Enterprise Java and Rational Rose - Part II

by Khawar Ahmed
Technical Marketing Engineer
Rational Software
Loïc Julien
Software Engineer
Rational Software

This is the second installment of a two-part series exploring the synergistic relationship between Java 2 Enterprise Edition (J2EE), a popular development platform for distributed enterprise applications, and Rational Rose, the industry's leading CASE modeling tool. Part I provided an introduction to J2EE from an architectural perspective. In Part II, we'll first explain how Servlets and Java Server Pages (JSPs) work within the J2EE architecture and then go on to show how Rational Rose can help developers build better J2EE applications.

What Is a Servlet?

As we saw in Part I, an enterprise application may (but does not necessarily) consist of several tiers. Tiers are primarily abstractions to help us understand the architecture. The J2EE architecture usually involves four distinct tiers, as shown in Figure 1.
Servlets reside on the Web tier and typically act as the intermediary between the client tier (end users) and the other tiers (e.g., the business or the EIS tier).

A servlet is really a Java class that is invoked in response to a user's command. The invocation of the servlet follows a procedural request/response process. The Servlet 2.2 specification introduces the notion of a Web Application: the component that is deployed within a servlet container. A Web Application is packaged as a file with the extension ".war" in a format similar to that of the Java Archive (.jar) file.

You've probably used or seen Web sites that use CGI scripts to create an HTML page on the fly. That is exactly the kind of functionality servlets provide, but they are much lighter weight. Unlike CGI scripts, servlets don't require a new process for each HTTP request. All they need is a thread. Plus, they are much easier to maintain.

When a servlet container starts up, it typically creates a pool of servlets. Once the servlets are initialized, they are ready to process requests. For an idea of how servlets work, let's say you visit a Web site that has a servlet for processing registration requests. When you fill out a registration form, and press "Submit," the servlet receives the request and extracts the information you provided in the form via a "request object." (The request object is created by the container and provides an object-based access to the form data.) Then the servlet does whatever it needs to do (save the information in a database, for example, or ask other objects such as EJBs to do so) and then composes an HTML page via the "response object." The page is then sent back to you, perhaps displaying a "registration successful" message and confirmation number.

By definition, a servlet is stateless. If you want a "stateful" servlet, then
you can use either the HttpSession object or the Servlet Context object. The HttpSession object is dedicated to a client session; the Servlet Context object lasts for the life of the application.

If you are familiar with the pre-J2EE servlet specification, you might be wondering where the servlet engine fits into the Servlet 2.2 architecture. As it turns out, the servlet engine is now called the servlet container, which we mentioned earlier. The new specification also introduces a few new concepts, such as an application that can be distributed within several containers.

Although servlets are wonderfully fast, lightweight, and powerful, servlet-based development suffers from one very fundamental flaw. The specification states that

\[ \ldots \textit{the servlet must contain the presentation aspects as well.} \]

This requirement can be difficult to implement and even more difficult to maintain, since the servlet developer and the presentation creators are, more often than not, different people. The presentation is embedded inside Java statements, which makes it difficult to comprehend. Moreover, the presentation is likely to change more frequently than the logic behind it.

So what's a developer to do? Use JavaServer Pages, or JSPs.

**JSPs: Stepping into the Breach**

Built on top of the Servlet architecture, the JSP technology allows for a clear separation between presentation logic and business logic. It provides all the power of servlet technology while allowing the servlet developer and the content developer to work independently.

JSPs are great for creating dynamic content. They excel at rendering dynamic HTML/XML documents and provide an easy way to associate template data (XML or HTML) with directives and actions that supply dynamic behaviors on Web pages. These behaviors can be tied to specific requests or entirely independent of any request.

Basically, the JSP technology is a mix of HTML code and Java bits of logic (called scriptlets). A JSP gets compiled dynamically into a servlet at runtime, essentially yielding servlet capabilities without the drawbacks.

Figure 2 shows a sample JSP.
The J2EE and the Model-View-Controller Paradigm

The J2EE programming model applies the Model-View-Controller design pattern to J2EE development. This allows enterprise application developers to separate tiers efficiently and associate many views with one model. In fact, for any client-server architecture, it's important to isolate the real logic of your application from your end-users so that you can easily change the look and feel of your application without impacting the logic.

Figure 3 shows one representation of this design pattern.
Many design patterns (best practices) recur in J2EE applications: shopping patterns, security patterns, transaction patterns, patterns that involve dependent objects such as value or data access objects, patterns that require a session-to-entity exchange, and so on. If you're designing your own e-commerce application with a Web-centric architecture, remember that an EJB-centric solution will be easier to adopt later if your design clearly separates presentation and business logic, as shown in Figure 4.

