ADVERTISER INDEX

ADVERTISER	URL	PH	PG
4TH PASS	WWW.4THPASS.COM	877-484-7277	89
ADRENALINE	WWW.ADRENALINE.COM		12
ADVANCED DATA, INC.	WWW.JVSEARCH.COM		46
AFFINITY	WWW.AFFINITY.COM/RESELLER	800-646-0662	41
AIR2WEB	WWW.AIR2WEB.COM	404-815-7707	25
AVANTSOFT	WWW.AVANTSOFT.COM/SEMINAR	408-530-8705	40
BEA	DEVELOPER.BEA.COM	408-570-8000	17
CAPE CLEAR	WWW.CAPECLEAR.COM		33
CAREER OPPORTUNITY ADVERTISERS		800-846-7591	114-137
CEREBELLUM SOFTWARE	WWW.CEREBELLUMSOFT.COM	888-862-9898	63
CODEMARKET	WWW.CODEMARKET.COM	646-486-7346	99
COMPUTERWORK.COM	WWW.COMPUTERWORK.COM	800-691-8413	52
COMPUWARE	WWW.COMPUWARE.COM/NUMEGA	800-4-NUMEGA	35
CORDA TECHNOLOGIES	WWW.CORDA.COM	802-802-0800	69
CTIA WIRELESS IT 2000	WWW.WIRELESSIT.COM	202-736-3241	105
CYSCAPE	WWW.CYSCAPE.COM/BHFREE	800-932-6869	52
ELIXIR TECHNOLOGY	WWW.ELIXIRTECH.COM/DOWNLOAD	65 532-4300	107
ENSEMBLE SYSTEMS	WWW.ENSEMBLE-SYSTEMS.COM	877-290-2662	38
ESMERTEC	WWW.ESMERTEC.COM		27
FATBRAIN.COM	WWWL.FATBRAIN.COM/FBT/OFFERS/JAVADEV		21
FIORANO	WWW.FIORANO.COM	800-663-3621	81
FLASHLINE.COM, INC.	WWW.FLASHLINE.COM	800-259-1961	67
FORT POINT PARTNERS, INC.	WWW.FORTPOINT.COM	415-395-4400	19
GEMSTONE	WWW.GEMSTONE.COM	503-533-3000	57
GENERIC LOGIC, INC.	WWW.GENLOGIC.COM	413-253-7491	89
HIT SOFTWARE	WWW.HIT.COM	408-345-4001	73
IAM CONSULTING	WWW.IAMX.COM	212-580-2700	85
INETSOFT TECHNOLOGY CORP	WWW.INETSOFTCORP.COM	908-755-0200	75
INSTANTIATIONS, INC.	WWW.INSTANTIATIONS.COM	800-808-3737	87
INT	WWW.INT.COM	713-975-7434	22
INTERNET WORLD FALL 2000	WWW.PENTONEVENTS.COM	800-500-1959	112-113
INTUITIVE SYSTEMS, INC	WWW.OPTIMIZEIT.COM	408-245-8540	101
JAVA DEVELOPER'S JOURNAL	WWW.JAVADEVELOPERSJOURNAL	201-802-3020	58,111
KL GROUP INC	WWW.KLGROUP.COM/GREAT	888-361-3264	15
KL GROUP INC	WWW.KLGROUP.COM/FASTER	888-361-3264	53
KL GROUP INC	WWW.KLGROUP.COM/MASTER	888-361-3264	95
KL GROUP INC	WWW.KLGROUP.COM/DEPLOY	800-663-4723	140
NO MAGIC	WWW.MAGICDRAW.COM	303-914-8074	7
NORTHWOODS SOFTWARE CORPORATION	WWW.NWOODS.COM	800-226-4662	102
OOP.COM	WWW.OOP.COM	877-667-6070	37
PACIFIC NORTHWEST NATIONAL LABORATORY	WWW.PNNL-SOFTWARE.COM	509-946-1110	47
PARASOFT	WWW.PARASOFT.COM/JDJ_OCT.HTM		61
POINTBASE	WWW.POINTBASE.COM/JDJ	877-238-8798	43
PRAMATI	WWW.PRAMATI.COM	408-965-5513	83
PROGRAMMER'S PARADISE	PROGRAMMERSPARADISE.COM/VISUALSOFT	800-516-0101	29
PROGRESS SOFTWARE	WWW.SONICMQ.COM/AD11.HTM	800-989-3773	2
PROTOVIEW	WWW.PROTOVIEW.COM	800-231-8588	3,97
QUADBASE	WWW.QUADBASE.COM	408-982-0835	45
QUICKSTREAM SOFTWARE	WWW.QUICKSTREAM.COM	888-769-9898	109
SANDSTONE TECHNOLOGY	WWW.SAND-STONE.COM	858-454-9404	40
SEGUE SOFTWARE	WWW.SEGUE.COM	800-287-1329	11,13
SIC CORPORATION	WWW.SIC21.COM	822.227.398801	91
SIERRA SYSTEMS	WWW.SIERRASYSTEMS.COM		59
SLANGSOFT	WWW.SLANGSOFT.COM	972-2-648-2424	139
SOFTWARE AG	WWW.SOFTWAREAG.COM/TAMINO	925-472-4900	79
SOFTWIRED	WWW.SOFTWIRED-INC.COM	41-14452370	65
THESHORTLIST	WWW.THESHORTLIST.COM	877-482-8827	39,47
THINAIRAPPS	WWW.THINAIRAPPS.COM/JAVA	888-609-THIN	49
TIDESTONE TECHNOLOGIES	WWW.TIDESTONE.COM	913-851-2200	77
TOGETHERSOFT CORPORATION	WWW.TOGETHERSOFT.COM	919-833-5550	6
UNIFY CORPORATION	WWW.UNIFYEWAVE.COM	800-GO UNIFY	93
VERGE TECHNOLOGIES GROUP, INC.	WWW.EJIP.NET	303-998-1027	103
VERIO	WWW.DEDICATEDSERVER.COM	877-624-7897	23
VIRTUALSCAPE	WWW.VIRTUALSCAPE.COM	877-VSCAPE4	31
VISICOMP	WWW.VISICOMP.COM/JDJ10	831-335-1820	51
VISUALIZE INC.	WWW.VISUALIZEINC.COM/JDJ	602-861-0999	58
WEBGAIN	WWW.WEBGAIN.COM	408-517-3815	4,55

Please visit the Web sites of our advertising partners who make it possible for us to bring you this Digital Edition (PDF) of ***JDJ***

USING SPACE BASED PROGRAMMING

JAVA™ DEVELOPER'S JOURNAL

The World's Leading Java Resource

October 2000 Volume: 5 Issue: 10

JAVADEVELOPERSJOURNAL.COM

XML DevCon FALL 2000 November 12–15, 2000

Announcing... Wireless DevCon December 3–5, 2000

SYS-CON MEDIA

ENTERPRISE
A Java engine for e-business development and deployment platforms
JAVA

EDITORIAL OFFICES
SYS-CON PUBLICATIONS, INC.
135 CHESTNUT RIDGE ROAD, MONTVALE, NJ 07645
TELEPHONE: 201 802-3000 FAX: 201 782-9600
SUBSCRIBE@SYS-CON.COM

JAVA DEVELOPER'S JOURNAL (ISSN#1087-6944) is published monthly (12 times a year) for $49.00 by SYS-CON Publications, Inc., 135 Chestnut Ridge Road, Montvale, NJ 07645. Periodicals Postage rates are paid at Montvale, NJ 07645 and additional mailing offices. POSTMASTER: Send address changes to: JAVA DEVELOPER'S JOURNAL, SYS-CON Publications, Inc., 135 Chestnut Ridge Road, Montvale, NJ 07645.

