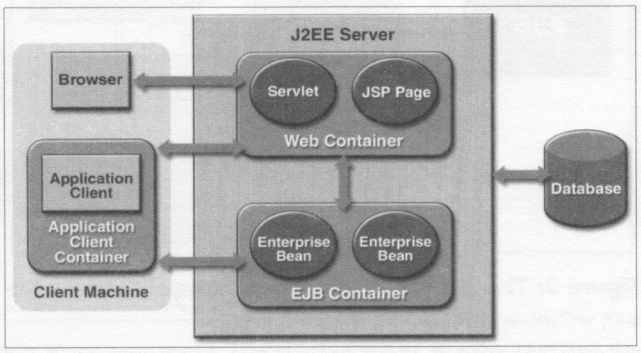

***Figure 2:* Traditional J2EE server**—The components used within a J2EE-compliant server. (Source: www.java.sun.com)

The EJB container looks after the security, transactions, persistence, concurrency, and resource management of an EJB. The purpose is to develop EJBs faster, because you don't have to have in-depth knowledge about distributed objects, transactions, or other enterprise systems. You can just concentrate on the business logic.

It's important for you to be aware that beans and EJBs are two distinct technologies. Beans are used for visual components whereas EJBs are non-visual components running within an EJB container.

You can store the business logic within servlets, but servlets are predominately used for generating dynamic Web pages. The purpose of a multi-tier structure is to separate the business logic from the presentation logic, leaving servlets involved with the presentation and EJBs handling the business logic. You can't access servlets as distributed objects, nor do they support transactions implicitly.

As you can see in figure 1, this means another developer can concentrate on the presentation level, without having to know the business logic.

Different types of Enterprise JavaBeans

There are three types of EJBs (figure 3).

Session beans

A session bean performs a task for a client. A session bean may be stateful or stateless, and only lives as long as the session of the client's code calling the bean. You use session beans to model the business logic.

Stateful session beans are dedicated to the client that created it. For example, in an e-commerce application, it would be a shopping cart dedicated to a particular user.

Stateless session beans aren't dedicated to a single client but can be used by many clients. In an e-commerce application, it may perform the financial calculations or the credit card processing.

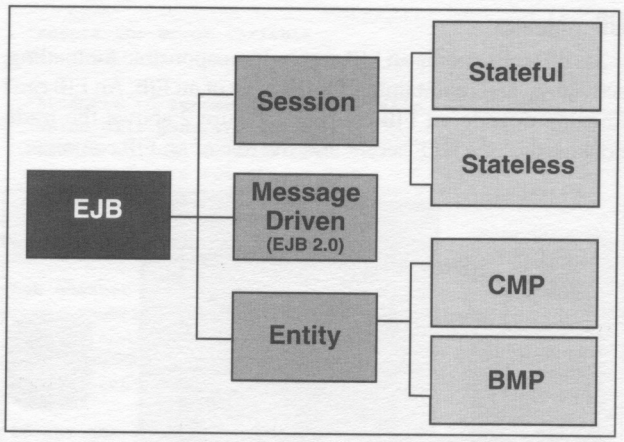

***Figure 3:* Three EJBs**—The different types of EJBs you can use within an EJB container.

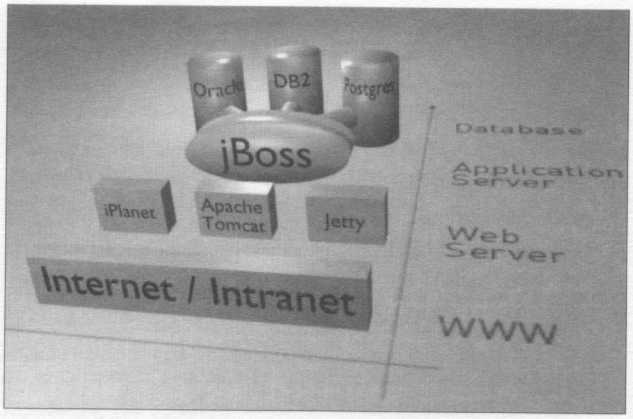

***Figure 4:* The role of JBoss**—Here's how JBoss fits into an IT environment. Domino would be located at the Web server layer. (Source: www.jboss.org)

