JAVA™ DEVELOPER'S JOURNAL

The World's Leading Java Resource

May 2001 Volume: 6 Issue: 5

RETAILERS PLEASE DISPLAY UNTIL JULY 31, 2001
$4.99US $6.99CAN

SYS-CON MEDIA

SEAN RHODY, FOUNDING EDITOR/CHIEF CORPORATE EDITOR

Son of a Sunset

My column two months ago, "Sunset on the Evil Empire," stirred up a great deal of controversy. Part of it was my fault, as I was trying to make two distinct points in the article, and that elicited a great deal of excitement directed at one point or the other. My first point, which I made in a curmudgeonly manner, was that Windows 95/98/Me is unstable, and that I was unhappy with that DOS-based platform. My second point concerned the Sun–Microsoft lawsuit settlement, in which we, the consumers, were the true losers.

With regard to the first point, I received a variety of responses. Several readers encouraged me to look at the Windows 2000 platform as a much more stable environment. I have, by the way. And I've been running its older brother NT for years. For a development platform I find it fairly stable, although I was able to crash it the other day by running five or six separate VMs and killing the virtual memory.

Other readers recommended the usual variety of operating systems. "Try Linux," they said. Or, "The Mac is a great platform." Agreed. I have Linux, Solaris, and HP-UX running at home. I like the Gnome interface on Linux, and find the CDE environment so similar that it's hard for me to tell whether I'm running the Sun box or the HP. I haven't tried a Mac in years, but OS X might tempt me.

Some readers were incensed that I wasn't impressed with the consumer version of Microsoft's operating system. "Spam," one reader called it. Another called me a "Charter member of the 'I hate Microsoft club,'" which I found pretty amusing as most of the folks on that side of the house wanted to shoot me when I ran a DCOM article a year or so ago.

In any case, I made a mistake ranting about operating systems, because doing that is like arguing religion – it can be fun, but nobody ever resolves anything.

Now concerning my second point, I received still more comments. I'm disappointed in the decision, because it means no Java integration into a very large set of applications that are used by the majority of end users in the world. I would have loved to see Java as an alternative scripting language to VBA, and certainly JSP would have been a nice alternative to ASP for IIS. Instead we get to use C# and .NET.

Once again readers were of two minds. Some felt that Java was making the Windows platform and its accompanying set of applications irrelevant. I disagree with some of that, mainly the part about the applications. Certainly the "write once, run anywhere" approach, combined with J2EE, has made selection of a server more a matter of hardware compatibility and scalability than software features, which is what I think Sun intended all along.

Other readers thought that Java was becoming irrelevant and blamed Sun for trying to control the hottest language in the market rather than moving it to a true open standard. I have a hard time believing the hottest language in the market will become irrelevant, but I do understand how the open standard issue is affecting acceptance…and not affecting it.

The one I liked best, though, came from a Microsoft employee in the IIS division. Of all the Microsoft-centric replies, his was actually the most polite. He suggested that there was no reason a vendor couldn't build a Java version of the common runtime for .NET, effectively creating Java on the Microsoft platform. And he's right, although from a Java perspective I think that's backwards. It's moving the mountain to Mohammed. Still, it was an intelligent, interesting comment, and I hope someone accomplishes it.

So why did I recap this at all? Because I wanted to let you all know that I do read your e-mails, and that we at ***JDJ*** value your input. Obviously, this was a topic near and dear to all of your hearts, and there were as many opinions as there are readers. Those of you who haven't written, I'd like to hear your views too.

sean@sys-con.com

AUTHOR BIO

Sean Rhody is the founding editor of Java Developer's Journal. *He is also a respected industry expert and a consultant with a leading Internet service company.*

ALAN WILLIAMSON, EDITOR-IN-CHIEF

All You Need Is Love

Last month when I sat down to write this editorial I had the good fortune to be staring out at the Golden Gate Bridge. Sadly, this month the view isn't quite as romantic; I'm sitting approximately 12 inches away from an elderly lady who has decided to push her seat back, reducing my air space by what seems a factor of 10. Yup, I'm on a plane heading for **SYS-CON** HQ to begin the layout for the JavaOne issue of ***JDJ***.

Since taking on this prestigious role, my understanding of the publishing world has jumped up a huge number of levels. It's a wonderful road to be traveling, and, as I busily scribble in my notebook, it's quite a responsibility that – if I were to sit and think about it – would keep me up at night.

Think about it. I and the ***JDJ*** team here have to prepare content for you that not only will educate but hopefully entertain. It's quite a task. Being a hard-core Java developer myself, I know how picky you are and what your attention to detail is like.

In fact, just yesterday my upcoming keynote at a Java conference in London was put on hold. The reason? The organizers got word there would be a major anticapitalist demonstration on the same day and they feared for our safety. When you hear it the first time it's quite funny, but when you give it a little more thought it's quite sobering to think you're involved in an event people feel strongly enough to actually picket. Kinda throws the notion that computers and the software we're all involved with are for the greater good. Or at least that's the dream I'm hanging – no, clinging – on to.

The hippy in me (or as close as a '70s/'80s child can get to being a hippy) thinks there isn't enough love in this world and all we need to do is just give a little hug to our neighbors to let them know we care. Wouldn't the world be a nicer place to live?

For me, I feel this whole Web Services revolution is the computer industry's hug to one another. At last vendors are waking up to the fact that there are solutions out there other than their own.

And here's the news flash: not all clients buy a single-vendor solution. Instead, it'll be a hodgepodge of solutions that claim they all talk to one another, but in reality will really sing only after thousands of dollars have been spent on consultants to get it working.

Web Services is the initiative to attempt to get all these solutions talking to one another without the extra overhead of the consultants. Wonderful goal, but I can't see it working quite as well in reality as the marketing/PR people are prophesying. "But it's just using Java and XML as the core," they say. So? What does that solve? My mother and I both speak English but, bless her, I doubt she'll ever understand what it is I do.

I support the notion of Web Services and I'm keeping my ear to the ground to see what's really going on. There's a lot of announcements in this area at the moment and it's quite daunting trying to keep up with it. Well, not only will we feature the odd Web Services piece, we're proud to announce that next month **SYS-CON Media** will launch a new title, ***Web Services Journal***, that will keep you up to date on this new arena. Look for it inside ***JDJ*** at JavaOne.

As you know, last month I was in San Francisco doing a tour of duty. I met a lot of people and did a lot of listening. I wanted to gauge the mood of the place, especially now that the bottom has most definitely fallen out of the dot-com market. Should we be worried or not? Well, I'm happy to report that the technical roles still need to be filled with skilled engineers. Although not as plentiful as they used to be – and merely turning up for the interview no longer serves you the job – on the whole, as one highly respected engineer put it, Silicon Valley is finally coming back down to earth. And what happens in the Valley generates ripples through to the rest of us.

So don't panic.

alan@sys-con.com

AUTHOR BIO

Alan Williamson holds the reins at n-ary (consulting) Ltd, one of the first companies in the UK to specialize in Java at the server side.

The AttachmentLayout Manager

Speed up your GUI construction

Written by *Valor Dodd*

Have you ever needed to write a simple graphical user interface (GUI) but didn't have the right kind of layout manager? Do you hate to use a layout manager that takes you longer to understand than to make your GUI? If this is the case, you're probably not a big fan of the GridBagLayout manager and its complexities.

This article describes AttachmentLayout, a layout manager that allows for GUI components (buttons, labels, lists, etc.) to be placed inside a container to its edges using spatial relationships. It's a simple but effective layout manager that can be easily visualized and implemented with only one line per component.

"The AttachmentLayout enables you to get to the heart of the software engineering that the simple GUI supports."

Background and Concept

The AttachmentLayout manager was developed for two reasons. The first is to allow construction of a GUI with only a few basic components placed inside a frame or dialog box – this can't be easily done using FlowLayout, BorderLayout, GridLayout, or BoxLayout. These first three layout managers are supplied by all implementations of the Java Development Kit (JDK) and are the easiest to use and understand. The last layout manager, BoxLayout, is included in the latest implementations of the Swing package and is more difficult to use. In many simple GUIs, GridLayout and FlowLayout can't produce the desired component layouts. BorderLayout and BoxLayout can produce many simple GUIs, but with additional overhead and less efficiency. Almost all GUIs can be constructed with the exact pixel placement of the components or using the GridBagLayout. Exact pixel placement would get the job done, but most experienced GUI developers wouldn't recommend this course of action as it can lead to less portable code.

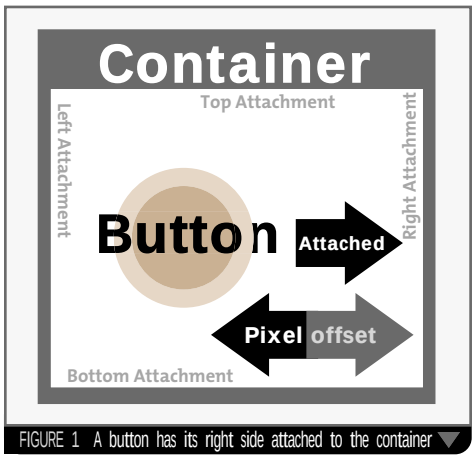


FIGURE 1 A button has its right side attached to the container

The second reason for the development of the AttachmentLayout manager is that the GridBagLayout is complex and can be cumbersome when constructing a simple GUI. The AttachmentLayout enables you to get to the heart of the software engineering that the simple GUI supports.

AttachmentLayout was designed to emulate certain aspects of the layout managers available in X-Windows/Motif without the complexities of a GridBagLayout manager or the inefficiencies of the others. The concept behind the AttachmentLayout manager is the ability to have a GUI element such as a button be connected (hence the word *attached*) to a container edge and, in some cases, to other GUI elements. In Figure 1 a button has the container as an attachment on its right side with a specified pixel offset between the button and the container. If the user stretches the right side of the container during program execution, the button keeps the same dimensions but follows the right side of the container as it's stretched. In another example, if the container is attached to the button's top, bottom, left, and right sides, the button will increase its width and height as the container's width and height are increased.

Implementation

The AttachmentLayout manager consists of two classes. The first, Attachment.java (see Listing 1), defines how a single component is attached to the container and to other components. It allows a component to have an attachment to its top, bottom, left, or right side. (Listings 1–4 can be found on the ***JDJ*** Web site, www.JavaDevelopersJournal.com.) An additional feature of Attachment.java is pixel padding. Padding is the number of pixels a component will always have between it and the component it has an attachment to on one of its sides. Attachment.java has three constructors defined, and the one most commonly used is shown below.

```
public Attachment( leftPad, leftComp, rightPad,
rightComp, bottomPad, bottomComp, topPad, topComp)
```

The first input parameter (leftPad) is the left pixel padding between a component and its left-side attached component. LeftComp is the component that will be attached to the left side. The same follows for the right, bottom, and top attachments, respectively. If you don't wish to have an attachment to one or more sides, the component input parameter for that particular side should be set to null. An instance of Attachment is assigned to each component when it's added to a container specifying an AttachmentLayout. Also stored in each instance of Attachment are the pixel location, width, and height of the component. All these fields are necessary to compute the location and size of each component when the container is resized. The size and location of each component is computed in AttachmentLayout.java.

The second class is AttachmentLayout.java (see Listing 2). To use the AttachmentLayout, a container must specify its use with setLayout(new AttachmentLayout()). AttachmentLayout has one constructor with no parameters. The constructor's only function is to initialize a Java Vector that will hold all the components to be placed inside the container. This class implements the LayoutManager2 interface in the java.awt package and must implement several methods to function properly as a layout manager. In the case of our layout manager, the most important methods of the interface that need to be defined are addLayoutComponent and layoutContainer. The Java Virtual Machine (JVM) calls addLayoutComponent when a program calls the add method (Component comp, Object constraints) to add a component to a container. For the AttachmentLayout manager, an object may be added to a container by calling the add method with the component to be added and an instance of the Attachment class as the constraint object.

As can be seen in Listing 2, the addLayoutComponent method takes the input component and adds it to the list of components in the container. This list is used in the layoutContainer method, which is the real guts of the AttachmentLayout class. It's called by the JVM to do the actual rendering of the container. This method computes the dimensions and the location of every component of the container. The first function the layoutContainer method per-

> "...the most important methods of the interface that need to be defined are addLayoutComponent and layoutContainer"

forms is to loop through every component in the container. If the component has an attachment to the container, it computes the upper-right and lower-left locations of the component based on the information stored in the component's associated instance of Attachment. From these two positions, the width and height can be calculated and the component can be placed inside the container with the proper spatial relationship to the container.

The second function of layoutContainer is to determine the component's relationship to other noncontainer components that it has attachments with inside the container. Again, looping through all components does this, and if it has a noncontainer attachment, it computes its upper-right and lower-left positions based on the components associated instance of Attachment. The remaining methods of AttachmentLayout are required by the LayoutManager2 interface, but as these are seldom used they're left to the reader.

Example

Figure 2 shows a simple GUI made using javax.swing components. The white area is a JPanel named mainPanel and consists of a JLabel. Along the bottom of the GUI are two JButtons named *Button 1* and *Button 2*. The main emphasis of a GUI of this kind is usually the contents of the large mainPanel, so it's afforded the largest area of the GUI. The content pane of the JFrame uses an AttachmentLayout manager to position mainPanel. The desire is to have mainPanel expand in every direction as the JFrame is resized. The following lines of code demonstrate how this is done.

```
Container p = JFrame.getContentPane();
p.setLayout(new AttachmentLayout());
p.add(mainPanel,new
Attachment(5,p,5,p,10,button1,5,p));
```

The first line of code gets the content pane of the JFrame. All components added to a JFrame are actually added to its content pane. Line 2 instructs the content pane to use an AttachmentLayout for its layout manager. Line 3 adds mainPanel to the content pane. Notice that the second argument to the add() method is an instance of the Attachment class. This form of add() is necessary since AttachmentLayout implements the java.awt.LayoutManager2 interface. The instance of Attachment has eight parameters. The first two parameters (5,p) tell AttachmentLayout how to construct the left attachment of

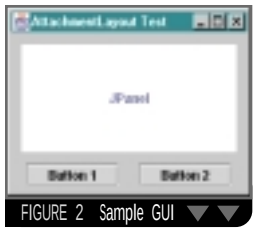


FIGURE 2 Sample GUI

FIGURE 3 GUI after JFrame is increased

mainPanel. The integer 5 is the number of pixels that will always be between mainPanel and the component it will be attached to on its left side (e.g., the content pane of the JFrame). The next two parameters are also 5 and pane. They tell the AttachmentLayout how to construct the right-side attachment of mainPanel. Parameters 5 and 6 describe the bottom attachment. In this case we want mainPanel to be spaced 10 pixels above Button 1. The mainPanel could also have a bottom attachment to Button 2 to provide the same results. And finally, the last two parameters construct the top attachment of mainPanel, always 5 pixels from the top of the content pane.

Along the bottom part of the GUI are Button 1 and Button 2. This GUI's intent is to have both buttons stay the same size and in relatively the same location during the resizing of the JFrame. Both buttons are also added to the content pane using the AttachmentLayout manager. The following lines of code demonstrate this:

```
pane.add( button1, new
Attachment(5,p,0,null,5,p,0,null));
pane.add( button2, new
Attachment(0,null,5,p,5,p,0,null));
```

The first line of code adds Button 1 to the content pane. It will be spaced 5 pixels away from the content pane on the left and bottom. The right and top attachments are set to null since we don't want the width or height to change as the frame is resized. Button 2 is added in a similar fashion but with attachments on the right and bottom. If you don't want to change a component's width or height, the size of the component must be set using the setSize() method of the component before it's added to the container. If this method isn't called before it's added, the preferred size of the component is used. Otherwise the component would still be placed inside the container, but at a default width and height of zero (see Listing 3 for the complete example).

Figure 3 shows the same GUI as in Figure 2 but the screenshot was taken after the JFrame increased in width and height. Notice the sizes and placement of the components in Figure 3 as compared to Figure 2.

As mentioned previously, this same GUI could be constructed using GridBagLayout. Listing 4 shows the steps necessary to construct the same GUI via GridBagLayout. It requires the use of a helper class, similar to Attachment.java, called GridBagConstraints. GridBagConstraints keeps track of all the characteristics describing the placement of a component inside a container. In our example, mainPanel required the use of eight different parameters and one class just to describe its placement.

For each button it also takes eight parameters and one class. The BuildConstraints() method simplifies the construction of the settings of the cell location, size, and proportions of the container it will fill. Using this method reduced the amount of code necessary to construct the GUI. Other

GridBagConstraints parameters to consider are the anchor and the fill. They tell the layout manager how they're positioned within the container and how they fill cells. Note that in GridBagLayout, components are placed in a grid of cells. The cells don't have to be the same size, which makes it even more difficult to mentally visualize the container layout. Finally, both buttons required an instance of the Inset class to be set inside the GridBagConstraints object in order to have the proper padding around the buttons.

The use of our simple GUI example provides a clear picture of the relative ease in using the AttachmentLayout manager versus the GridBagLayout manager. The AttachmentLayout manager essentially required only four lines of code to arrange the main panel and both buttons in the JFrame's content pane. In comparison, using the GridBagLayout required 18 lines of code and a helper method (seven additional lines and two instances of the Inset class).

Using AttachmentLayout, this example GUI can be easily visualized and the code written in less than five minutes. Constructing the same GUI, but using GridBagLayout, took at least twice as long and the use of a manual to remember all the options.

AttachmentLayout was also easier to use than the BorderLayout and the BoxLayout manager. The BorderLayout required seven lines of code and the BoxLayout required eight to produce the exact same GUI, but both of these required the use of additional panels and layout managers.

Limitations

As with all layout managers, the AttachmentLayout manager has its limitations. It's good for simple GUIs that may not be easily handled by other layout managers. More complex GUIs will require a layout manager such as the GridBagLayout. Also, it should be used only when components inside a container don't depend on the resizing of other noncontainer components that it is has an attachment to. This limitation is what some experienced Motif/X-Windows developers know as *circular dependency*. As an example, if Button A has a right-side attachment to Button B and Button B has a left-side attachment to Button A, there's a circular dependency. Circular dependencies will produce unexpected results and in some implementations of Motif/X-Windows they will produce an infinite loop. In our case the size and placement of both components depends on the changes of the other, but only one will take effect, depending on the order in which the components were added to the container.

Summary

In my five years of Java programming I've been required to construct relatively simple user input or informational display GUIs on many occasions. But I never have quite the perfect, easy-to-use layout manager that would help me construct my GUI in the shortest amount of time.

The basic layout managers that come with Sun Microsystems' JDK cover both ends of the spectrum when it comes to use and functionality – easy and simple to hard and complex. The simpler ones are GridLayout, FlowLayout, and BorderLayout. The harder and more complex one is the GridBagLayout. BorderLayout or BoxLayout can produce our simple example GUI or more complex GUIs, but will require the use of additional nested panels and layout managers. AttachmentLayout.java with its concept of attachments is a simple layout manager, but with a different kind of capability, and as such it's another useful tool that can be put in the GUI developer's arsenal.

Author Bio

Valor Dodd is a Java-certified senior software engineer at Lockheed Martin in Denver, Colorado. He has more than 20 years of software experience developing computer graphics and GUI applications for the telephone and defense industries.

vsdodd@mho.com

Working with Dynamic XML Documents

What XML/value mapping lets us do

WRITTEN BY
JON SIEGEL

XML gets mentioned a lot as an interoperability "platform." By itself, of course, XML can't be a platform because it's a document format. It may be flexible, human-readable, dynamic, popular, and cool because it looks a lot like HTML, but it's still just a document format, and there are a lot of differences between a document format and an interoperability platform.

To interoperate using XML, you either have to build an infrastructure around it or incorporate it into an infrastructure that already exists. While other folks build yet another infrastructure around XML, we show in this column how XML has been incorporated into the enterprise-ready, mature CORBA infrastructure. This is **CORBA Corner**, after all.

The W3C has supplemented XML with the Document Object Model (DOM), defined in OMG IDL. OMG members used the DOM as the basis for their XML mapping, but made one change along the way: instead of keeping the representation of each node in the XML document tree as a full-blown CORBA object, OMG's version represents a node as a CORBA valuetype. Passable by value but not a first-class CORBA object, the valuetype is the CORBA multilanguage equivalent of the Java serializable. And valuetypes are tailor-made to represent an XML document's structure: graphs of valuetypes, sent over the wire by including their root node (or any node, if the node structure links both up and down the tree) in the argument list of a CORBA call, will be reconstructed properly, in their entirety, at the receiving end. Send an XML document to a remote application and suddenly all navigation up and down its tree is done with local invocations instead of dozens of network round-trips.

In this article we investigate OMG's XML/value mapping and the things it lets us do with our XML documents. More than just a bridge between XML and CORBA – although it certainly is all of that – the valuetypes and their structure provide such an elegant API into the XML document (structure and content alike) that (in our opinion, anyhow) this deserves to be the way everyone works with XML content from a program, even in a non-CORBA environment. Here's what you can do with XML documents using the mapping:

- Create a new XML document from scratch.
- Read in an existing XML document from storage or from the network.
- Parse the document into a multiply linked list of CORBA valuetypes.
  - –Parsing can be done dynamically if there's no DTD with structural information about the document.
  - –Parsing can take advantage of a DTD if there is one.
  - –If the document and DTD versions are out of synchronization, the parsing can take advantage of the DTD as far as it goes.
- Edit the document, including adding or deleting elements; adding, deleting, or changing attributes; and editing text.
- As a linked list of valuetypes, the document may be sent around the network in CORBA calls with its structure intact. This includes secure, transactional CORBA calls and asynchronous calls using CORBA messaging. This is a great way to send XML data in an invocation.
- Serialize the in-memory representation, generating a revised version of the Unicode-based XML format document that you're used to.

We don't have space to demonstrate all of these, but we'll look at as many as we can in the form of a programming example. We haven't included the specification details here to save room for example code, which isn't available free off the Web as the specification is. To get the specification, download doc.omg.org/orbos/00-08-10 (the specification document) and doc.omg.org/orbos/00-11-01 (zipped IDL file). Where the two files disagree, the IDL in the zip file supersedes.

Listing 1 is an example XML document that we use throughout this article. If your XML knowledge is a little hazy, point your browser to www.w3.org/XML. And to learn about the DOM, surf to www.w3.org/DOM.

Initializing and Reading the Document

We're not going to list the code that gets us started in XML document processing mode. Instead, we'll just list what we did:

- ***Representation of XML documents as strings:*** Even though both XML and Java use Unicode, CORBA represents XML as a special DOMString type (typdef'd to sequence<short>). Why? Because you can use CORBA to go from any language to any other. Pass a Java string to a C program and you (probably) end up with an array of 8-bit chars; pass it to COBOL (unlikely, we admit, but possible) and the system attempts to translate your Unicode into EBCDIC! Yucko. So we've created a convenience function makeDOMString that converts a Java string into the programming language–independent DOMString type.
- ***Reading in the document:*** We read the document in as a Java string and con-

This article is condensed from the forthcoming book *Quick CORBA 3* by Jon Siegel, ©2001 by Object Management Group, and appears here by permission.

...the valuetype is the

CORBA multilanguage equivalent of the Java serializable.

And valuetypes are tailor-made to represent an XML document's structure

verted it into a DOMString.

- ***Parsing the document:*** The parser is defined by the specification and supplied with your implementation. After locating the parser (probably via a call to resolve_initial_references), you invoke, for example,

```
Document PO_doc =
parser.parse(PO_Stream);
```

- ***Error checking:*** The XML specification requires a parser to return an error, with no partial results, if a document contains even one XML structure/format error. (It doesn't care if you had the price of the bolts wrong, though.) The OMG specification is well prepared for this, with its definition of exception XMLException and 38 specific parsing error codes (numbered 2 through 39, of course). You should definitely check for these errors on return from parse.

On return from parse, if the routine found no errors during parsing, our document is stored in a multiply linked list of valuetypes starting at the root node PO_doc. Now let's do some things with it.

Editing the XML Document

If we're the company writing the PO, we need to edit it – adding or deleting items, changing quantities or POitem numbers or names, or whatever. To our programmer, the XML/value mapping structures the PO data to make it all easily available; using these program structures, the programmer will present the data to our clerk for editing via a GUI. The operation getElementsByTagName returns a list of Elements selected by Tag Name (duh!), so we'd probably start by retrieving all (that is, both) of the POitems this way:

```
DOMString name =
makeDOMString("POitem");
// Retrieve items in Purchase
Order:
NodeList elms =
PO_doc.getElementsByTagName
(name);
```

Now elms, a sequence of Nodes, contains two elements – the two items in our Purchase Order. Each contains four child elements – the POitem_name, POitem_number, POitem_size, and POitem_quantity. We could easily display a POitem in a window for editing, or count the number of POitem nodes that we got back and display the number on the screen, or print it for confirmation when we print the PO.

Changing the Text in an Element

The specification uses OMG IDL attributes, which aren't the same as XML attributes. Here's a quick review in case you forgot how IDL attributes work: if you declare a variable to be an IDL attribute, the IDL compiler generates a get and set operation for it automatically (unless you declare it read-

only, which eliminates the set operation). The get and set operations are mapped to programming languages just like all other operations. The Java mapping overloads the operations on the name of the variable: if you include an input argument, it's a set; leave it out and it's a get.

To demonstrate how we can change the text associated with a particular element, let's change the quantity of Bolt POitem_number B01420 to 150 gross. Listing 2 contains the code in a single block, with a few comments. The rest of this section explains it in more detail.

After defining two DOMStrings for use later, we start our loop over POitems in elms. elms.item(i) returns the ith Node in NodeList elms. (The operation name *item* comes from the XML/Value specification and has nothing to do with the fact that we're retrieving a POitem.) elms.item returns a Node; we have to cast the return value to an Element in order to assign it to element POitem.

Each POitem element has four children, tagnamed (from the strings in our XML document) POitem_name, POitem_number, POitem_size, and POitem_quantity. getElementsByTagName returns a list, so we declare ino and iqty to be NodeLists even though we're certain that only one element is going to come back from each call here. After checking that we have a valid POitem (even though we didn't bother to check that we had a valid PO!), we're ready to check and change the number of items we want to buy.

One of these lines of code (at least!) needs a little explanation. It's this one:

```
if
(((Text)(ino.item(0).firstChild()))
.data().equals(checker))
```

The four Element valuetype children of POitem that we're working with here don't contain text – they have children that contain the text. Here's how we burrow down to the text itself.

ino is a one-element NodeList containing our POitem_number. item is the operation defined by the specification on NodeList that returns an item in the list by index number. (Once again, the operation name *item* has nothing to do with its being an item on our PO.) So ino.item(0) returns the first Element in our (one-element!) list.

Fortunately for us, this Element (and its brothers and sisters) has only a single Text Node, so we can retrieve it using the get operation of the readonly attribute Node firstChild defined on the Element. In Java the get operation for an attribute maps to the name of its parameter so the operation firstChild gets that node.

The firstChild is a Text Node, so we have to cast it to (Text) in order to retrieve the text from it.

The text that it contains is in attribute data, so we can retrieve it using the get operation for data, which in Java maps to the operation name data. Fortunately it's a DOMString, the same type as checker, so we don't have to do any more casting to do the comparison. Phew!

Naturally, we've strung all of these fetch operations together in a single line of code to show you how elegantly you can program with this specification and Java!

In the next line of code (not counting the comments) we use the set operation of the attribute data of the Text Node of the POitem_quantity Element to set the new quantity. Except for this, the tricks in this line are the same ones as in the line above it.

Adding a New Element

We can add a new element easily. Operations to create new Nodes of all types – that is, Node factories – are defined on our root Document node, so we invoke on PO_doc to create Elements and the Text Nodes. When you create an

Element, you specify its tagName; when you create a Text Node, you pass in its text data.

Passing the Document in a CORBA Invocation

To pass our document as a tree of valuetypes, all we have to do is insert the root node as an argument in a CORBA call. For example, suppose our purchasing department runs a server that supports the operation PlaceOrder with this IDL:

```
Interface PurchasingServer  {

    Document ThisPO;
    boolean PlaceOrder(in
      dom::Document order);
};
```

In this operation ThisPO is a Document valuetype, and is an input argument to the CORBA invocation PlaceOrder. (We're not executing one of the Document methods.) When our client application invokes, in Java, the code in Listing 3, the entire purchase order tree gets sent over the wire to the server where it gets reconstructed exactly as it was in the client application, even though we've only included the root Document node of our purchase order in the argument list of PlaceOrder. This follows from the representation of the document node tree as a multiply linked list.

Writing Out the New or Revised XML Document

Once your user finishes editing the PO, you may want to write it out as an XML data file in Unicode. The operation to do this, serialize, is parallel in form to the parse operation discussed at the beginning of this article. Also, like the parse operation, serialize doesn't exist in the DOM at either Level 1 or Level 2. DOM Level 3 is supposed to introduce this functionality when it arrives.

Flyweight Pattern

It's not much of an issue for short XML documents, but long ones that repeat elements many times (and some documents may have hundreds, thousands, or even more instances of a given element) use up many bytes repeating element name text. The XML/value mapping uses the flyweight pattern to conserve this space: one instance of each element name (and other types of repeated text) is saved in an indexed array, and only the index number is saved with each element. The array is another valuetype, included in the structure of the document, so it goes over the wire along with everything else when you ship your valuetype tree around.

What About Documents with DTDs?

The specification treats static documents – that is, documents defined by a DTD – very well indeed, generating not only the IDL for a set of document-specific valuetypes but also their implementation. All you have to do is program the editing operations around these elements tailored to your DTD. We think the static mapping will be used a lot more than the dynamic mapping, but we had to present this first because it's the foundation for the static, which is based on the dynamic valuetypes with DTD-specified names. We'll present the static mapping in an upcoming column, so watch for it.

Acknowledgments

I'd like to thank Alan Conway and Darach Ennis of IONA Technologies, who wrote the example code for our sample XML file and answered many questions about the specification as we wrote the book chapter from which this article is excerpted.

siegel@omg.org

Author Bio

Jon Siegel, PhD, is director of technology transfer at OMG where he writes and teaches about OMG's specifications: CORBA, the CORBAservices and CORBAfacilities, and the modeling specifications UML, the MOF, XMI, and the CWM. He is the author of CORBA 3 Fundamentals and Programming *and the new book,* Quick CORBA 3 *(Wiley), from which this article was condensed.*

Listing 1

```
<purchase_order company="Enjay Manufacturing" number="01239876">
    <ship_to_address>
      <street>21 Pine Street</street>
      <city>Cleveland</city>
      <state>OH</state>
      <postcode>44113</postcode>
    </ship_to_address>
    <POitem_list>
      <POitem>
        <POitem_name>bolt</POitem_name>
        <POitem_number>BO1420</POitem_number>
        <POitem_size>1/4X20</POitem_size>
        <POitem_quantity>120gross</POitem_quantity>
      </POitem>
      <POitem>
        <POitem_name>nut</POitem_name>
        <POitem_number>NU14</POitem_number>
        <POitem_size>1/4</POitem_size>
        <POitem_quantity>120gross</POitem_quantity>
      </POitem>
    </POitem_list>
</purchase_order>
```

Listing 2

```
// Modify any Bolt items quantity values to 150 gross
    // where their POitem_number is 'BO1420'
    DOMString checker = makeDOMString("BO1420");
    DOMString change  = makeDOMString("150gross");
    // Loop over the items in our PO:
    for (int i = 0; i < elms.length(); i++)
    {
      Element poItem = (Element)elms.item(i);
      // ino is the POitem_number element for this poItem:
      NodeList ino =
        poItem.getElementsByTagName(makeDOMString("POitem_number"));
      // iqty is the POitem_quantity element for this poItem:
      NodeList iqty =
        poItem.getElementsByTagName(makeDOMString("POitem_quantity"));
      if (ino.length() != 1 || iqty.length() != 1)
      {
        System.err.println("Invalid purchase Order");
        System.exit(1);
      }
      // This next line is explained in detail in the text
      if (((Text)(ino.item(0).firstChild())).data().equals(checker))
      {
        // Compare successful: this poItem needs its gross changed
        ((Text)(iqty.item(0).firstChild())).data(change);
      }
    }
```

Listing 3

```
{
    // Retrieve a purchasing server object reference
    // from the naming service...
    PurchasingServer server = whatever;
    // Set up the document root of our PO tree structure:
    //
    dom.Document thePO = whatever; // set equal to our PO document root
    // Here we go...
    if (server.PlaceOrder(thePO))  // this line sends the entire document
    {
      // Success!
    }
    else
    {
      // Whoops.
    }
  }
```

Download the Code! www.JavaDevelopersJournal.com

power jMs

Written by Tarak Modi

Applying the facade pattern to JMS using a custom protocol handler

What is a facade?