SEAN RHODY, EDITOR-IN-CHIEF

Who Will Be King of J2001?

Lately I've received a number of e-mails and had conversations regarding J2EE compliance and what it means to the industry. Each conversation or message has a slightly different slant depending on whether the person on the other end is a vendor or a reader or a colleague. What almost everyone seems to agree on is that the J2EE standard has done more to create a basic parity among vendors than any other event in the short but colorful history of Java.

Parity excites end users (in this case developers or IT departments) and depresses vendors. IT departments love the portability of J2EE because it allows them to move from one vendor to another as they see fit. Vendors dislike parity because it reduces their ability to charge premiums – they're selling a commodity.

To avoid the commodity problem, vendors naturally attempt to differentiate themselves from one another by providing additional features and new functionality. With J2EE this is a complex task and one fraught with danger. Yet there are several areas where vendors can add value without necessarily forsaking compliance with the specification. It's this differentiation that in fact will drive IT users to move from one implementation to another as they look for specific features. In a sense, the very cause of commoditization is also the driver behind specialization.

I see a number of areas that vendors will be likely to address as they begin the process of differentiating themselves. For lack of a better name I've termed this *J2001* – next year's J2EE. This has nothing to do with the EJB 2.0 specification or anything else official: it's my own word. What follows is my view of what's likely to be used to sell products in 2001.

Clustering capabilities will be highlighted as vendors continue to push on the topics of reliability, availability and scalability. Some vendors are pursuing multiple VMs running on a single machine (or processor) as a way of achieving better clustering. Others are clustering at the machine level. I expect major players in the industry to have a clustered solution and for them to beat on each other about which one is best. I wouldn't be surprised to see integration of hardware clustering solutions as the logical next step in the quest for speed and scalability.

Personalization services will become a hot topic. Let's face it – the Web abounds with personalized sites. The integration of personalization into product offerings will be another area where vendors begin to establish dominance. The degree of integration and flexibility of personalization will be vital in determining the best solution.

Closely aligned with personalization but truly another topic is the idea of business rules engines. Such engines already exist as stand-alone products, and several vendors have either written their own or integrated them into their products as add-ons. I expect the critical factor surrounding this area to be the ability to define rules that affect and are aware of EJBs. Another key to success will be the ability to provide a rules interface that business users, not IT technicians, can understand.

Integration into systems management systems such as Tivoli, Unicenter or OpenView will also be a key consideration as large IT organizations begin to demand greater control of J2EE services from network management consoles.

Transaction/commerce engines will also become a key factor. The whole point of EJB is to do transactions. But, from the EJB perspective, this is a somewhat artificial, programmatic transaction. From the B2B or B2C standpoint a transaction involves the exchange of cash or other valuables. Commerce engines already exist, and many of them run on EJB servers. Look for vendors who have commerce offerings to extend their product, and for vendors who don't, to seek partners to provide this necessary functionality. I see 2001 as the year that commercially available negotiation (not auction) engines finally hit their stride.

I could go on, but this list is probably big enough. Every project I see these days asks questions along these lines. It's the answers to these questions that will decide who will be king of J2001.

sean@sys-con.com

AUTHOR BIO
Sean Rhody is editor-in-chief of Java Developer's Journal. He is also a respected industry expert and a consultant with a leading Internet service company.

WRITTEN BY KEITH SCIULLI

While Giants Lie Sleeping

A technology tug-of-war for smart devices

This year's battle in the technology field resembles an election year – people are choosing sides and leveraging their power. The big decision for developers will be selecting a protocol to build into their smart devices. Each camp has its pundits and its naysayers. Sun and Microsoft are deeply entrenched in the market. Both have too much at risk and are refusing to concede ground. This stalemate has placed developers right in the middle of the quandary. What protocol will emerge from this stubborn battle as the de facto standard? Who will produce the most widely accepted platform? Who has the resources to bring it to the mass market first and make it stick? What's tearing at the heart of this next big tech explosion will likely be the catalyst for the evolution. For now the market looks like a technology tug-of-war with the developers as the rope.

Consortiums are being established to outline the codes and specifications that will be used. Many of the larger companies are sharing seats on more than one committee simply to stay informed, or to sway the momentum of the market. Their persuasion is obviously self-serving; nonetheless their power is indisputable. Groups like OSGi (Open Service Gateway Initiative) resemble "arms dealers," arming their members for a good round of competition on a level playing field.

While maintaining a neutral tone, yet satisfying the individual interests among the members, Universal Plug and Play groups are establishing a strong user base, and secret society alliances are popping up everywhere.

Stuck in the middle of the mess are the developers who are feeling the heat from business developers. These developers see smart devices and the new revenue streams they represent as the eminent wave of new commerce in the new economy. For now, smart devices have found their place in industrial automation. Next, we'll see it in luxury automobiles, at least in those cars that can absorb the cost and size of the stealthlike boards in their componentry to create an Internet-on-wheels scenario, and eventually morphing into complete vehicle self-navigation without human intervention. However, the real market-in-waiting is the residential gateway market of the post PC–era, when devices will be controlled remotely, proactively and automatically through a central processing unit located just about anywhere. This is where the market expects to become a multibillion-dollar cash cow and the future of consumer electronic appliances. "They have said that it is expected to be the biggest explosion with the longest fuse," says George Reel, sales director for ProSyst. "We've been evangelizing and nurturing this market while we feed the high-end commercial and industrial uses first."

ProSyst is 100% Java and platform independent. Not entirely agnostic, ProSyst, a two-year-old company from Germany, maneuvered into embedded to capitalize on the expected turn in the market. ProSyst feels it has discovered the answer to the dilemma by being the first to introduce a working Jini into its embedded server, gradually customizing its product to satisfy the market. Though it's designed to be OSGi compliant, it also integrates Universal Plug and Play and is planning on successfully bridging the two protocols like an interpreter for the United Nations. ProSyst has remained ahead of the curve and successfully managed to predict and incorporate the changes into its embedded server. ProSyst's solution is "mBedded server," a small 60KB footprint that can sit on practically any JVM and enable it to sing. They've introduced WAP, HAVI, HTML and WML, and with its latest version they've teamed up to offer Directory Services. This is in response to the demand for the encrypted security of proprietary information over the Net and, perhaps more important, network management of these billion devices as they proliferate into the market.