Entity beans

Entity beans are persistent, transactional, server-side components. Entity beans model the business data. In an e-commerce application, it models orders or products. There are two types of entity beans:

1. Bean-managed persistence (BMP) beans are responsible for managing their own relationships with a database (or persistence state). In other words, you write the database access logic code.
2. Container-managed persistence (CMP) beans leave the management of its persistent state and relationships to the EJB container. In other words, the EJB container handles all the database access logic.

Message-driven

This is a new addition to the EJB 2.0 specification. They are stateless, server-side components used to receive and process messages that have been sent to them via JMS. These are mostly used to integrate an EJB system with a legacy system, or to enable business-to-business interactions.

JBoss

JBoss (http://www.jboss.org) is a free, open source J2EE-based implementation. JBoss with Tomcat provides a popular and low cost J2EE solution.

Even though JBoss isn't a complete J2EE environment in its own right, the reviews on http://www.theserverside.com rate the product highly against application servers with considerably more functionality. Figure 4 shows the relationship of JBoss within an IT environment. Rnext would be located where Tomcat is placed.

Installing JBoss is straightforward:

1. Install the latest version of the Java 2 Runtime Environment, if you don't already have it installed (http://java.sun.com/j2se/1.3/jre).
2. Download and uncompress the latest version of JBoss (http://www.jboss.org/binary.jsp).
3. Run the file "run.bat" within the bin directory of JBoss.

That's it! Your EJB container is now running and ready to process any requests.

Now let's put together a simple EJB in JBoss. But before you do that, you have to create an EJB jar.

Creating a session bean

An enterprise bean consists of four components (figure 5).

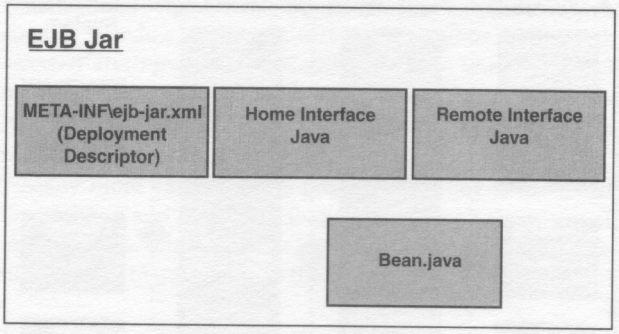

***Figure 5:* The files required within an EJB jar**—These components are required to develop an EJB.

1. The home interface contains the methods used by Java clients to create new enterprise beans, locate existing beans, and destroy beans.
2. The remote interface contains the business method of the bean, used by Java clients at runtime.
3. The bean class contains the application logic and the business method defined in the remote interface.
4. The deployment descriptor is an XML configuration file that describes the enterprise bean and its runtime attributes when it's deployed. Figure 6 shows how the parts fit together.

The shaded boxes are the responsibility of the EJB developer. The EJB client is any Java application that activates and uses an EJB, which could be an agent, servlet, or JSP.

As shown in the diagram, Java Naming and Directory Interface (JNDI) is a directory server that lets Java programs look up objects or data by name. You'll find the required EJB