Figure 4: Class Diagram of a Typical Shopping Pattern

Figure 5 shows how to map your Web-centric application to an EJB-centric application.
Developing J2EE Applications with Rational Rose

Rational Rose introduced support for J2EE via the Rose Tech Preview release in July 2000. Rational Rose supports the following capabilities related to J2EE:

- Servlet development, including generation of a J2EE compliant deployment descriptor.
- Entity Bean development, including generation of a J2EE compliant deployment descriptor.
- Session Bean development, including generation of a J2EE compliant deployment descriptor.
- Ability to invoke a Java compiler to compile files.
- Creation of Web and EJB archive files from within Rose.
- Synchronization of home and remote interfaces with the EJB class.

We introduce you to these capabilities in the following examples but do not cover the dialogs in great depth.

Creating and Modifying Servlets

Rational Rose provides a quick and easy way to create and modify servlets. To create a new servlet, right click on a class diagram. Then, from the shortcut menu, select J2EE>New Servlet. This brings up the dialog shown in Figure 6, which allows you to quickly configure the servlet you want to create. Note that the dialog is multi-level (Servlet Class Properties, Advanced Properties, Deployment Descriptor). You can move back and forth among the different levels by clicking the desired item listed on the left.
In the Class Properties dialog, you can specify the name of the servlet, whether it is based on the generic servlet or the http servlet class, and so on. You can also choose the exact operations you want to generate for the servlet: e.g., doGet, doPost, destroy, init, etc. Let's specify the name as "myServlet," a subclass of the http servlet class, and select the following operations from the list: init, service, doGet, destroy.

If we click on "Advanced Properties" on the left, we bring up the Advanced Properties dialog shown in Figure 7. This lets us specify whether we want to use a context object, and the exact operation of the servlet in which we want specific activities to take place. For example, if we check getSession and choose doGet from the list, a call to `getSession` will be placed in the doGet operation at code generation time. Similarly, if we set a BufferSize limit of "7," then a `setBufferSize()` call will be placed in the doGet operation at code generation time.

The resulting doGet operation is shown below:

```
/*
 Method: doGet
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    /**
 Servlet Code Generation Process
   -- Content Type --
    **/
    response.setContentType("text/html");

    /**
 Servlet Code Generation Process
*/
```
--- Response Buffering ---
/**
 * response.setBufferSize(7);
 **/

** Servlet Code Generation Process
--- ServletRequest.getSession ---
/**
 * HttpSession session = request.getSession(true);
 */

--- Servlet - Class Properties
- Advanced Properties
- Deployment Descriptor

![Advanced Properties Dialog](image)

**Figure 7: Servlet Configuration - Advanced Properties Dialog**

Next, let's click on Deployment Descriptor on the left, which brings up the dialog shown in Figure 8. Servlets use XML-based deployment descriptors as a way to communicate various configuration details to a container or Web server. At code generation time, Rose J will generate the XML deployment descriptor based on the information we specify. This may include a more descriptive name for the servlet, initialization parameters required for the servlet, a session timeout value, and so on. We can specify the Init and Context parameters by using the radio buttons. We can toggle between Name and Value, and enter the name and value in the text field on the right.
We can also modify an existing servlet (created via the process described above) by selecting the servlet class on a class diagram, right clicking, and then choosing J2EE>Servlet-configuration from the shortcut menu. This brings up the same dialogs, pre-populated with the appropriate information.

Creating and Modifying Enterprise JavaBeans

Working with EJBs is equally simple, and we can create them in a couple of ways. One way is to click on the class diagram and then choose J2EE>New EJB from the shortcut menu. The preferred method, however, is to make a design/implementation decision by turning an analysis class into an EJB. We can select the class, right click to bring up its shortcut menu, and select J2EE>EJB-Configuration. This simplifies the process because we do not have to specify names for the EJB or the remote and home interfaces. Rose uses the name assigned to the class we are converting as the basis for establishing the other names!

Selecting J2EE>EJB-Configuration brings up a dialog similar to the one for creating a servlet. The dialog has three levels: EJB Class Properties, Persistence Properties, and Deployment Descriptor.

The Class Properties dialog, shown in Figure 9, allows us to choose the type of EJB we want to create, as well as the bean, home, and remote interfaces. For instance, we can choose to create an entity bean with container-managed persistence. We can call it MyEntityEJB, and name the home and remote interfaces MyEntityHome, and MyEntity, respectively.
Clicking on Persistence Properties on the left side brings up the Persistence Properties dialog shown in Figure 10. When we create entity beans, we can specify that the primary key class should be based on some pattern, or we can assign a specific name for the class. We can also define finder methods. (Note that the first method is defined this way; the rest can be defined by selecting the EJB, right clicking, and then selecting J2EE->EJB- New ejbFind Method from the shortcut menu, which brings up a dialog similar to the one in Figure 10.)
Clicking on Deployment Descriptor on the left side brings up the Deployment Descriptor Configuration dialog shown in Figure 11.