—continued on page 41

k.sciulli@prosyst.com

AUTHOR BIO

Keith Sciulli is the director of marketing and strategic operations at ProSyst. He holds two degrees in technical communications from Carnegie-Mellon University.

USING SPACE BASED PROGRAMMING for Loosely Coupled Distributed Systems

WRITTEN BY TARAK MODI

One of the problems of highly distributed systems is figuring out how systems discover each other. After all, the whole point of having systems distributed is to allow flexible and perhaps even dynamic configurations to maximize system performance and availability. How do these distributed components of one system or multiple systems discover each other? And once they're discovered how do we allow enough flexibility, such as rediscovery, to allow their fail-safe operation?

Space-based programming may provide us with a good answer to these questions and more. In this article I'll describe what a space is and how it can be used to mitigate some of the issues mentioned above. And I've included a technique to convert an ordinary message queue into a space.

What Is a Space?

Conventional distributed tools rely on passing messages between processes (asynchronous communication) or invoking methods on remote objects (synchronous communication). A space is an extension of the asynchronous communication model in which two processes are not passing messages to one another. In fact, the processes are totally unaware of each other.

In Figure 1, Process 1 places a message into the space. Process 2, which has been waiting for this type of message, takes the message out of the space and processes it. Based on the results, it places another message into the space. Process 3, which has been waiting for this type of message, takes the message out of the space.

Following are highlights of the preceding discussion:

1. The space may contain different types of messages. In fact, I used the term *message* for clarity. These messages are actually just "things" (the message may be an object, an XML document or anything else that the space allows to be put in it). In Figure 1 the different shapes in the space illustrate the different types of messages.
2. The three processes involved have no knowledge of one another. All they know is that they put a message in a space and get a message out of the space.
3. As in the message-passing scenario, we aren't limited to two processes communicating asynchronously, but rather any number of processes communicating via a common space. This allows the creation of loosely coupled systems that can be highly distributed and extremely flexible, and can provide high availability and dynamic load balancing.

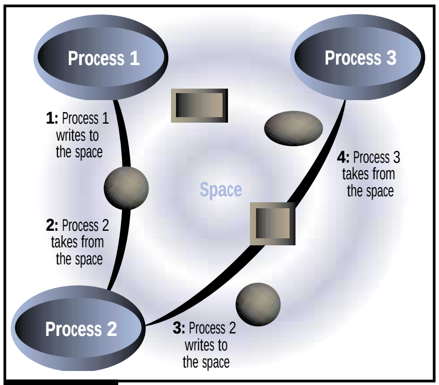

Process 1
Process 3
1: Process 1 writes to the space
Space
4: Process 3 takes from the space
2: Process 2 takes from the space
Process 2
3: Process 2 writes to the space

FIGURE 1 Multiple processes interacting with a typical space

Any number of processes can communicate via a common space

Let's look at a more specific example this time. A common encryption method is the use of "one-way" functions, which take an input and, like any other function, generate an output. The distinguishing feature of such functions is that it's extremely difficult to compute the input that was given to the function to get the output (i.e., to compute the inverse of the function); hence, the term *one-way* function. Instead of trying to figure out the inverse of the function to get the input required for the given output, an easier way may be to take all possible inputs and compute the output for each one. When we get an output that matches the one we have, we've found the right "input." But this can be extremely time consuming given the vast number of possible inputs.

Assume that passwords can't be more than four characters in length and only alphanumeric ASCII characters are used. This gives us 14,776,336 possible passwords (62^4). Using the brute force technique to break the password, assume that the main program breaks the input set into 16 pieces and puts each piece – along with the encrypted password – in the space. The password-breaking programs watch the space for such pieces and each available program immediately grabs a piece and starts working. The programs continue until no more such pieces are available or until the password has been broken. If the password is broken, the breaking program puts the solution in the space, which is picked up by the main program.

The main program then proceeds to pick up the remaining pieces, since it has already found the solution it needs. The program never knew how many password-breaking programs were available, nor did it know where they were located. The password-breaking programs had no knowledge about one another or about the main program. If there were 16 password-breaking programs available, and each one was on a separate machine, we would've had 16 machines working on breaking the password simultaneously!

No change to any configuration of the system is required to add new password-breaking programs. This is why spaces are so good for fault tolerance, load balancing and scalability.

As you can see, spaces provide an extremely powerful concept/mechanism to decouple cooperating or dependent systems. The concept of a space isn't new, however. Tuple spaces were first described in 1982 in the context of a programming language called *Linda*. Linda consisted of tuples, which were collections of data grouped together, and the tuple space, which was the shared blackboard from which applications could place and retrieve tuples. The concept never gained much popularity outside of academia, however. Today spaces may be an elegant solution to many of the traditional distributed computing dilemmas. In recognition of this fact, JavaSoft has created its own implementation of the space concept, JavaSpaces, and IBM has created TSpaces, which is much more functional and complex than JavaSpaces. (We won't discuss IBM's TSpaces in this article.)

We're now in a position to describe some of the key characteristics of a space:

- ***Spaces provide shared access:*** A space provides a network-accessible "shared memory" that can be accessed by many shared remote/local processes concurrently. The space handles all issues regarding concurrent access, allowing the processes to focus on the task at hand. At the very least, spaces provide processes with the ability to place and retrieve "things." Some spaces also provide the ability to read/peek at things (i.e., to get the thing without actually removing it from the space, thus allowing other processes to access it as well).

- ***Spaces are persistent:*** A space provides reliable storage for processes to place "things." These "things" may outlive the processes that created them. It also allows the dependent/cooperating processes to work together even when they have nonoverlapping life cycles, and boosts the fault tolerance and high-availability capability of distributed systems.
- ***Spaces are associative.*** Associative lookup allows processes to "find" the "things" they're interested in. As many processes may be using/sharing the same space, many different "things" may be in the space. It's important for processes to be able to get the "things" they require without having to filter out the "noise" themselves. This is possible because spaces allow processes to define filters/templates that instruct/direct the space to "find" the right "things" for that process.

These are just a few key characteristics of spaces. Many commercial space implementations, such as the ones from JavaSoft and IBM, have additional characteristics such as the ability to perform "transacted" operations on the space.

JavaSoft's Implementation: JavaSpaces

JavaSpaces technology, a new realization of the tuple spaces concept described above, is an implementation that's available free from JavaSoft. JavaSpaces is built on top of another complex technology, Jini, a Java-based technology that allows any device to become network aware. Jini provides a complex yet elegant programming model that realizes the Jini team's vision of "network anything, anytime, anywhere."