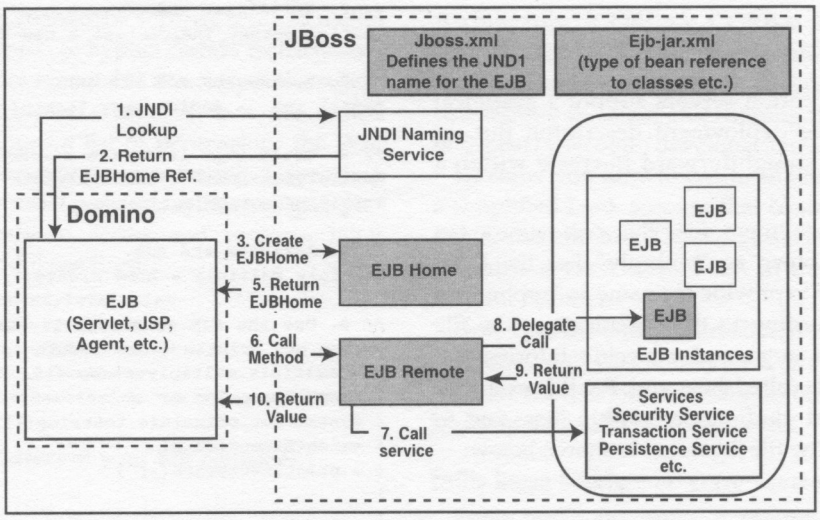

***Figure 6:* How it fits together**—The relationship between an EJB client and JBoss. The shaded boxes are the sections the EJB developer must implement.

by passing a name to the JNDI server, which returns a reference to the object.

As an example, you'll create a stateless session EJB that will multiply two values. It isn't a useful bean, but it demonstrates the process of creating a simple EJB. I'll discuss each source code file for this EJB in detail.

Remote interface

The remote interface is the class that exposes the business methods to the outside world. The EJB client accesses this class to call the business methods. Because this example is very simple, there is only one method: multiplyvalues.

```
package com.domino.multiply;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Multiply extends EJBObject
{ public double multiplyvalues(double value1, double
value2) throws RemoteException; }
```

Bean class

The bean class contains the code for the multiplyvalues method:

```
package com.domino.multiply;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class MultiplyBean implements SessionBean
{
 public double multiplyvalues(double value1, double
value2)
 { return value1 * value2; }

 public MultiplyBean() { }
 public void ejbCreate() { }
 public void ejbRemove() { }
 public void ejbActivate() { }
 public void ejbPassivate() { }
 public void setSessionContext(SessionContext sc) { }
}
```

In particular, notice there are a lot of empty methods. These methods are used by the EJB container to activate or remove EJBs. You can use them as "hooks." In other words, you can add a System.out.println statement to each method and watch when the EJB container calls each method for the EJB. Although the methods are blank, the EJB specification requires them to be defined.

Home interface

This interface activates the EJB when it's called by an EJB client:

```
package com.domino.multiply;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface MultiplyHome extends EJBHome
{ Multiply create() throws RemoteException,
CreateException; }
```

All beans require the home and remote interface except message-driven beans.

Deployment descriptor

You must create a deployment descriptor file 'ejb-jar.xml,' and it must be in the directory called /META-INF. This file defines the name of the EJB, what type of bean it is, and where the classes are located.

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar>
 <description>A sample stateless EJB for jBoss
 </description>
<display-name>Stateless Multiply</display-name>
   <enterprise-beans>
    <session>
     <ejb-name>Multiply</ejb-name>
     <home>com.domino.multiply.MultiplyHome</home>
     <remote>com.domino.multiply.Multiply</remote>
     <ejb-class>com.domino.multiply.MultiplyBean
     </ejb-class>
     <session-type>Stateless</session-type>
     <transaction-type>Bean</transaction-type>
```

```
    </session>
  </enterprise-beans>
</ejb-jar>
```

Commercial EJB application servers supply a graphical tool for constructing the deployment descriptor. But, as you can see, it is fairly straightforward to create within a text editor.

According to the deployment file, the <ejb-name> tag states that the EJB is known as "Multiply." For the JNDI lookup, it's good practice to provide the name as "application name/bean name." According to the specification, the file ejb-jar.xml can't contain any vendor-specific information; thus you have to use a file called jboss.xml. For this example, you can use a tag called <jndi-name> within jboss.xml to override the EJB name. The file jboss.xml is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
 <enterprise-beans>
 <session>
  <ejb-name>Multiply</ejb-name>
  <jndi-name>demo/Multiply</jndi-name>
</session>
 </enterprise-beans>
</jboss>
```

Assuming you have the latest Java Standard Edition SDK (http://java.sun.com/j2se/1.3) installed, you can compile the classes by issuing the following command (don't forget to add the jboss classes to the classpath):

```
javac -classpath c:\jboss\client\jboss-j2ee.jar com\
domino\multiply\*.java
```

After you've compiled the classes, you can create the .JAR file:

```
jar cvf multiply.jar com\domino\multiply\*.class META-INF
```

Deploying the EJB .JAR package to JBoss is easy because it supports "hot deployment." Copy the .JAR file into the directory jboss\deploy, watch the messages on the JBoss server, and that's it! If you want to un-deploy the EJB from JBoss, delete the file.