In software engineering it's a design pattern. One possible definition of a facade is "A higher-level interface that provides a unified way of accessing a subsystem and as a result makes the subsystem easier to use."

Thus, in constrast to most design patterns that help break the system up into subsystems, the facade design pattern rolls up a complex subsystem into one, easy-to-use system. A facade can provide a simple default view of the subsystem that's good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

Why Does JMS Need It?

JMS serves as an excellent foundation for enterprise applications. Although the JMS API is very concise, using JMS effectively can be challenging and there are many potential booby traps that novice users may fall prey to. Furthermore, it may not be very appealing to force every developer in your organization to learn JMS. Most organizations would avoid the above "issues" by creating a reusable library (i.e., a facade) that encapsulates all the JMS-related code/knowledge. Unfortunately, even this library would require a learning curve, albeit a smaller one (hopefully).

In this article I present an alternative approach based on the Java protocol handler architecture. The major advantage of this approach is that most Java developers are already familiar with using this architecture via the URL class in the java.net package. For example:

```
URL url = new URL("http://www.javasoft.com/index.html");
```

An additional, and by no means minor, benefit is that this is a time-tested architecture built into the Java language/platform itself. Best of all, this architecture is flexible enough to allow the "plug-in" of new protocol (handlers). That's exactly what I demonstrate in this article by creating a protocol handler for JMS.

This JMS protocol handler will make programming with JMS as simple as the example shown in Listing 1.

Notice that Listing 1 doesn't contain any JMS-specific code. In fact, the only giveaway is that the name of the protocol is "jms" in the URL; otherwise the code is exactly the same as that for HTTP or FTP from a Java program. The JMS protocol handler takes care of all the JMS grunt work for us. After all, that's what a good facade is supposed to do, right?

An Overview of the Protocol Handler Architecture

One of the benefits of using Java is the excellent support it provides for networking. Not only is this support extensive, but it's easy to leverage, in most cases requiring the developer to master only a few classes, such as the java.net.URL class. This support is built on top of an elaborate (and extensible) architecture called the *Java Protocol Handler* archi-

tecture. In this section I discuss this architecture and how to extend it with your own protocol handler. Then in the next section, we actually implement a JMS protocol handler using this knowledge.

The gateway to this architecture is (you guessed it) the java.net.URL class, which encapsulates a URL string. The general form of a URL is:

```
protocol://host:port/filepath#ref
```

Examples include "http://www.javasoft.com/index.html" or "file:///C:/temp/junk.txt". (The three slashes "///" is not an error; you'll see why in a moment.) In the first example, the protocol is HTTP, the host name is www.javasoft.com, and the filepath is index.html. Since no port number has been specified, the protocol handler will use a protocol-specific/default port, which in this case will be port number 80. In the second example, the protocol is file and the filepath is C:/temp/junk.txt. Note that in this case neither the host name nor the port number has been specified (which would have been between the second and third slashes), so the file protocol handler will use protocol-specific/default values for these.

As an aside, the format of the URL string for the JMS protocol will be:

```
jms://Queue/<QueueName>
```

or

```
jms://Topic/<TopicName>
```

Note that the JMS protocol does not have a concept of a host name, port number, or filepath. Instead the host name is actually the messaging style and the filepath is the destination.

The URL class does not know how to access the resource stream represented by the URL string. Instead, it relies on a set of other classes to handle this. When a new URL class instance is created it resolves the URL string to a protocol-specific handler (e.g., the protocol handler class). This protocol handler knows how to create a connection to the resource represented by the URL string and return an object corresponding to this connection. Since this resolution occurs at construction time, any attempt to construct an instance of URL with an unknown/invalid protocol will throw a MalformedURLException during the construction itself. The relevant portion of the URL constructor is shown below:

```
 if(handler == null &&
(handler = getURLStreamHandler(protocol)) == null) {
throw new MalformedURLException(
"unknown protocol: " + protocol);
 }
```

I'll discuss the getURLStreamHandler method in detail later in this article.

Sun provides protocol handlers for several standard and widely used protocols such as HTTP, FTP, mailto, and Gopher. Protocol handlers must follow a strict naming convention. The class must always be named *Handler*. The package name must always have the protocol name as its last part. For example, Sun's protocol handler for the HTTP protocol is called Handler and is in the package sun.net.www.protocol.http. Note that the package name ends with "http". In our case the "jms" protocol handler will be in the jmsbook.jms package and will be called Handler. To make the Java runtime aware of your own protocol handlers, you must use the java.protocol.handler.pkgs system property. This property is set equal to a "|" delimited list of package name prefixes. These prefixes will be used to resolve the specified protocol name to a protocol handler object. Note that these package names must not include the last part (e.g., the protocol name). So in our case this property will be set to jmsbook (and not jmsbook.jms), as follows:

```
java.protocol.handler.pkgs=jmsbook
```

The getURLStreamHandler Method

Why does Java impose such a strict naming convention for protocol handlers? The answer is found by examining the URL class source code, more specifically the getURLStreamHandler method implementation, which is called to resolve a protocol name to the corresponding protocol handler. The relevant portion of this method is shown in Listing 2. Read the inline comments for the explanation of the code fragment.

Only one protocol handler object is created per VM per protocol. A new protocol handler is created the first time it's required and is then cached for later use. This means that multiple threads may use the same protocol handler simultaneously. Thus the protocol handler implementation must be thread- safe. The URL class instance caches the protocol handler in a static hash table, which allows any URL instance to access this handler. To get a better feel for this, let's take a look at the remainder of the getURLStreamHandler method. Once again, read the comments for the explanation (see Listing 3).

The openConnection and openStream Methods

At this point the URL has successfully resolved the protocol string to a protocol handler. The openConnection method may be used to gain access to a connection object. A connection object must implement the URLConnection interface and is used to send and receive data to and from the resource stream, respectively. The URL instance merely delegates the openConnection method call to the protocol handler as shown below:

```
 public URLConnection openConnection()
     throws java.io.IOException {
  return handler.openConnection(this);
}
```

The URL class also provides a helper method, openStream, for clients interested only in receiving data from the resource stream. This method is shown below:

```
public final InputStream openStream()
     throws java.io.IOException {
  return openConnection().getInputStream();
     }
```

Our JMS Protocol Handler

As discussed above, the URL class serves as the gateway into Java's protocol handler architecture. The URL class itself has very limited functionality beyond resolving a protocol string into a protocol handler. There are two key pieces that a protocol handler implementer must provide/implement: a Handler class and a URL Connection class. Having said that, let's take a look at the implementation for our JMS protocol handler. A class diagram showing all the pieces of the JMS protocol handler architecture and how they fit together is shown in Figure 1.

NOTE: This article is based on a chapter in Tarak Modi's upcoming book *The Essence of JMS: A Developer's Guide* (to be published by Manning Publications). According to the author, the actual chapter includes a much more detailed discussion of the concepts presented in this article along with the complete source code for the JMS protocol handler and associated test programs. Modi has offered to provide the complete source code and test programs to all ***JDJ*** readers who request them from him at tmodi@att.net.

The Handler Class

As with all protocol handlers, the JMS protocol handler conforms to the following rules:

1. The class name is Handler.
2. It extends the URLStreamHandler class and provides a concrete implementation of the openConnection abstract method.
3. Its package name has the protocol name as its last part (i.e., it's in a package whose last part is "jms". In our case the handler is in the jmsbook.jms package.

Since the JMS protocol handler extends the URLStreamHandler class, it must provide an implementation of the openConnection method. The openConnection method is responsible for returning a connection object (an instance of JmsURLConnection, which we'll see next) to the caller. The caller may then use this connection object to access the resource stream. The handler decides how to initialize the JmsURLConnection object based on the host name portion of the URL as shown below in pseudo-code:

```
// pseudo-code
if(u.getHost().equals("Queue"))
     return a new JmsURLConnection to use with JMS Queues.
else if(u.getHost().equals("Topic"))
     return a new JmsURLConnection to use with JMS Topics.
else
     throw new IOException("Host name must be Topic or
Queue.");
```

Configuring the Handler

The handler has two member variables that must be initialized: a reference to a queue connection and a reference to a topic connection. These references are obtained via a queue and a topic connection factory, respectively. The JMS specification does not define a standard way of getting the initial queue and topic connection factories. As a result, each vendor that provides a JMS-compliant messaging product must define its own way of allowing clients to get these initial connection factories. One of my primary design goals is not to get tied to a specific JMS provider. To achieve this I must isolate any vendor-specific code so that it does not affect the core architecture and can easily be replaced and tested. I have defined two interfaces, JmsQueueConnectionFactory and JmsTopicConnectionFactory, which are shown below:

```
public interface JmsQueueConnectionFactory {
  public javax.jms.QueueConnectionFactory
    getQueueConnectionFactory(java.util.Properties props)
     throws javax.jms.JMSException;
}

public interface JmsTopicConnectionFactory {
  public javax.jms.QueueConnectionFactory
    getQueueConnectionFactory(java.util.Properties props)
     throws javax.jms.JMSException;
}
```

These interfaces are defined in the jmsbook.jms package. Both have one method each. For example, the JmsQueueConnectionFactory interface has a method called getQueueConnectionFactory that returns the initial queue connection factory. This method gets an instance of a java.util.Properties object as its parameter, which contains all the vendor-specific messaging product information that the class implementing this interface will require to get the queue connection factory. One such class that implements the JmsQueueConnectionFactory interface is created for every vendor's messaging product that's to be supported. An example of such a class for Sun Microsystems' Java Message Queue product is shown below:

```
package jmsbook.jms
public class SunJmsQueueConnectionFactoryImpl
  implements JmsQueueConnectionFactory {
  public javax.jms.QueueConnectionFactory
    getQueueConnectionFactory(java.util.Properties
         props) throws javax.jms.JMSException {
    javax.jms.QueueConnectionFactory factory =
         new com.sun.messaging.QueueConnectionFactory();
  return(factory);
 }
}
```

Similarly, an example of a topic connection factory class for Sun Microsystems' Java Message Queue product is shown below:

```
 package jmsbook.jms
public class SunJmsTopicConnectionFactoryImpl
implements JmsTopicConnectionFactory {
 public javax.jms.TopicConnectionFactory
getTopicConnectionFactory(java.util.Properties
   props)
throws javax.jms.JMSException {
        javax.jms.TopicConnectionFactory factory =
        new com.sun.messaging.TopicConnectionFactory();
                return(factory);
      }
}
```

The JMS protocol handler class is configurable through a properties file, the name of which is specified in the jmsbook.jms.propertiesFile system property. An example is shown below:

```
java -
Djmsbook.jms.propertiesFile=C:/temp/jmsProtocol.
properties
. . . the rest of the java command
```

The properties file must have two properties, JmsQueueConnectionFactory and JmsTopicConnection-

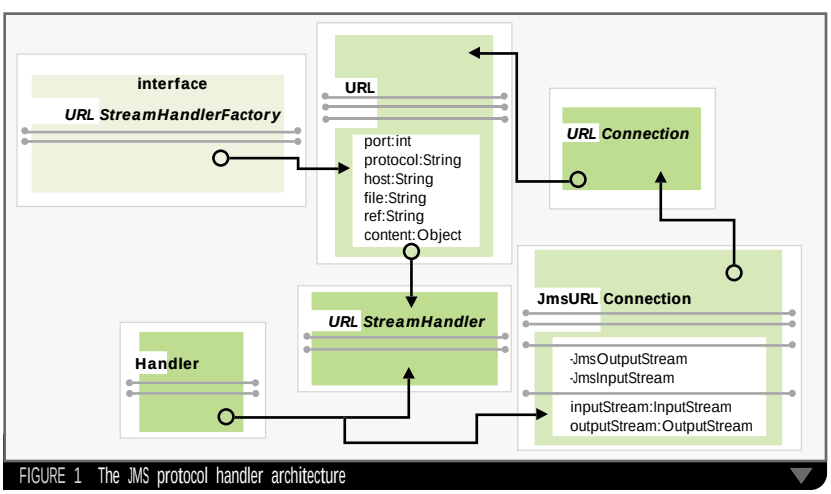


FIGURE 1 The JMS protocol handler architecture

Factory, that specify which queue and topic connection factory classes to use. The properties file may contain any other properties in addition to these, such as JMS provider-specific properties that may be used by the connection factory class implementations. The handler loads the properties from the properties file, creates a new instance of the specified factory class, and calls the appropriate get method on the factory, passing the entire properties collection to the method as its parameter. As a result, this method is able to gain access to any JMS provider–specific properties defined in the properties file. A sample properties file for configuring the JMS protocol handler to use Sun Microsystems' Java Message Queue product is shown below:

```
# The factory to use to get the initial Connection Factory
  JmsQueueConnectionFactory=
jmsbook.jms.SunJmsQueueConnectionFactoryImpl
  JmsTopicConnectionFactory=
jmsbook.jms.SunJmsTopicConnectionFactoryImpl

# Optional
# Sun Microsystems' Java Message Queue product specific
properties
.
.
.
```

Once the handler has references to the vendor's connection factories, it can get the queue and topic connections from these as specified by the JMS.

The JmsURLConnection Class

As you've seen before, the openConnection method in the JMS protocol handler class returns an instance of the JmsURLConnection class. This class extends the URLConnection class and overrides the set/getRequestProperty, getInputStream, getOutputStream, and connect methods. I'll discuss each one of these methods next.

The set/getRequestProperty Methods

Messages in JMS are associated with a delivery mode, a priority, and a time-to-live. The JmsURLConnection class provides default values for three "request" properties and allows the client (e.g., user) to configure these properties at any point in time. The default values of these properties are shown below:

```
// The delivery mode; default is persistent.
 private int persistence = DeliveryMode.PERSISTENT;
// The priority; default is 9 (highest)
private int priority = 9;
// The time-to-live; default is 0 (forever)
private int ttl = 0;
```

A client can get the value of any of these properties at any time by calling the getRequestProperty method and passing in the name of the property required. An example of querying the connection for the delivery mode is shown below:

```
// uc is a JmsURLConnection
String persistent = uc.getRequestProperty("persistent");
```

In the above code fragment, if the value of persistent is "true" after the statement is executed, then the delivery mode is persistent. Note the name of the persistence property is *persistent*. Similarly, to find out the priority and time-to-live properties, call the getRequestProperty method with the property names "priority" and "timeToLive", respectively.

To change the value of any of these properties use the setRequestProperty method. For example, to change the delivery mode to nonpersistent, change the priority to 4, and change the time-to-live to 10 seconds:

```
setRequestProperty("persistent","false");
setRequestProperty("priority","4");
setRequestProperty("timeToLive","10000");
```

If an invalid property name or value is passed in to either of these methods, a RuntimeException will be thrown.

The Connect Method

All this method does is set the connected member variable in the base class to true.

```
public void connect() throws IOException {
  this.connected = true;
}
```

Remember, all the connection-related work has already been performed in the handler class, so we don't need to do anything here.

The getInputStream Method

This method checks to see if the JMS connection it has is a queue or a topic connection. Based on this it creates the appropriate session and message consumer. It then creates a new instance of the JmsInputStream class, passing in the session and the message consumer. JmsInputStream is a private class that extends the java.io.InputStream class and provides a concrete implementation of the read method. When the client calls the read method's (or a method that results in the read method's being called, such as readUTF, readDouble, or any other <read> method on any decorating input stream), it first checks if there are any more bytes in the buffer. If no more bytes exist, it calls the private method readMessage. This method checks if there are any more bytes remaining in a previous (partially read) message. If no such message exists, it calls the receive method on the message consumer that was passed in during construction.

The getOutputStream Method

This method checks to see if the JMS connection it has is a queue or a topic connection and then, based on this, creates the appropriate session and message producer. It then creates a new instance of the JmsOutputStream class, passing in the session and the message producer. JmsOutputStream is a private class that extends the java.io.OutputStream class and provides a concrete implementation of the write method.

A client then calls the write method (or a method that results in the write method's being called, such as writeUTF, writeDouble, or any other <write> method on any decorating output stream). A client can call *write* as many times as needed. Finally, when the entire message is written, the client must call *flush* to actually send the message. The flush method will call the private writeMessage method that contains all the logic for dealing with the message producer.

For example, let's assume that a message consists of some basic information about a person, such as name, age, and sex. Such a message could be sent as shown in Listing 4.

A client could receive this message as shown in Listing 5.

Summary

Probably one of the most significant advancements in software engineering in the past few years has been the widespread acceptance of

component-based programming and design by contract. I've absolutely bought into these concepts and always emphasize designing your enterprise applications without relying on or getting married to a specific JMS provider by remaining loyal to the JMS specification. In this article I've taken the design-by-contract concept one step further by creating a facade that allows you to decouple your application not only from specific JMS providers, but from JMS itself by using Java's protocol handler architecture. Additional (and by no means minor) benefits of this technique include a reduced learning curve for most Java developers, since they're already familiar with using URLs and the ability to leverage a well-designed and time-tested architecture of the Java platform.

Reference

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Author Bio

Tarak Modi, a certified Java programmer, is a lead systems architect at Online Insight where he's responsible for setting, directing, and implementing the vision and strategy of the company's product line from a technical and architectural perspective. Tarak has worked with Java, C++, and technologies such as EJB, Corba, and DCOM, and holds a BS in EE, an MS in computer engineering, and an MBA with a concentration in IS.

tmodi@att.net

Listing 1

```
// create a new URL with our custom "jms" protocol.
URL url = new URL("jms://Queue/ModiQueue");
 URLConnection uc = url.openConnection();

// Send a message
 DataOutputStream dos =
    new DataOutputStream(uc.getOutputStream());
 dos.writeUTF("Hello!");
 dos.flush();

// Receive the message
DataInputStream dis = new DataInputStream(uc.getInputStream());
String message = dis.readUTF();

// close the streams.
dos.close();
dis.close();
```

Listing 2

```
// Get the list of package prefixes
// protocolPathProp has been defined as
// "java.protocol.handler.pkgs"
String packagePrefixList = null;
 PackagePrefixList = (String)
  java.security.AccessController.doPrivileged(
   new sun.security.action.GetPropertyAction(
    protocolPathProp,""));

 // Add the standard protocols package to the list!
 // First, if any package prefixes were found, append
 // another delimiter, "|", to the end.
 if (packagePrefixList != "") {
     packagePrefixList += "|";
 }

 // and now append "sun.net.www.protocol" to the end.
 // Important:
 // Since this package is appended at the end of user
 // specified packages, a user can override any of the
 // Sun provided protocol handler implementations, such
 // as the one for http.
 packagePrefixList += "sun.net.www.protocol";

 // And now parse through the list…
 // Remember, "|" is the delimiter.
 StringTokenizer packagePrefixIter =
     new StringTokenizer(packagePrefixList, "|");

 // Keep going until either we get a handler or
 // no more tokens remain.
 // Note that there will always be at least
 // one token, sun.net.www.protocol.
 while (handler == null && packagePrefixIter.hasMoreTokens()) {

  // Get the next token
  String packagePrefix =
   packagePrefixIter.nextToken().trim();
  try {
   // Create the fully qualified class name.
    // Eg. jmsbook + jms + ".Handler"
           String clsName = packagePrefix + "." +
           protocol + ".Handler";

  Class cls = null;
   try {
 // Now try loading the class with that name.
                                  clsClass.forName(clsName);
    }
catch (ClassNotFoundException e) {
    ClassLoader cl =
       ClassLoader.getSystemClassLoader();
   if (cl != null) {
      cls = cl.loadClass(clsName);
     }
    }
    if (cls != null) {
 // create a new instance.
     handler  = (URLStreamHandler)cls.newInstance();
    }
}
catch (Exception e) {
    // any number of exceptions can get thrown here
    // move onto the next token…
  }

} // while loop.
```

Listing 3

```
// This is the static hash table used
// to cache the protocol handlers.
// All access to this table must be synchronized.
static Hashtable handlers = new Hashtable();

static synchronized URLStreamHandler getURLStreamHandler(
  String protocol) {

// Have we already resolved this protocol?
URLStreamHandler handler =
  (URLStreamHandler)handlers.get(protocol);
// Maybe not…
if (handler == null) {
// Use the factory (if any)
// We will not consider this case.
// In a nutshell, a factory implements the
// URLStreamHandlerFactory interface and is
// registered with the URL instance either during
// construction or using the
// setURLStreamHandlerFactory method.
// A factory can only be set once and similar to the
// protocol handlers is shared by all URL instances.
if (factory != null) {
   handler =
    factory.createURLStreamHandler(protocol);
}

// still don't have a handler…
if (handler == null) {
// All the logic to
resolve a protocol
// string to a protocol handler.
// Plug in the implementation that
// we saw above here
    }

// Cache the handler if one was found.
           if (handler != null) {
           handlers.put(protocol, handler);
           }
  }

// Return the handler to the caller.
return handler;
    }
```

Listing 4

```
// uc is a JmsURLConnection
// Wrap/Decorate the JmsOutputStream with a DataOutputStream
 DataOutpputStream dos =
   new DataOutputStream(uc.getOutputStream());
// Write the name (string), sex (string)
// and age (long) to the stream.
 dos.writeUTF(name);
 dos.writeUTF(sex);
 dos.writeLong(age);
// send the message.
 dos.flush();
// done
 dos.close();
```

Listing 5

```
// uc is a JmsURLConnection
// Wrap/Decorate the JmsInputStream with a DataInputStream
 DataInpputStream dis =
   new DataInputStream(uc.getInputStream());
// Read the name (string), sex (string)
// and age (long) from the stream.
 String name = dos.readUTF(name);
 String sex = dos.readUTF(sex);
 Long age = dos.readLong(age);
 // done
 dis.close();
```

Download the Code! www.JavaDevelopersJournal.com

Beyond the JMS Specification

Real-world issues for large-scale B2B deployments

WRITTEN BY
DAVID CHAPPELL &
BILL CULLEN

The Java Message Service (JMS) is a specification put forth by Sun to define a common set of APIs and common semantics for messaging-oriented middleware providers. An increasing number of MOM vendors have embraced this specification, and new vendors are building messaging products suitable for doing business-to-business communication across the Internet.

Part 3 of 3

The result is a landscape where developers can feel comfortable about writing an application using a standard set of APIs while still having an ample selection of JMS-compliant vendors to choose from. However, the JMS specification is intentionally agnostic when it comes to deployment architectures. Today's B2B environment requires much more than is commonly dictated by the specification, and all JMS vendors have been racing to the finish line to build the additional infrastructure to keep pace with the demands of large-scale e-business messaging deployment over the Internet.

In this article we'll discuss some of the real-world issues beyond the JMS specification. In particular we'll focus on large-scale B2B deployments, and explain how the "beyond JMS" capabilities are a key component of the Commerce One e-marketplace solution.

AUTHOR BIOS

Dave Chappell is chief technology evangelist for Sonic Software's SonicMQ, and coauthor of O'Reilly's Java Message Service.

Bill Cullen has been developing and managing software projects for over 18 years. As director of software engineering for Sonic Software, he plays a key role in engineering the architecture and performance for SonicMQ, one of the first products to implement the Java Message Service specification.

About Commerce One…

Commerce One, through its software, services, and Global Trading Web of interconnected business communities, enables worldwide commerce on the Internet. They are the leading force behind the Global Trading Web, the world's largest business-to-business global trading community, connecting thousands of e-marketplaces around the globe. Through Commerce One.net, Commerce One provides the technology and business services that allow these global trading partners to work together sucessfully.

Commerce One's solutions include:

- ***Enterprise Buyer:*** An e-procurement application
- ***MarketSite:*** Assists Internet market makers in building open marketplaces and linking them to the Global Trading Web
- ***MarketSet:*** Based on the MarketSite platform; provides a specific set of infrastructure and applications for the direct goods and supply chain management market segments

The JMS Specification

The JMS spec prescribes a clear set of rules of engagement between an application and an enterprise messaging middleware. They include:

- Standard APIs for establishing connections, and the sending and receiving of messages
- Loosely coupled asynchronous communication between applications or between components of applications
- A publish-and-subscribe model for a one-to-many broadcast of information, and a point-to-point queuing model for establishing many-to-one or a one-to-one conversational pipe between distinct endpoints
- Quality of service options for message delivery, including once-and-only-once guaranteed delivery, at-most-once delivery, all-or-nothing transactional grouping of messages, and guaranteed ordering; determined by a strict set of rules governing message persistence and store-and-forward capabilities, which are held together by a well-defined set of message acknowledgement semantics
- Failure conditions, error handling, crash recovery, message redelivery
- A broad range of message types capable of transporting any kind of application data in a convenient fashion, and a rich set of methods for constructing and deconstructing messages on either side of a conversation

Beyond JMS: Requirements of E-Business Messaging

Traditional MOM vendors built products intended to carry data between a relatively small set of applications within the four walls of an enterprise. However, if you consider the kind of messaging infrastructure that is capable of supporting the likes of the Commerce One e-marketplace solution, a whole new set of requirements is necessary. Consider these requirements in the context of the diagram shown in Figure 1:

- ***Massive scalability:*** Concurrently connecting tens of thousands of trading partners in a single trading exchange, and connecting multiple trading exchanges together
- ***Fault tolerance and high availability***
- ***Geographically dispersed applications:*** Using the Internet as a means of communicating in a secure and reliable fashion; being secure and firewall-friendly via the use of Secure Sockets Layer (SSL) and HTTP protocols
- ***End-to-end security:*** Through the use of PKI, digital certificates, mutual authentication, Access Control Lists (ACL)

- ***Segregation of application domains:*** Protecting data that is private to an application domain while still being able to selectively expose message queues to the other participants across the domains; this is key, whether communicating between geographically dispersed locations within a single company or between trading partners participating in a trading exchange
- ***Simple and flexible deployment configurations:*** Separation of physical deployment topology from the client applications that use the messaging system
- ***Server-to-server–based architecture:*** Minimum number of "moving parts" or processes reduces number of failure points and network hops; preferable over an IP multicast architecture when conducting business across corporate boundaries. Considering the number of gateways or bridges that need to be installed/maintained in order to move across firewalls and network router boundaries, the perceived benefits of IP multicasting diminish quickly

Commerce One chose SonicMQ as the way to connect these pieces. The basic benefits of JMS, combined with the e-business messaging capabilities of the SonicMQ, allowed them to delegate the responsibility of scalable, secure, guaranteed delivery of data to a best-of-breed JMS provider suitable for the task.

The key enabling technology that fulfills the e-business messaging requirements of this equation is referred to as *dynamic routing architecture* (DRA)

Massive Scalability via DRA

DRA is a messaging server technology that allows increased message volumes to be handled as needed without reconfiguring application programs or requiring significant administrative overhead. The DRA approach eliminates the need to accommodate topology changes and connectivity issues by dynamically adjusting as needed to support changes in messaging configuration. Additional message servers may be added transparently to support additional external connections or to scale up internal systems to handle increased message traffic.

As illustrated in Figure 2, groups of servers, called *clusters*, may connect to other groups of servers as needed, creating highly distributed deployments across loosely coupled locations. The high throughput performance of each JMS server minimizes the number of servers needed in these configurations to handle large messaging volumes.

DRA provides robust, on-demand connection management to eliminate excessive resource use on single servers and the corresponding communication infrastructure. Message senders and receivers may be distributed dynamically across groups of servers to fully utilize resources. DRA connections between clusters enable distributed groups of servers to communicate seamlessly when needed. A connection may be established only when there are messages to send, or may be connected continuously. DRA determines the best path for a message destination dynamically and creates a connection if necessary. As connections are made, messaging destinations dynamically become available throughout the DRA system.

Hardware and communication failures are also handled by DRA, with failover for connections and transparent store-and-forward capabilities for destinations that are temporarily unavailable. The features listed above allow messaging servers to be deployed with maximum efficiency while maintaining high availability.

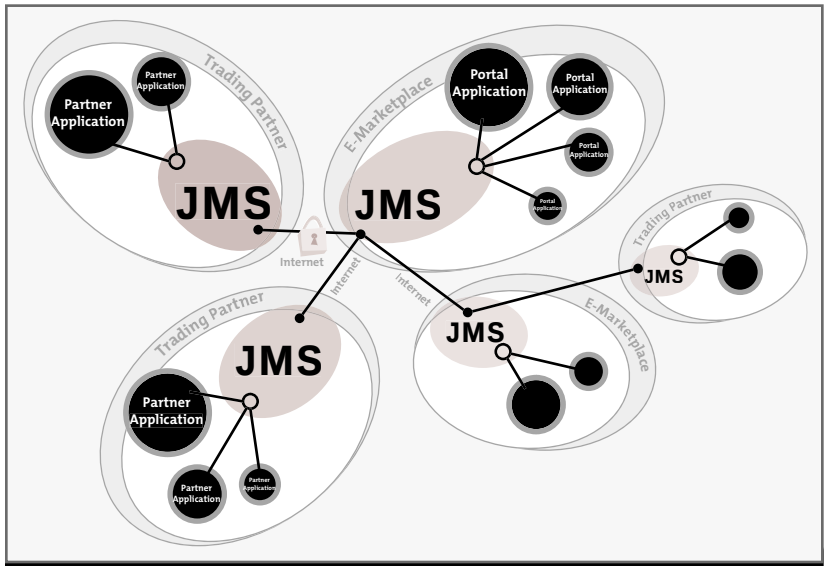


FIGURE 1 Trading partners participating in a supply chain through an e-marketplace

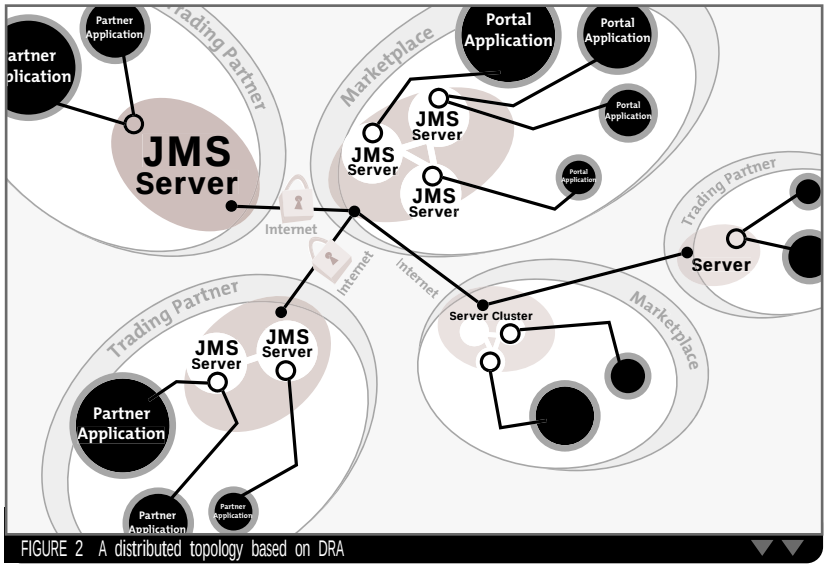


FIGURE 2 A distributed topology based on DRA

Parallel Cluster Technology

Parallel clustering allows clusters of one or more servers to be created as needed within the messaging infrastructure. Clusters allow multiple servers to support as many JMS connections as necessary, while providing transparent cluster-wide access to messaging destinations (see Figure 3).