The goal of JavaSpaces is to provide what might be thought of as a file system for objects. Like other JavaSoft APIs, JavaSpaces provides a simple yet powerful set of features to developers. As I see it, however, JavaSpaces has four drawbacks:

1. The implementation of JavaSpaces is complex to install.
2. The fact that it builds on top of Jini makes it a little too heavy, especially if there are no plans to use Jini elsewhere in the project.
3. JavaSpaces relies on Java RMI, the suitability of which for highly scalable commercial applications is a topic of debate among many software gurus.
4. JavaSpaces works only with serializable Java objects.

Creating Your Own Space Implementation

Even though commercial implementations of spaces are available in the market, there are several reasons to create your own. If you work in a start-up company, budget constraints may be a big reason. Also, the functionality offered by a commercial implementation may be too much for the job at hand. Not only may this result in a larger learning curve, it may even slow down your application due to the sheer size of the memory footprint. Finally, it's always fun to create your own implementation.

At Online Insight we decided to create our own implementation. The primary reasons for our decision were our limited set of requirements and the extremely lightweight implementation we required to achieve our scalability and performance goals.

Our requirements can be summarized as follows:

1. The space must support shared access.
2. The space must be persistent.
3. The space must provide the ability to specify a filtering template.
4. The space must allow one "thing" to be accessed by only one process/application at a time (i.e., we don't support the "read" operation).
5. The space must perform and scale well under load.
6. The space must be accessible to other CORBA objects.
7. The space must not impose a limitation on what you can put in it (unlike JavaSpaces, for example).
8. The space must not impose size limitations on what you can put in it (the underlying hardware, however, may impose a limitation).

Note that the first three requirements are, in addition, key characteristics of a space.

Java Message Service

At the time we were evaluating message queue–type software – specifically, Java Message Service (JMS) implementations – we realized that we could build our space facility on top of one of these queues.

JMS is an API for accessing enterprise-messaging systems from Java programs. It defines a common set of enterprise-messaging concepts and facilities, and attempts to minimize the set of concepts a Java language programmer must learn to use, including enterprise-messaging products such as IBM MQSeries. JMS also strives to maximize the portability of messaging applications. It doesn't, however, address load balancing/fault tolerance, error notification, administration of the message queue or security issues. These are all message queue vendor–specific and outside the domain of the JMS.

By using message queues that expose a JMS interface, we allow ourselves the flexibility to switch vendors of message queues if we discover that the selected one doesn't meet our scalability requirements. This separation of implementation from interface is an important design pattern (see the Bridge design pattern in *Design Patterns* by Gamma et al., published by Addison-Wesley). Since each JMS implementation has its own unique way of getting the initial connection factory, we defined a Java interface with one method, "getConnectionFactory", which returns the initial connection factory.

Each space is configured through a properties file. One property in this file is the fully qualified name of the class that implements this interface. There is one such class for each JMS implementation supported by the space. For example, we created one class for Sun's Java Message Queue and one for Progress Software's SonicMQ. By doing this, changing the underlying message queue used by the space is simply a matter of changing the name of the Java class in the properties file for the space. Therefore, if one vendor's message queue doesn't live up to our expectations, we can quickly switch to another.

The space implementation itself is a CORBA object that has the following interface:

```
interface Space
{
void write(in ByteStream blob) raises (SpaceException);
       ByteStream take() raises (SpaceException);
       void write_filter(in ByteStream blob, in FilterSeq f)
raises (SpaceException);
       ByteStream take_filter(in FilterSeq f) raises (SpaceExcep-
tion);
       ByteStream take_filter_as_string(in string f)
raises (SpaceException);

       void shutdown();
};
```

The type ByteStream simply evaluates to a stream of bytes. Hence, anything that can be represented as a stream of bytes, such as a CORBA object IOR, a serialized Java object or an XML document, can be stored in the space and retrieved.

Each space instance has three properties: a name, a property that indicates if this instance of the space is persistent and a property that indicates if this instance of the space allows filters. The reason there are properties to turn the persistence and filtering off is purely for performance.

Not all spaces in our application domain are required to be persistent, in which case persistence is a performance bottleneck because it involves writing out to a database or similar storage mechanism. Similarly, if filtering isn't required, it's a performance bottleneck. As mentioned above, each space is configured through a properties file,which has the property indicating the space name, the persistence status (on/off) and the filtering status (on/off) of the space.

An example of the properties file used in configuring the space is shown below:

```
SpaceName=MySpace
AllowFilter=true
Persistent=true

# The factory to use to get the initial Connection Factory
SpaceFactory=SonicMQSpaceFactoryImpl
```

The "SpaceName" property is the name of the space, "AllowFilter" is a boolean property where true means the space turns filter support on and "Persistent" is a boolean property where true means the space turns persistence on. "SpaceFactory" is set to the fully qualified name of the class that allows us to get the initial connection factory from the message queue. In the foregoing example, this property is set to a class that works with SonicMQ implementation.

During start-up each space installs itself in the CORBA Name Service using its name property as the binding name and in the CORBA Trader Service with the name, persistence and filter properties. Thus interested applications/processes can find a space by using a well-known name from the CORBA Name Service or the space properties from the CORBA Trader Service. For example, an application that wants filtering but isn't interested in persistence can indicate these requirements to the CORBA Trader Service, which will then provide the application with a list of CORBA space references that match these requirements. The application may then choose one from that list based on some further screening.

Our implementation of the space gains all its persistence and filtering capabilities from the underlying messaging queue provider. Our space is the only client of the message queue. In our implementation the only purpose the message queue serves is as a high-quality storage/retrieval mechanism that also provides filtering capabilities. We aren't relying on the queuing facilities per se.

Each method of the CORBA interface is detailed below:

- **write:** This method is called by an application when it wants to put a stream of bytes into the space and doesn't want to attach filtering properties to the stream.
- **write_filter:** This method is used by an application when it wants to put a stream of bytes into the space and wants to attach filtering properties to the stream. The type FilterSeq evaluates to an array of filters that are attached to that bytestream. A filter is a name-value pair. Hence, a FilterSeq is an array of name value pairs.
- **take:** This method is called by an application when it wants to retrieve a stream of bytes from the space. No filtering is performed since none is specified.
- **take_filter:** This method is called by an application when it wants to retrieve a stream of bytes from the space. However, in this case a FilterSeq is provided. For a match to occur, the bytestream must have a subset of the filters provided in the method call, and the value of each filter attached to the bytestream must match the value for the corresponding filter in the method call.
- **take_filter_as_string:** This method is called by an application when it wants to retrieve a stream of bytes from the space. In this case a string that specifies the exact filter is provided. For a match to occur, the filter properties attached to the bytestream must satisfy the filter string provided in the method call. This method is used when the filtering conditions can't be specified as a FilterSeq.
- **shutdown:** This method is called to shut down the space. The shutdown is clean, which means the registration with the Name Service and the Trader Service is removed.