Compiling Java Files from Within Rose

Rose J also introduces a modeless, auto sync editor for use with Java. Once it’s installed, the new editor becomes the default browser for Rose, accessible via the Java>Browse Java Source command.

Via the new editor, we can link in an external Java compiler and then compile files from within Rose. This is useful for quick compilation and is not intended as a replacement for a full-featured IDE. To link in our Java compiler, we’d enter the path to it via Build>Java Compiler Options. The dialog is shown in Figure 12.
To invoke the compiler, we choose the Build>Compile menu option from the modeless editor main menu.

**Building EJB and Web Archive Files from Within Rose**

Once we've compiled our Java classes to .class files, Rose J offers a convenient way of building EJB and Web archives. Access to this capability is via the J2EE>New EJB-Jar file, the J2EE>New War file menu options off the class shortcut menu, or the Tools>Java>J2EE menu. Figure 13 shows the dialog.

![Figure 13: Dialog for Creating an Archive](image-url)
In the first field, we can enter the path to the jar utility. Then, we click on the [...] next to Archive Name field, choose the directory path, and enter the file name for the jar file via the file dialog. We can then select and add the appropriate .class files and the XML deployment descriptor we generated from Rose for the EJB (or servlet, if we're dealing with a Web archive file).

**Automated Interface Synchronization**

One of the challenges developers face when using EJBs is the need to keep the EJB class and the home and remote interfaces in sync. For example, each time you add a new business method and want to expose it via the remote interface, you must also add the method to the remote interface class. Rose automates this process by providing a synchronization menu option that's invoked from the J2EE>EJB-Update>Update Interfaces (from EJB) or from J2EE>EJB-Update>Update EJB (from Interfaces menu options).

Rose relies on stereotyping to distinguish between different types of methods, so if you create the methods yourself, be sure to assign them appropriate stereotypes. For example, business functions must be stereotyped <<EJBRemoteMethod>>. Find methods must be stereotyped <<EJBFinderMethod>> and so on.

**The Future of J2EE and Rational Rose**

The J2EE platform is evolving rapidly to meet the demands of current users and vendors. A new J2EE 1.3 specification is now in development via the Java Community Process, and drafts are already available. This new version includes revisions and refinements for many of the technologies and features we have discussed, introducing greater consistency and tighter integration throughout the platform.

The new specification also introduces connector architecture. A weakness of the J2EE 1.2 specification is that it doesn't provide an easy way for enterprise

---

**A Quick Preview of the New EJB 2.0 Architecture**

- This new version emphasizes portability. Although a goal for the EJB framework is the ability to move ejb-jars from one container to another without any rewriting or recompiling, an area such as persistence, for example, might be managed differently in different environments. The container-managed contract was rewritten to resolve this problem.

- In the EJB 1.1 architecture, an entity bean was likely to be represented as a row in a database. In the 2.0 container-managed architecture, the persistent state schema of the bean is much richer.

- The EJB 2.0 architecture brings an asynchronous and messaging dimension to the EJB framework with Java Messaging Service technology.
and the definition of a new, message-driven type of bean. A message-driven bean is defined as a single class with no home and no remote interfaces (to avoid any internal or external direct access).

- The new specification introduces EJB QL, an SQL-like query language for finder and select methods. This powerful language will operate at the logical level, not the data level, and is a portable definition for finder methods.

- The EJB 2.0 specification introduces support for an additional type of method: the home methods, which are equivalent to static methods. They are not attached to one particular instance. Home methods are declared in the home interface and defined in the implementation class.

- Interoperability between J2EE application servers has been enforced with a requirement to support the CORBA/IIOP protocol. In this configuration, IORs (interoperable object references) are associated to the EJBHome and EJBRemote, and IIOP messages are sent from one J2EE server to another. This powerful requirement allows EJBs to be accessed by non-Java clients.

- Other technologies have also been revised, including the Java Servlet specification, which introduces new capabilities (e.g., support for event notification and filtering), and the JSP specification, which has added mapping from JSP to XML.

As a future direction, J2EE v1.3 points to the XML data binding specification, which is a way to realize XML schemas by Java classes. Although it's still under development, this specification previews the important role that XML will play in the future specification of the Java 2 platform, Enterprise Edition.

Given J2EE's responsiveness to user demands, its popularity as a development platform -- which is already growing rapidly -- is likely to accelerate in the future. Although the learning curve for J2EE is steep,
developers will find that Rational Rose can ease the transition to this platform by automating and simplifying many aspects of the development process. Visit the Rose Upgrades page at http://www.rational.com for free Rational Rose Evaluation software.

References


For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!