The clustering architecture of DRA minimizes overhead through a loosely coupled design. Topology and destination changes are broadcast in parallel with participating clusters and servers, minimizing overhead as routing paths change dynamically.

Minimal administrative overhead within and between DRA clusters results in the full utilization of server capacity for messaging tasks. Communication between servers is predominantly used to deliver messages, providing predictable, linear scalability as servers are added to the cluster.

Active Route Optimization

DRA features active route optimization, which allows named destinations to be reached within the system by message senders regardless of connection and topology changes. The ability to actively locate destinations by name eliminates the need to reconfigure application code as messaging servers are changed or scaled to higher volumes.

Active route optimization allows clients to be distributed across multiple servers and reach message destinations dynamically from the server they're connected to, without configuration. To handle additional connections for an application, a server may be added at any location in a distributed system, which is immediately used to handle additional client traffic (see Figure 4).

Message routing is a key element of the DRA server cluster technology. Through routing, when a message destination is added, it's immediately accessible by all messaging clients regardless of the machine within the cluster they're currently connected to. For example, a new incoming connection from a trading partner can immediately be used throughout the system to reach message destinations at that trading partner. Destinations within a firewall or across the Internet are immediately available throughout the cluster once they're connected with DRA.

As illustrated in Figure 5, active route optimization is used to route a request for a quote to an e-marketplace pricing application, which then creates a message destined for a supplier's quote application. The messages are delivered through the optimal path to their destination.

Active route optimization enables applications to be run and replicated as needed. Replication allows application services to be added to support increased application traffic. Within a cluster, duplicate destination names on different servers may be created to replicate application services on multiple machines and achieve high scalability. Messages will be routed to the nearest application service when received at a server (see Figure 6).

Application services may be configured to receive messages from multiple messaging servers, allowing additional load on one message server to be handled on demand by multiple application services. On-demand load balancing ensures that no application waits for work while any JMS server has messages available. On-demand load balancing also enhances system resiliency; application services continue operation in the event of a message server machine failure, and message servers can continue to handle clients if an application machine fails.

DRA allows messaging servers and application services to be deployed in the topology most suitable to achieve the performance and resiliency necessary for high-volume applications.

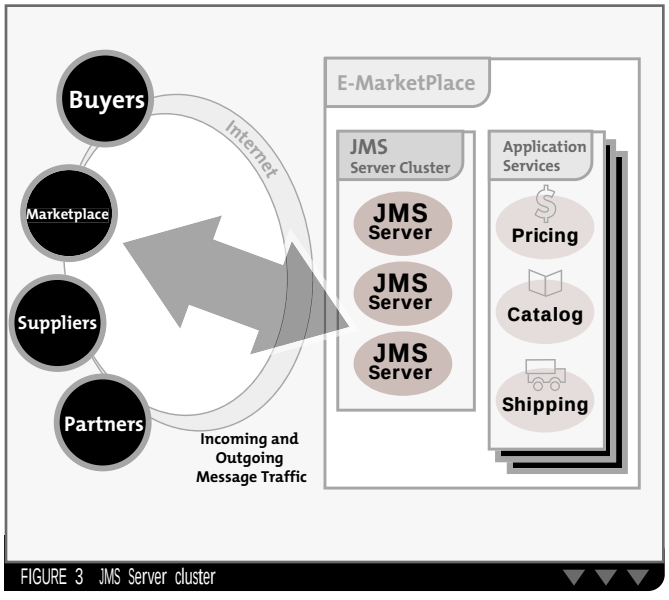


FIGURE 3 JMS Server cluster

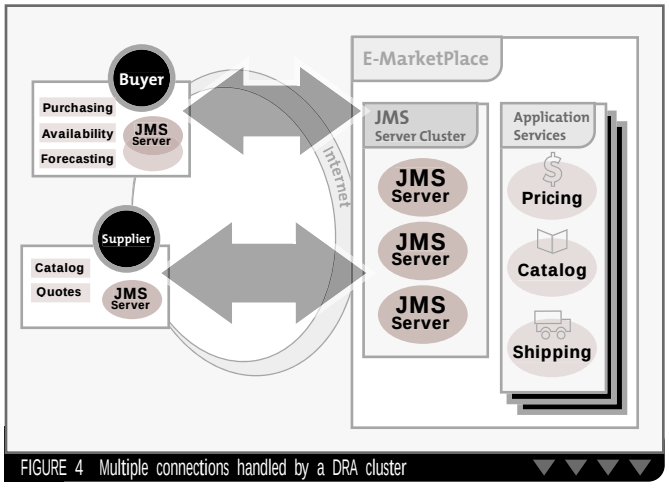


FIGURE 4 Multiple connections handled by a DRA cluster

Multiple Cluster Configuration and Naming

Connections between servers allow all destinations within a cluster to be reached from other clusters. Each cluster is given a routing node name that uniquely identifies it within the rest of a DRA-based system. Destinations are named in a manner similar to e-mail addresses.

Configurations with tens of thousands of clusters can be supported within a single naming system using DRA. Unique destinations across a routing connection may be found by specifying a routing node name and a destination name, allowing for unique names to be created and reached on a global scale. Destinations specified without a node name are resolved within the cluster to allow for simple local operation, while node names are required to reach destinations across cluster-to-cluster connections. Duplicate destination names in different clusters don't conflict, and may be reached individually by specifying the appropriate node name.

Internet Connection Management

To maximize server use, DRA supports load balancing for incoming client as well as server connections. Load balancing spreads incoming connections across the servers in a cluster after an initial connection is made. For resiliency in the case of a server failure, load-balanced connections may have multiple initial points of connection and support a failover reconnect to other servers should a connection be lost.

Connections in DRA are made on demand for messaging between servers, and can be made from either side of a routing connection when needed to deliver messages. An inactivity timeout can be specified for a connection to either maintain or avoid a long-duration connection when no messages are being sent.

When a connection can't be made to a destination, messages are stored in the sending server until forwarded for delivery. The store-and-forward capability allows guaranteed messaging to continue despite short- or long-term interruptions in connectivity. Messages saved for forwarding may be persisted to disk when large volumes must be retained. When a cluster is able to establish a connection to a destination, all messages will be delivered to the newly accessible location, regardless of which server connected to the destination last, which side initiated the connection, or where messages were stored.

End-to-End Security

Server and cluster-level security allows full access control between groups of users and destinations in the DRA architecture. A company may provide a trading partner with access to any selected part of its messaging system and be assured that other aspects of the system are fully protected. DRA ensures the isolation of security domains from each other by preventing information about DRA clusters from reaching unwanted destinations.

Mutual authentication of clients and other DRA clusters is handled by a choice of authentication technologies to guarantee the identity of incoming connections; full privacy is supported through the use of SSL and payload encryption features. DRA provides a system-wide PKI digital certificate capability that fully secures trading partner relationships. Through secure DRA, JMS servers may be employed both inside and outside firewalls with HTTP tunneling capability and forward and reverse proxy support.

Administration

A cluster of servers may be managed as a single entity, allowing user information, routing connections, and security settings to be administered with a single operation. Multiple clusters may be managed using a single-configuration database, and from a single administrative console screen. Notifications of system events are available to assist in the management of distributed operation. All administrative functions, including notifications, are also available from a programmatic interface.

Servers maintain full configuration information locally, eliminating the need for server interaction when handling messages. Local configuration allows each server to function even if centralized configuration information is temporarily unreachable.

Exception Handling Beyond the JMS Spec: Dead Message Queue

For each server, a dead message queue (DMQ) is used as the ultimate destination for any message not delivered in the DRA system. A message that can't be routed due to changing configuration will end up in this queue. Messages that exceed their time to live can also be placed in the DMQ. Destinations that aren't reachable within predetermined time limits may have pending messages added to the DMQ. The DMQ is accessible to applications as a normal queue and provides notifications as messages are added to it. A message in the DMQ will remain intact, including its intended destination and a code explaining why it arrived in the DMQ. A custom application can be written to read messages from the DMQ and decide what to do based on business rules.

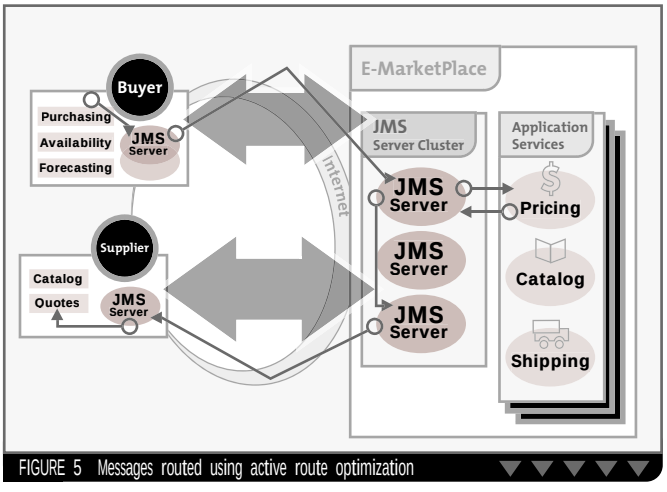


FIGURE 5 Messages routed using active route optimization

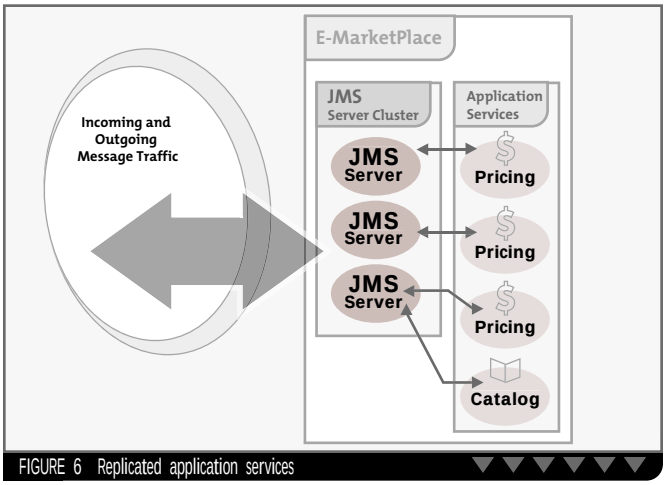


FIGURE 6 Replicated application services

Conclusion

JMS, combined with the proper deployment architecture, such as dynamic routing architecture, solves today's e-business messaging needs of connecting enterprises in a secure, reliable, and highly scalable fashion.

DRA provides the simplicity and flexibility needed to deploy e-business applications within an infrastructure that can grow and scale. Application functionality may be added as and where needed, with the appropriate level of messaging performance to support it. Dynamic routing ensures that applications are insulated from infrastructure changes, and that new messaging connections may be added at any time.

Resources

1. *SonicMQ*: www.SonicMQ.com
2. *MiddlewareSpectra's article on Commerce One's middleware selection criteria:* www.sonicmq.com/whitepapers/mws.pdf

chappell@progress.com

bcullen@progress.com

universal wrapper for entity beans

a design approach for multitier applications

written by andrei povodyrev and alan askew

This article presents a design approach for multitier applications implemented with Enterprise JavaBeans. These entity EJBs inherit bulk set-and-get methods from a single parent class that takes advantage of the java.lang.reflect package. This approach reduces the number of network round-trips, simplifies application maintenance, and significantly reduces the lines of code in an EJB application.

Introduction

Any funky new Web site, no matter how appealing its dynamic introduction or its promise of a more efficient, exciting, and communal lifestyle, must culminate eventually in "the form." Internet users surf new sites with a subconscious dread of "the form," that page where, in spite of 21st-century technology, you must still type in last name, first name, middle initial, address, and so on to receive mailings.

Distributed Web application developers share this dread, hoping to avoid the coding ennui that accompanies such pages. Furthermore, developers must contend with the performance and maintenance of the distributed code that handles these pages. While we can't yet help users save the time invested in last name, first name, and middle initial, we'd like to offer relief to application developers trying to efficiently and elegantly move bulk data from EJB clients to their database.

Setup

Let's consider a Web-based EJB architecture in which the user's browser submits HTTP requests to a servlet, which processes these requests via calls to EJBs, which in turn communicates with persistent storage (see Figure 1).

Suppose your application users arrive at a page on which they'll subscribe to a magazine. After they submit the 15 fields your application requires, your servlet will service a request laden with 15 parameters or attributes ready for processing. There are several common ways to write this new data to the database. Each presents its own performance or maintenance problems, all of which our technique eliminates.

Typical Approaches

In the most direct approach your client code can find the appropriate remote interface of your fine-grained entity bean and call a set method for each of the required data fields. Explicit transaction management (Java Transaction API) preserves the transactional integrity of this operation, but each set call on the stub still requires a round-trip network interaction with the skeleton resulting in some latency, a delay between command execution and completion. The more method invocations used to perform a logical piece of work, the greater the latency in the application (see Listing 1, Style 1).

To take advantage of an application server's transaction management, you might hide all the set methods required for this operation behind a single EJB method with an extended parameter list. This approach reduces your network communication to a single round-trip. As form fields on the page change, however, the maintenance of these methods renders this approach frustrating and cumbersome (see Listing 1, Style 2).

Note: This article assumes a thorough understanding of Java and the Enterprise JavaBeans component model and is intended for enterprise application developers.

Custom data classes, whose data members mimic those of an entity bean, prove more elegant than the other approaches on the client side, but rather awkward and repetitive on the EJB side. First, every bean requires an individually tailored class that developers must update and recompile whenever the bean changes. Second, the bean developer must still retrieve each field from the wrapper object and call the corresponding set method on the bean. Thus the client developer calls 15 set methods on the wrapper, then the bean developer calls 15 get methods on the wrapper and 15 set methods on the bean. The code requires 45 get-and-set calls for a bulk update of only 15 fields (see Listing 1, Style 3).

Universal Wrapper Objects: Hashtables

To alleviate the maintenance problems custom classes create, we now consider using hashtables to pass update information to beans. You need not develop a separate wrapper class for each bean, and you can easily identify those data elements the client wished to update, leaving other attributes untouched. However, you must still duplicate attribute calls. The client user would have to use a put call for every attribute, then the bean developer would turn each of those into a pair of get (on the hashtable) and set (on the bean) calls. Considering the dull and systematic task of converting hashtable values with keys such as “lastName” into method calls such as setLastName(), you might begin to anticipate our solution.

EntityBeanServices: The Basics

Any Java class can be dynamically inspected using Reflection, which allows the programmer to obtain information about the fields, constructors, and methods of any class at runtime. Java implements Reflection via the java.lang.reflect package. Any object method can be called dynamically using invoke() on the java.lang.reflect.Method class. We’ve developed an elegant and powerful pair of methods that take advantage of Reflection and automatically translate a hashtable key into a bean method invocation.

Client code assembles a hashtable with all the bean attributes to update, using as hashkeys the actual method names on the entity bean (e.g., “setLastName”). Then they call a single method, setBeanAttributes(), on the entity bean, passing this hashtable of bean attributes as an argument (see Listing 1, Our Approach). In the setBeanAttributes() method, the code examines each of the keys provided in the hashtable, invoking the corresponding method with the key’s paired object as the single argument.

The companion method, getBeanAttributes(), works in a similar way, looping through the bean’s methods, finding all those whose names begin with the conventional “get” substring and putting the return object in the hashtable with a key named for the invoked get method.

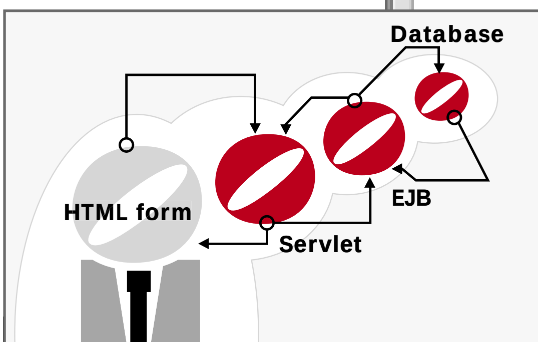


FIGURE 1 Simple workflow for a multitier, EJB-enabled application

Implementation Details

As shown in Figure 2, application architects can easily include these methods in their application design. We write the setBeanAttributes() method implementation in a parent class called EntityBeanServices (EBS) (see Listing 2). Each entity bean extends this class as well as implements javax.ejb.EntityBean, which the EJB specification requires. The remote interfaces can also extend an EntityBeanServicesInt interface (see Listing 3). All entity beans will then have setBeanAttributes() and getBeanAttributes() defined in the remote interface and implemented in the bean class, just as the EJB specification requires. Each entity bean now has access to setBeanAttributes() and getBeanAttributes() without any further coding on the beans (other than the two class extensions). Should you need to update attributes on more than one bean in a single transaction, you can write a session bean method that takes as arguments one hashtable per entity bean. The body of that method then looks up each bean and calls setBeanAttributes() on it, preserving all the set methods inside a single, container-managed transaction.

When you pass a hashtable to setBeanAttributes(), the key value pair consists of the set method’s name and an object intended as the argument to that set method. Even if you use primitive bean attributes, setBeanAttributes() can handle the required manipulation from, for example, an Integer object to an int primitive (in fact, the java.lang.reflect package provides this service).

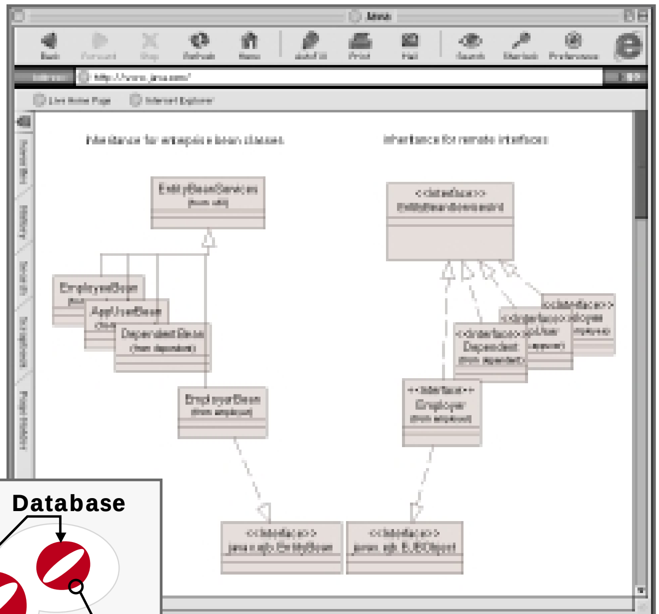


FIGURE 2 Universal wrapper class diagram

The methods of the EntityBeanServices class inherited by a particular enterprise bean uses the default isolation level and transactional attribute declared in the bean’s deployment descriptor. For example, if the default transaction attribute is set to TX_REQUIRED, a call to setBeanAttributes() invokes all set methods together in a single transaction. Thus, if a single set method invocation fails in this bulk operation, the entire transaction will be rolled back. Given the flexible nature of the transactional attributes model for EJB, developers can implement specific transactional strategies for both the bean’s own methods as well as those of the parent class.

Extensions of EBS

Most bean developers have faced the following problem posed by the ejbCreate() method. ejbCreate() asks the developer to define (via the

method signature) which data fields the client code must provide to create an entity. As those requirements change, old ejbCreate() definitions must be changed or preserved as obsolete versions. With EntityBeanServices in place, we can call ejbCreate() passing only a hashtable and make a setBeanAttributes() call within that method. In this way we require only a single ejbCreate() method that we never have to change. Of course the developer must be sure to include key/value pairs for all not null attributes.

Another modification to EntityBeanServices will improve the performance of your calls to getBeanAttributes(). Rather than invoke each and every get method on the bean, even when you need only a subset of the attributes returned, you could overload getBeanAttributes(). Provide a second signature that takes as an argument a hashtable with keys specified for those get methods you hope to execute. Should you require only five attributes from the bean, this overloaded form will allow you to execute precisely five methods and no more.

We have considered numerous other features that beans might inherit from this parent class that we hope to share in future articles. We look forward to any extensions of EBS you work out as you incorporate this pattern into your applications.

Drawbacks

While we've found EntityBeanServices an excellent tool for our distributed EJB applications, we should note the price to pay for this convenience, even though we find it relatively small. The greatest inconvenience of EntityBeanServices is the absence of compile-time checking. Since attributes are put into a hashtable with method names as hashkeys, a simple misspelling can cause runtime grief. We've tried to minimize this concern by providing detailed output when throwing exceptions from EntityBeanServices. We offer several likely causes for the runtime error and use the Reflection class to be sure the developer knows which bean the client was trying to update when the error occurred. To improve error handling, we encourage developers to provide the class with a set of custom exceptions.

We also experienced some problems initially when trying to get or set null object references as bean data members. Java 1.1.7 developers can avoid this by creating their own NullObject, which represents a null reference. Java 2 resolves this problem easily with the java.util.HashMap class, accepting key value pairs in which the value is a null object reference. You may also find Java 2 options such as java.util.Vector and java.util.LinkedList more efficient for setBeanAttributes(). Any of these will work as the argument type and eliminate inefficiencies caused by the initial memory allocation and code synchronization of java.util.Hashtable.

Note that changes to EntityBeanServices method signatures require the architect to rebuild all the descendent beans. Since we still experiment with new concepts in EBS, we wrote a simple shell script to rebuild our beans and run it after hours. If you use only the two methods we provide or carefully plan your implementation of EntityBeansServices ahead of time, you shouldn't encounter this problem.

Finally, those of you developing beans for commercial use will have to package EBS along with your beans. This may create minor inconveniences for you.

Conclusion

Our need for transaction-aware bulk updates led us to this bean superclass, which has served us faithfully. We've eliminated some of our developers' tedious coding responsibilities and eased the maintenance required when we augment or prune our entity beans. Although we always hope they won't, data structures may shift and change even deep into the development process, and EntityBeanServices help minimize the impact of these changes. We've controlled the risks associated with runtime errors by providing clear direction to the test developer as to the likely nature of the problem and have found numerous relevant extensions for this class. We encourage you to take this concept, work with it, and share your results and experiences with us.

Acknowlegements

We would like to thank FCG Doghouse for their support; Shaheem Sait for inspiring us to write this article; and Art Solano, Jennifer Terry, and Jonathan Leibundguth for useful comments.

Author Bios

Andrei Povodyrev is a senior software development specialist with FCG-Doghouse, Inc., experienced in the analysis, design, development, and maintenance of business information systems. He has specific expertise in Java, EJB, JSP, PowerBuilder, C++, Sybase, and Oracle systems and is a certified programmer for the Java 1.1 Platform.

Alan Askew is a senior software development specialist with FCG-Doghouse, Inc., and has worked in both management and technical consulting in a variety of industries. He has particular expertise in Java, EJB, JSP, PowerBuilder, and Oracle. He's a certified programmer for the Java 1.1 Platform and has worked through half of his Oracle DBA Certification.

apovodyrev@doghouse.com

aaskew@doghouse.com

Listing 1

```
public class EJBClient{
 public static void main(String [] args){
  try{
   Member ourMember;
   //Put code here to get the remote interface for a particular
     Member object

   //To change Member data before, you might have used one of the
      three styles:

 //Style 1. Calling set methods directly...Requires three network
    round trips
 //javax.transaction UserTransaction  ut = //get the
    UserTransaction
 //ut.begin
   //ourMember.setFirstName("Jane");
   //ourMember.setLastName("Doe");
   //ourMember.setMagazineName("Computer Weekly");
   //ut.commit();

   //Style 2. A long-parameter list set method is inflexible as
     the number of arguments is //fixed
   // ourMember.setSubscriptionInformation("Jane", "Doe",
     "Computer Weekly");

   //Style 3. A set method with a custom wrapper class
     (MemberWrapper)is cumbersome to maintain: changes in
     entity bean will require corresponding changes on wrapper
     class; each entity bean requires its own wrapper
   //MemberWrapper mWrapper = new MemberWrapper();
   //mWrapper.setFirstName("Jane");
   //mWrapper.setLastName("Doe");
   //mWrapper.setMagazineName("Computer Weekly");

   //ourMember.setSubscriptionInformation(mWrapper);

   // OUR APPROACH
   //Using EntityBeanServices, you prepare a hashtable
   java.util.Hashtable ht2 = new java.util.Hashtable();
   ht2.put("setFirstName","Jane");
   ht2.put("setLastName","Doe");
   ht2.put("setMagazineName","Computer Weekly");
   //Then make a single call, uniting the set methods in a single
    transaction. //Universal form for every entity bean in your
    application
   ourMember.setBeanAttributes(ht2);

  } //end try
  catch(Exception e){
   e.printStackTrace();
  } //end catch
 } //end main method
} //end class
```

Listing 2

```
public class EntityBeanServices {

 public void setBeanAttributes(Hashtable ht) throws
java.rmi.RemoteException{

  //invoke set methods based on keys in ht
  String key, name, methodName = null;
  boolean methodNotFound = true;
  Object arglist[] = new Object[1];
  Enumeration keys = ht.keys();

  try{
    name  = getClass().getName();

    Method m[] = getClass().getDeclaredMethods();

    while(keys.hasMoreElements()){
    key =  (String)keys.nextElement();
    methodNotFound = true;
    for(int i = 0; i < m.length; i++){
     methodName = m[i].getName();

// loop through bean's methods to find a match
// some filtering based on common sense is desirable
//match name of the method
    if(key.equalsIgnoreCase(methodName))
//get methods must be public
       if(m[i].getModifiers() == Modifier.PUBLIC)
//set methods have a single argument
    if((m[i].getParameterTypes()).length == 1)
//set methods must not  start with "get"

    if(!methodName.startsWith("get"))
    if(!methodName.startsWith("ejb")) {

            methodNotFound = false;
            arglist[0] = ht.get(key);
//invoke matched method
            m[i].invoke(this, arglist);
         }// end if
    }//end for
    if(methodNotFound) throw new Exception("Attempt to
     invoke a set method " +
         key + " on " + name + " failed. \n"  +
     "   Possible reasons: \n" +
     " 1) it is not a set method;\n " +
     " 2) method has more than 1 argument;\n" +
     " 3) method name does not start with 'set'\n" +
     " 4) no such method in the class" );

   }//end while

  }catch(Throwable e){
   e.printStackTrace();
    throw new java.rmi.RemoteException("Exception is rethrown");}
 }


 public Hashtable getBeanAttributes(){
  // identify and invoke all get methods for this entity bean
  Hashtable ht = new Hashtable();
  Object arglist[] = new Object[0];

  try{
   //get all methods of the EJB
    Method m[] = getClass().getDeclaredMethods();


    // filter get methods and invoke them

   for(int i = 0; i < m.length; i++){
    if((m[i].getParameterTypes()).length != 0)    continue;
 //get methods do not have parameters
    if(m[i].getModifiers() != Modifier.PUBLIC)    continue;
//get methods must be public                      continue;
    if(!m[i].getName().startsWith("get"))
//get methods must start with "get"
    if(m[i].getName().equals("getBeanAttributes")) continue;
//should not be itself

    Object ob = m[i].invoke(this, arglist);
    if (ob != null)
                ht.put(m[i].getName(), ob);

   }
  }catch(Throwable e){
   e.printStackTrace();
  }
  return ht;
 }

}
```

Listing 3

```
/**
* EntityBeanServicesInt is designed as a parent interface for
   EJB remote interfaces.
* Interface declares methods that are implemented in
   EntitybeanServices class
**/
public interface EntityBeanServicesInt {
 public void setBeanAttributes(java.util.Hashtable ht) throws
  java.rmi.RemoteException;
 public java.util.Hashtable getBeanAttributes() throws
  java.rmi.RemoteException;
}
```

Download the Code! www.JavaDevelopersJournal.com

Implementing J2EE Security with WebLogic Server

The benefits and ease of use

Part 2 of 2

WRITTEN BY
JASON WESTRA &
CHRIS SIEMBACK

In the March issue of *JDJ* (Vol. 6, issue 3) we discussed the basics behind J2EE security, including coverage of role-based security for both the Web and EJB tiers. In Part 2, we provide an example of implementing J2EE security in the WebLogic Server.

While this article and the examples contained within are specific to WebLogic 6.0, all of the deployment code and standard descriptors should be portable to any J2EE-compliant server. We won't cover encryption and SSL this month, as they're articles in their own right.

Trader Application

How many securities trading examples have you seen to date? Too many, I bet you'd say. Well, we decided airline reservation systems and "hello, world" programs are way too overused. Even though the stock market is not a hot topic these days, in an effort to spend more time explaining J2EE security and less time talking about business requirements, we opted for the trading program as well.

The application we'll be building represents a stock trading system in which users may buy, sell, and view securities. We'll grant all paying customers the ability to purchase and view securities. However, we'll reserve the ability to sell securities to just traders, who may also view and purchase securities. Basically, traders are the power users of our system! Furthermore, we'll provide a customized login page that can be tailored to the look and feel of our site rather than relying on a basic browser authentication window.

To build our application we've chosen the WebLogic Server 6.0. With WebLogic 6.0, we'll take advantage of features such as a security domain based on a relational database, Web application security, and method-level EJB security.

Implementing J2EE Security with the WebLogic 6.0 Server

There are several things we'll need to accomplish to get our end-to-end application up and running with the security constraints we require. Listed below are the basic steps for creating and implementing this functional example:

1. Configure a security domain
2. Secure the Web tier
3. Secure the EJB tier
4. Demonstrate end-to-end security in a J2EE application

Configure a Security Domain

Before we can secure anything, we must have a security domain that contains our users and groups. WebLogic Server provides several methods to construct and use security domains (or realms). The default WebLogic domain is file-based, storing users, groups, and ACLs (access control lists) for the server in separate properties files. The file-based domain is easy to read and configure in the server, but it's not a scalable solution for an organization with any real user-base. The File Realm solution is designed for systems with 1,000 or fewer users. Furthermore, if the file itself becomes corrupt, you'll need to reconfigure security again. This is unacceptable in an enterprise-level application, which must have a robust, scalable security domain.

WebLogic provides plug-in capabilities for more scalable solutions including LDAP, UNIX, and NT security. In this example, we'll use the RDBMSRealm that comes with the standard WebLogic install. This plug-in domain uses a relational database to store security information. The RDBMSRealm is easy to use, offers configurable caching of security information, and refreshes its data without downtime – features we're particularly interested in exploiting for our production J2EE application. So, how easy is it? Let's see.

Our first step in configuring a security domain is to create the relational tables necessary to store our information. The RDBMSRealm consists of three database tables: USERS, GROUPMEMBERS, and ACLENTRIES. WebLogic provides the DDL (data definition language) necessary to create the tables in a file called rdbmsrealm.ddl. Use the WebLogic utility, utils.Schema, to run the DDL or create the tables by hand. This file will also insert some sample data entries in each of the three tables to ensure that we understand the format of the data that's expected by the domain's tables. *Note:* You may need to modify the scripts to work with your particular database or the default data that's being inserted – but changing table or column names will force you to make some modifications to the schema properties (described later). We advise just using

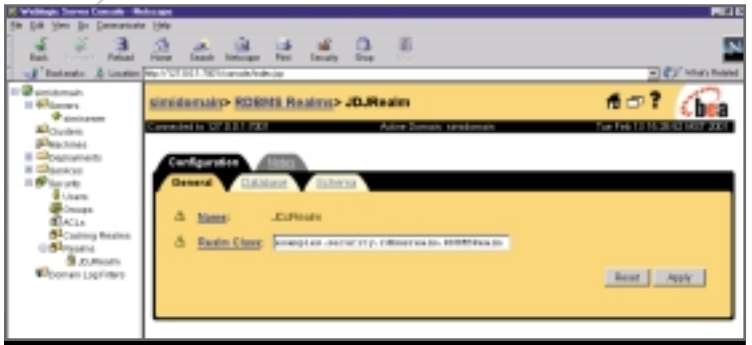


FIGURE 1 WebLogic console

FIGURE 2 Database configuration

the schema provided for your initial look into the security example.

For our example we're creating two groups, onlineinvestor and trader. Once the database tables have been created, we can create and register the domain for use within WebLogic.