Write an agent to access the EJB

You've created the EJB .JAR and deployed it within JBoss. How do you call it using an agent in Rnext? You can access it using this code:

```
import lotus.domino.*;
import java.util.Properties;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import com.domino.multiply.*;

public class JavaAgent extends AgentBase {

public void NotesMain() {

try {
Session session = getSession();
AgentContext agentContext = session.getAgentContext();

// 1. Set up the IP address of the EJB container and
// the context factory to use.
Properties props = new Properties();
props.put("java.naming.factory.initial",
   "org.jnp.interfaces.NamingContextFactory");
props.put("java.naming.provider.url","127.0.0.1:1099");

// 2. Initialize the context.
InitialContext jndiContext = new InitialContext(props);

// 3. Lookup the EJB via JNDI.
Object ref  = jndiContext.lookup("demo/Multiply");

// 4. Check that it is the correct object.
MultiplyHome home = (MultiplyHome)
PortableRemoteObject.narrow (ref, MultiplyHome.class);

// 5. Activate the EJB.
Multiply multiply = home.create();

// 6. Use the EJB as though it was an ordinary object!
System.out.println ("The result is: " _
  + multiply.multiplyvalues (10, 10));
} catch(Exception e)
{ System.out.println(e.toString()); }
} catch(Exception e)
{ e.printStackTrace(); }
}
}
```

Unfortunately, R5 runs an older version of Java and can't perform the lookup without amending the standard Java RMI classes. Running the agent code for Rnext requires you to add the following line to Notes.ini (change the directories to your installation of JBoss!):

```
JavaUserClasses=c:\jboss\client\jboss-j2ee.jar;
c:\jboss\client\jnp-client.jar; c:\jboss\client
\jbosssx-client.jar; c:\classes\multiply.jar;
```

The code tells Notes where to find the required packages for compiling and running the agent. Alternatively, copy the files into the directory jvm/lib/ext.

Use a servlet to access the EJB via the browser

You can use the same code shown above within a servlet:

```
public class SampleServlet extends HttpServlet {

public void doGet(HttpServletRequest request,
HttpServletResponse response) throws IOException,
ServletException
{
 Session session = null;
PrintWriter out = response.getWriter();

 response.setContentType("text/html");


 Properties props = new Properties();
props.put("java.naming.factory.initial", _
  "org.jnp.interfaces.NamingContextFactory");
props.put("java.naming.provider.url","127.0.0.1:1099");
InitialContext Context = new InitialContext(props);
Object ref  = Context.lookup("demo/Multiply");
 MultiplyHome home = (MultiplyHome)
 PortableRemoteObject.narrow (ref, MultiplyHome.class);
 Multiply remote = home.create();

 out.println ("The result is: " + _
  remote.multiplyvalues(10, 10));
 out.close();
}
```

The servlet code shown above is condensed and isn't recommended for a live environment. For example, in a live environment, you should use the servlet's init event to initialize the EJB.

The difference between the servlet and the agent is security. Running a servlet means it is running under Tomcat, not

under Notes. You have to change the security settings within the JVM to give it permission to bypass certain restrictions.

Located within the Notes executable directory is the subdirectory jvm\lib\security, and within it is a file called java.policy. This file contains a list of permissions the JVM can perform without a security error occurring. Depending on your existing security settings, you may have to give permission for the EJB code to access and perform JNDI lookups. For example, add these lines:

```
permission java.util.PropertyPermission
"java.naming.factory.initial", "write";
permission java.util.PropertyPermission
"java.naming.provider.url", "write";
permission java.net.SocketPermission "xx.xx.xx.xx:8083",
"connect,resolve";
```

(xx.xx.xx.xx is the IP address of your EJB container.)