The space implements all methods in the interface as synchronized. Furthermore, the take implementations are nonblocking, that is, if there's nothing to take, the method returns with nothing.

Conclusion

Distributed applications can be notoriously difficult to design, build and debug. The distributed environment introduces many complexities that aren't present when writing stand-alone applications. Some of these challenges are network latency, synchronization and concurrency, and partial failure.

Space-based programming, although not a silver bullet, is an excellent concept that can lead to an elegant solution to these problems. It takes us one step closer to achieving our goals in a distributed system, namely those of scalability, high availability, loose coupling and performance. It also helps us face the challenges mentioned above. Best of all, you don't have to buy an expensive implementation to get started with this excellent concept. It's fairly easy to create a homegrown implementation that satisfies your requirements...and it's fun, too!

Resources

1. *Linda Group:* www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html
2. *JavaSpaces homepage*: www.javasoft.com/products/javaspaces/
3. *IBM, TSpaces*: www.almaden.ibm.com/cs/TSpaces/
4. Carriero, N.J. (1987). "Implementation of Tuple Space Machines," PhD thesis, Yale University, Department of Computer Science.
5. Segall, E.J. (1993). "Tuple Space Operations: Multiple-Key Search, Online Matching and Wait-Free Synchronization," PhD thesis, Rutgers University, Department of Computer Science.
6. Gul, A., et al. "ActorSpaces: An Open Distributed Programming Paradigm," University of Illinois at Urbana-Champaign, ULIUENG-92-1846.

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 technological 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.

tarak.modi@onlineinsight.com

CORBA for Real-Time Systems

Plus...news from OMG

Real-Time (RT) systems are, in the temporal sense, predictable. They're not necessarily fast, though many are; they don't necessarily deal with high throughput, though many do. Their defining characteristic is their temporal predictability. They run glamorous, high-risk, high-speed applications such as fly-by-wire airplane and missile controls, military data collection and display, and manufacturing process control, but they also run more mundane applications such as e-commerce transaction systems and materials-handling facilities.

WRITTEN BY
JON SIEGEL

For example, RT systems ensure that:

- When the high-fluid-level sensor in a tank is triggered, the pump will receive a shutdown message within 10 seconds.
- When engine RPM exceeds 6,000, fuel flow will be lowered within 20 milliseconds.
- 95% of credit card sales approval requests will receive a response within one minute.

These examples also demonstrate some of the differences among RT systems. They can work in time intervals that are long compared to a machine cycle. They can be fast (some RT systems work in microseconds). Or their reliability can be measured statistically, rather than absolutely.

Real-Time CORBA Architecture

RT CORBA doesn't define a magical environment that makes a non-RT CORBA application run with RT predictability. Instead, predictability is achieved through control of resources and load. RT systems must be built on controllable, predictable hardware running controllable, predictable RT operating systems. Network delay, although frequently not controllable, must at least be taken into account. Local application behavior is carefully controlled using RT OS capabilities, while distributed behavior is controlled using RT CORBA.

Real-Time CORBA is a set of optional extensions to the standard CORBA ORB. It supports end-to-end predictability for distributed applications by:

- Respecting thread priorities between client and server during CORBA invocations
- Bounding the duration of thread priority inversions
- Bounding the latencies of operation invocations

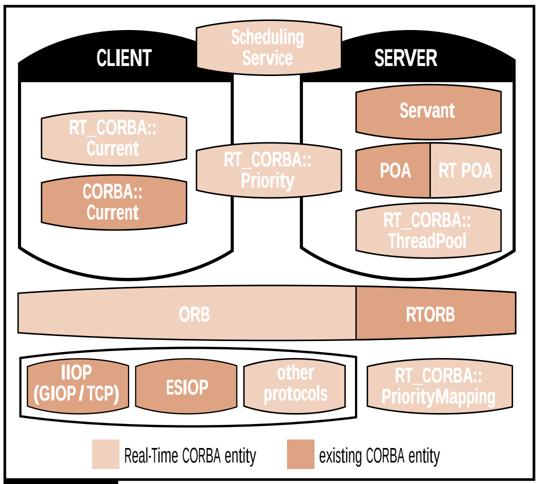


FIGURE 1 RT CORBA adds many interfaces.

Basic Real-Time Architecture

As Figure 1 shows, RT CORBA extends the client's current interface, the Portable Object Adapter (POA) interface, and adds interfaces for priority settings, ThreadPool management, priority mapping, communications features and a scheduling service.

Real-Time Priorities and Threading

When you write an RT application, you never have enough computing resource to get everything done right away so you have to prioritize. You divide your application's activities by priority and, when they run, you assign each activity a priority based on how urgent or important it is or how often it has to be run. High-priority tasks get first crack at resource – CPU, network, even database rows. You prioritize CPU access by allocating threads with assigned priority values, a function supported by your RT OS. Then you assign priorities to these threads based on a scheduling approach.

RT CORBA runs on many RT OSs. In each OS priorities are represented by integers, running from a minimum to a maximum value. Trouble is, each RT OS defines its own minimum and maximum value and, in a distributed system, you'll probably have a combination of RT OSs linked by the network. To enable interoperable and portable applications, RT CORBA defines a mapping from its own priority scheme termed *Real-Time CORBA Priority* to the scheme of the underlying RT OS.

What happens to the priority of a task that migrates over the wire from one CPU to another? There are two priority propagation models in RT CORBA:

1. ***Client Priority Propagation Model:*** The client's priority is propagated to the server.
2. ***Server-Set Priority Model:*** The server determines the priority of requests based on its own prioritization rules.

There are also two basic ways to manipulate threads in RT CORBA. One set of interfaces, derived from CORBA::Current, affects threads for its object (that's what Current refers to, of course). The other affects ThreadPools. Threads in pools can be preallocated and partitioned among your active POAs.

Priority inversion (PI) is the bane of the RT programmer. Here's an example: What if a low-priority task, as part of its function, locks a database row in preparation for a read-and-change and then stops executing because a high-priority task takes over the CPU. Suppose, further, that the high-priority task needs the same database row. It can't proceed because it can't get the mutex lock on the row, but the low-priority task can't get the CPU cycles it needs to finish and unlock because other medium-and high-priority tasks keep getting the CPU time.

RT CORBA defines a number of mechanisms that reduce both the occurrence and duration of PIs. Priority inheritance sets the priority of a thread based in part on the priorities of other threads (and especially threads doing work that might interfere). Multibanded communications also work to prevent PI, as we'll see in the next section.