After creating our domain tables, we need to ensure our security domain classes are built and available to the WebLogic Server. Again, we're using the example security classes provided by WebLogic. You'll find the security classes in the following directory:

```
/WEBLOGIC_HOME/samples/examples/security/rdbmsrealm
```

They'll need to be compiled and put in the server classpath for the domain to function properly.

Enable the Domain for WebLogic

The RDBMSRealm relies on properties contained within the config.xml file used for your domain for the database connection information and on the SQL statements that will be used with such actions as adding or deleting users and groups. To create the RDBMSRealm, you should launch the WebLogic Server Console. The WebLogic Server Console is a Web-based administrative utility that allows developers to remotely manage nearly all the facets of the WebLogic Server. It's a welcome upgrade from previous versions of this tool. To enable the domain, we simply select the "Realms" node located under "Security" (see Figure 1). You'll need to follow the "Create a new RDBMSRealm" link and enter the appropriate information. For this example, we've chosen to name our domain "JDJDomain" and we've also provided the fully qualified class name of the RDBMSRealm (examples.security.rdbmsrealm.RDBMSRealm).

Now that we have the domain defined, we must provide WebLogic with the database connection information that's required to access the database. To do this, click on the "Database" tab and enter the appropriate information for the connection. An example of this is shown in Figure 2.

After you've entered the appropriate data, click on the "Apply" button. Now that we've constructed our domain and supplied WebLogic with the necessary connection information, we must provide some additional properties that our Domain classes will use to manipulate the data. The schema properties are the values for the PreparedStatements that the RDBMSDelegate uses to make modifications to the data store. After selecting the "Schema" tab and entering the appropriate properties, we can apply them.

Now that we've completed the domain, we need to select the "Caching Realms" and create a new Caching

Realm, selecting the "JDJDomain" that we just created from the drop-down list box. We have the option of modifying the caching properties, which will affect the performance of our security domain.

Last, we need to select the Caching Realm we just created from the drop-down list box under the "Security" node. Once we restart the server, we should be ready to rock and roll!

Securing the Web Tier

With the Servlet 2.2 specification, J2EE application deployers and system administrators have the ability to apply declarative security to Web content. This method of security is portable to any J2EE-compliant server and allows developers to define custom login and error pages. In fact, we can assign custom error pages for specific Java exceptions and HTTP codes (e.g., 404 page not found error) so we can hide the error behind a nice message such as, "We are currently experiencing high volume. If you are a venture capitalist looking to fund us, please forward money to account XYZ. Thank you."

Create Web Deployment Descriptors

To apply security at the Web tier, we edit the Web application's standard, web.xml, and WebLogic's proprietary WebLogic.xml deployment descriptors to define the Web resources, security restrictions, a custom login page, and error page. Both the web.xml and WebLogic.xml files are located within the WEB-INF directory in the HTML document root. Breathe easy, securitymongers.... This directory is not accessible to browsers, even though it's contained within the HTML root.

The web.xml file (see Listing 1) is the standard Web application deployment descriptor. It defines a Web application's Web resources, essentially a collection of pages, and their attached security constraints. Also, we've defined a custom login page when users need to be authenticated. When users first hit a restricted page, the custom login page is presented to them. If the user is authenticated and granted access, the user is automatically brought to the page he or she was originally intended to get to. If authentication fails or the user doesn't have access to the page, then he or she is brought to the form-error page. With this method, developers don't have to explicitly carry and insert the user's credentials or security attributes, a J2EE server such as WebLogic 6.0 handles this functionality for you.

After defining the web.xml file, we need a corresponding WebLogic Web descriptor, called WebLogic.xml appropriately enough. This file is proprietary and is used to link external references such as security principals from the standard web.xml descriptor to our deployment environment in the WebLogic Server. Listing 2 provides an example of the WebLogic.xml file that we'll use. Notice how generic role descriptions in the web.xml such as "role-onlineinvestor" is mapped to onlineinvestor, an actual principal in our security domain. This mapping is expressed visually in Figure 3.

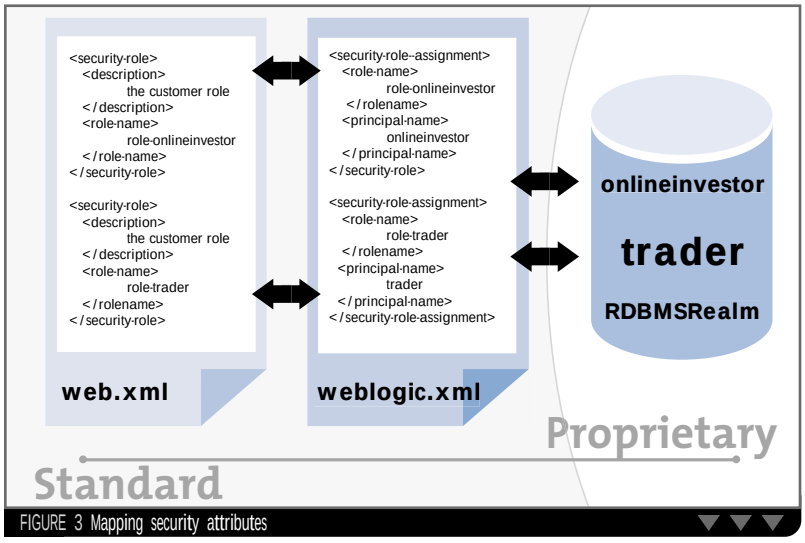


FIGURE 3 Mapping security attributes

METHOD	AUTHORIZED ROLE	DESCRIPTION
getSecurities	everyone	This method returns a list of securities
buy	onlineinvestor trader	Returns a String upon successful completion of purchasing a security
sell	trader	Returns a String upon the successful completion of selling a security

TABLE 1 Security resource to role mappings

EXCEPTION	DESCRIPTION
javax.naming.AuthenticationException	AuthenticationException is thrown if the username and password of the individual cannot be found in the security domain.
java.rmi.RemoteException	A RemoteException is thrown if the user exists, but doesn't match the security requirements for the method he or she is attempting to access. You can check the message in the exception to verify that a security issue was the cause.

TABLE 2 WebLogic security exception details

Construct a Custom Login Page

Now that we have our Web security defined in our descriptors, we need to construct the login.jsp page we defined in the web.xml file. This file allows us to construct a login page that doesn't rely on the browser's native authentication method and enables us to design the page to look like the rest of our site. The requirements for this page are explained in detail in the Servlet 2.2 specification, but let's briefly cover the basic ones here.

Listing 3 is the most basic login JSP we can create. In this page, we have a form that will send user credentials to the server. The form ACTION attribute must be set to "j_security_check". We also require two text fields to contain the username and password. They're required to have the attribute value of NAME equal to "j_username" and "j_password" respectively. These naming conventions allow the Web container that's servicing an authentication request to handle it generically.

The login.jsp works like this: when users first attempt to access a restricted Web resource, the request is sent to the container, which stores the URL of the request and sends back the login form. The user then inputs his or her credentials and submits the form back to the container. If the user has valid permissions, he or she is brought to the URL that was originally requested. If the login fails, the user is brought to the login-error page.

Securing the EJB Tier

Now that we have a working security domain and have successfully secured our Web tier, let's add security to our TraderEJB in the EJB tier. Even though we have Web-tier security, since EJBs are remote objects they may be accessed from clients other than a Web server. For this purpose, we want to also apply restrictions on our EJB methods.

To do so we can modify our TraderEJB component to implement the security restrictions specified by our application requirements.

J2EE security is designed as a container-based service and set at the thread level, which enables the context to be propagated to other components on subsequent calls. This feature simplifies the security of J2EE applications by implicitly passing the security context of the user to each component without requiring the developer to code security logic. Security constraints can be added, removed, or updated without affecting the components that utilize the container's security services. Since J2EE security is declarative, we need only update XML descriptors to secure the TraderEJB.

Table 1 describes the security restrictions placed on the TraderEJB's methods in the XML descriptors. For demonstration purposes, these methods are hard-coded; however, in a real application, these methods would probably access a real-time quote feed for securities, and store shares bought and sold with JDBC or entity beans.

The Web tier doesn't have to authenticate a user until a secured Web resource is requested. At some point the Web tier may contact an EJB even if a user has never been established; however, the EJB tier must have a user. To solve this dilemma, the J2EE specification requires that a J2EE product supply a principal in place of the empty user. WebLogic defines the group *everyone* and the users *guest* and *system* to support this functionality. Because they're inherent in WebLogic, they don't need to be defined in the RDBMSRealm.

The ejb-jar.xml file (see Listing 4) is where we assign permissions to methods in the EJB. From within the method-permission tag we define which roles have access to which methods. For example, we've added the role-onlineinvestor and role-trader (which map to the onlineinvestor and trader groups, respectively, in the domain) to the buy method. You may add as many groups as necessary to the method's permission. For the sell method, we've allowed only the *trader* group. The last method, getSecurities, provides permission to the implicit *everyone* group. Remember that J2EE security allows access to restricted methods as long as the user is in at least one of the groups defined.

The WebLogic-ejb-jar.xml file (see Listing 5) maps the actual group/user name contained within the domain to a role name that's used within the XML files. Here we're assigning a role-name, which will be used in the ejb-jar.xml, to the actual principal name contained within the database domain. We're currently mapping the trader group to the role-name *role-trader*. The same is true for the onlineinvestor and everyone.

How Did We Do?

We now believe we have a functional application that meets the security requirements stated. Let's test our application to be sure. Where do we begin? Since we defined in the web.xml that only the /trade/index.jsp page should have security, we should be able to freely access any other portions of our site without being prompted for authentication. It's pretty simple testing security when there really isn't any!

Now, let's attempt to access resources that are reserved for online investors and traders. Since we've secured access to the resources in the trade directory, we should

Author Bios

Jason Westra is CTO at Verge Technologies Group, Inc., a Java consulting firm specializing in e-business solutions with Enterprise JavaBeans.

Chris Siemback, an Enterprise Java consultant at Verge Technologies Group, Inc., specializes in Web-based EJB applications, various development methodologies, and distributed architectures.

get sent to the custom login page we created when we access these pages. Enter the credentials for an onlineinvestor, and we should get forwarded back to the page we were attempting to access in the first place. Now, we should be able to list and purchase securities. So far, so good.

Last, let's attempt to access these resources as a trader. To be safe, let's close our browser and then access the restricted pages again, logging in as a trader this time. We should now be able to perform all three of the EJB methods that we've secured, including selling securities. *Note:* If we had originally logged in as a trader, we would not have been prompted again unless we tried to access resources traders don't have access to.

Understanding WebLogic's Security Exceptions

As stated in Part 1, there are two steps involved when the user attempts to access a controlled resource, authentication, and authorization. First, WebLogic authenticates the user by determining whether the specific user even exists in the domain. If the user isn't present, an AuthenticationException is thrown. If the user exists, WebLogic checks the permissions to determine if the user has authorization to access a service. If the user is authorized, the service is accessible and methods may be executed. Table 2 lists the possible exceptions that can be thrown when checking security access.

Conclusion

This month we took the basic J2EE security knowledge we learned in our last column and applied it toward a working example on the WebLogic Server's implementation. Specifically, our example was based on WebLogic's RDBMSRealm, a security domain that utilizes a relational database to store its information. We hope this month's **EJB Home** improved your comfort level with J2EE security and enlightened you on its benefits and ease of use when applying it through the WebLogic Server.

westra@sys-con.com

csiemback@vergecorp.com

Listing 1: web.xml

```
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>TradeApp</web-resource-name>
            <url-pattern>/trade/*</url-pattern>
      <http-method>POST</http-method>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>role-onlineinvestor</role-name>

    </auth-constraint>
  </security-constraint>

   <login-config>
        <auth-method>FORM</auth-method>
        <domain-name>JDJDomain</domain-name>
    <form-login-config>
                  <form-login-page>/jsp/login.jsp</
                    form-login-page>
                  <form-error-page>/jsp/loginerror.jsp</
                    form-error-page>
    </form-login-config>
  </login-config>

  <security-role>
        <description>the customer role</description>
        <role-name>role-onlineinvestor</role-name>
  </security-role>

  <security-role>
        <description>the customer role</description>
        <role-name>role-trader</role-name>
  </security-role>
```

Listing 2: WebLogic.xml

```
<security-role-assignment>
       <role-name>role-onlineinvestor</role-name>
       <principal-name>onlineinvestor</principal-name>
     </security-role-assignment>

     <security-role-assignment>
       <role-name>role-trader</role-name>
       <principal-name>trader</principal-name>
     </security-role-assignment>
```

Listing 3: logic.jsp

```
<HTML>
<BODY>
The page you're attempting to access is restricted, please login:
<BR>
<FORM METHOD="post" ACTION="j_security_check">
Username: <INPUT TYPE ="text" NAME="j_username"><BR>
Password: <INPUT TYPE ="password" NAME="j_password"><BR>
<P><INPUT TYPE="Submit" NAME="Submit" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

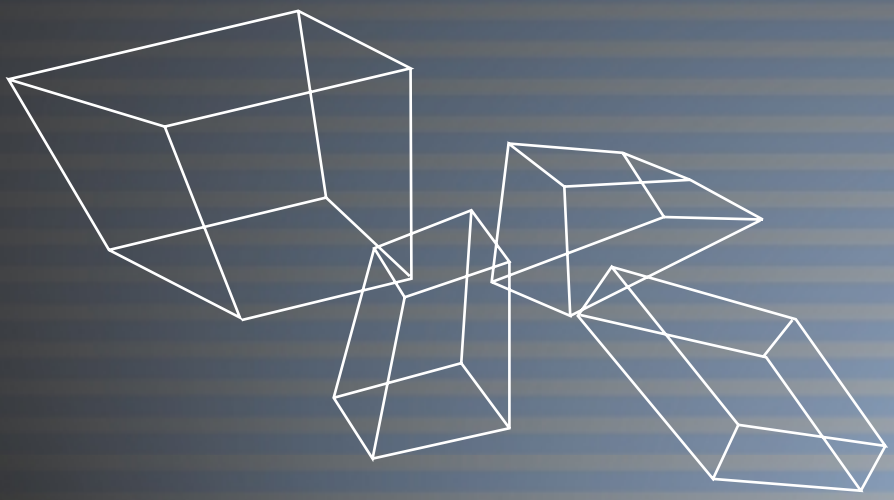
Listing 4: WebLogic-ejb-jar.xml

```
<security-role-assignment>
        <role-name>role-onlineinvestor</role-name>
        <principal-name>onlineinvestor</principal-name>
  </security-role-assignment>

  <security-role-assignment>
        <role-name>role-trader</role-name>
        <principal-name>trader</principal-name>
  </security-role-assignment>

  <security-role-assignment>
        <role-name>role-everyone</role-name>
        <principal-name>everyone</principal-name>
  </security-role-assignment>
```

Listing 5: ejb-jar.xml

```
<assembly-descriptor>
    <security-role>
        <description>Investor in the application</description>
        <role-name>role-onlineinvestor</role-name>
    </security-role>
    <security-role>
        <description>A stock broker, or trader</description>
        <role-name>role-trader</role-name>
    </security-role>
    <security-role>
        <description>Anyone in the RDBMSDomain</description>
        <role-name>role-everyone</role-name>
    </security-role>
    <method-permission>
        <description>
        This permission gives the right to purchase shares.
        </description>
        <role-name>role-onlineinvestor</role-name>
        <role-name>role-trader</role-name>
        <method>
              <ejb-name>jdj.security.SecureTradeMgr</ejb-name>
              <method-name>buy</method-name>
        </method>
    </method-permission>
    <method-permission>
        <description>
        This permission gives the right to sell shares.
        </description>
        <role-name>role-trader</role-name>
        <method>
              <ejb-name>jdj.security.SecureTradeMgr</ejb-name>
              <method-name>sell</method-name>
        </method>
    </method-permission>
    <method-permission>
        <description>
        This permission gives the right to view the list of
        securities.
        </description>
        <role-name>role-everyone</role-name>
        <method>
              <ejb-name>jdj.security.SecureTradeMgr</ejb-name>
              <method-name>getSecurities</method-name>
        </method>
    </method-permission>
  </assembly-descriptor>
```

Download the Code! www.JavaDevelopersJournal.com



Building Thread-Safe GUIs with Swing

Written by Neal Ford

Create a rich user interface library?

Remember the old axiom, Be careful what you ask for, you just might get it? That's what happened with the Abstract Windowing Toolkit (AWT), GUI controls, and threading. Developers were tired of always worrying about multithreaded access to GUI elements, so it sounded like a good idea to create an application framework that was always thread-safe.

What do we mean by thread-safe? Two separate threads of execution can access the control at the same time without the developer having to worry about the threads interfering with one another. AWT made this possible...and was consequently very sluggish. The original designers of Java built a lot of thread safety into the language and its libraries. For example, the collections classes from the original JDK (Vector and Hashtable) are always thread-safe. However, that safety comes at a cost. Because there's a great deal of overhead necessary to build thread-safe artifacts, they tend to be much slower than nonthread-safe alternatives. This is true of the collections classes (which is why we now have ArrayList and Hashmap, the nonthread-safe alternatives) and Swing.

When it came time to build JFC and Swing, one of the design decisions was that thread safety would be eschewed in favor of speed. This doesn't mean the controls can never be accessed from multiple threads, but the developer is now responsible for adding code to ensure that no ill effects occur. This article shows how to build thread-safe GUIs in Swing. First, however, I'll show what happens if you don't take care of threading.

Thread Collisions

Consider the application shown in Figure 1. It's a simple list box whose items are updated via a thread. For the first version of this example, no thread safety is built into the code (see Listing 1).

As you can see, the thread updates the contents of the list box continuously. What's bad about this? When it runs, you get the result shown in Figure 1. As the thread updates the list box, the drawing thread in Swing also accesses the elements to keep the GUI representation updated. Because both threads can access the content at the same time, the worker thread can pull items off the content at the same time the drawing thread is accessing them for display. This causes the cascading series of exceptions you see in the figure.

How can this be prevented? By using one of the static methods in the EventQueue class – invokeLater() or

invokeAndWait(). These methods were originally in the SwingUtilities class (in JDK 1.1.x) and are now accessible from either class. These methods can take a thread as their parameter and are responsible for executing the thread that's passed to them in sync with the main Swing thread. These two methods determine how you want the update to occur. The invokeLater() method returns immediately, placing the update code in the regular event queue of Swing. This means the updating will take place as soon as possible, but it doesn't make your code wait for the update to occur. There may be situations in which you want your code to wait until the update has occurred. The invokeAndWait() method won't return until the update is complete. These methods are the secret to handling graceful threaded Swing code.

Listing 2 demonstrates how to solve the original problem. I've added code to call invokeLater() to the code that must update the Swing control's contents. Now when the application runs, no exceptions occur because both the update and drawing threads are no longer in conflict. The pertinent change to the code appears in the body of the run() method of the WorkerThread class and is shown here:

```
EventQueue.invokeLater(new Runnable() {
    public void run() {
        if (model.contains(i))
            model.removeElement(i);
        else
            model.addElement(i);
    }
});
```

This is a common technique for updating Swing controls from within a thread. A new anonymous class that implements the Runnable interface is created. The code in the run() method is the code placed in the main event thread in Swing. Notice that this is the same code used in the previous example to update the list box's model – the difference here is the call to EventQueue.invokeLater().

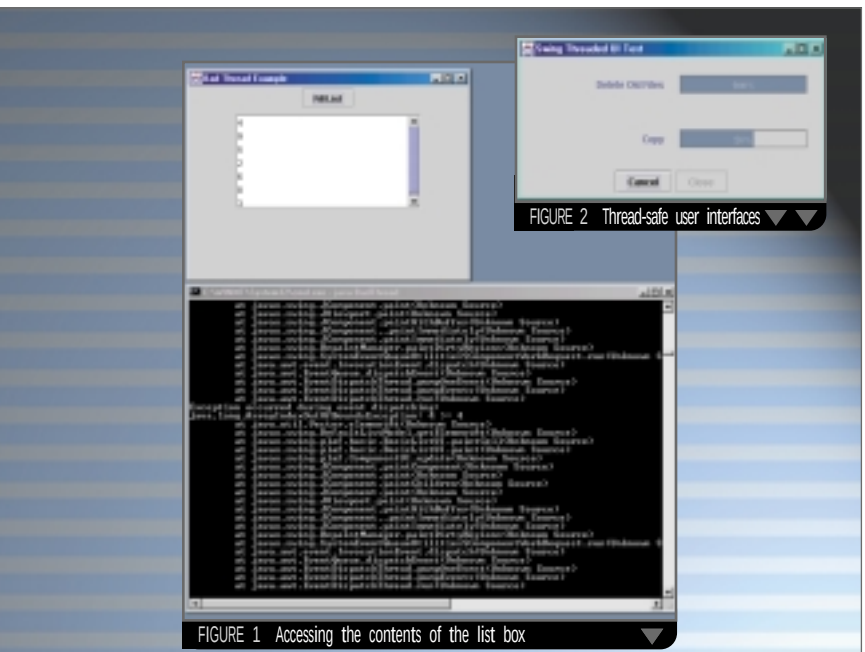


FIGURE 1 Accessing the contents of the list box

FIGURE 2 Thread-safe user interfaces

Progress for Long-Running Processes

Here's a practical example of the need to have a thread update in real time. A common chore in applications is to show the users the progress of some long-running process. There are two ways to create such a process. If coded directly into the application (i.e., not in a thread), the application can't show progress because the process hasn't finished yet. In other words, if you do the work in an event handler, you're occupying the main event thread. If the process was spawned in a thread, the thread safety of the GUI must be considered. The second solution is the only real choice if you want to provide feedback, and armed with the EventQueue methods listed above, it's easy to accommodate.

For this sample application I copy a collection of files from one location to another. This is a classic example of a process for which you want to provide feedback. This application copies all the source files from the Java SwingSet2 demo to the temp directory (hey, I had to copy something from somewhere to somewhere else, didn't I?). As you can see in Figure 2, the user interface provides feedback on the percentage of completion for the two tasks. First, the application deletes the files from the target directory (if they've been previously copied there) and then copies the source files to the target.

Because there are two threads with common traits at work in this application, I first declared an abstract thread class to encapsulate the similarities. This class appears in Listing 3. Two characteristics are required from each thread. First, they must have a reference to the frame class to access the UI widgets to notify them of the progress. Second, a terminate request flag will appear in this base class. As you probably know, the thread stop() method has been deprecated in Java 2 because it can lead to undesirable situations. Now, to be able to stop a thread, you must provide a suicide watch flag – not so much a command to stop as a request that the thread commit suicide. Both worker threads subclass the abstract WorkerThread class.

The delete thread appears in Listing 4. One item to note in both threads is the manner in which they interact with the frame class. The thread doesn't directly access the JProgressBar on the frame. Instead, frame methods are called to handle the actual initialization and updating. The frame's initCopyProgress() method is shown here:

```
void initCopyProgress(int numFiles) {
  jprgrsbrCopy.setMaximum(numFiles);
  jprgrsbrCopy.setMinimum(0);
}
```

The updateCopyProgress() method handles the updating of the progress bar's progress.

```
void updateCopyProgress() {
  jprgrsbrCopy.setValue(
      jprgrsbrCopy.getValue() + 1);
}
```

Making the frame's methods update the progress bar is a good idea so the thread doesn't have too much knowledge of how the user interface is handling the progress mechanism. The UI could now change to incorporate a gauge or

some other progress technique without affecting the threads. The remainder of the DeleteThread deletes the files in the target directory. Notice the call to EventQueue.invokeLater() to update the frame. The last act of the DeleteThread is to instantiate and call the CopyThread (see Listing 5). Both threads could have been created at the same time and the CopyThread made to wait on the DeleteThread. However, because these two operations must run serially, it makes sense to spawn one from the other.

The CopyThread performs many of the same housekeeping operations as the DeleteThread. Most of the code in the CopyThread concerns itself with copying files, which is not relevant to this discussion. For our purposes note the call to update the user interface in a call to EventQueue. invokeLater().

Building user interfaces that gracefully spawn threads and in turn report progress along the way is easy once you understand the requirements built into the Swing application framework. Now we have a rich user interface library that offers better speed and capabilities than the old one. When necessary, the developer can make it thread-safe to give it the capabilities of the old AWT framework without the disadvantages.

Author Bio

Neal Ford, vice president of technology at the DSW Group, is also the designer and developer of applications, instructional materials, magazine articles, and video presentations. He's written two books, Developing with Delphi: Object-Oriented Techniques *and* JBuilder 3 Unleashed.

nford@thedswgroup.com

Listing 1

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class BadThread {
    public static void main(String[] args) {
        JFrame frame = new TestFrame();
        frame.show();
    }
}

class TestFrame extends JFrame {
    public TestFrame() {
        setTitle("Bad Thread Example");
        setSize(400,300);
        setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        model = new DefaultListModel();

        JList list = new JList(model);
        JScrollPane scrollPane =
                new JScrollPane(list);

        JPanel p = new JPanel();
        p.add(scrollPane);
        getContentPane().add(p, "Center");

        JButton b = new JButton("Fill List");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(
                    ActionEvent event) {
                new WorkerThread(model).start();
            }
        });
        p = new JPanel();
        p.add(b);
        getContentPane().add(p, "North");
    }

    private DefaultListModel model;
}

class WorkerThread extends Thread {
    public WorkerThread(DefaultListModel aModel) {
        model = aModel;
        generator = new Random();
    }

    public void run() {
        while (true) {
            Integer i =
                new Integer(generator.nextInt(10));

            if (model.contains(i))
                model.removeElement(i);
            else
                model.addElement(i);

            yield();
        }
    }

    private DefaultListModel model;
    private Random generator;
}
```

Listing 2

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class GoodThread {
    public static void main(String[] args) {
        JFrame frame = new TestFrame();
        frame.show();
    }
}

class TestFrame extends JFrame {
    public TestFrame() {
        setTitle("Good Thread Example");
        setSize(400,300);
        setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        model = new DefaultListModel();

        JList list = new JList(model);
        JScrollPane scrollPane =
                new JScrollPane(list);

        JPanel p = new JPanel();
        p.add(scrollPane);
        getContentPane().add(p, "Center");

        JButton b = new JButton("Fill List");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(
                    ActionEvent event) {
                new WorkerThread(model).start();
            }
        });
        p = new JPanel();
        p.add(b);
        getContentPane().add(p, "North");
    }

    private DefaultListModel model;
}

class WorkerThread extends Thread {
    public WorkerThread(DefaultListModel aModel) {
        model = aModel;
        generator = new Random();
    }

    public void run() {
        while (true) {
            final Integer i =
                new Integer(generator.nextInt(10));
            EventQueue.invokeLater(new Runnable() {
                public void run() {
                    if (model.contains(i))
                        model.removeElement(i);
                    else
                        model.addElement(i);
                }
            });
            yield();
        }
    }

    private DefaultListModel model;
    private Random generator;
}
```

Listings 3 - 5

Listings 3, 4, and 5 can be found online at www.sys-con.com/java/source/

Download the Code! www.JavaDevelopersJournal.com

Fitting the Pieces into the Enterprise Java Jigsaw

Techniques for applet-servlet communication

Part 2 of 3

WRITTEN BY
TONY LOTON

In Part 1 of this series (*JDJ* Vol. 6, issue 4) I developed a simple access control mechanism for my application using HTTP authentication and servlets. In my view, servlets have always been underrated as a technology.

Their use has sometimes been limited to replacing traditional CGI scripts for the processing of HTML form submissions. However, the fact that you can send and receive serialized Java objects to and from servlets means they can be combined with applets as part of simple distributed object architecture, competing with RMI, CORBA, and EJB. No tunneling is required, and it works in browsers that don't support RMI.

Despite my enthusiasm for servlets, for some applications the interactivity and graphical capabilities of applets make them invaluable pieces in the jigsaw. I once wrote an applet that displayed aviation flight paths on a globe projection that could be zoomed and rotated about each axis. Try doing that with a servlet!

Although applets have fallen somewhat out of favor, it's possible to write one without regard to whether it will be downloaded to Internet Explorer or Netscape, on Windows or UNIX. I was doing that four years ago, and now with the Java plug-in down to a relatively trim 5MB in version 1.3 and Java WebStart on the way, the promise of true "write once, run anywhere" applets may yet become a reality.

Applet-Servlet Communication with Applet Parameters

To accommodate the first applet within my architecture, I'll first make a change to the LoginServlet. The code for writing out the user's options as HTML links (see April ***JDJ***) will be replaced by code to download an applet with a set of parameters representing the user's options (see Listing 1).

The resulting HTML, containing an applet tag and a set of parameters, looks like this:

```
<applet code=com.lotontech.applets.
OptionsApplet
 height=30, width=600>
<param name=user
value=bill></param>
<param name=option1
value=getTasks></param>
<param name=option2
value=transferTasks></param>
</applet>
```

This passing of parameters to the OptionsApplet represents the simplest possible servlet-to-applet communication mechanism, although I'll cover a more useful alternative later in this article.

The OptionsApplet will now be responsible for displaying the user options menu, with each option presented as an AWT button (see Figure 1) rather than as an HTML link. I know it's not very fashionable to use AWT, but for this series I've decided to concentrate on the Enterprise aspects and leave the GUI advice to someone else.

There are two important points to look for in the OptionsApplet code included in Listing 2. The first is the retrieval of the applet parameters representing the current user's menu options. The second is the technique I have used for launching a new servlet into a browser frame – via the AppletContext – when an option button is clicked.

The AppletContext is the applet's handle on the browser it's running in. By calling the showDocument() method it's possible to fetch and display any HTML document the server can provide. You could replace the current document – which isn't such a good idea if it contains your applet – or display a new document in a separate frame or a new browser

> "...my preferred technique is easy to understand, **doesn't suffer from tunneling issues,** and may just be worth a try"

AUTHOR BIO

After graduating with a degree in computer science and management, Tony Loton worked for almost 10 years as a consultant, course instructor, and technical author. He uses his company - LOTONtech Limited (www.lotontech.com) - as a vehicle for researching, developing, and commercializing innovative software solutions. Tony is currently writing a book, Web Mining with Java, *to be published by Wiley later this year.*

FIGURE 1 OptionsApplet presentation

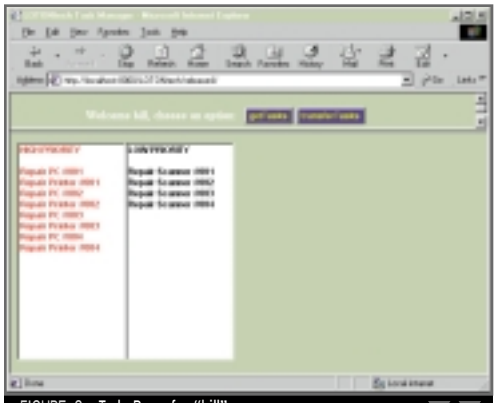


FIGURE 2 Task Page for "bill"

window. I'm doing the latter, and when I say *displaying a document* in this context, I actually mean *running a servlet and displaying its response*, where the response might contain yet another embedded applet.

This applet is only slightly more impressive than the original HTML links because I've kept it simple to demonstrate the technique. In applying this to any real scenario, I've often used an applet to display a multilevel menu via a Swing JTree that allows users to expand and contract the various levels in a way that can't be achieved easily with HTML alone.

This kind of multilevel menu wouldn't be as easy to specify as a flat series of applet parameters, so the applet-servlet communication channel would need to be enhanced to handle the transmission of more descriptive data. Two methods for transmitting descriptive data spring to mind: XML and Java serialized objects. I'll cover the latter and, rather than tire you with yet another menu variation, I'll find a different use for serialization within my application.

Applet-Servlet Communication with Serialized Objects

When user "bill" clicks on his getTasks button (see Figure 1), he sees a list of the tasks that are assigned to him. When a different user, "ben", clicks that button, he'll see a different list. A sample Task Page for "bill" can be seen in Figure 2.

Once again I'll use an applet and servlet combination, but this time in a slightly different way. When the TaskServlet is invoked via the getTasks button, with no parameters, it serves up an HTML tag to download the TaskApplet to the browser. As soon as the applet is instantiated in the browser, it makes a second call directly to the same servlet. This time the applet specifies a getdata parameter as part of the URL connection string, and expects the servlet to return a vector of serialized objects representing the user's tasks. In Listing 3 take a look at the first highlighted code, which tests the presence of the getdata parameter. You can then follow the alternate behavior that depends on the presence, or absence, of the parameter.

After constructing the two AWT Lists – one for high-priority tasks and one for low – during initialization, the TaskApplet (see Listing 4) reinvokes the TaskServlet by opening an InputStream on its URL, this time with the getdata parameter included in the URL connection string.

Note that when the applet requests the task data from the servlet, there's no need to specify which user's tasks should be returned because the HttpSession object – accessible from any servlet – already holds the user name for the current session, as set by the LoginServlet at the time of authentication. (For more about the LoginServlet, please refer to my earlier article.)

The tasks for the current user are returned in the form of a vector of serialized Java objects – in this case simple Strings – that are immediately ready for use as objects in the receiving applet. The contents of the vector don't need to be Strings, and the container doesn't need to be a vector. The only limitation is that any Java objects transmitted in this way must be serializable, which for your own objects means that their class(es) must implement java.io.Serializable. For Java runtime classes it means that you can't transmit things like JDBC connection objects.

Thus an applet/servlet combination allows a style of distributed object programming that in my opinion is much underused.

Conclusion

This check-in/checkout style of object programming using servlets and serialized objects isn't suitable for every situation. Many applications are more suited to the RMI/CORBA model, in which objects are passed between distributed clients and servers by reference rather than by value (serialized), and remain at all times at their point of origination. The issues to consider include performance (references are smaller than objects in transmission) and consistency (a remotely referenced server-side object appears in the same state to all its clients).

However, my preferred technique is easy to understand, doesn't suffer from tunneling issues (since you're using HTTP anyway), and may just be worth a try before you consider something more complex.

On the subject of complexity, no Enterprise Java series would be complete without the appearance of Enterprise JavaBeans. My next article will introduce the simplest possible EJB architecture – consisting of a lone session bean – and will suggest (but not recommend) how this bean's functionality could instead be provided by a more complex combination of session beans, entity beans, and distributed transactions.

tony@lotontech.com

Listing 1: Writing User Options as Applet Parameters