Access the EJB via JSP

You can use the same code via JSP, but you should use tag libraries. JSPs were intended to have a minimal amount of Java code, whereas servlets have a minimal amount of presentation code. For example, if you use a customized tag library, the JSP looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">
<%@ page contentType="text/html; charset=UTF-8"
import="lotus.domino.*,java.util.*" %>
<%@ taglib uri="MultiplyEJBTags.jar" prefix="ejb" %>
<%@ taglib uri="domtags.tld" prefix="domino" %>
<%@ taglib uri="domutil.tld" prefix="util" %>
<ejb:useMultiplyBean/>
<% multiply = multiplyHome.create(); %>

<html>
<body>
 Result is: <%= multiply.multiplyvalues(10, 10);%>

</body>
</html>
```

Notice there is little Java code within the JSP. This makes it easier for Web designers to update and maintain pages without having to know Java. They only have to know a few tags to call the EJB.

Creating an entity bean

The previous bean was a simplified example of a session bean, so let's write a bean that does something more useful—such as a BMP Entity bean that uses a Notes database.

As mentioned earlier in the article, there are two types of entity beans: bean-managed persistence (BMP) and container-managed persistence (CMP). BMP requires the data access to be supplied within the bean, whereas CMP's data access is managed by the EJB container.

CMP beans usually work with JDBC drivers to connect to the data source. Unfortunately, the JDBC driver supplied by Lotus to access Notes databases (http://www.lotus.com/home.nsf/welcome/jdbc) isn't a JDBC-compliant driver. This means, for now, you can't develop a CMP Notes bean without a lot of inconvenience.

Instead, you can use BMP and access Notes data via a Domino R5 server using Common Object Request Broker Architecture (CORBA). CORBA is a standard that lets you access distributed objects programmatically. For this example, I assume you have the Lotus Java/CORBA toolkit successfully installed and running (http://www.lotus.com/home.nsf/welcome/developernetwork).

I'll show you how to create an entity bean that represents a simplified bank account that contains an account number, name, and amount. The Notes database is simple because it contains a view with a list of names, amount, and the account number.

Note that the entity bean shown in this example has been simplified and isn't suited for a live environment. It doesn't support transactions or rollback, but it is a good source for learning the basics of entity beans.

Entity home interface

Entity beans are similar to session beans but require additional methods to support the management of data. The bean, as defined by the EJB specification, must provide a method that serves as a primary key lookup. This method contains the information that uniquely identifies an account within your Notes database. The home interface for the sample entity bean is defined as:

```
public interface AccountHome extends EJBHome
{
  // Create a new account
public Customer create(final String id, final String
name) throws CreateException, RemoteException;
// Find an account
  public Customer findByPrimaryKey(final String id)
throws FinderException, RemoteException;
}
```

Entity remote interface

The remote interface is straightforward and contains all the business logic functions of retrieving and setting the account details:

```
public interface Account extends EJBObject
{
  public String getAccountNo() throws RemoteException;
  public String getName() throws RemoteException;
  public String setName() throws RemoteException;
  public String getAmount() throws RemoteException;
  public String setAmount() throws RemoteException;
}
```

Entity bean

You must supply a number of methods to let the EJB container access Notes data. The most important, required by the EJB specification, is the FindByPrimaryKey method. It provides a means of finding the required data—in this case, an account. (The sample download has another class called findByNameContaining, which finds all accounts containing a particular name, but it isn't outlined in this article.)

The method is straightforward: Connect to a Domino server, look for an account with the requested ID, and, if it's found, return the ID. Otherwise, raise an error.