Real-Time Communications

Communications Quality of Service (QoS) is crucial to end-to-end predictability, and RT CORBA gives you lots of ways to control it:

- You can set timeout values for your remote invocations, using the timeout definitions in OMG's recently adopted asynchronous messaging service specification. These let you define timeouts for the invocation, the response and the round-trip.
- You can use the concept of priority-banded connections, which are just what the name says. This helps keep lower-priority invocations from interfering with higher-priority work.
- You can ask for a nonmultiplexed connection from a client to a server. The low-priority connections serving the riffraff may be totally jammed, but this connection will always be clear.
- Both client and server can select and configure the protocols used to communicate.

Real-Time CORBA Management

A number of interfaces are used to set up and manage the features we've just listed. Most are policies derived from POA::policy, so they'll look familiar to CORBA programmers when they use RT CORBA for the first time. For real-time behavior on the server side you'll have to create a Real-Time POA, that is, one that uses the RT::POA interface and bears policies that make it execute in real-time mode.

The rest of this article focuses on news from OMG's RT CORBA workshop.

OMG's RT CORBA Workshop

OMG held its first RT and Embedded Systems Workshop this past July in Falls Church, Virginia, sponsored by Highlander Communications, Objective Interface Systems (OIS) and Vertel – three companies that market RT ORBs. (They're not the only ones; look for a longer list of RT ORBs when we discuss the ORB vendors' roundtable.) Indicating a high interest in distributed RT and embedded systems, the event drew over 140 attendees. If you missed it, plan for next year's, which is already in the works. In fact, there will be two events: one for RT and one for embedded systems. You'll find the workshop's URL at the end of this column.

The four-day workshop started with two days of tutorials: the first on CORBA, the second on the RT extensions. Each of the final two days consisted of several sessions of papers grouped by topic, followed by a roundtable. Every session allowed plenty of time for Q&A. I can't possibly summarize all the papers in this column, so I'll concentrate on a few and summarize the roundtable discussions too, since they're always spontaneous and fun. OMG posts the papers on its public Web site 30 days after each workshop (see www.omg.org/news/meetings/realtime/2000/presentations.htm).

Please don't view the papers and think you have the full benefit of the workshop; most of the benefit (and the fun!) comes from the interaction between the speakers and the audience, and the opportunity to network with many of the world's experts in distributed RT computing.

Scheduling and Resource Management

The first session covered scheduling and resource management. Here's a definition of scheduling: in a basic RT system you set priority levels for each of your tasks, prioritizing important or urgent tasks higher than less important ones. Since a task probably involves more than one computation step, setting priorities for every step of every task may be daunting – kind of like assembler programming before higher-level languages were developed. A scheduling service (such as the one in the RT CORBA specification) lets you group multiple steps into an activity and assign all of the steps a single priority. Whether you assign priorities to each computational step yourself or do it via a scheduling service, this is still static scheduling because each task's priority is assigned statically and can't change even if runtime conditions warrant. (If you recall our previous point about client-propagated and server-defined priority models, you'll recognize that a scheduling service will have to take the priority propagation model into account.)

Dynamic scheduling adapts computational priorities to changing loads, resource availability and urgency. By collecting and analyzing data at runtime and being clever about the way they assign resources to tasks, dynamically scheduled systems can optimize resource usage and maximize the number of activities that meet their deadlines. If the system is sufficiently clever, it might even be able to deliver predictable performance when our requirements of strict control over the system aren't quite met. Dynamic scheduling is not straightforward! There's a lot of current research in this area, and many ways to achieve (or, it seems, almost achieve) its goals.

The first paper, from UCSB, described a feedback system that adjusted task priorities based on runtime data. The second, from BBN, added QoS detection functionality to objects' stubs and skeletons, enabling the system to adapt dynamically to changing loads.

Case Studies/Experience Reports

Case studies and experience reports were spread over several sessions. These included descriptions of some of the exciting applications associated with RT computing. For example, Boeing Aircraft described the Weapon Systems Open Architecture (WSOA), a project that also includes Honeywell Technology Center, BBN and Washington University (which contributed the TAO ORB). Additional technology came from DARPA and the Air Force Research Laboratory. CORBA portions of WSOA include a hard RT application in an F-15E1 jet fighter and a soft (that is, statistical) RT application in a 737 AFL support plane. A flight demonstration is scheduled for next summer.

The Software-Defined Radio Forum (SDRF), a group of companies, is developing a standard for a family of two-way radios that can be reconfigured in software to work at different frequencies and provide different services. (Think of a handset that can be a cell phone, wireless digital pager, Web browser or two-way radio for military, aircraft or marine use, depending on the software you download to run it.) Presenters from Mitre, Software Technology, Inc., and Exigent International presented the group's work including contributions from RT CORBA and the CORBA Component Model (CCM). The forum also incorporated U.S. military work on the Joint Tactical Radio System (JTRS). These embedded systems run in Real-Time, and thus highlighted both of the workshop's main topics.

Possibly the most impressive combination of RT CORBA and advanced CORBA features in a running, commercial, embedded distributed RT system is the pick-and-place surface mount technology (SMT) assembly machine described by Bruce Trask of Contact Systems. Competing in a rough market, Contact Systems' goal is to produce machines that place electronic components onto printed circuit boards faster, more accurately and less expensively than their competitors'.

The machine, which fits on a cart, includes multiple-part feeders on both ends and an assembly in the center where the parts are mounted on a board. Two robot arms alternate between picking up parts from the feeders and placing them on the board. While the arm makes its way from the feeders to the board, an electronic camera takes a picture of the three to five components that it has picked up and adjusts the arm's movement to place the components in their exact places on the board. Although the arm takes 200 ms to move from the feeder to the board, it takes only 50 ms to move from the camera position to the board, giving the system little time to recognize the components and calculate the movement correction.

Their current machine has five Pentium processors, seven special-purpose processors and 30 microcontrollers, all networked and controlled by an RT CORBA system based on the TAO ORB. The system also uses the new OMG asynchronous messaging service and a (as yet nonstandard) pluggable protocols framework and RT event service.

Another paper, hopefully with ramifications for the future, covered microCORBA – CORBA for the kind of system-on-a-chip that can be produced so cheaply it can be used once and thrown away (or at least not recovered as might happen if you're probing ocean temperatures with networks of sensors that are released to the currents, or battlefields with networks of sensors that may be blown up, or body parts with networks of sensors that are swallowed). Near its smallest, this kind of system-on-a-chip may have to function with only 64 bytes (not a typo!) of RAM, but its size and correspondingly small cost enable a market with nearly unbelievable numbers of units: sales estimates range up to 11 billion units per year. OMG members plan to standardize a version of CORBA for these chips; look for an RFP before the end of 2000.