```
//-- write the options out as an applet --

out.println(
 "<applet code=com.lotontech.applets.OptionsApplet
  height=30, width=600>");

out.println("<param name=user value="+user+"></param>");

for (int opNum=0; opNum<options.size(); opNum++)
{
  String thisOption=(String) options.elementAt(opNum);

  out.println("<param name=option"+(opNum+1)
   +" value="+thisOption+"></param>");
}

out.println("</applet>");
```

Listing 2: Options Applet Java Code

```
public class OptionsApplet extends Applet
 implements ActionListener
{
 public void init()
 {
  // -- get the user from parameter --
  String user=getParameter("user");

  // -- GUI would be initialized here --

  int opNum=0;
  boolean finished=false;
  while (!finished)
  {
   opNum++;

   String thisOption=getParameter
    ("option"+opNum);

   if (thisOption==null) finished=true;
   else
   {
    Button newButton=new Button(thisOption);
    this.add(newButton);
    newButton.addActionListener(this);
   }

  }

 }

 // -- ActionListener Method --
 public void actionPerformed(ActionEvent event)
 {

  if (event.getSource() instanceof Button)
  {
   Button sourceButton=(Button)event.getSource();

   // -- tell browser to launch selected app. --

   AppletContext theBrowser
    =this.getAppletContext();

   try
   {
     URL documentBase=getCodeBase();

     URL newURL=new
      URL(documentBase,sourceButton.getLabel());

     theBrowser.showDocument(newURL,"main");
   }
   catch (Exception ex) {ex.printStackTrace();}
  }

 }

}
```

Listing 3: TaskServlet

```
public class TaskServlet extends HttpServlet
{
 public void doGet(HttpServletRequest req
  , HttpServletResponse res) throws IOException
 {

  // -- get the current http session --
  HttpSession session=req.getSession(true);

  String user=(String)
   session.getAttribute("user");

  // -- check for null user omitted –

  String getdata=req.getParameter("getdata");
  if (getdata==null) getdata="false";
  if (!getdata.equals("true"))
  {
   // -- download HTML with task applet tag
  }
  else
  {
   // --task applet calling me again for data --

   Vector tasks=new Vector();

   // -- get connection from datasource --
   InitialContext ic = new InitialContext();

   DataSource ds = (DataSource)
    ic.lookup("java:comp/env/jdbc/LOTONtech");

   Connection con = ds.getConnection();

   // -- select user tasks from the database --
   Statement st=con.createStatement();

   ResultSet results=st.executeQuery(
    "SELECT username, task , priority
    FROM usertasks WHERE username='"+user+"'");

   while (results.next())
   {
    Task thisTask=new Task(results.getString(2)
     , results.getString(3));

    tasks.addElement(thisTask);
   }

   // -- return serialized vector to the applet --
   ObjectOutputStream taskStream = new
    ObjectOutputStream(res.getOutputStream());

   taskStream.writeObject(tasks);
  }
 }
}
```

Listing 4: TaskApplet

```
public class TaskApplet extends Applet
{
 public void init()
 {
  // -- get the user from parameter --
  String user=getParameter("user");

  // -- set up the GUI Lists --

  this.setBackground(new Color(0,0,128));
  setLayout(new GridLayout(1,2));

  java.awt.List highList=new java.awt.List();
  highList.add("HIGH PRIORITY");
  highList.add("");
  add(highList);

  java.awt.List lowList=new java.awt.List();
  lowList.add("LOW PRIORITY");
  lowList.add("");
  add(lowList);

  try
  {
   // -- get user tasks from task servlet --

   URL servletURL=new URL(this.getCodeBase()
    ,"getTasks?getdata=true");

   ObjectInputStream taskStream=
    new ObjectInputStream(servletURL.openStream());

   Vector tasks=(Vector) taskStream.readObject();

   // -- and display them in 2 lists --

   for (int taskNum=0; taskNum<tasks.size();
   taskNum++)
   {
    Task thisTask=(Task) tasks.elementAt(taskNum);

    if (thisTask.priority.equals("high"))
     highList.add(thisTask.taskName);
    else if (thisTask.priority.equals("low"))
     lowList.add(thisTask.taskName);
   }

  }
  catch (Exception ex)
  {
   // -- send error to Java Console --
   ex.printStackTrace();
  }

 }
}
```

Download the Code! www.JavaDevelopersJournal.com

Building a Telephone/Voice Portal with Java

It's quick, and it's easy – lightweight telephony applications using Java

Telephone access to the Web is the latest craze sweeping the dot-com landscape. Voice portals with names like BeVocal, Quack.com, Tellme, and AudioPoint are promising all callers easy access to news, traffic reports, stock quotes, and driving directions. Some of these services may flash and burn as quickly as a California brushfire, but they represent the leading edge of a much larger trend that began several decades ago and has accelerated with advances in audio- and speech-processing technologies: the convergence of voice and data networks.

Part 1

Written by Kent V. Klinner III & Dale B. Walker

With 2 billion telephones and over 400 million cell phones in use today, the sheer scale of this convergence may lead to the most profound changes in our communication networks since the advent of the Web browser.

Emerging standards for voice-over IP and wireless access protocols present systems designers with many choices for remote access technologies, but the predominant communication device, the plain old telephone, means that voice gateways can open data services to a larger population of users than any other available technology.

Java developers are in a strong position to drive voice and data network convergence with cross-platform software for servers, consumer devices, Internet appliances, and mobile handsets. With server-side components for telephony and voice processing, Web content, online databases, and e-mail will be just a phone call away. Client-side applets for mobile devices, handsets, and Internet appliances will deliver applications directly to users at the mobile edges of the Net.

Bridging the circuit-switched world of the telephone and the packet-switched world of the Internet can be a daunting task. Controlling telephone devices, detecting touch tones, and processing caller ID packets and digitized voice often require programming in C or an assembly language for arcane platform-specific hardware. With the information and resources in this article, Java developers can add basic telephone and voice services to their applications quickly and easily.

The market is still testing the value of voice-recognition access to driving directions and cinema schedules. The real value in voice/data convergence technologies may emerge with the development of more personal services, like remote control of your office network, telephone access to your home security system, or mobile access to e-mail, faxes, and voice messages through your existing office or home telephone number.

This article describes a fast and easy way to develop lightweight telephony applications with Java. In the process we hope to demonstrate that Java is the most powerful and robust language for developing and deploying sophisticated services for voice and data networks. With the phonelet framework for telephony services you'll be able to get started immediately.

We've defined a phonelet as the most basic element of a voice portal service application. Servlet developers will immediately recognize the similarity between phonelets and servlets. The phonelets described here shouldn't be confused with Java applets that run within cell phones and other portable devices running a Java VM.

What Is a Voice Portal?

A voice portal is really nothing more than a call center with connections to Web services, some voice recognition services for navigation, a speech synthesis engine for converting text output to the caller, and, usually, some kind of programming or scripting capability. A voice portal application provides a service that can be accessed from the convenience of a telephone or a cell phone. VoiceXML (VXML) is emerging as a popular standard for scripting dialogs. This article focuses primarily on the use of Java to develop voice applications, but we'll discuss the advantages of VXML in Part 2 of this series.

A voice portal application must talk to its user. All user input and output is through narrow-bandwidth audio channels. A voice application must operate efficiently and reliably with nothing more than audio input and output in a narrow range between 30 and 4,000Hz.

A voice portal application developer is essentially a signal processing engineer. Developers must write applications that can filter audio input streams for user commands and convert all output to an audio stream that can be understood by the caller. Fortunately, the developer's problem is made a bit simpler by telephone touch-tone signals, speech-processing components, and the digital signal-processing functions of telephony boards and voice modems. An understanding of audio signal processing will help, but isn't necessary.

How to Program Voice Portal Applications

There are two ways to program voice portal applications: (1) use a standard *programming* language like C or Java, or (2) use a standard *scripting* language like VXML or one supplied by the call center manufacturer or voice portal service. The usual trade-offs apply. Using the former offers the developer the most power, but may impose the burden of complexity or machine dependence. The latter approach is often easier, but may not allow the flexibility and sophistication available to C and Java programmers.

VXML is emerging as an excellent choice for scripting cross-platform phone/voice services. IBM has announced that their forthcoming VoiceServer will support VXML, and Tellme networks, one of the high-profile voice portals, offers developers access to their VXML programming toolkit. Installing your own VXML server can be complicated and expensive. This article describes a lightweight phone/voice service framework that you can use to develop personal voice services on your own home or office network. We'll demonstrate how to add telephone access to servlets and applications with an inexpensive voice modem and a simple Java framework for processing caller ID, touch tones, and digitized voice. The example code, found on the ***JDJ*** Web site, www.javadevelopersjournal.com, will demonstrate how to process caller ID packets to announce incoming calls and how to detect touch tones, play and record audio, and generate speech using a third-party speech synthesis application like IBM's ViaVoice. With the phonelets framework included in the Resources section at the end of this article, Java developers can create cross-platform applications combining phone, voice, and Web services.

Phones Are from Venus, PCs Are from Mars

To grasp the challenges and limitations of a voice portal service, you should understand something about telephone technology. Telephone line signals span a broad range of voltages from 12 to 110v. Computer interfaces operate in a relatively narrow range of voltages under 5v.

The connection between your telephone and the local branch exchange is called the *local loop* and is simply two twisted copper wires. The audio from both parties and the signals to dial another party, signal a busy connection, or initiate ringing must be conveyed on those two wires in a narrow audio bandwidth channel between 30 and 4,000Hz.

Telephone company switching systems have advanced as rapidly and as dramatically as computers. Actually, telephone switching requirements drove much of the computer revolution. By contrast, our telephone handsets and the local loop have changed little. The challenge for telephony engineers has been to extend new services to local loop customers while maintaining compatibility with a design specified almost a century ago. The simplest solution has been to overload the local loop connection with audio control tones. Touch tones and caller identification packets are familiar examples of audio control signals conveyed through the local loop.

The telephone interface circuits of modems and telephony cards have been bridging these two worlds for decades. Even in an age of digital PBXs and broadband services, a good analog modem with voice and fax capabilities can be a powerful tool for application developers.

Modems Are Signal Processors

While cell phones, PDAs, and Web browsers have captured our hearts and our wallets, the common modem has continued its slow march of progress in relative obscurity. We take it for granted. It's an essential part of every PC and laptop because any PC or laptop without dial-up access to the Internet would be seriously limited. Today's modems bear little resemblance to their raucous, slow, and limited forebears of the 1970s and '80s. Today's modems have almost as much processing power as a PC of 15 years ago. The best modems are marvelous little packages of signal processing and telephone line control. Understanding them is essential to understanding the capabilities and limitations of your system.

In the early 1990s modem manufacturers began adding voice processing and tone-detection capabilities to the basic tone-generation function necessary for touch-tone dialing.

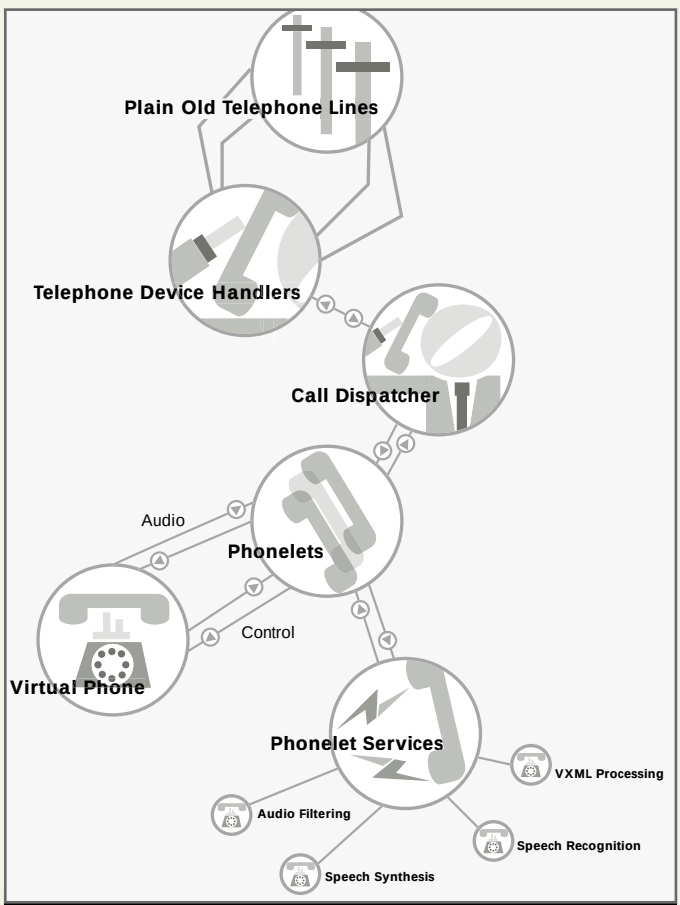


FIGURE 1 Voice portal building blocks

Today inexpensive voice modems offer audio encoding and decoding at a variety of rates and resolutions and in a variety of formats. Many modems will deliver out-of-band events like keypad tone detection, silence detection, busy-signal detection, and fax synchronization tone. Some modems offer caller ID reporting, automatic volume control, and full duplex operation.

Unfortunately, modems are often poorly documented and their behaviors don't adhere to anything more stringent than de facto standards. The phonelet framework simplifies modem programming with a GenericVoiceModem class that you can extend to adjust to the specific characteristics of your modem.

While modems simplify many of the routine tasks of call initiation and termination, they don't insulate the software developer from all the vagaries of an analog communication channel. Java developers who have never written telephony software or programmed a modem may be surprised at how difficult it is to write an application that can handle the demands of streaming audio as well as the complexities of telephone and modem control and still remain responsive and robust to unpredictable events like DTMF (Dual Tone Multiple Frequency) tones, caller hang-up, or device failure. The phonelet development framework hides much of the complexity of modem control and signaling behind a simplified interface so that Java developers can concentrate on the application logic.

Voice Portal Building Blocks

A basic understanding of the components of a voice portal system will help you understand the capabilities and limitations of voice and telephone access to digital networks. Figure 1 illustrates the basic architecture of a voice portal system. The major components are (1) telephone device handlers, (2) a phone call dispatcher, (3) a set of voice portal applications, and (4) a set of application services.

A virtual phone abstraction can simplify the application's interaction with the caller and hide some of the details of telephone line control and signaling. Application services might include libraries for audio signal processing, speech processing, and application scripting support for VoiceXML. VoiceXML is beyond the scope of this article, but you can find lots of information and support on the Web sites of TellMe Networks, Nuance, and IBM.

Emerging standards in telephony and speech processing promise to simplify the development of robust scalable applications, but many of these standards are in flux and not fully implemented. You may need to work around limitations in components and APIs.

DTMF	1209Hz	1336Hz	1477Hz	1633Hz
697Hz	1	2	3	A
770Hz	4	5	6	B
852Hz	7	8	9	C
941Hz	*	0	#	D

TABLE 1 DTMF frequency table

Code Char	Description
0-9, A-D, * #	DTMF Tones
a	Answer tone
b	Busy tone
c	Fax calling tone
d	Dial tone
e	Data calling tone
h	Local phone on hook
H	Local phone off hook
R	Ring
s	Silence timer has expired

TABLE 2 Sample list for Lucent modem chips

Call Dispatcher – PhoneServerLite

PhoneServerLite (see Resources section) is a simple phonelet host that implements the most important features of a voice portal server.

- It acts as the call dispatcher, passing incoming phone calls to the phonelet applications for service (the first phonelet to answer the call gets the service).
- It supports multiple telephone lines and multiple phonelet applications.
- It's threaded for efficient multiuser applications.
- It's resilient to phonelet service failures (bugs in developer code).
- It's resilient to device failure and disconnection.
- It's scalable from laptops to multiprocessor servers.
- It's easy to configure.
- It's easy to extend with custom device handlers and phonelets.

PhoneServerLite configures itself at start-up, and expects the following file folders to contain configuration information, phonelets, and device handlers:

- ***Phonelets:*** Contains the class files for each phonelet application. The file phonelet.txt identifies the phonelets to be loaded at start-up of PhoneServerLite. The sample file documents the format.
- ***Devices:*** Contains the class files for your custom voice modem device handlers. The file devices.txt specifies the devices (i.e., modems) and the parameters. The sample file documents the format.
- ***Files:*** PhoneServerLite creates a folder for each phonelet specified in the phonelet.txt configuration. A phonelet may store permanent files in its file folder. A phonelet can get a file reference to its permanent file folder with the PhoneletConfig.getFileFolder() method.
- ***Temp:*** Temporary files go here. Each instantiation of a phonelet will have its own folder within the "temp" folder. A phonelet may store temporary files in its temp file folder. A phonelet can get a file reference to its temporary file folder with the PhoneletConfig.getTempFolder() method.
- ***Resources:*** Contains resources available to all phonelets, including prerecorded sounds.

The PhoneServerLite start-up procedure is simple.

1. It loads the voice modem device handlers specified in the devices/devices.txt configuration file.
2. It loads and initializes each phonelet specified in the phonelets/phonelets.txt file.
3. It listens for incoming phone calls on each line owned by a device handler and dispatches a service notification to each phonelet when a line rings.

PhoneServerLite dispatches phone calls to phonelets in the order in which the phonelets are loaded (the order in which they appear in the phonelets.txt file). The first phonelet to answer a call handles it. When the phonelet has finished, PhoneServerLite resets the device (hanging up the line in the event that the phonelet fails to terminate the call) and returns it to the waiting condition. Phone calls are dispatched after the second ring because PhoneServerLite gives the device handler an opportu-

nity to detect the caller ID packet, which arrives as a sequence of signals between the first and second ring.

Telephone Device Handlers: GenericVoiceModem

We've chosen to present the phonelets framework with device handlers for voice modems instead of more sophisticated telephony boards because: (1) modems are inexpensive and readily available; (2) they meet the minimum hardware requirements for telephone line control, audio capture, audio play, and tone detection; and (3) they're relatively easy to program through a serial port interface. Telephony cards from manufacturers like Dialogic, Periphonics, and Lucent are far more sophisticated, often including onboard signal processors, but they're more expensive, not as readily available, and, most important of all, not always programmable in Java. Voice modems will keep our discussion at a stick-and-rudder level. Everything you learn from this article applies to more sophisticated hardware, but along the way you'll experience the thrill of flying low with minimal equipment.

Applications communicate with modems through serial interfaces, either physical or virtual. PhoneServerLite includes a SerialPort class and a native driver for Windows 95/98/NT users who can handle the relatively high serial data transfer rates required for digitized audio. JavaSoft's javax.comm serial package for Java has failed with buffer overrun and underrun errors at speeds far below those necessary for audio applications. SerialPort and SpeedSerialWin32.dll simplify the developer's interface to serial ports and provide a stable driver for sustained high-speed serial transfers.

Modems are notoriously difficult to program, and their command sets are often poorly documented and inconsistent. PhoneServerLite includes a GenericVoiceModem class that developers can extend to support the voice modem of their choice. GenericVoiceModem implements the VoiceModem interface and makes no attempt to support data and fax command sets. Its sole function is to provide a simple and reliable interface for common voice modems. Check our Web site for future updates that will add support for fax and data capabilities.

PhoneServerLite loads all VoiceModem device handlers specified in the devices.txt configuration file. GenericVoiceModem implements support for voice modems based on the popular Rockwell/Conexant chip set (e.g., Best Data Smart One, Comtrol RocketModem, and some 3COMHz modems).

Voice Portal Servlets: Phonelets

A phonelet is a lot like a servlet: it's the bare essence of an application. Phonelets embody only the essential logic and data of an application and they can live only in the nurturing environment of a host. The phonelet's simplicity frees the developer from the complexities of telephone device management, call dispatching, scheduling, and error handling. Phonelet developers can focus on the essential components of an application and forget about the details.

A phonelet, like a servlet, has a simple life cycle: (1) init, (2) service, and (3) destroy. It depends on its host to feed it with service requests and provide support functions. Phonelets can provide textual descriptions of themselves with the getPhoneletInfo() method.

- ***Init (PhoneletConfig config):*** The phonelet host calls the init() method only once before invoking the service() method.
- ***Service (PhoneCall call):*** The phonelet host (e.g., PhoneServerLite) delivers incoming phone calls to the phonelet's service() method. When the phone rings, the phonelet can check the ring count, the caller ID, and the incoming line to determine whether it will answer the call.
- ***Destroy ():*** The phonelet host calls the destroy() method only once, and only after it has called the phonelet's init() method. The phonelet host won't invoke the phonelet's service() method after the destroy() method is invoked.

The javadoc files for the phonelet API are included in the Resources section. Developers should be familiar with just six principal components:

1. ***PhoneletConfig:*** Passed as the argument to the init() method. Contains accessors for initialization parameters, the file base, the temp file base, and the PhoneletContext object.
2. ***PhoneletContext:*** Contains accessors for system properties and the call dispatcher. All PhoneServerLite phonelets share the same context.
3. ***CallDispatcher:*** Manages phonelets, devices, and phone calls.
4. ***Phone:*** The virtual phone.
5. ***PhoneCall:*** Encapsulates the virtual phone and the caller ID information.
6. ***CallerID:*** Encapsulates the caller ID packet info.

Getting Started Quickly with HelloCaller Phonelet

HelloCaller is a very simple example of a phonelet. Think of it as the audio equivalent of "hello world."

```
public class HelloCaller extends GenericPhonelet{
public void service (PhoneCall call) throws IOException
{
  Phone phone = call.answer(this);
  phone.play(hello.wav);
  phone.hangup(this);
 }
}
```

The phone.play() plays the WAV sound file to the caller through the voice modem's audio transmit capabilities. The sample rate and resolution of the audio WAV file must match the voice capabilities of your modem. If you try to play a 22kHz 16-bit WAV file through a modem that only supports 8kHz 8-bit audio, your caller will be very displeased with the result. We'll discuss audio streaming in more detail in the next part of this series, but you can check the Java documentation for API details. Writing a simple phonelet that speaks to a caller is very easy with the phonelet framework.

How to Detect Touch Tones

When you dial a telephone number or punch keys to navigate a voice mail system, your telephone generates a tone for each key that comprises two distinct audio frequencies. These DTMF frequencies, carefully chosen as unlikely components of human vocalizations, are listed in Table 1. Note that there are actually 16 combinations of frequencies, four of which aren't commonly used because they're not available from a telephone touchpad.

These signals propagate unhindered through the telephone network and are decoded on the receiving end by tone detectors. The DTMF system is a relatively new development in a telephone system that has undergone only about three user interface upgrades in a century.

In 1941 AT&T introduced touch-tone dialing for central office operators in Baltimore, Maryland. The speed advantage of touch-tone over rotary dialing offset the significant cost of the electronics. The first affordable touch-tone telephones were introduced in 1962. Touch-tone service would not be widely available in the U.S. until the 1970s.

DTMF tone generation and detection fostered the development of a wide array of touch-tone services and equipment, including voice mail, automated attendants, and telephone banking. It's now a part of our lives and a foundation component of any voice portal system. Touch tones may be the single most useful signaling component of a voice portal system. When was the last time you called a business and a human answered immediately?

Modems listen for caller DTMF. When a modem detects a DTMF tone, it sends a 2-byte data packet to the computer. The

first byte of the packet is the shield code – 0x10 for most modems – which lets the computer know that the next byte contains a code that specifies a signal event on the line. For touch-tone events the second byte is the ASCII character of the key pressed. Other event codes are also transmitted in this matter. The exact set of codes supported varies by modem chip set. A sample list for Lucent modem chips is given in Table 2.

Shield codes are passed to the computer in the incoming data stream. If the modem is also in voice-receive mode, the shielded data packet is inserted into the audio stream and must be detected and separated. Otherwise, not only will the information be lost, but the audio will sound really weird. The GenericModem class provides this filtering.

The difficulty in handling touch tones in a voice portal system is that they may come at any time. A caller isn't a computer that can be directed to deliver a touch tone within a specific time window. It's a human being whose clumsy fingers might not press the telephone keypad with the frequency and precision that we demand from machines. The phonelet framework gathers touch tones into a buffer and dispatches an event to the phonelet, which is responsible for monitoring and clearing this buffer.

Every phonelet implements the PhoneUser interface. Touch tones and other shielded codes are delivered as asynchronous events through the phoneEvent() method. Your phonelet is responsible for processing touch-tone event characters, delivered one character at a time. The default PhoneHandler buffers touch tones and provides an accessor method, getTouchtones(), that allows you to retrieve the contents of the touch-tone buffer as a string. There is also a method for clearing the touch-tone buffer, clearTouchtones(). You may, however, manage your own touch-tone buffer, as illustrated in the code snippet below.

```
StringBuffer touchtoneBuffer = new StringBuffer();
public void phoneEvent (PhoneEvent event) {
 if (event.getType() == PhoneEvent.TOUCHTONE)
  touchtoneBuffer.append((char)event.getValue());
}
```

As a convenience to developers, the play() and record() methods can always be interrupted by a single touch-tone character without additional processing by the phonelet. Interrupting an outgoing message or an audio recording with a specific sequence of touch tones, however, is a more complicated function and must be implemented by the phonelet developer.

Putting It Together

Listing 1 demonstrates a simple answering service that plays a greeting, detects touch tones, and records a message after a beep.

In the Next Part...

The next article in this series will describe audio streaming in more detail, demonstrate how to incorporate caller identification services, demonstrate remote control of home appliances with touch tones, and discuss some of the challenges of speech recognition and synthesis.

Resources

- ***MessagePhonelet.java:*** Source code demonstrating how to record and play voice messages, how to generate synthetic speech from text, and how to use touch tones to control program flow.
- ***AnnouncePhonelet.java:*** Demonstrates how to process caller ID packets. Announces an incoming call and the identity of the caller if the phone number (as delivered by caller ID) is contained in a contacts database.
- ***Talker.java:*** Source code for a wrapper to simplify use of a JSAPI speech synthesis engine. Has been tested with IBM's Via Voice and Speech for Java SDK.
- ***CallerID.java:*** Source code for a caller ID object that parses unformatted caller ID packets.
- ***PhoneServerLite:*** A multithreaded phonelet host with loadable device handlers and phonelets. Includes javadoc documentation for phonelet framework.
- ***SpeedSerialWin32.dll:*** Win 95/98/NT native library for high-speed serial access. JavaSoft's javax.comm package provides adequate support for serial data transfers at low speeds, but fails at the relatively high speeds necessary to record and play digitized voice with an external voice-enabled modem. The javax.comm package is unnecessarily complicated by an attempt to wrap the parallel and serial ports into a single package. SpeedSerialWin32 simplifies the programmer's view of the serial port and provides reliable high-speed serial transfers.
- ***CommonVoiceModemCommands.txt***

Recommended Reading

1. Lindley, C. *Digital Audio with Java*. Prentice Hall.
2. McClellan, J.H., et al. *DSP First: A Multimedia Approach*. Prentice Hall.
3. Pierce, J.R., and Noll, A.M. *Signals: The Science of Telecommunications*. Scientific American Library Series.
4. Rorabaugh, C.B. *DSP Primer*. McGraw-Hill.

Web Links

1. *A brief history of TouchTone:* www.research.att.com/history/64touch.html
2. *VoiceXML forum:* www.vxml.org/
3. *Nuance is rich with downloads and information for developers*: www.nuance.com/
4. *IBM speech technologies:* www-4.ibm.com/software/speech/
5. *Caller ID FAQ:* www.ainslie.org.uk/callerid.htm
6. JTAPI: www.javasoft.com/products/jtapi/
7. JSAPI: www.javasoft.com/products/java-media/speech/

Author Bios

Kent V. Klinner III, chief technical officer at TransPhonic, Inc., develops platforms and components for portable and wireless devices. An electrical engineer who still likes to get close to the hardware, he's been developing Java since 1995.

Dale B. Walker is principal engineer at TransPhonic, where she develops applications for mobile and wireless devices. Dale is an electrical engineer with 20 years of broad experience.

kvk@transphonic.com

dbw@transphonic.com

Listing 1