```
public String ejbFindByPrimaryKey(final String id)
throws FinderException
{
  // 1. Connect to Domino via CORBA.
  Database db = getDominoConnection();
```

```
    // 2. Get the view with a list of accounts.
    View view = db.getView("Index");

    // 3. Find the requested account.
    Document doc = view.getDocumentByKey(id,true);

// If it exists, return the ID, otherwise raise
// an error saying the account does not exist.
    if (doc!=null)
      return id;
    else
      throw new ObjectNotFoundException
        ("No Customer with id=" + id);
}
```

The EJB container requires a number of other functions, and they all follow the same pattern. For example, connect to Domino and perform the relevant tasks via CORBA. You should look at the complete listing, which you can find in the sample database available for download from ADVISOR.COM.

To comply with the remote interface, you have to implement the the business logic functions. They are straightforward in that you can amend or return the values of the entity bean class's private variables. These values are populated when the ejbCreate or ejbLoad events are called.

```
public String getId()
  { return id; }
public String getName()
  { return name; }
public void setName(final String name)
  { this.name = name; }
public String getAmount()
  { return amount; }
public void setAmount(final String amount)
  { this.amount = amount; }
```

Use the entity bean

The process of accessing the bean is the same as accessing a session bean—by performing a JNDI lookup for the Account object:

```
Context c = new InitialContext();
Object o = c.lookup ("Account");
AccountHome h = (AccountHome) PortableRemoteObject.narrow
(o, AccountHome.class);
```

After the object is retrieved, you can ask the EJB container to get a specified account and amend or delete its details:

```
// 1. Finding customer with account number 1
Account account = h.findByPrimaryKey ("1");

// 2. Deleting the customer.
account.remove();

// 3. Create new account with ID '1'
Account account1 = h.create ("1", "joe bloggs", "100");

// 4. Retrieve the details of the account.
System.out.println ("Account No:" + account1.getId());
System.out.println ("Name: " + account1.getName());
System.out.println ("Amount: " + account1.getAmount());
```

EJB issues

Every tool has its place, and good design is the key to the success of a development project. EJB is another tool you should use only in the right circumstances. There are many implementations where money and resources have been wasted when a simple alternative solution would have been sufficient.

EJBs provide transactions, security, pooling, etc. without you having to know all the details of how these services work, but there is a cost.

If the application you're developing requires these services, EJBs are a good fit. They also offer scalability, so if the application has lots of users, they can get to their data efficiently.

It's possible to provide identical solutions with just JSP and servlets. If you want to deliver data from a database into Web applications—without the need for automatic transactions or pooling within your application code—then EJBs are probably excessive for your requirements.

The best way to decide whether to use EJBs for an application is to look at what EJBs achieve and ask yourself whether you require any of those features. Some of the features are:

- Concurrent read, update access to multiple shared enterprise data sources.
- A standard, portable, component-based architecture for multiple client types.
- Support for components that will work on different application servers by different vendors.

Can Rnext handle this type of performance with JBoss? I don't think anybody knows at the moment, because it hasn't been tested on a large scale. But where JBoss does fit is providing EJBs to a number of applications, not just Rnext.

As you can see, using and calling EJBs is straightforward. Developing the EJB is the complex part. Sun assumes that, for the most part, you won't be developing EJBs but using and calling components, built by third-party developers, from your own code. You can find some of these components at http://industry.java.sun.com/solutions/browse.

If you're looking for further information about EJBs, check out http://www.jguru.com. If you're looking for optimization tips, try http://www.precisejava.com. (This site also provides optimization tips for servlets and JSPs.) Finally, a good resource for all things EJBs and J2EE application servers is http://www.theserverside.com.

Wrap up

In a nutshell, EJBs provide a server-side component environment defined by a pre-written infrastructure: the EJB specification.

WebSphere Application Server and Domino are complementary technologies, but for most Domino "workshops" using WebSphere Application Server just for EJBs is a bit excessive when there is a stable, free, open source alternative: JBoss. JBoss can provide services to a variety of applications within your environment.

There is a problem with EJBs in that there is a lot of online information with too many "experts" offering differing opinions of what, where, and when EJBs should or shouldn't be used. Too many experts with different opinions on the same topic always cause confusion. Be warned: When looking for EJB information, you will encounter a lot of hot air! Like they say, "the truth is out there ... all you need is a better search engine!" ■