Panel Discussion Winds Up the Day

A panel discussion with six ORB implementers concluded the first day. Each started by introducing his company and product:

- **Ben Watson, Tri-Pacific Software**, said they have a compliant RT CORBA 1.0 scheduling service that guarantees all activities will meet their schedules. It works with embedded non-RT as well as RT ORBs. They use UML to do RT modeling. (OMG members are now working on a specification that will add RT concepts to UML.)
- **Doug Schmidt** spoke about his crew back at **UCI**, now working on an RT CCM implementation that will include dynamic scheduling, RT notification, fault tolerance and load balancing. Doug, now working at **DARPA**, on leave from UCI, produced the TAO RT ORB while he was a professor at the University of Wisconsin.
- **Malcolm Spence** represented Object Computing, Inc. (**OCI**), a company that sells and supports TAO to customers in telecommunications, aerospace, defense, finance and health care.
- **David Barnett**, **Highlander Communications**, focused on his company's adaptation of Inprise's VisiBroker ORB for RT embedded systems. This ORB supports the minimum CORBA specification (an official stripped-down version of CORBA suitable for embedded systems), pluggable protocols and a logging facility.
- **Bill Beckwith**, CTO of **OIS**, makers of ORBexpress, described his company's fast RT ORB (remember, we pointed out that *RT* and *fast* aren't the same thing) which provides priority management, priority inheritance and replaceable transports, and is suitable for hard RT and embedded applications. They foresee use in fiber optic switches, software-defined radios and transportation systems including boats, trains and airplanes.
- **Sam Aslam-Mir, Vertel,** said that his company's primary customer base is telecommunications, which requires both long mean-time between failure and long equipment lifetimes. (A lot of 30-year-old telecommunications equipment is still in service, although the pace of technology is bringing this down.) Convergence of telecommunications and computer networking, and the expanding Internet infrastructure, is changing the industry.

Questions from the Audience

Panelists then answered questions from the audience, including these:

- Asked about ORBs that combine RT and fault tolerance, Beckwith, Schmidt and Aslam-Mir said their companies were all close to providing a solution.
- What about security in RT CORBA applications? Bill Beckwith pointed out that this was a hard problem: RT requires a security implementation able to provide encryption, access decisions and other functions via bounded-time invocations. Currently, no research indicates that this is even possible, although none says it isn't, either. A number of panelists said there was research underway on this topic. Curiously, several of them wouldn't say where it was being done, or by whom.
- Communications is a hot topic in distributed RT, so it was no surprise when the topic of pluggable protocols came up. A pluggable protocols module allows the application programmer to insert his or her own network software interface, extending the ORB to communicate with a new protocol. Several panelists, including Schmidt, Aslam-Mir and Beckwith, said their products now have this or similar functionality. Some OMG members are planning an effort to standardize extensible transport frameworks.

To close out the workshop on the last day, **Brad Balfour** of **OIS** moderated an end-user panel. The first panel presenter, **Craig Rodrigues** of **BBN**, described the Future Scout Cavalry System, a military vehicle program, while **Ron Snyder** of **General Dynamics Land Systems** (GDLS) presented a series of slides emblazoned with pictures of the military vehicles and hardware that use the most exciting if not the most peace-loving RT hardware and software.

• • •

OMG and its member companies hold around five workshops each year. To see what's coming up, go to OMG's homepage www.omg.org, mouse over calendar, and select the top item, *meeting schedules*. This workshop will split into two next year: one on embedded systems in January and a separate one on RT now scheduled (albeit not yet predictably!) for May or June. You can download the RT CORBA 1.1 specification from www.omg.org/technology/documents/recent/ by clicking on CORBA/IIOP and then on Realtime CORBA in the table that comes up next. The Fault Tolerance specification also appears on this table. The RT specification will move to www.omg.org/technology/documents/formal/ with the CORBA 2.4 release; Fault Tolerance will follow with the CORBA 3.0 release now scheduled for early 2001. To follow the Extensible Transport Frameworks RFP, click on www.omg.org/techprocess/meetings/-schedule/.

siegel@omg.org

AUTHOR BIO
Jon Siegel, the Object Management Group's director of technology transfer, also writes articles and presents tutorials and seminars about CORBA. His new book, CORBA 3 Fundamentals and Programming, has just been published by John Wiley and Sons.

The Sharp Tongue of Microsoft

WRITTEN BY
ALAN WILLIAMSON

When all's said and done, August was a pretty uneventful month in the world of Java. No major acquisitions, CEO resignations or significant announcements. In the press world this period is often referred to as the "silly season": basically, nothing's happening, so they have to dredge up silly wee filler stories.

But instead of going into nonsense about some dog that can drive a car across four states, we'll take a quick look at Microsoft's new programming language, C#, some excellent customer service and what our Nordic cousins are up to with their mobiles.

It's Not Java, Honest!

This past summer, on June 26, Microsoft announced its new programming language to complement its .NET platform, called "C#" (pronounced *sharp*, as in the musical note). The new language promises all the latest features that a modern-day language should have without compromising security or functionality. I, like many of you, saw the initial press regarding its launch, then didn't give it much thought. After a couple of months, though, I thought the time had come to take a proper look at the language and see what the story is. I'm sure in the future there'll be many articles that will take you through wonderful, natty tables comparing C# with Java – with lots of ticks and crosses detailing various pros and cons. So as not to steal the thunder from that brigade of literature, let me just go through some of the broader points.

At the time of this writing (September), documentation on C# is thin on the ground. It took me over 15 minutes to find information on the main Microsoft Web site. Searching on "C#" – as in c sharp – yielded no results. Nor was it listed on any of their standard product pages. I was beginning to think that C# was simply Scotch mist or, as the Americans would probably say, *vaporware*. But eventually I stumbled on a link that took me to some literature I could refer to. Let me save you some time and point you in the right direction: http://msdn.microsoft.com/vstudio/nextgen/technology/csharpinto.asp.

No doubt about it...if you were to run your eyes quickly over a piece of C# code you'd think it was Java. The syntax is very familiar, and I don't believe this is by accident. I'm guessing that Mr. Gates wants us in the Java community to embrace this new kid on the block with much love and give it a chance to grow on us. Mmmmmm, we'll see.

Looking at it, Microsoft has put some cool features in there...and some that really shouldn't have been included.

```
using System;
public class ArrayClass {

  public static void
PrintArray(string[] w) {
   for (int i=0; i < w.Length; i++)
     Console.Write(w[i] + " ");
  }

  public static void Main() {
   // Declare and initialize an
array:
   string[] WeekDays = new string []
{"Mon","Tue","Wed","Thu","Fri"};

   // Pass the array as a parameter:
   PrintArray(WeekDays);
  }
}
```

As you can see from the sample C# code, which prints out the contents of a string array, there are only subtle differences with Java. The first one you notice is the word *using* as opposed to *import*. The declaration of the class is the exact same – no difference there. However, C# has no notion of "extends" or "implements." All classes and interfaces are extended in the same way:

```
public class userClassX : baseClassY,
interfaceClassZ{ ... }
```

Declaring methods takes on the same guise, so no major problems on that front. C# introduces a cool feature known as *delegation*. This is where you obtain a pointer to a particular object and can then call the necessary methods that this object has access to. This is what polymorphism is all about. In Java, however, you can forward- and backward-cast that reference to access other methods in the hierarchy. This isn't allowed in C#. You'd have to obtain a new reference to the object to accomplish it. You can argue whether this is good or bad, but it'll allow designers to maintain much stronger control over the use of their objects.