```
public void service (PhoneCall call) throws IOException {
 Phone phone = call.answer(this);
 if (phone == null) return;
String one = "1";
 int loopCounter = 0;
 double beepLength = 1.2; // seconds
 int beepFreq = 1000; // cycles/second
 while(loopCounter < 3){
  phone.play(new File("pressOneToLeaveAMessage.wav"),one);
 if (getTouchtone().equals(one)){
   phone.play(new File("leaveYourMessageAfterTheBeep.wav"));
   File messageFile =
record(maxTime,maxSilence,beepFreq,beepLength,null);
   break;
  }
  pause(5); // wait 5 seconds
  loopCounter++;
  }
 phone.play(new File("goodbye.wav"));
 phone.hangup(this);
}
```

Download the Code! www.JavaDevelopersJournal.com

Distributed Logging Using the Java Message Service

A flexible solution for enterprise computing environments

WRITTEN BY
DAVID CHAPPELL &
GREG PAVLIK

Every software system has logging requirements so application processing can be monitored and tracked. Modern distributed systems, which are usually based on application frameworks, require a logging solution that can cope with multiple processes on multiple hosts sending logging information to a single logging service.

Many application frameworks widely used today, whether they're high-level frameworks like J2EE application servers or low-level frameworks like CORBA ORBs, don't provide a distributed logging facility for application code. Using JMS queues to log application messages is a portable, framework-independent way to efficiently log messages in a distributed system.

Distributed Logging Solutions

It's usually a given that a distributed application needs to keep a centralized application log. We've seen many ad hoc solutions, which are often implemented on a per-application basis. A common way to develop these logging servers is to use low-level APIs, often with the C or Java socket APIs. Logging clients connect by opening a socket and pushing bytes to a log service. Since socket programming is low-level and often error-prone, the logging services are sometimes constructed with an RPC-based distributed object framework such as CORBA or RMI. This provides a higher layer of abstraction to work with, but it still means application developers have to build fundamental application services instead of focusing on the most important task at hand – building real business solutions.

Homegrown distributed logging services are often based on synchronously logging API calls. This means the logging client is forced to block while the logging service processes the message and makes a persistent record in the log. Implementations that support concurrent clients can encounter performance problems related to lock contention in the logging server. In some cases logging services will have internal message queues, so that blocking occurs only through the log message queuing and not throughout the entire logging process. While this approach takes care of the synchronous blocking problem, it's time-consuming and difficult to implement efficiently and reliably, particularly if the solution needs to be coordinated with global or distributed transactions. There are many issues to consider with regard to failure and recovery scenarios for the queue itself and the rules of interaction between the logging client and the logging service under such undesirable conditions.

This matter can be further complicated by geographically dispersed deployments. A distributed application may not be localized to one physical location. Globally distributed applications would presumably need to communicate with the centralized logging system in a secure and reliable fashion over the Internet. As illustrated in Figure 1, you'd likely funnel logging information through intermediate aggregation servers in order to play nice in a firewall environment. These intermediate logging services act as a common conduit that all local applications communicate through. Ideally, these intermediate aggregate servers would be capable of storing log information in case the centralized logging server became temporarily unavailable.

If you were to build a subsystem from scratch that solves all these issues, you'd wind up with something similar to a full-blown JMS queue implementation. Why not use one from the start? It makes perfect sense to base the logging server and its message queue on a common middleware standard and use a common off-the-shelf solution. JMS is an ideal middleware layer that enables distributed logging clients to log messages asynchronously in a uniform and platform-independent way.

It's a natural and pleasant experience to start using JMS to do the same kinds of things that are often done with system-level protocols. JMS has an added advantage as it provides multiple message types for dealing with different kinds of data formats, each with its own set of helper APIs for constructing and deconstructing messages. JMS also accounts for the problems that arise when the intended receivers aren't

currently up and running – a crucial advantage for systems that require high reliability and accurate application logging. With JMS, senders and receivers are abstractly decoupled from each other. An application may send a message to an intended receiver, even when the receiver is not available. The JMS system stores messages on the receiver's behalf until the receiver is available. These are important system-level services that would otherwise have to be written by application programmers who could be more productive developing the actual business applications.

In addition, using JMS as the means for a logging mechanism provides the following benefits:

- Simple, yet flexible standards-based API to be commonly shared among all applications.
- Nonblocking asynchronous placement of log data into the log queue.
- Guaranteed once-and-only-once delivery of critical log data to the centralized logging application.
- Guaranteed ordering of log messages. Messages will automatically be received by the logging application in the relative order in which they were originally placed in the log queue by the senders.
- Well-defined messaging models and message delivery semantics.
- High availability of logging services. Error conditions and the complexities of failure scenarios are handled transparently by the JMS provider or in the interface between the application code and the JMS provider.

As shown in Figure 2, substituting a JMS system as the mechanism for delivering the log data to the centralized logging server removes a great deal of complexity that you would have had to build and manage.

JMS provides support for two messaging paradigms, publish/subscribe and queuing. *Publish/subscribe* is a broadcast model, which is analogous to an event service. Messages are published to virtual channels called *topics* and every client registered as a listener for a topic receives the message. *Queuing* is a point-to-point model. Clients send messages to designated endpoints where messages are enqueued. The message queue is persistent and can be thought of logically as a stack; a message pushed on to a queue will be delivered to a single message consumer. This article uses JMS queues for building application logs.

Logging Queue

Since we're using JMS, the hard work is already done. There's no need to write any infrastructure code at all: JMS provides virtually everything needed for a robust logging service. We need to provide only a logging service implementation that reads the log messages from the queue and does whatever is appropriate for the application. Since the queue is persistent, we don't worry about losing messages. For some JMS implementations, it's necessary to use an administrative console to set up the queue before clients can successfully connect to it. If that's the case, creating an administered object through the console is generally as simple as assigning a name. Self-administered JMS implementations don't require any setup.

Generic Entry Point

To use the queue as a basis for distributed logging, we'll need to define a mechanism for the logging client to write the JMS queue. In general, it's good programming practice to provide a layer of indirection between application code and protocol-specific APIs – the fact that we're using a JMS queue to support distributed logging should be completely transparent. This may be important if you already have a logging subsystem in place.

Migrating each application toward a JMS-based solution can be done separately, obviating the need to coordinate the upgrade of all applications in tandem. In other cases, your application server may provide distributed logging and management capabilities already. Preferably the transport mechanism is dynamically configurable. In Java, this is accomplished with interfaces and Factory classes. Finally, the logging API

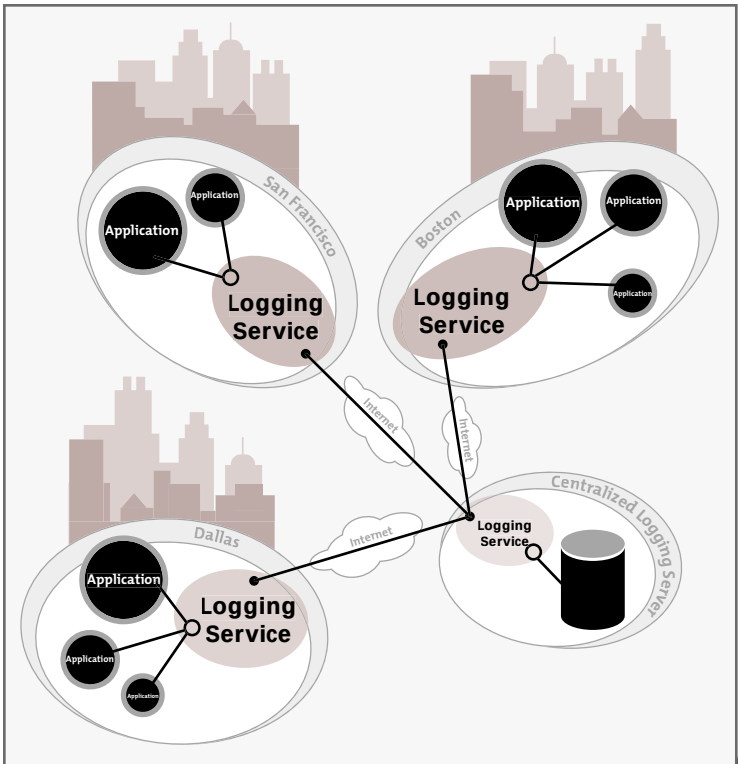


FIGURE 1 The challenge of centralized logging in a geographically dispersed environment

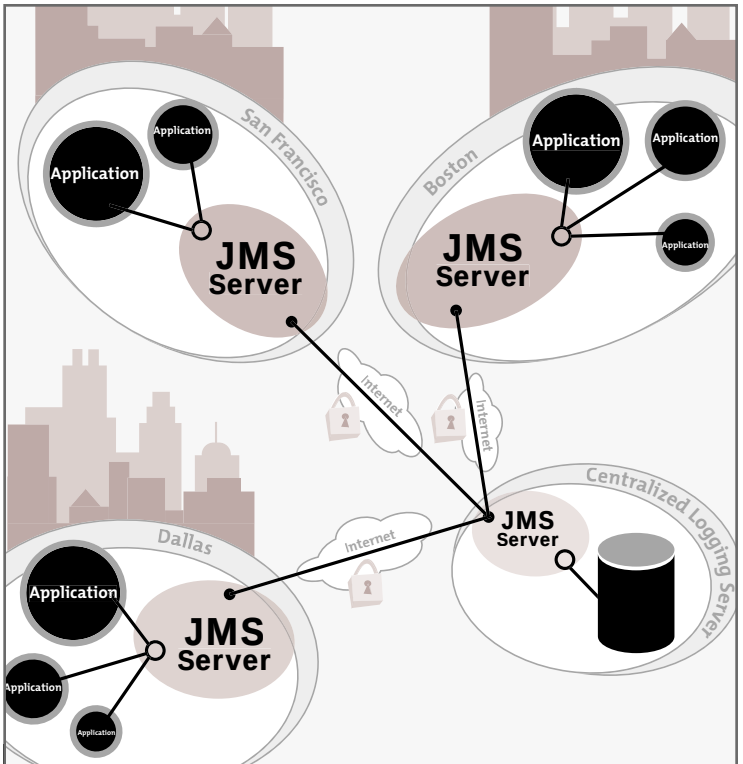


FIGURE 2 JMS handles the complexity of reliability and security across geographical dispersion

It's a natural and pleasant experience to start

using JMS to do the same kinds of things

that are often done with system-level protocols

should be simple and straightforward. Listings 1 and 2 show a simple logging interface and implementation that uses a JMS server to write a log message to a queue. Many applications have more complex log message requirements, so this is an illustrative example.

Access to the client logging implementation is provided by a factory class (see Listing 3).

AUTHORS BIO

Dave Chappell is chief technology evangelist for Progress Software's SonicMQ, and coauthor of O'Reilly's Java Message Service.

Greg Pavlik is a senior architect in HP's application server division.

Log Processing

The logging server may send the data to any number of sources: files, databases, a terminal console, and more. It depends on the specific requirements of the application. In general, simple serialized logging to a file or a terminal console can be accomplished using a JMS MessageListener. The JMS server will automatically serialize messages, eliminating the need for lock management in the logging service code. Listing 4 provides an example of a MessageListener that logs messages to standard err on the terminal screen.

A more complicated logging service might interact with the queue and a database log using global transactions. It might also want to process many messages off the queue concurrently. For these kinds of requirements, an EJB 2.0 message-driven bean may be a more appropriate way to implement the processing logic of the logging service. The EJB container can provide support for global transaction management and concurrent message processing, greatly simplifying the development of the logging service. In addition, the EJB server should provide fault tolerance for the log service itself. In this case, the logging service might have to manage lock contention for writing to log files, but since writing to the log file has been decoupled from application processing by the JMS queue, this doesn't present a performance issue.

J2EE

J2EE-based applications are hosted by application servers that often run a single logical application in many different virtual machines. This allows the application server to transparently provide scalability and fault tolerance to applications built using J2EE components. Application servers are a perfect use case for a distributed logging facility because the replicated application server instances are all servicing clients of a single application. In most cases it's optimal for the application to use a single log. Servlets and EJBs can simply access a singleton logging client API similar to the one we presented above. JSP developers, on the other hand, shouldn't be forced to write Java code unless it's absolutely necessary. The JSP 1.1 specification provides a facility for writing custom tag extensions. A logging tag could be implemented as shown in Listing 5.

The tag we've defined can be used in a natural way by a JSP developer. Logging to the JMS queue in a JSP becomes as simple as adding a new element to an XML document:

```
<app:log message="Application successfully processed request." />
```

Another advantage to using JMS as the basis for distributed logging in a J2EE application is that JMS is a part of J2EE, so a JMS implementation will be provided with the application server. As a practical matter this means a JMS-based logging solution should not incur a large expense.

Beyond J2EE

A J2EE-based application is only one example of a distributed architecture, and J2EE accounts for only a fraction of distributed Java applications. Many Java applications rely on Java RMI or CORBA, directly on JMS, or on a low-level protocol such as Sockets for tying together distributed components. Applications based on any of these protocols and the architectures they suggest can benefit from a distributed log service. All the advantages of building a log service around JMS apply equally well to these applications. Many JMS vendors provide a set of C APIs for their JMS server implementation, which means that JMS can be used as a communication protocol with non-Java applications as well. Thus a JMS-based logging service can be used in a very broad context. It provides a flexible solution for large, heterogeneous enterprise computing environments.

Conclusion

A distributed logging service provides an ideal use-case for JMS. Using JMS, application information can be easily logged to a persistent queue and then processed asynchronously. Application-specific development is pushed to the boundaries of the log processing – time-consuming development of fundamental application services is avoided altogether. JMS also provides fault tolerance and scalability, so the application log can provide highly reliable information. Since EJB 2.0 now integrates JMS into the EJB container, global transactions and support for concurrent message processing can be provided transparently in the logging service.

chappell@progress.com

gpavlik@bluestone.com

Listing 1: Interface for the logging client to send messages to a log service

```
package util.log;
/**
 * Title: ClientLog
 * Description: Interface defining the logging API used by
   logging clients.
 * @author Greg Pavlik
 * @author David Chappell
 */
public interface ClientLog
{
  public void logMessage(String message) throws LogException;
}
```

Listing 2: The JMS-based implementation of the ClientLog interface

```
package util.log;
import javax.jms.*;

/**
 * Title: JMSClientLogImpl
 * Description: Implementation of logging service logging
   client API; uses JMS
 * to send the application log.
 * @author Greg Pavlik
 * @author David Chappell
 */

public class JMSClientLogImpl implements ClientLog
{
  /** queue sender for connection */
  private QueueSender m_sender = null;
  /** queue session for connection */
  private QueueSession m_session = null;

  /**
   * No arg constructor for log service implementation.
   * Acquire a reference to administered queue via standard
   * JNDI lookups.
   * @param initialContextFactory initial context factory
   * @exception thrown if constructor cannot initialize queue
     sender
   */
  public JMSClientLogImpl(String initialContextFactory) throws
LogException
  {
      try
      {
        //acquire JMS queue reference via JNDI
        java.util.Hashtable env = new java.util.Hashtable();
        env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        initialContextFactory);
        javax.naming.InitialContext context = new
        javax.naming.InitialContext(env);
        QueueConnectionFactory qcf = (QueueConnectionFactory)
        context.lookup("QueueConnectionFactory");
        QueueConnection connection = qcf.createQueueConnection();
        QueueSession session = connection.createQueueSession
        (false, QueueSession.AUTO_ACKNOWLEDGE);
        Queue queue = (Queue)context.lookup("LogQueue");
        m_sender = session.createSender(queue);
      }
      catch (Exception e)
      {
        throw new LogException(e.getMessage());
      }
  }

  /**
   * Takes an input string and pushes message on to log queue
   * @param logEntry the input String
   */
  public void logMessage(String logEntry) throws LogException
  {
      try
      {
        TextMessage message = m_session.createTextMessage();
        message.setText(logEntry);
        m_sender.send(message);
      }
      catch (Exception e)
      {
        throw new LogException(e.getMessage());
      }
  }
}
```

Listing 3: A factory class for the implementation of the ClientLog interface

```
package util.log;
/**
 * Title: ClientLogFactory
 * Description: Acts as a factory maintaining implementation
   of ClientLog as a singleton instance.
 * @author Greg Pavlik
 * @author David Chappell
 */

public class ClientLogFactory
{
  /** client log implementation returned by factory */
  static private ClientLog m_logImpl = null;

  static
  {
      //hard code for example
      m_logImpl = new JMSClientLogImpl("JMS Initial Context
Factory");
  }

  /**
   * provides access to singleton instance
   * @return implementation of ClientLog interface
   */
  static public ClientLog instance() throws LogException
  {
      return m_logImpl;
  }
}
```

Listing 4: A message listener that sends text to standard error on monitoring console

```
package util.log;
/**
 * Title: LogServiceMessageListener
 * Description: Message Listener for distributed log example.
   This example extracts the message text from the message
   and writes it to standard err.
 * @author Greg Pavlik
 * @author David Chappell
 */
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.TextMessage;

public class LogServiceMessageListener implements MessageListener
{
  /**
   * Send message contents to standard error
   * @param message the text message sent from
   */
  public void onMessage(Message message)
  {
      try
      {
        String text = ((TextMessage)message).getText();
        System.err.println(text);
      }
      catch(Exception e)
      {
        System.err.println("SYSTEM ERROR: could not process
        message: " + message);
      }
  }
}
```

Listing 5: A tag extension for JSPs

```
package util.log;
/**
 * Title: LogClientTag
 * Description: A tag extension for JSPs that uses the
   ClientLog implementation.
 * This can be used to send log messages to the JMS queue without
 * writing any Java code.
 * @author Greg Pavlik
 * @author David Chappell
 */

import javax.servlet.jsp.tagext.TagSupport;

public class LogClientTag extends TagSupport
{

  /** Client log message set by JSP engine during tag processing*/
  private String m_message = null;

  /** no arg constructor */
  public LogClientTag()
  {
    ;
  }

  /**
   * After tag has been encountered, simply send text to ClientLog
   */
  public int doEndTag() throws javax.servlet.jsp.JspException
  {
        try
        {
            ClientLogFactory.instance().logMessage(m_message);
            return javax.servlet.jsp.tagext.Tag.SKIP_BODY;
        }
        catch (LogException e)
        {
            throw new
javax.servlet.jsp.JspException(e.getMessage());
        }
    }

    //ATTRIBUTE ACCESSOR/MUTATORS
    public void setMessage(String value)
    {
        m_message = value;
    }

    public String getMessage()
    {
        return m_message;
    }
}
```

Download the Code! www.JavaDevelopersJournal.com

The J2EE Connector Architecture

Leverage JCA in your code

WRITTEN BY
BRADY FLOWERS

Most companies have a large investment in legacy systems for ERP, transaction processing, and database applications. Everyone's talking about how they can leverage these systems and integrate them into their modern, multitier, e-business application architectures.

There's an old saying about the weather – everyone talks about it but no one ever does anything about it. Fortunately, this is not true for Enterprise Information Systems (EISs). In fact, IBM, Sun, and a number of other companies are doing something about it in the Java 2 Platform, Enterprise Edition (J2EE) with something called the J2EE Connector Architecture (JCA). VisualAge for Java contains the tooling that was, in large part, the inspiration for JCA. J2EE and JCA are not, as of this writing, finalized and thus no application server or development tool can claim support for either. We'll see, however, that VisualAge for Java's Enterprise Access Builder for Transactions, together with some common design patterns, enables us to leverage JCA in our code.

> VisualAge for Java's Enterprise Access Builder for Transactions,
>
> **together with some common design patterns,**
>
> **enables us to leverage JCA in our code**

First let's look at some of the challenges of integrating an EIS into our e-business architecture:

- Almost all EIS vendors provide APIs for their products. In general these are proprietary interfaces, which may or may not interoperate well with other software, and they tend to be quite complex.
- Every time an application server vendor wants to support a given EIS, the vendor has to build and maintain a separate interface just for that EIS.
- There's no standard process for managing issues of security, transactional integrity, or connection pooling within the applications; the application developers must reinvent these wheels for each EIS.

What Is JCA?

Just as JDBC defines a standard API for relational database access for Java developers, JCA aims to do likewise for connecting to and accessing EISs. The JCA defines a standard architecture for connecting the J2EE platform to EISs by specifying a set of scalable and secure mechanisms for integrating EISs. The JCA also defines a Common Client Interface (CCI) for EIS access.

The Connector Architecture

The two key concepts of the architecture are resource adapters, usually provided by the EIS vendor, and application servers, which the resource adapters "plug into." The architecture defines contracts for transaction management, security, and connection management. A resource adapter must support these contracts or it won't be allowed to plug into a compliant application server. This architecture allows any number of application servers to support a given EIS and one application server to support many EISs (see Figure 1). The J2EE Connector Architecture stipulates that the EIS will be the resource manager in instances where transactions are an issue. The architecture also allows the application server to support connection management and connection pooling when scalability and performance are an issue (when are they not?).

Common Client Interface

CCI defines a common API so that each application server can supply tooling to support any and all JCA-compliant EISs. Some of its features are:

- Definition of a remote function-call interface that focuses on executing functions on an EIS and retrieving the results
- A simple, powerful, and extensible API
- Consistency with various facilities defined by the J2SE and J2EE platforms
- Independence from any specific EIS

The CCI consists of classes and interfaces for the abstraction and manipulation of connections, interactions, records, and connection metadata. You can find these classes and interfaces in the following packages:

- ***Connection-related interfaces:***

```
javax.resource.cci.ConnectionFactory
javax.resource.cci.Connection
javax.resource.cci.ConnectionSpec
javax.resource.cci.LocalTransaction
```

- ***Interaction-related interfaces:***

```
javax.resource.cci.Interaction
javax.resource.cci.InteractionSpec
```

- ***Data representation–related interfaces:***

```
javax.resource.cci.Record
javax.resource.cci.MappedRecord
javax.resource.cci.IndexedRecord
javax.resource.cci.RecordFactory
javax.resource.cci.Streamable
javax.resource.cci.ResultSet
java.sql.ResultSetMetaData
```

- ***Metadata-related interfaces:***

```
javax.resource.cci.ConnectionMetaData
javax.resource.cci.ResourceAdapter
MetaData
javax.resource.cci.ResultSetInfo
```

- ***Additional classes:***

```
javax.resource.ResourceException
javax.resource.cci.ResourceWarning
```

What Is CCF?

As the vendor of CICS, MQSeries, Encina, IMS, and Host on-Demand, IBM is a prominent EIS supplier. It worked with Sun on the development of J2EE, along with other vendors such as Inprise, Oracle, BEA, Motorola, and Sybase. While J2EE was still under development, IBM delivered a working solution to its customers. The Common Connector Framework (CCF) and the Enterprise Access Builder for Transactions (EAB) have been part of VisualAge for Java since 1998. (In addition to the IBM EISs mentioned, the EAB also contains connectors and tooling to support SAP R/3 servers.)

If we examine the components of the Common Connector Framework, we'll see that most of the details and nearly all the philosophy have made the transition from CCF to JCA. The Common Connector Framework features all the same constructs as CCI plus a few extra features:

- Connection/communication specifications
- Interaction specifications
- Records and record types
- Mappers to map records to managed business objects
- Command and navigator beans

If we examine the components of the Common Connector Framework,

we'll see that most of the details and nearly all the philosophy

have made the transition from CCF to JCA

Command and Navigator Beans

Figure 2 illustrates how the CCF command bean wraps connection, interaction, and record beans to encapsulate one transaction with the EIS. The command provides getters and setters for the input and output records, a possible façade for other input and output properties, an execute() method to begin the processing, and events for the successful and unsuccessful completion of the transaction. Not shown in Figure 2 but also very important is the ability to commit or roll back the transaction, if applicable.

Navigator beans are simply an extension of the command beans, which can contain other command beans and navigators. These objects allow us to group multiple EIS transactions and entire subgroups of transactions into meaningful units of work that can be committed or rolled back as one.

The command and navigator beans have a simple, well-defined API that's designed to make them easy to work with in a visual builder tool such as VisualAge for Java's Visual Composition Editor. In the next article we'll make good use of this feature as we build a sample transaction program.

SmartGuides

In addition to supporting Java record structures, which is part of the CCI, the Enterprise Access Builder for Transactions also supplies several tools to help automate the creation of certain record types. CICS programs written in COBOL generally have a communications area called a COMMAREA or their screen is defined as BMS maps; IMS screens are defined via MFS maps. The Java Record Builder SmartGuide knows how to read and parse COBOL, BMS, and MFS source files to create input and output record types and records from these structures. In addition, the 3270 Importer is an integrated tool that imports a 3270 terminal source and generates a record type and a record. Figure 3 shows all the options available in the Enterprise Access Builder for Transactions.

The EAB and EJBs

The Enterprise Access Builder can be used in the development of session beans and BMP entity beans in two ways:

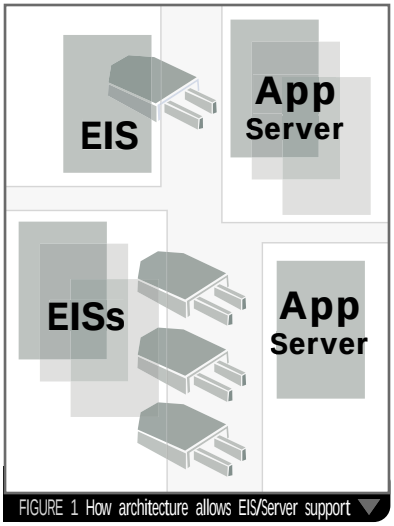


FIGURE 1 How architecture allows EIS/Server support

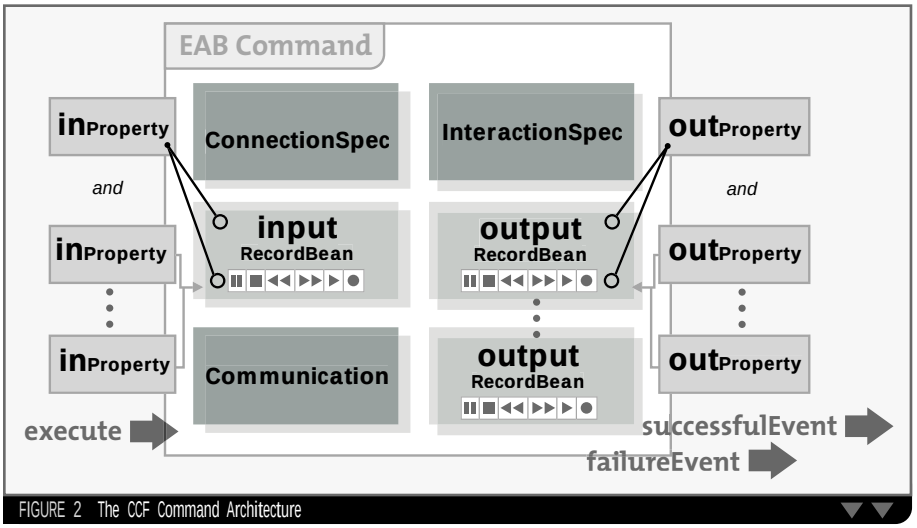


FIGURE 2 The CCF Command Architecture

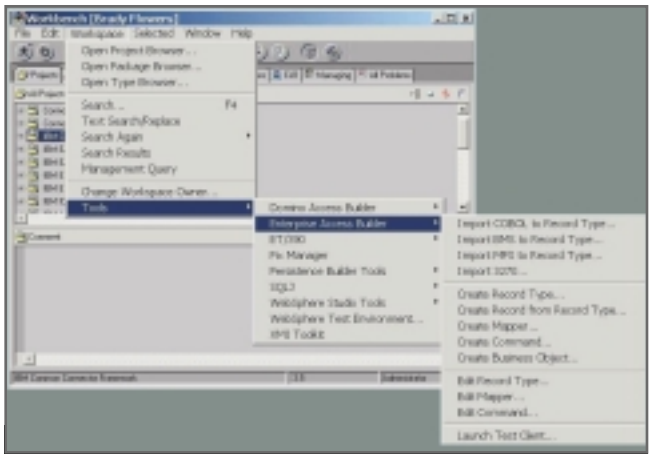


FIGURE 3 The Enterprise Access Builder in VisualAge for Java

1. The EAB features a session bean SmartGuide, which can be used to create a session bean that encapsulates an EAB command or navigator, and an EAB session bean editor, which can be used to further refine the enterprise bean.
2. After creating a BMP entity bean in the VisualAge EJB development environment, the persistence methods of the entity bean – ejbCreate(), ejbFind(), ejbLoad(), ejbStore(), and ejbRemove() – can be implemented either by invoking methods on an EAB-generated session bean, or by directly executing EAB commands and navigators.

The Path from CCF to JCA

The JCA specification will be finalized in J2EE 1.3, which is expected to be released in 2001. As of this moment, however, JCA and CCI are not implemented by any application server or EIS vendor, and the IBM CCF is not yet ported to JCA. So the question becomes: How do we write code that works now but won't need to be thrown away in the immediate future? We can take a clue from one of the structures provided in the EAB: the Command Pattern.

Command Pattern

Command Pattern is one of the patterns discussed in the revered book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides. The EAB command bean is an implementation of this pattern. If we use the EAB's command bean directly, however, we may risk the maintainability of our code in the future should we decide to move to a different implementation of the JCA or if the EAB changes to fit the JCA as the specification becomes final.

The Command Pattern is one of the shortest and easiest to grasp patterns in *Design Patterns*. Basically, a Command should provide setXXX() methods, execute(), and getXXX() plus the possibility of thrown exceptions. We would do well to define our own Command interface, which may inherit it from one of the existing command interfaces or serve as a façade. That's exactly the course we'll set in the next column.

Next time we'll build a back-end transaction using EAB/CCF and wrap the transaction in a custom command to make it JCA-ready.

Resources

1. *The JCA specification and information on the CCI:* http://java.sun.com/j2ee/connector/
2. *IBM documentation for the Enterprise Access Builder for Transactions:* www7.software.ibm.com/vad.nsf/Data/Document3852
3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns.* Addison-Wesley.

AUTHOR BIO

Brady Flowers is a software IT architect with IBM's WebSpeed team specializing in WebSphere, Java, and the rest of IBM's suite of e-business applications.

bradenf@us.ibm.com

zeroCode

by Ampersand Corporation

REVIEWED BY BRIAN R. BARBASH

AUTHOR BIO

Brian R. Barbash is a consultant for the Consulting Group of Computer Sciences Corporation. He specializes in application design and development, business and technical analysis, and Web design.

bbarbash@csc.com

JDJ WORLD CLASS AWARD

Ampersand Corporation
700 N. Central Ave, Suite 270
Glendale, CA 91203
Phone: 818 548-9100
E-mail: info@zerocode.com
Web: www.zerocode.com

Pricing Information:
Subscription basis
$500 per concurrent designer per month
Payment plans available
Considerations for site licenses for companies with 10 or more users

Test Environment:
Development:
Internet Explorer 5.5 running against zeroCode's development site
Runtime:
Windows NT 4.0 SP 5
256MB RAM
Database: Oracle 8*i*
Web Server: Apache 1.3.12
Servlet Engine: Allaire JRun 3.0

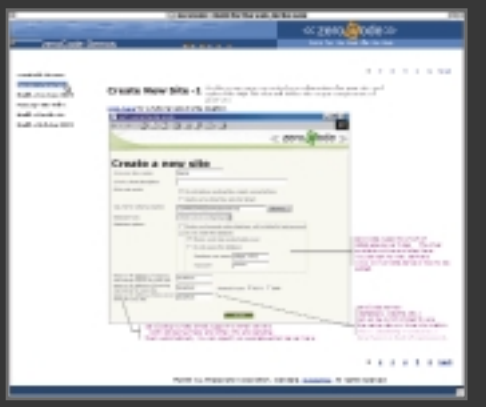
zeroCode is a Web-based development environment that allows a team to graphically develop a Web-based database application with minimal handwritten code. The development environment is Web-based and housed on zeroCode servers. Developers build the application via the Web interface and when complete, the entire site is downloaded and installed on servers in the production environment.

While developing with zeroCode, a heavy emphasis is placed on an application's design. To optimize code generation capabilities, the zeroCode environment defines a set of design rules and guidelines the development team must follow. Some examples of rules to follow include using Java naming conventions for database objects, normalizing the table structure as much as possible, and using foreign key constraints to identify relationships among tables.

zeroCode is designed to isolate the tiers within an application, shown in Figure 1 from the zeroCode documentation. Users interact with zeroCode applications through servlets. The servlets interact with the application objects, which in turn interact with the database through a data-access layer. Manipulation of data for UI display purposes is handled via FreeMarker, an open-source tag expansion engine designed to interface HTML and application objects.

All data to support the application is stored in a JDBC-compliant database (zeroCode has been tested with Oracle 8/8*i* and MS SQL Server). The database design should closely follow the application's object model to provide the best possible generation of components. All data relationships must be modeled with foreign keys to allow zeroCode to construct relational objects for the application. The data model is uploaded to zeroCode, which then generates the tables, business logic components, and default HTML views for each database object in the schema. At this point, the user has access to the system and may manipulate data at a table level.

Once the schema is uploaded and the database is in place, there are several concepts and related terminology that must be understood to develop applications with zeroCode. The user interface data model (UDM) is an object that relates an HTML page to data in the database. UDMs are hierarchical in nature and represent data accordingly. An example might be a collection of albums by a single musician. zeroCode provides basic UDMs for common actions and data representations. Customized UDMs may be created to perform more complex operations. zeroCode automatically generates HTML pages to view data in the database schema. These



pages are called templates and may be modified individually to provide a customized look and feel. Templates include HTML, JavaScript functions for validating form input, and FreeMarker tags. Metatemplates are the files that contain instructions for generating templates. By modifying metatemplates, a large number of HTML pages may be changed. For example, if a company logo and common header were required for all pages, the HTML can be added to the metatemplates and the site regenerated to update all pages.

Predicates are objects that are analogous to "where" clauses in SQL. They're used to constrain data for display and are applied to nodes within a UDM.

As mentioned earlier, all development takes place on the zeroCode Web site and when complete, the application and all tools for runtime support are downloaded and installed on local production boxes. The two main components to zeroCode are the zeroCode development environment and the runtime environment. The development environment is currently available for Linux only and was therefore not included in this review. The runtime environment is a pure Java implementation and will run on any supported platform. Currently, zeroCode is tested for Linux and Windows NT with UNIX evaluations forthcoming. For this review I created a very simple site to access a database schema and proceeded to download the environment for deployment.

When downloaded, the site is packaged as a .tgz file, readable by WinZip. The finished application package will include all runtime required files (zeroCode JAR files, stylesheets, HTML files, UDM files, etc.) in their appropriate directories. To bring the site up, the database schema file must be executed within a local instance of Oracle to create the database structure. Then the configuration files must be modified to fit the runtime environment. Settings include the name, port, user, and password settings for the local database and the paths to the root of the zeroCode download. Finally, JRun must be configured to recognize a servlet URL for the site. The process was relatively easy to complete and the site was up and running locally in less than one hour. Currently, zeroCode runtime has been tested for Linux, Solaris, and Windows NT.

The zeroCode development environment presents a powerful alternative to developing database-enabled Web sites. As zeroCode evolves and is enhanced further, more and more complex sites will be able to be generated using this development platform. It's a product that takes a significant step toward the goal of developing applications with minimal handwritten code.

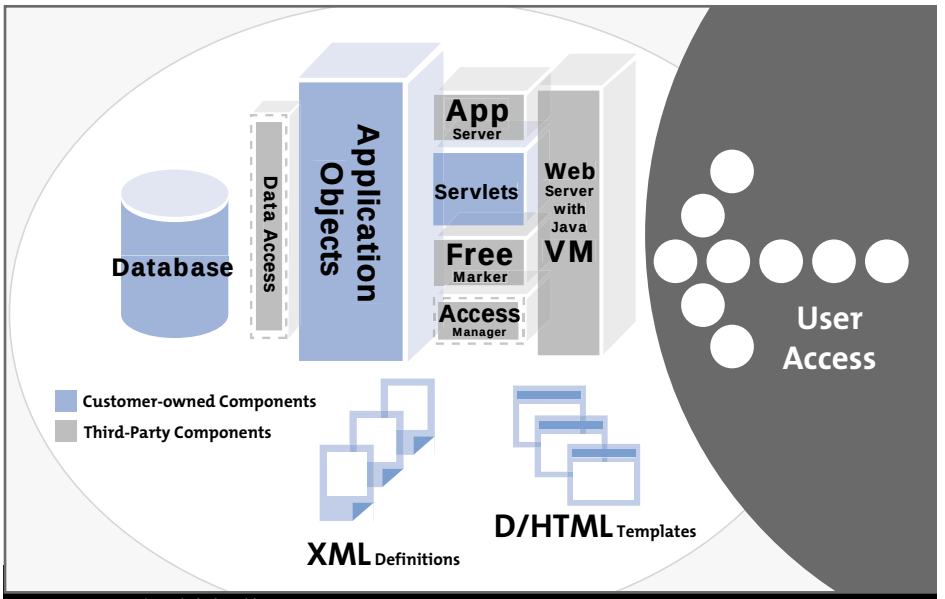


FIGURE 1 zeroCode technical architecture

A Practical Solution for the Deployment of JavaServer Pages

Download Web apps without compromising your security

Part 3 of 3

WRITTEN BY
ALEXIS GRANDEMANGE

A Client Company implements an archive-downloading package. It downloads presentation archives from its providers, Server Companies 1 and 2. Server Company 1 archive includes Enterprise JavaBean client code; Server Company 2 includes Java Message Service (JMS) client code. The archives are either JAR files or Web Archive (WAR) files.

Since Client Company downloads its archives through the Internet:

1. It wants to be sure downloaded archives can be sent only by its identified providers.
2. It wants to be aware of the security requirements of its providers. For instance, does a downloaded archive need to write or remove files?

• • •

The first article in this series (***JDJ***, Vol. 6, issue 1) described how to download servlets and JSP archives like applets. Part 2 (***JDJ***, Vol. 6, issue 2) showed how to administrate these presentation archives remotely, in particular to force a refresh at a scheduled time. I also indicated how to handle special cases, such as resources.

> A better solution would be to declare somewhere that
> **code coming from a given source with a given signature**
> **is granted a set of permissions**

The last issue to address is security. Let's consider the situation presented above, illustrated in Figure 1.

The solution I present to address these requirements is to enhance the archive-downloading package to host archives in sandboxes in the way that browsers host applets. I used the standard Java 2 security described in the Java Security Architecture document downloadable from http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html.

Java 2 Security

Figure 2 depicts Java 2 security components. The hub component is the Security Manager. Its purpose is to determine whether particular operations should be permitted or denied. The Security Manager is implemented by SecurityManager class and subclasses, which contain methods with names that begin with the word *check*. These methods are called by various methods in the Java libraries, including Java API, before those methods perform potentially sensitive operations. The invocation of such a check method typically looks like this:

```
SecurityManager security =
System.getSecurityManager();
if (security != null) {
security.checkXXX(argument,  . . .
); }
```

There is at most one active SecurityManager instance that a method retrieves with System.getSecurityManager(). If getSecurityManager() returns null, the method doesn't check and is slightly faster. This is often the case in Java application servers. SecurityManager check methods rely on another component, the Access Controller, and call AccessController.checkPermission(perm).

The AccessController class manages Java 2 security, whereas the Security Manager that already existed in Java 1 remains mainly for upward compatibility.

A fundamental concept of Java 2 security is the protection domain. A domain encloses a set of classes whose instances are granted the same set of permissions. Permissions here are instances of Permission subclasses. They represent access to a system resource. For instance, the permission to read a file C:\TEMP\FileAccess.txt can be produced by:

```
new
java.io.FilePermission("C:/TEMP/
FileAccess.txt", "read").
```

As you could guess, a protection domain is implemented by a ProtectionDomain class, whose constructor

is ProtectionDomain(CodeSource codesource, PermissionCollection permissions). Let's start with the simplest parameter, permissions. This is simply the collection of the permissions the protection domain will have. The parameter of codesource is slightly more complex. It represents the origin of the protection domain classes and its credentials. Codesource is characterized by a set of public keys and a codebase URL, and its constructor is CodeSource(URL url, Certificate[] certs).

I need to give a short explanation of the credential issue here. To be sure a piece of code is coming from a source, we must check whether it contains something only the source can generate. The most common and standard mechanism is asymmetric keys. The source encrypts a signature with a private key and the destination uses a corresponding public key to decrypt and check that the signature is correct. It retrieves the public key from a certificate, which certifies that the key belongs to the source. As only the source has the private key, it is the only entity able to generate a signature that can be decrypted with the public key. It is the solution implemented in Java, and Certificate is a class wrapping a certificate.

A last point to consider: we could build our PermissionCollection programmatically, but it wouldn't be flexible and convenient. A better solution would be to declare somewhere that code coming from a given source with a given signature is granted a set of permissions. This is the purpose of the policies, implemented as Policy subclasses. A policy is a way to get a PermissionCollection, given a CodeSource, based on a configuration file or database. Sun provides a default policy implementation, PolicyFile, that relies on configuration files.

Java security is comprehensive and well documented. Accordingly, I used it to support sandboxes inside a Java server in a way that was as close as possible to the applet distribution model.

Use

A Server Company generates a key pair and stores it in a key store, using the keytool command as indicated in Listing 1. (*Note:* All listings appear on the Web at www.javadevelopersjournal.com.) Here it creates a key pair whose alias is hello. It's protected by a password helloPswd and stored in a key store named D:\JSPservlet\keystore. The Server Company then signs its archive with "jarsigner," for instance:

```
jarsigner -keystore
D:\JSPservlet\keystore -storepass
keystorePswd -keypass helloPswd
helloMisc.jar hello
```

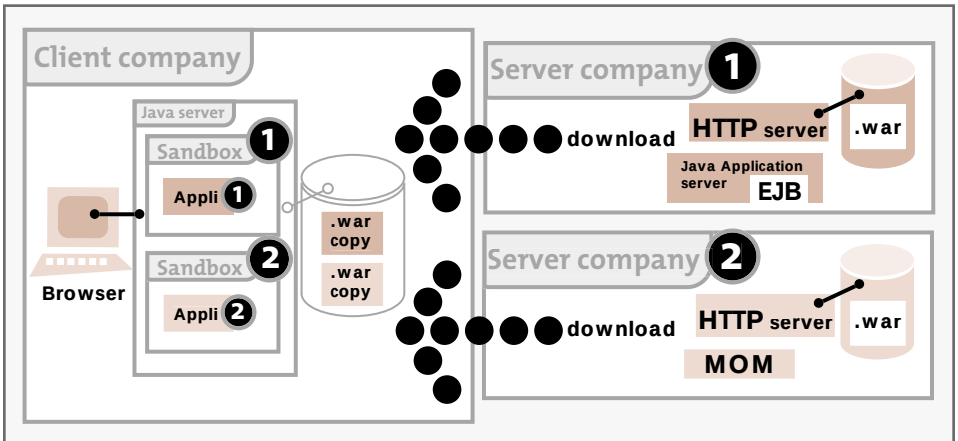


FIGURE 1 Sandbox need

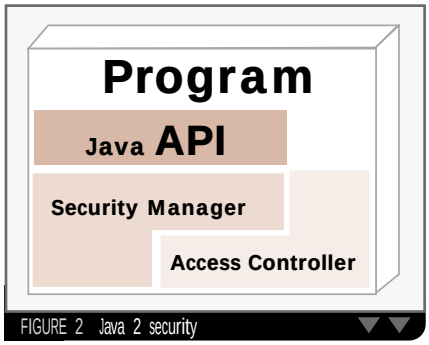


FIGURE 2 Java 2 security

to sign a helloMisc.jar archive, using its hello key pair.

The Server Company also describes which authorizations the archive needs to run properly in a standard policy file. For instance, it describes the helloMisc.jar security requirement in Listing 2. This policy states that the archive classes, represented here by their codesource, require all permissions. It also states that archive classes must be signed with the public key defined in key store and aliased by hello.

The Client Company administrator imports the Server Company certificate into its key store. It can verify the required permissions in the policy file and check the certificate with the commands in Listing 3. If it agrees on permissions and certificates, it can add them to its environment without restarting its Java server because the archive-downloading package finds the policy files and key stores in the cache directory specified by the JSPservlet deployment descriptor. More precisely, the package first looks for an archive.policy file or, if that doesn't exist, for a java.policy file. Therefore, the administrator can choose to merge policy files of different providers or to keep them separate.

Once the administrator has completed this task, it or the Server Company can initiate a first download and users can access the Web application.

Implementation

Let's start with a reminder of the tool structure (see Figure 3).

JSPservlet, a special servlet, handles HTTP requests toward a Web application and forwards them to target servlets and JSPs with the help of a set of objects:

1. JSPhandler objects manage Web applications and maintain a ClassEntry map. They also cache initialization parameters.
2. ClassEntry objects manage archives and maintain a cache of target objects.
3. JSPloader objects are target servlets class loaders and maintain a cache of target classes.

The security support is implemented in the class loader, JSPloader.

It implies, however, a minor modification of JSPhandler to support another parameter, allPermissionPolicy. This parameter has two functions: first, if present, it means that the archive classes must run in a sandbox; second, it gives the name of a default policy file that's used if the Java server doesn't set a Security Manager, which is often the case.

Setting a Security Manager is a JVM-wide action. As we saw before, if it isn't set, then it can't be called by Java API and hence can't get the opportunity to invoke the Access Controller and to enforce a policy. So we need to set a Security Manager if no one is active, but before we have to set a policy. Otherwise the Java server will get an AccessControlException, because Java 2 security doesn't have a built-in concept that local code is trusted. Therefore we need to grant our Java server permis-

sions such as the right to read local disks. Listing 4 is the simplest allPermissionPolicy file we can define. It grants all permissions to all classes, which is the same behavior as running without a Security Manager.

First let's look at the constructor in Listing 5. If allPermissionPolicy property isn't set, it loads the archive classes and resources and returns without attempting to apply a security. Otherwise, if a Security Manager isn't set, it creates a new policy with new sun.security.provider.PolicyFile(). It must be set before a java.security.policy system property that PolicyFile constructor uses to locate the policy file. Next the constructor sets this policy as the current policy and creates the security manager.

The constructor then creates a base URL, which is used later in the classes definition. We'll examine this point shortly. For the moment, simply note that the base URL is made up of the archive download location and the archive name. It's the codeBase the administrator specifies in the archive policy file, as you can see in Listing 2.

The next step of the constructor consists of loading the archive policy file from either archive.policy or java.policy, as described above. It creates a policy but doesn't set it as the current policy: the Access Controller continues to apply allPermissionPolicy.

Eventually the constructor builds classes through the invocation of loadClassDataFS(). This method gets a JarInputStream on a local copy of the remote archive and passes it to a parseStream() method shown in Listing 6. parseStream(), which defines classes and implements the sandbox, relies on the ability to define a class inside a Protection Domain. If classes inside the domain are trying to do something the domain isn't allowed to do, the Access Controller throws a java.security.AccessControlException.

Let's look at a back trace (see Figure 4). Here I tried to run a TestServlet.FileAccess servlet from an unsigned archive. TestServlet.FileAccess invoked File.exists(). File.exists() invoked SecurityManager.checkRead(), which invoked AccessController.CheckPermission(). AccessController.CheckPermission() found that the Protection domain didn't have the needed permission and threw a java.security.AccessControlException.

The interesting part of parseStream() starts when it has identified a class. If allPermissionPolicy property isn't set, no sandbox is enforced and parseStream() defines the class with defineClass(name, buf, 0, buf.length). Otherwise it extracts the class certificates with JarEntry.getCertificates(). If it finds no certificate, it defines the class in a protectionDomain() domain. Though this domain has no certificates, it doesn't necessarily mean it has no permissions. If the domain codesource was granted permissions without a signature (SignedBy parameter), such as in the Listing 7 policy file, policy.getPermissions() returns these permissions.

To handle a situation in which JarEntry.getCertificates() found certificates, parseStream() maintains a protectionDomains HashMap whose keys are certificates arrays and values are Protection Domains. Thanks to this HashMap, parseStream() uses only one Protection Domain per certificate array. It enumerates protectionDomains and checks to see if a domain already has the same certificate array. If it does, parseStream() defines the class in this domain. Otherwise it creates a new Protection Domain with the permissions returned by policy.getPermissions(), defines the class in this domain, and records the new domain in protectionDomains. The new Protection Domain can be granted permissions because a certificate matched a signature as seen in the Listing 2 policy file.

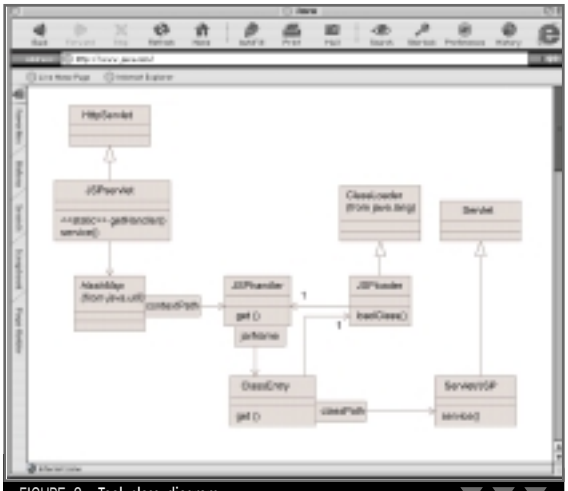


FIGURE 3 Tool class diagram

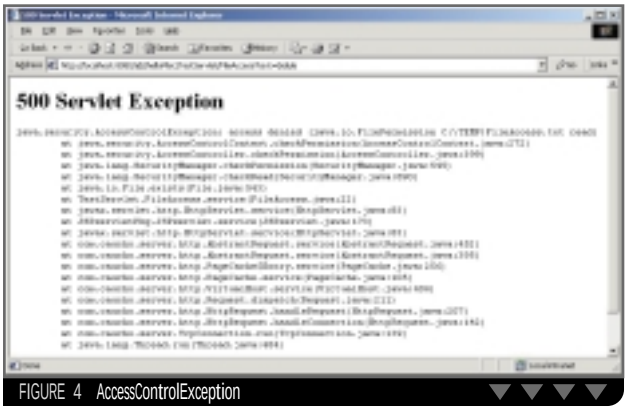


FIGURE 4 AccessControlException

Callback Issue

When the servlet container invokes JSPservlet, which invokes the target servlet, the solution we presented above is enough. However, we also have to handle the case when the target servlet asks for a resource or a class, or calls back JSPservlet for another servlet using RequestDispatcher.include() or RequestDispatcher.forward().

The callback is likely to fail in a disconcerting way. If you call servlet A first, you get an AccessControlException in a class loader; if you call it again, you get AccessControlException in JSPservlet. If you invoke servlet B before servlet A, however, it works fine.

This problem happens because the target servlet code calls a class – JSPservlet, for example – requiring privileges it doesn't have. As we saw above, AccessController.CheckPermission is invoked. It checks to see that all code traversed by the execution thread up to its call has permission for that access. When the target servlet calls back JSPservlet, a piece of code – the target servlet itself – doesn't have the proper permission and AccessController throws an AccessControlException. The problem occurs randomly because, depending on the invocation order, resources have or have not already been accessed.

To address this issue, we need to use "privileged" code (see Figure 5). Let's refine the explanation above. AccessController checks whether all code traversed by the execution thread has permission for that access unless some code on the thread has been marked privileged. If a caller whose code is granted a permission is marked privileged and all code invoked by this caller also has this permission, then AccessController allows the access.

We granted allPermissionPolicy permissions to the tool code, so the only thing we need to do is mark the code privileged when it can be called by the target servlets at resource retrieval, class retrieval, and servlet handling. To do this we must use AccessController.doPrivileged. This method is thoroughly described in the JDK doc.

Listing 8 shows how I mark servlet handling as protected. As I use an anonymous inner class, I must declare local variables used in the privileged block as final. I also use PrivilegedExceptionAction interface to handle exceptions raised in the privileged block.

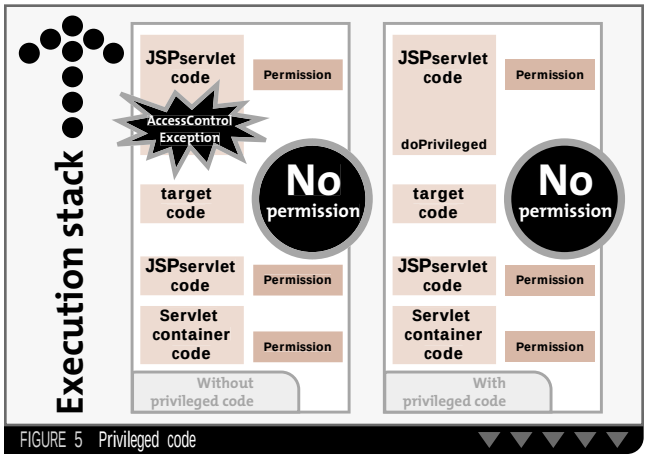


FIGURE 5 Privileged code

Certificate Authority Considerations

Let's assume you are the server company administrator. You run the Listing 1 command to populate your key store with a self-signed certificate, and a private-key JARsigner needs to sign the archive. You should:

1. Never distribute this key store, as it contains the private key allowing signing archives.
2. Keep the key store in a safe location. If you lose it, you'll never be able to sign archives again, and if someone with malicious intent reads it, your security will be compromised.

Let's assume your customers accept only certificates issued by a given Certificate Authority (CA). You need to get a certificate that is issued by this CA and that wraps your public key, whose corresponding private key is known only to you. To do it, first build a PKCS#10 certificate request with:

```
keytool -certreq -alias
alias -keystore keystore.
```

PKCS stands for Public-Key Cryptography Standards, which are RSA Laboratories specifications. PKCS#10 specifies the Certificate Request (CR) standard. Once you have your CR, you query a certificate to the CA, typically through an HTML form that prompts you for your CR. You then import this certificate in your key store with the command:

```
keytool -import -file
certificate_issued_by_CA -alias
alias -keystore keystore.
```

You also send the certificate to your customers, who import it in their key stores using the same keytool command. It's important to note the difference between your and your customers' key stores. Yours contains both the certificate and the private key. You can sign archives. Theirs contains only the certificate. They can check only archive signatures.

Now consider the distribution of a signed archive. Assume that many customers have anonymously downloaded a signed archive that someone published on a repository. They followed instructions, installed the certificate in their key store, and put the archive in production. Later someone revokes the certificate. If a CRL checking mechanism is embedded in the solution, it's possible to disable the archive classes without impacting other Web applications or needing to stop the Java server.

We can implement this mechanism using freely downloadable material because:

1. Revoked certificates are stored in Certificate Revocation Lists (CRL), generally accessible through Lightweight Directory Access Protocol (LDAP).
2. LDAP repositories and keytool use X509 certificates and CRL.
3. Java 2 provides helper objects for X509 certificates and CRL.
4. JNDI supports LDAP.

Listing 9 presents a CRLchecker class, which checks CRLs. Its core method, refresh(), gets an initial Directory Context, providing parameters like the LDAP URL, where the CRL is defined; the principal; and the password to use to connect to the CA directory. It then retrieves the CRL in a certificateRevocationList context attribute, creates a Java X509CRL object, and populates it with the attribute value.

The refresh() method is invoked by the constructor and by getNextUpdate(), whose purpose is first to refresh the repository periodically and second to return the next scheduled CRL update. CRL update can become complicated and expensive, especially if many CAs are involved. Most revocations aren't critical. An employee certificate, for example, can be revoked when the employee moves to another department. It's therefore often practical to update the CRL only once a day or once a week. As it's also expensive for an application to poll the CRL repository, CRL standard specifies a next update field containing a date the application can use as a hint to poll the repository.

The last CRLchecker method, check(), uses CRL getRevokedCertificate to learn whether a certificate has been revoked using its serial number. It throws an exception if this is the case.

Summary

The Java 2 framework has the flexibility required to implement sandboxes in an application server, still relying on the Java 2 policy files, keytool, and JARsigner. It also provides classes to check on whether the credentials you use are still valid. The major difficulty in this area is that it encompasses traditionally separate spheres of knowledge.

Regarding the requirement of supporting archives that are downloaded like applets, we see that this method clearly enhances its area of application. You can select Web applications on the Web and download them in your application servers, local or remote, without compromising your security. Readers can go to http://pagebox.net/ASversion.hxm for sources, documentation, and binaries for Tomcat and Resin.

Author Bio

Alexis Grandemange is an architect and system designer. A Java programmer since 1996 with a background in C++ and COM, his main interest is J2EE with a focus on design, optimization, and performance issues.

alexis.grandemange@pagebox.net

Effective Application Deployment with Embedded Java Technology

A significant advantage for systems designers and developers

WRITTEN BY
MARC R. ERICKSON

Connectivity changes everything, especially with embedded computing technology. Since we're entering a world in which things will link and think, it's clear that many new projects will begin to incorporate more advanced and sometimes complex technology.

Options are available for implementing whole new classes of applications with embedded devices. Development engineers must judge these options carefully to accommodate possible resource constraints and emerging standards and specifications.

Development Issues

As smart devices become more capable, it's becoming increasingly important to adhere to industry-oriented standards and specifications while avoiding the task of building and supporting basic components as part of the overall project. The issue is one of focus. If development engineers are mired in trying to complete the low-level facilities of a platform, they could run out of time and resources when it's finally time to focus on project-specific components.

When working with embedded Java technology, developers get a head start toward project conformity and shorter time-to-market thanks to the well-defined and well-known base Java class libraries. When developers target embedded platforms, these libraries can be chosen with different configurations, considerably reducing the size of Java applications. This impacts both the number of classes and methods included in libraries, and the amount of code needed to implement a method. For example, when a small library configuration without security support is selected, the java.security classes are eliminated, as well as the internal references and manipulation of security-related data within other classes throughout the library.

Several Java Community Process Expert Groups are now working on "Micro Edition"–related class library configurations. Vendors are providing different approaches to implementing configurations, including Java compatible libraries that adhere to JCP specifications and other libraries that can be used to meet special customer requirements. Both types of implementations are available in IBM's VisualAge Micro Edition tools and runtime components.

"Personal configuration" components add several additional libraries to those found in the base Java class libraries. For example, components for the following use are included:

- Windows-based user interfaces (AWT and SWT technology)
- Image map-based user interfaces (MicroView model/event/view-based technology)
- Relational databases (with JavaSQL parts that interface with embedded databases like DB2/Everyplace)
- Remote Method Invocation (CORBA RMI technology)

Additional components are available to provide access to cross-industry and industry-specific services, for example:

- Transaction messaging (with products like MQSeries Everyplace)
- Component bundle management (with OSGi-compliant components)
- Automotive bus interfaces (compatible with MOST, CAN, and IEEE-1850 protocols)
- GPS and cellular phone control interfaces
- Home Internet gateways

The OSGi bundle management technology is of particular interest (see www.osgi.org). Using tools that package and identify bundles of components with appropriate descriptive data, bundle management works with a virtual machine and bundle servers to provide bundle delivery and hot-code activation for maintaining applications. This can be used to add and remove the features of an offering as well as update data and replace code. Bundle management can be completely automated, allowing a system to be designed with the minimum usage of space on a device. For example, a product-diagnostic package could deliver in-depth analysis components upon recognition of a given error code.

Sharing the Development Workload

Many developers considering pervasive solutions will turn to partners to create and deploy the portions of the system that reside on embedded devices. Object-oriented Java and the use of virtual machines provide several advantages for fast project development, reduced time-to-market, and ongoing product maintenance and update. Thanks to the relative portability of components written in Java, it's possible to reuse existing logic from earlier projects and do much of the development in a cross-platform environment.

An embedded device is equally adept at running both client and server components. Deeply embedded devices may not even have a user interface, responding instead to requests from other devices in the network that gather infor-

mation and render the interface to the user. Much of the advantage of pervasive computing solutions comes from completely automating activities, with servers communicating directly with smart connected devices.

In some cases an experienced embedded developer will have access to major program assets designed in direct executing languages such as Assembler, C, and C++. These are usually specific to platform CPU architectures and RTOS environments. Examples include speech recognition software, specialized device drivers, and existing applications that need to be adapted for pervasive system integration. The embedded Java environment makes the Java Native Interface (JNI) available for this work. In implementations, such as IBM's J9 virtual machine, special attention has been paid to making this interface compact and fast.

With the use of middleware and the ability to reuse existing logic through JNI, the elements of an end-to-end pervasive computing solution can be incorporated quickly, improving time-to-market. While embedded platforms are not personal computers (lacking the PC's uniform approach to device attachment and large resource pools), powerful connected-device solutions can be constructed and maintained. Today's embedded platforms and processors are achieving remarkable power while using space efficiently, reducing heat generation, and conserving power – often enabling battery-operated mobile use.

Each embedded device platform is unique. In many cases the connected device will be deployed with custom-engineered hardware. Designers can select from a broad range of processors, including 32-bit PowerPC, X86, Pentium, SuperH, ARM, DragonBall, and MIPS. Clearly, the improved portability of Java makes a difference in reuse on embedded platforms.

Embedded software is usually developed using personal computer–based developer workstations (VisualAge Micro Edition tools run on both Windows and Linux platforms), and functional testing can be done on the same workstation using a Windows or Linux version of the J9 virtual machine. Cross-platform development is consistent because IBM's J9 VM is generated from the same source for all target platforms, including the one supplied for use on developer workstations.

The actual device hardware does change some aspects of the execution environment. For example, on a device that doesn't have floating point math hardware, the VM and RTOS will emulate these facilities. Obviously, applications will run more slowly in emulation. There are other considerations as special devices are supported. Flash memory (for file system simulation and direct execution memory), graphic user interface layers, and communication interfaces may have different characteristics on different platforms.

Getting Started

During early project development, it's often wise to include a reference platform that integrates the actual target-device processor and several related devices. This allows the application to be tested and tuned on the processor and the devices that will be used in the deployment of the actual project.

In many cases the vendors of embedded Java and RTOS technology will perform this integration on a range of standard reference platforms. Offerings that include services and reference boards are available from vendors such as QSSL (www.qssl.com), Motorola (www.motorola.com/mobileGT), MontaVista (www.mvista.com), and IBM's Object Technology International, Inc. (www.embedded.oti.com). These offerings help developers quickly start their projects and focus on application instead of integration issues.

Embedded platforms usually have limited resources available to application programs. Beyond applications that occupy about 500KB, Java byte code technology is actually more compact than raw machine language. By combining compact and well-engineered base Java class libraries, middleware components, and adaptive compilation techniques (Just in Time [JIT] and Ahead of Time [AOT]), it's possible to create more compact and efficient applications in the Java language.

Embedded RTOS components, the virtual machine, Java class libraries, and extended components are often configurable. This allows components on a platform to be tuned for efficient memory utilization and fast execution. IBM's SmartLinker technology even allows the automatic elimination of unreferenced classes from the base Java class libraries. The J9 VM includes "plug-in" technology for features such as dynamic loading, debugging, JIT, and analyzers.

Deployment Issues

Clearly, it'll be necessary to provide scalable server support for pervasive solutions. Existing networks and servers may not be able to handle the workload of thousands to millions of devices. For this reason, an early phase of project design must consider the architecture and deployment hierarchy of the ongoing processing demand of devices and the periodic maintenance requirements of bundle management technology.

Java technology is ideal for this use thanks to support for servlet and applet program structures. Advanced techniques that automate server workload distribution have already been tested and deployed in products such as IBM's WebSphere servers. For example, workloads of millions of interactions per day were handled efficiently during IBM's support of the Web sites for the Olympic Games in Nagano and Sydney.

After product deployment, bundle management involves the selection of appropriate components from those available on bundle servers, and the delivery and activation of those components on embedded-device platforms.

Often network hierarchy will include high-capacity devices close to the embedded platform that can be used to

Author Bio

Marc R. Erickson is a project manager for Object Technology International, a subsidiary of IBM. He holds a degree in data systems management from Southern Illinois University.

stage and store bundles. Examples include the Telematics computer that provides the user interface, mobile interconnection, and data storage resources on a vehicle, and the Internet gateway processor that will be part of home automation networks. These designs enable developers to reduce the component size on deeply embedded devices. For example, connection security facilities may only be needed at the point in which a vehicle or house attach to the external Internet, since the network inside a house or within a vehicle is protected by physical access control.

Keeping Up with Changes

The task of maintaining components for embedded devices involves several distinct steps. First, a deeply embedded device must determine where its bundle server is located on a network. This is done using service discovery protocols. There are now about 23 competing implementations of service discovery components. One of the most interesting is the "Salutation" protocol, since it's robust and open.

Bundle management allows developers to work with connected devices after a product is in customer hands. It usually can accomplish this without the inconvenience of returning a product to a service center. The OSGi specification for bundle management defines how metadata is stored and used to identify appropriate bundles for a device. This is based upon device and platform architecture and related component release levels. The selection is based upon a secure exchange of authentication information, which is used to authorize and select the appropriate bundles for a device. More details are available at OSGi's Web site, www.osgi.org.

When a bundle has been delivered to a device, it's best to avoid interrupting the user. When a virtual machine–based technology is used, it's possible to suspend a class's execution and then hot-code replace the component, restarting it at the interrupted method. With careful design, the user need not even be aware of the product upgrade or maintenance activity.

Conclusion

Several important extensions of embedded Java have recently become available to pervasive solution developers. Since the environment of connected embedded devices is quite different from the personal computer and server environments that many systems engineers are familiar with, it's wise to consider working with specialists familiar with embedded software design and integration. Java offers a significant advantage to systems designers and developers with its well-known base class libraries, configurable components, bundle management, and advanced middleware options.

mre@us.ibm.com

DATABASE PROGRAMMING WITH JDBC AND JAVA

BY GEORGE REESE

PUBLISHED BY O'REILLY & ASSOCIATES

Some Good New Features, But Not Up to Expectations

REVIEWED BY JAMES MCGOVERN

Database Programming with JDBC and Java was originally published four years ago and is now in its second revision. Significantly improved over the first edition, the book is targeted toward those who want ideas on how database programming works with Java. This latest edition also covers advanced topics such as serialization, persistence, and security.

A growing trend popular with book publishers is to incorporate Javadoc printouts of the Java APIs into new books – even though you can download them for free. This practice allows publishers to command high prices for their books…and make them look larger. This book is no exception. It totals 330 pages, with about 217 pages of actual content. One third of this book is stuff you can get free.

In writing this review I compared the information in the book to the Sun JDBC tutorial. The title and the content definitely don't match. The book discusses the use of RMI (Remote Method Invocation), EJB (Enterprise JavaBeans) containers, and JNDI (Java Naming and Directory Interface). It opens with a brief overview of ANSI standard SQL, and is written primarily for developers who've never worked with a relational database before. The next chapters deal more with working with databases than with JDBC itself.

Chapters 3 and 4, which discuss JDBC briefly, cover typical programming scenarios such as working with stored procedures, batch processing, updatable result sets, and advanced data types. The JDBC 2.0 specification now recommends a standard way of dealing with connection pooling. In the past, individual developers addressed this mainly with reference to the application server you were deploying. Chapter 5 contains just six pages of text on the JDBC optional package that contains this functionality.

I thought Chapter 7, which covers architecture and design patterns, was good. Except for the same stereotypical example of how an applet and Swing use the model/view pattern, it provided some useful information. If you're not familiar with design patterns, I recommend *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley).

Chapter 8, on distributed component models, covers security, transactions, lookup, searches, and entity relationships. The chapter contains some useful code examples for

generating unique sequence numbers, generic facade (a type of pattern), and implementing collections. Chapter 9 is about persistence and provides solid examples of a framework that is very useful in high-throughput database applications.

Chapter 10 covers the design of a UI that interfaces with business objects. Today 99% of applications developed in Java use HTML for the interface. The examples in this chapter use Swing and don't address any concerns of developers whose interface is HTML. I did learn some tips about Swing from this chapter – they'd be useful in pursuing the Sun certification exam – but otherwise it didn't provide any insight into how to develop a typical Web application.

The book doesn't provide many opportunities to learn anything other than what you could figure out for yourself from the Sun JDBC tutorial and reference. The publisher, O'Reilly, is known for delivering high quality, but this book fails to live up to the standard I've come to expect as a buyer of their books. I recommend you save your money and buy a good J2EE book.

jmcgovern@enherent.com

AUTHOR BIO

James McGovern, coauthor of several Java-related books, is a senior technical architect for Enherent Corporation in their software development center in Windsor, Connecticut. His focus is on strategy and architecture for high-profile e-business Web sites.

JDJ ADVERTISER INDEX

ADVERTISER	URL	PHONE	PAGE
Appeal Virtual Machines	www.jrockit.com	46 (0) 8 50630900	55
BEA	www.bea.com	800.817.4BEA	4
Borland	www.borland.com	800-632-2864	49
Brokat	www.brokat.com	408-275-6900	19
Cape Clear	www.capeclear.com/download	866-CAPE226	71
Career Opportunities Section		201-802-3028	125
ComponentSource	www.componentsource.com/java	888-850-9911	13
Compoze Software	www.compoze.com	866-COMPOZE	101
Corda Technologies	www.corda.com	801-805-9400	77
Dynamic Buyer Incorporated	www.dynamicbuyer.com	845-620-9800	113
Elixir Technology	www.elixirtech.com/download	65 532-4300	89
Esmertec	www.esmertec.com	877-751-3420	53
Fiorano	www.fiorano.com	800-663-3621	99
Flashline, Inc.	www.flashline.com	800-259-1961	79
FuegoTech	www.fuegotech.com	800-355-7602	23
Hit Software	www.hitsw.com	408-345-4001	81
HostPro	www.hostpro.com	877-467-8464	17
IAM Consulting	www.iamx.com	212-580-2700	105
ILOG	www.ilog.com/jdj	800 FOR ILOG	37
Informix	www.cloudscape.com/freedev	888.59.JAVA1	31
Infragistics, Inc.	www.infragistics.com	800-231-8588	14-15
INT	www.int.com	713-975-7434	22
IntraNet Solutions	www.intranetsolutions.com		27
IONA	www.iona.com	781-902-8000	67
JavaEdge Conference & Expo	www.sys-con.com/javaedge	201-802-3069	110-111
JavaOne 2001	http://java.sun.com/javaone/		103
JDJStore	www.jdjstore.com	888-303-JAVA	119, 123
Jinfonet Software	www.jinfonet.com	301-990-6330	51
JustComputerJobs	www.justjavajobs.com	877-905-NERD	16
Leapnet	www.leapnet.com	(312) 528-2400	115
LOOX Software, Inc.	www.loox.com	800-684-LOOX	61
Macromedia	www.macromedia.com	888-939-2545	33
NetDive	www.netdive.com	415-981-4546	109
New Atlanta Communications	www.servletexec.com	678-366-3211	83
No Magic	www.magicdraw.com	303-914-8074	7
Northwoods Software Corporation	www.nwoods.com	800-226-4662	70
Parasoft	www.parasoft.com/jdj4	888-305-0041	69
Pingtel	www.pingtel.com/javachallenge		39
Pramati	www.pramati.com	877-PRAMATI	93
PreEmptive Solutions	www.preemptive.com	800-996-4556	85
Programix	www.jthreadkit.com		48
Progress Software	http://www.sonicmq.com/jdj501.htm	800-989-3773	2
ProSyst	www.prosyst.com	866-PROSYST	75
Quadbase	www.quadbase.com	408-982-0835	54
QuickStream Software	www.quickstream.com	888-769-9898	96
Rational Software	www.rational.com/jdj2	800-728-1212	41
RSA Security	www.rsasecurity.com/go/paint	866-432-7233	25
Sandia National Laboratory	http://herzberg.ca.sandia.gov/jess		20
Segue Software, Inc.	www.segue.com	800-287-1329	47
SilverStream	www.silverstream.com	800-465-5680	35
Sitraka Software	www.sitraka.com/greatapp/jdj	888-361-3264	93
Sitraka Software	www.sitraka.com/j2ee-jdj	888-361-3264	21
Sitraka Software	www.sitraka.com/serverchart/jdj	888-361-3264	128
Sitraka Software	www.sitraka.com/promise/jdj	888-361-3264	63
Softwired	www.softwired-inc.com	41-14452370	73, 97
Sun Microsystems	www.sun.com/service/suned/training	800-422-8020	57
Sybase	www.sybase.com	800-8-SYBASE	43
SYS-CON Media Reprints	www.sys-con.com	201-802-3024	121
Talarian	www.talarian.com/jms	650-965-8050	11
Tidestone Technologies	www.tidestone.com	800-884-8665	59
Togethersoft Corporation	www.togethersoft.com	919-833-5550	6
TopCoder	www.topcoder.com	866-TOPCODE	29
Unify Corporation	www.unifywave.com/jdj	800-GO UNIFY	91
Valtech	www.valtech.com	888 240 6028	45
Verge Technologies Group	www.verge.com		107
VM Gear	www.vmgear.com	888-655-0055	87
WebGain	www.webgain.com	877-WEBGAIN	127
Zero G	www.ZeroG.com/installs	415-512-7771	3

Sims Computing Releases Flux 2.1

(Billings, MT) – Sims Computing has announced the release of Flux, the Enterprise Job Scheduler, version 2.1. The new version remains lightweight yet has several new features such as queuing, scalability, holiday calendars, and an audit trail.

Flux is application server and database independent.
www.simscomputing.com

Kada Systems Offers Java Apps for Palm OS

(*Andover, MA*) – Kada Systems, a leading provider of solutions that enable Java applications for mobile e-business, has introduced the Kada Mobile Platform for the Palm OS, reportedly the industry's smallest, fastest, most complete and easily ported Java application platform for mobile devices.

Kada Mobile, which is half the size of any competing, full-function Java Virtual Machine, offers the only just-in-time compiler for handheld devices. It includes a full implementation of the Java APIs, and supports standard development tools such as Visual Café, Forte, PowerJ, and Code Warrior.
www.kadasystems.com

INT Upgrades Java 3D Technology

J/View3DPro 2.0 is now available from Interactive Network Technologies, Inc. (INT). A graphics toolkit based on Java 3D, it allows programmers to visualize, manipulate, annotate, and edit complex 3D scenes.
www.int.com.

LOOX Software Releases LOOXGIS 1.3

(Paris, France / Burlingame, CA) The new version of LOOX Software's high-performance Java map-rendering component for Java 2 developers offers new levels of performance and accuracy for applications including management of wireless networks, fleet management, and intelligent transportation systems.
www.loox.com.

Sitraka Software Integrates JClass with Forte for Java Development Environment

(Toronto, ON) – Sitraka Software's JClass Java technology-based components are fully integrated with Sun Microsystems' Forte for Java. The components are now available through Flashline's Components Marketplace within the Forte for Java portal.

Through the Forte for Java Extension Partners Program, Sitraka Software (formerly KL Group) will provide tight product integration across all its product lines, beginning with its JClass family of components, also creating modules designed specifically for use with the Forte for Java product.
www.sun.com
www.sitraka.com

Sun Releases JMF 2.1.1

(Los Angeles, CA) – Sun Microsystems has released the Java Media Framework API software (JMF), a "one-stop shop" for multimedia on the Java platform. JMF 2.1.1, an optional API package for J2SE, provides a unified architecture for the capture, playback, streaming, and transcoding of media content across most major operating systems. JMF source code will be released under Sun Community Source Licensing (SCSL).
java.sun.com

Development Tool Extends OLAP Capabilities to Java Platform

(Sunnyvale, CA) – A Java 2 Enterprise Edition–compliant application development tool that enables the rapid creation, management, and deployment of Web-centric custom business analysis applications is available from Hyperion.

Hyperion Application Builder allows companies to reduce application development time and the total cost of ownership for building and deploying custom analysis applications in cross-platform environments.
www.hyperion.com

Infragistics Releases New Products

(Cranbury, NJ) – Infragistics, formed by the merger of ProtoView Development and Sheridan Software, has released JSuite 5.0 and JFCSuite 5.0. The products allow Java developers to add a feature-rich presentation layer to their applications quickly, easily, and cost effectively. Infragistics has also released PowerChart 2.0, included with JSuite and JFCSuite.

Each suite includes charting, a grid, data input components, an explorer, a tree, and calendaring.
www.infragistics.com.

IPlanet Extends Portal Server Offering

(Hannover, Germany) – iPlanet E-Commerce Solutions, a Sun–Netscape Alliance, has extended its portal platform with the addition of integrated wireless capability via the iPlanet Portal Server: Mobile Access Pack. The product lets service providers offer a full spectrum of portal services through any Internet-based device and allows enterprises to channel corporate services and applications to mobile employees to help increase productivity and promote rapid access to business-critical information.

The product is fully integrated with the iPlanet Messaging, Calendar, and Directory Server software products.
www.sun.com www.iplanet.com

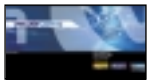
Internetwork Introduces Sm@rttest

(Worcester, MA) – Addressing the need of large- to mid-range networks to determine the responsiveness of their load capacity and efficiency, interNetwork, Inc., has introduced sm@rtTEST, a test suite for mid- to large-scale enterprises, inter-intranets, and telecommunications networks. The test engine creates the behavior of an unlimited number of real users through automated scenarios stressing the network and application services.
www.internetworks.com

Graphics and Virtual Studio Solution

(New York, NY) – vi[z]rt Ltd. has entered into a strategic alliance to bring unique visual presentation tools to the corporate market under the name vi[z] presenter.

As part of the turnkey corporate presentation solution, vi[z]rt will feature three packages of its



vi[z] presenter product line: modeling for the real-time creation of 2D or 3D graphics; content management for the coordination of graphics through specifically designed corporate and financial templates; and a virtual studio package that allows for the integration of the corporate presenter into a virtual set using the Chromatte system. This system reduces the need to set up complex and controlled lighting conditions, and enables anyone to achieve high quality chroma-keying.

www.vizrt.com
www.viewercom.com

Marketplace Manager Adds Classifieds

(Salt Lake City, UT) – Infopia, Inc., developers of Marketplace Manager, have added the Excite Classifieds Network to their list of online shopping destinations where merchants can list their products.

www.infopia.com.

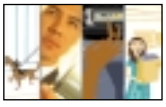
Maxim I/T Integrates Autovue for Java into Findview

(Lake Buena Vista, FL) – Maxim I/T and Cimmetry Systems have integrated Cimmetry's Web-based viewing technology into FindView2.0. Cimmetry's technology will aid users of Maxim I/T's enterprise search technology, FindView, to view a broad range of formats.

www.maxim-IT.com
www.cimmetry.com

Motorola Reinvents Wireless Phone as Digital Personal Companion

(Schaumburg, IL) – Motorola has a new generation of wireless handsets that bring computing power to the palm of your hand



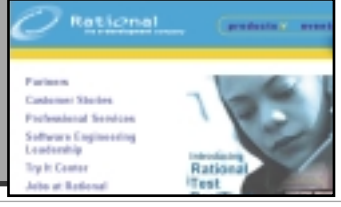
and combine many of the capabilities of a handheld computer, two-way radio, interactive pager, and Internet-ready mobile phone in a single device.

Two new handsets are the first wireless phones in North America to incorporate Java 2 Platform, Micro Edition, the software environment from Sun Microsystems. The offline capabilities of J2ME technology enable users to run applications even when disconnected from the network. For more information or to obtain a free CD-ROM, visit www.motorola.com/.

Rational Software and BEA Announce Add-In

(Lexington, MA / San Jose, CA) – Rational Software and BEA Systems, Inc., have announced a Rational Unified Process Add-In for the BEA WebLogic Server 6.0.

The add-in offers comprehensive software development guidelines, examples, and templates for building high-volume, mission-critical e-business applications on the BEA WebLogic Server. A free download is available at www.rational.com and developer.bea.com.



Compuware Expands Testing Solutions

(Farmington Hills, MI) – Compuware Corporation has released QAHiperstation 7.0, providing support for the IBM WebSphere software platform for e-business. The release enables organizations to leverage their S/390 investments for e-business deployments.

QAHiperstation provides a complete solution for focused testing of S/390 business logic. QAHiperstation 7.0 now includes native support for S/390 applications accessed from a Web browser. This new release also reveals the TCP/IP and APPC messages that drive the S/390 business logic in multitier applications.

www.software.ibm.com/websphere
www.compuware.com

WebGain Accelerates Java App Development, Introduces Quality Analyzer

(Santa Clara, CA) – WebGain, Inc., has recently acquired Metamata, Inc., a supplier of Java

development environment and productivity software. The addition of Metamata allows WebGain to enhance its distributed debugging capabilities with additional code analysis, metrics, and coverage functionality.

WebGain also introduced WebGain Quality Analyzer, a Java-based product that enables Java development teams to improve the quality and performance of enterprise-class Java applications.

The product consists of three main components: WebGain Audit, WebGain Cover, and WebGain Metrics. Each is designed to assist managers and developers at specific stages of application development.

www.webgain.com

PointBase Releases 3.5

(Mountain View, CA) – PointBase 3.5, the latest upgrade to Pointbase's family of pure Java, small-footprint database products, has been released.

The new version includes JDBC 2.0 API support for batch operations and BLOB/CLOB, scrollable cursors, and SQL caching. All are essential for decreasing development time and providing more efficient programmatic development.

www.pointbase.com

O'Reilly Offers New Edition of Java Book

(Sebastopol, CA) – "Since I wrote the first edition of this book, servlets and the server-side Java platform have grown in popularity beyond everyone's wildest expectations," writes Jason Hunter, author of *Java Servlet Programming, Second Edition*. "The servlet world has changed over the last two and a half years, and [this edition] brings readers up-to-date."

Like the bestselling first edition, *Java Servlet Programming, Second Edition*, by Jason Hunter with William Crawford, covers the servlet life cycle; how to use servlets to maintain state information effortlessly; how to serve

dynamic Web content, including both HTML pages and multimedia data; and advanced topics like integrated session tracking, efficient database connectivity using JDBC, applet-servlet communication, and internationalization.

In addition, Hunter's book introduces JavaServer Pages and explains how a servlet programmer can (and should) use them. It also provides tutorials on several servlet-based content creation frameworks currently used in production sites, including WebMacro/Velocity, Tea, XMLC, and ECS.

www.oreilly.com

CocoBase Enterprise

O/R V3.1 Service Release 9.0

by THOUGHT Inc.

REVIEWED BY JIM MILBERY

AUTHOR BIO

Jim Milbery is a software consultant with Kuromaku Partners LLC (www.kuromaku.com), and is based in Easton, Pennsylvania. He has over 17 years of experience in application development and relational databases. Jim is the applications editor for Wireless Business & Technology, *the product reviews editor for* Java Developer's Journal, *and a coauthor of* Making the Technical Sale *(Muska & Lipman).*

jmilbery@kuromaku.com

THOUGHT Inc.
657 Mission Street
San Francisco, CA 94105
Web: www.thoughtinc.com
E-mail: info thoughtinc.com
Phone: 415 836-9199

Test Environment: Sony Vaio Pentium II 366MHz 256MB RAM

Last month in ***JDJ*** (Vol. 6, issue 4) I introduced the topic of object/relational mapping. Databases such as Oracle8*i* or DB2 store data in tables and columns. Thus, customer data is stored in a "customer" table and information relevant to the customer such as ID, name, and address are stored as columns. All the data for a single customer within the customer table is equivalent to a "record." From the EJB perspective customer data is represented by a customer "class" and the data elements are represented by "attributes." Conceptually, the mapping process is a simple one. Each database table is an EJB class (CMP or BMP), and each and every column in the table becomes an attribute. Individual customer records are instantiated as EJB objects as necessary.

This month I tested the latest release of THOUGHT Inc.'s CocoBase Enterprise O/R mapping tool on my Windows 2000 server. Java-centric software companies such as THOUGHT Inc. take full advantage of both Java and the Internet when it comes to software development. They continually enhance their products with new features and bug fixes.

Last month I previewed "Service Release 8," and this month I was able to upgrade to "Service Release 9" – which included more than a dozen feature enhancements. The installation of CocoBase is packaged as a Java class file; it's a simple process to extract the installation files and install the software. The Java installer isn't fully Windows 2000 compatible yet, so CocoBase doesn't create desktop icons or menu items – all the software must be accessed via a series of command scripts. Nevertheless, it's a simple process to create your own desktop icons that point to the most common functions. The starting point for working with CocoBase is the Enterprise Administration Interface. THOUGHT Inc.'s choice of terminology here is a little off the mark as the admin interface is really the heart of CocoBase. All the major functions of the software are accessible from within this one interface – which is itself a Swing-based GUI program (see Figure 1).

CocoBase can connect to any of your enterprise data through a powerful JDBC interface. The software also comes equipped with a built-in SimpleText, Hypersonic SQL, InstantDB, database, and you can use this data source to get accustomed to the many tools within the CocoAdmin interface. I elected to work with the SimpleText database and to create a new database map (as shown in the upper window in Figure 1).

You're free to map multiple CocoBase objects to a single database table, and I quickly created a variation on the Employee Table that would list only those employees that make over $20,000 annually. I called this new object EmployeeGT20000 and generated a new data map for this object from within CocoBase within a few minutes. I chose to build this object based on a CocoBase-supplied "where clause" against the salary field using a variable for the salary. Technically speaking, this object could be used to represent any group of employees by supplying different values for this parameter. This new object extends the Object class and either doesn't implement any proprietary interfaces or implements a number of interfaces including Cloneable, CBProp (a CocoBase class), and java.io.Serializable.

Once I had defined the EmployeeGT20000 map, I used CocoAdmin to generate Java source code into a package that I defined as "jmpkg". I could easily have added custom attributes and derived fields as part of the generation process. In either case, the resulting code is straightforward Java code as shown in the right-hand panel in Figure 1. The generated code includes get/set methods for all the attributes, a "where-clause" facility, and a facility for managing query information. Feel free to extend and modify this code. CocoBase can be used to manage data using both the BMP and CMP entity bean types. THOUGHT Inc. provides interface code for a wealth of third-party application servers including Allaire, Borland, IBM WebSphere, HP/Bluestone Sapphire, iPlanet, BEA WebLogic, and Sybase. (The development environment integrates with a number of popular Java IDEs such as VisualAge for Java, Borland's JBuilder 4, and iPlanet's Forte for Java.) This review covers only a fraction of CocoBase's capabilities and I'd encourage you to review the PDF documentation for additional details.

Summary

CocoBase's O/R tool can greatly simplify the task of interfacing Java code with relational data. In contrast with complete "frameworks," CocoBase improves productivity without sacrificing low-level control. You're free to modify the Java source as necessary to address your specific application needs. I'd encourage you to put CocoBase on your short list of products to consider when building a data-centric J2EE application.. ●

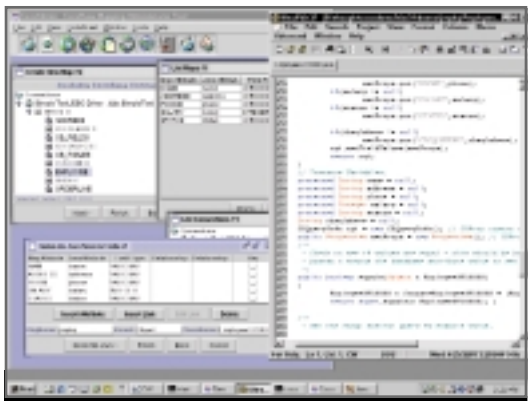


FIGURE 1 CocoAdmin with its many functions on display

The Paper Chase

Résumés: Greatest ally . . . or worst enemy?

WRITTEN BY
Bill Baloglu &
Billy Palmieri

As staffing professionals we read a lot of résumés. The most common problem is that they misrepresent candidates as being more skilled and experienced than they really are…or, worse, they misrepresent seasoned candidates as less experienced than they are. A good résumé is a clear, detailed visual representation of who you are, what you've done, and what you want to do. The key to writing a good one lies not in the beauty of your creative writing, but in highlighting how your skills and experience are relevant to the position you're applying for.

Objective

There are pros and cons to including an objective. If you're applying for a full-time job and have specific goals that this position will help you achieve, an objective could be helpful. However, most objective statements are too generic and therefore meaningless.

Omit the objective statement and let your skills and experience speak for themselves.

Skills

In a technical résumé the specific skills and proficiencies are key. Put the skills section right up front. And, *please*, organize it!

Group skills in order of most recent use and according to specific categories. An organized list indicates that you know which tools are related to each other. It can also suggest how you've used the skills and for what purpose. And break out the skills into categories, such as Platforms, Languages, and Operating Systems.

Skills are keywords in more ways than one. Once you've listed your skills, continue to list them throughout the experience section. It's always helpful to know what tools you've used most recently, most often, and when and where. Most online recruiting is based on keyword searches that produce results based on the number of times those keywords appear in the résumé.

If you have a lot of antiquated tools and technologies on your résumé, think twice about including them. Focus your skills and experience around what you're doing now and what you want to do next. The résumé doesn't need to be your whole life story. Pick and choose what you want to include to get the job.

Functional Résumés

Some people recommend a résumé that begins with detailed descriptions of the functions you've performed, followed by a minimal list of employment dates and experience. From our perspective, three words come to mind: *Don't Do It*.

Many staffing and hiring professionals scan résumés to zero in on the most important information: the experience section. They need to know what you've done and when and where you've done it.

Experience

The experience section is the résumé's meat and bones. Each of your positions should be listed with your title, what you did, what tools you used to do it, and how long you did it.

If you oversell yourself, the worst thing that can happen is that you'll get the job. Time and time again, junior to mid-level developers get jobs they're not qualified for based on misleading résumés. This person is quickly overwhelmed, can't perform the job he or she was hired to do, and is soon revealed to be incompetent.

A detailed paragraph of what the company did is not meaningful information. A detailed paragraph about what your specific role was at that company is both pertinent and meaningful.

Dates of hire should include both month and year. Employers are wary of many short-term positions, so if you've had a series of contracts, list them as such.

It's always better to be honest about your career history than to appear to be hiding something.

Don't list every job you've ever had. The résumé should represent who you are and what you do now.

Entrepreneurial ventures can be an asset if presented in light of the job you're applying for. But they can also be a liability. You may have been CEO of your own venture, but the title won't help you get an engineering job – and it may hurt you.

If you've done a lot of management, emphasize it only if you're applying for a management role. Management skills listed on a developer's résumé might cause the hiring manager to think: "This person's going to want to manage everything – and will probably want my job."

Remember, whatever you list in the skills section must be listed in the experience section to give it validity.

Education

Include degrees earned, and the names and locations of all educational institutions, but leave dates off. The law forbids employers from considering candidates based on age. Dates of graduation provide information that by law they must ignore.

Bachelor's and master's degrees in computer science and related fields are always a plus. But MBAs can send up a red flag to employers who are just looking to hire a hands-on developer.

A PhD on a résumé can also be tricky if you're looking for a straight developer job. Some managers think of PhDs as being more interested in exploring complex theories than in creating practical real-world solutions.

Military

This experience usually suggests a well-disciplined candidate with a good work ethic. But describe your military experience only as it relates to the job you're applying for.

Hobbies and Personal Information

Although hobbies and sports can be good conversation starters in interviews, they can also backfire. And leave off all references to age, marital status, family, and health as employers are forbidden by law to consider any personal information in their hiring decisions. For the same reason, leave off that little photo of yourself and let your work experience show your stability.

Author Bios

Bill Baloglu is a principal at Object Focus (www.ObjectFocus.com), a Java staffing firm in Silicon Valley. Prior to that he was a software engineer for 16 years. Bill has extensive OO experience ,and has held software development and senior technical management positions at several Silicon Valley firms.

Billy Palmieri is a seasoned staffing industry executive and a principal of ObjectFocus. Before that he was at Renaissance Worldwide, a multimillion-dollar global IT consulting firm where he held several senior management positions in the firm's Silicon Valley operations.

billb@objectfocus.com

billp@objectfocus.com

Bean-test 3.1

by Empirix, Inc.
(Formerly RSW Software)

REVIEWED BY CEDRICK W. JOHNSON

AUTHOR BIO

Cedrick W. Johnson works with object-oriented programming projects at Acxiom Corporation in Downers Grove, Illinois. Johnson is also studying computer science at Northern Illinois University.

cedrick@cedrick-johnson.com

JAVA DEVELOPER'S JOURNAL JDJ WORLD CLASS AWARD

Empirix, Inc.
1430 Main Street
Waltham, MA 02451
Phone: 781 993-8500
Web: www.empirix.com
E-mail: info@empirix.com

System Requirements:
Memory: 128MB recommended minimum
Hard Disk Space: 50MB recommended minimum

Test Environment:
BEA WebLogic 5.1.0 Service Pack 8, AMD K6-2 500 MHz, Windows 2000, 190MB RAM.
For extra Bean-test remote "agent":
Pentium 120, JDK 1.3, RedHat Linux 7 with 48MB RAM.

In this new era of rapid application development (RAD), there's an ever-increasing push to get applications into production without adequate testing. This methodology does meet deadlines, but it can also lead to serious implications for your business's future. For example, many Internet companies deploy applications that lack the ability to handle high loads, or their applications aren't scalable enough to grow with the increasing demands of the business. With a little testing these simple mistakes can be caught before an application goes into production.

I've been developing applications since 1996 and have seen an increasing trend within many development teams to rush a product out the door without adequate testing. Recently I reviewed a new version of a testing tool named Bean-test from Empirix, Inc. Bean-test is an application tool built from the ground up to test your EJBs on BEA WebLogic, IBM WebSphere, Bluestone's Total-e-Server application servers, or any other EJB 1.1 application server.

Installation of Bean-test is straightforward. Since it's built using 100% Java, installation of the server and associated client agents are relatively simple on many platforms. Once the installation is complete, starting Bean-test on Windows is a breeze. Within a matter of seconds the Bean-test user interface pops up in your default Web browser, and you're ready to begin testing EJBs.

Getting Started with Bean-test: Sample EJBs

Empirix provides an excellent tutorial session on using the demo beans. During your evaluation period you have the option of spending one hour with an Empirix representative who will show you the entire program and how to generate a test case. To use the demo beans you need to deploy them to your application server; in my case, I deployed them to a WebLogic 5.1.0 SP8 server running on the same machine. To deploy the sample beans to the application server, click on a file located in the Bean-test installation directory and add a couple of WebLogic JAR files to the Bean-test classpath configuration page. Now you're ready to begin generating test cases (see Figure 1).

Once the Bean-test user interface is loaded, you should first set up your servers. A neat feature within the Bean-test product is that you can install Bean-test clients (agents) on remote machines. In the setup tab you can add or configure the machines that will participate in the test process. With the agents you can specify the number of Java Virtual Machines (JVMs) that will be spawned on those machines as well as the weight of that system. This comes in handy if you have several machines with different amounts of memory or processing power, as it provides an accurate test of your EJB. Also, under the setup tab you can edit the application server properties. This enables you to add, delete, or modify application servers, classpath variables, and other information.

With Bean-test, project creation is easy and quick. You need to indicate the project name, select the type and version of application server you wish to use, and specify the classpath to the EJB JAR files that are installed on the application server. Once this is completed, Bean-test will automatically find and display the EJBs it found within your JAR files. You can even tell Bean-test to automatically create test cases for all the beans it finds. This is especially handy if you have 50 beans and wish to test them all.

Now that a test case has been generated, you can view the generated client code. A feature of Bean-test is that all code is generated in a standard open fashion, enabling you to either edit the client code using the built-in editor within the Bean-test UI, or load the file into your favorite development tool for further editing.

Bean-test is very flexible with regard to actual test data. For supplying test data you can choose between randomly generated data and data from a data table, or enter the data directly on the screen. Data tables are typically comma-delimited files and can be edited with any type of editor. This allows for greater control over the application data that will be supplied to your EJB.

Running the Test Case

If you have distributed clients set up, you can request the server to pack up the appropriate client files and ship them out to each remote client. When your tests are done, Bean-test's extensive reporting system allows you to see where problems can develop within your application. It enables you to also designate a test result as a baseline so you can compare different test results to gain insight as to how your EJBs perform under varying situations.

Bean-test by Empirix is a robust testing tool that's flexible enough to meet the testing needs of today's rapid development market. You can download an evaluation copy of Empirix's Bean-test product at www.empirix.com.

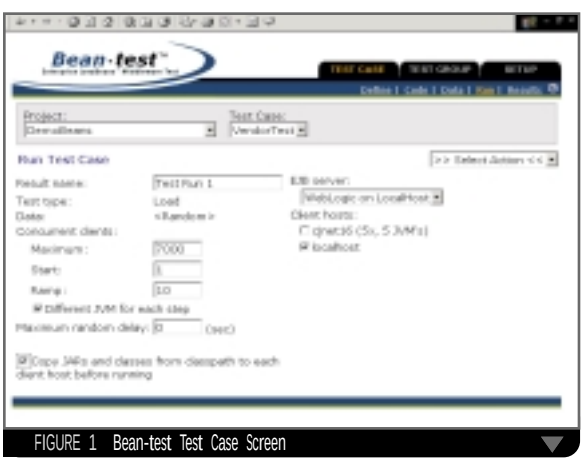


FIGURE 1 Bean-test Test Case Screen