I'm confident that Java developers looking over the rest of the code will be able to understand it with no problem whatsoever. Barring a couple of wee case changes, it reads pretty much like any other Java program.

Another cool feature, which I find particularly useful, is variable initialization. As soon as you declare a variable, it's set to a default value. Sadly, Java has no standard for this one. It depends on what JVM you run whether or not a value will be set, so it's always safer to set it yourself and not have to rely on a third party. But with C# it's built in there from the start, so no worries.

Now before you throw down your Java and start running toward C#, it's not all a bed of roses. There are some quirks you need to be made aware of. So stay put for a moment.

One of Java's greatest strengths is its ability to run pretty much anywhere there's a virtual machine, and with the availability of JVMs reaching saturation point, this is no longer a major issue. C# doesn't share the same luxury. Unlike C++/C, it doesn't compile to a native program but instead needs to use Microsoft's .NET platform. So right away you can be pretty sure that getting your C# to run on a Solaris box isn't going to be smooth sailing. It can be argued that Java suffered from the same fate when it took its first steps into the wide world, so we'll have to wait and see which vendors implement .NET platforms for their boxes.

Another major stumbling block as far as I'm concerned is the retention of direct memory pointers. The days of accessing direct memory are back again. Hurrah! I think not. May I introduce a good friend, Dr. Watson; he'll be helping you with your C# development and deployment! It's going to be ugly. Consider yourself warned.

I highly recommend that you read about C# yourself. I personally like the look of it, but I'm not in the Windows world where I need to look at it seriously. However, I do know a lot of you are under major client pressure to stick with Mr. Gates, so I guess some manager somewhere will soon read about and start dictating that the next project has to be C# enabled. Before you denounce it, have a look and make up your own mind. At least that way, when you have to argue for Java, you'll be on far stronger footing.

Service, Please?

Sadly, we're not known for our customer service in this great nation of ours – the U.K., that is. We continually have to put up with poor service, rude suppliers and no sense of urgency. I could write books on what we have to endure, but I won't bore you with such moans. What I'd like to comment on is the quality of service from America. Damn, you're lucky!

We recently moved our offices into a nice old building that dates back to the 1800s. It's an ex-bank building with loads of character and we love it. But in this move we had to have a new Class-C allocated to us, so the pain of the form-filling at Network Solutions had to ensue. Well, thanks to the Labor Day holiday in the U.S., the DNS root servers weren't updated when they were meant to be. So we had to wait an extra day for the update. I learned this after a pleasant phone conversation with customer service.

The chap's name was David, and I was very impressed by the level of care devoted to sorting out our problems. He even asked for our phone number in case we were cut off. Now that's service! It was on speakerphone and Keith, after overhearing the whole thing, asked: "Why can't we have that level of service in this country?" Very impressive. The Brits could learn a lot from this customer care.

Big Brother Is Watching

I'm sure most of you have caught at least a glimpse of the great human social experiment of throwing 10 people into one house and observing how they get on. Stick a camera in any university dorm and you have your Big Brother. But it would appear that Big Brother is starting to monitor our movements for us, via our cellular phones.

Every time we move around with our mobile phone, some computer somewhere is tracking the location so it can route calls to and fro. Now, as was recently announced, a company in Sweden is selling this as a service. Bikeposition.com will guide you to your destination, detailing the direction you have to take and all necessary adjustments. Very cool technology. Scares me, this sort of thing. For every positive use a particular technology can be applied to, there is an equal, if not more negative, application. For example, slip a powered-on mobile into someone's pocket, and *boom*, you can monitor every movement. Too Orwellian for my liking.

On that note, I'm now reaching to turn off my Nokia as I prepare to head home. Remember to keep an eye on www.n-ary.com/industrywatch/ for updates.

See you next month.

Author Bio

Alan Williamson is CEO of n-ary (consulting) Ltd (www.n-ary.com), the first pure Java company in the U.K. A Java solutions company specializing in delivering real-world applications with real-world Java, Alan is the author of two Java Servlet books and contributed to the Servlet API.

alan@n-ary.com

The Evolution of Connecting

The right application leads to powerful new functionality

WRITTEN BY
ROBERT J. BRUNNER

So here you are, the eager Java developer, about to embrace JDBC (Java Database Connectivity), the next item on your Java technology checklist. If you followed my last article (*JDJ*, Vol. 5, issue 9), you've selected a database system and a JDBC driver to help you master this technology. Now you want to jump in and start writing code. Perhaps you've bought a JDBC book, read your JDBC driver documentation or collected various JDBC articles from magazines (such as this one). Unfortunately, many of these resources skim the introductory topics or, worse, offer seemingly conflicting code examples. This article discusses the details of connecting a Java application to a database using JDBC, including how the process has changed with the evolution of the Java programming language.

In theory, the basics of connecting a running Java application to a database are quite simple. Fundamentally, an application developer merely has to code to the JDBC API (see http://java.sun.com/j2se/1.3/docs/api/java/sql/package-summary.html for specific information on the J2SE 1.3 JDBC API). This API has seven classes. Only one, the DriverManager class, is commonly used by beginners. The bulk of the API is dominated by interfaces that must be implemented by the JDBC driver vendors. As previously discussed, this allows for a wide range in performance and flexibility due to specific implementation details. As is often the case, however, the devil is in the details, and there are a lot of details whenever databases are involved.

Before delving into them, however, a mental picture (see Figure 1 for an actual picture) of the process involved in connecting to a database can be useful in understanding why things behave the way they do. First, while seemingly obvious, the fundamental point to start with is that all Java code runs inside a specific Java Virtual Machine (JVM).

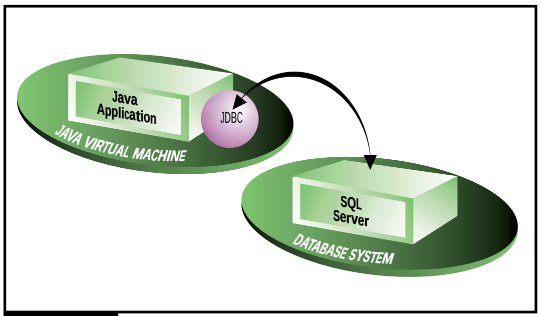


FIGURE 1 Outline of the JDBC architecture

Meanwhile, the database system of interest has its own interfaces and protocols, which are generally controlled by a server or daemon process. In order for these two server processes to communicate, a bridge must be established and that's accomplished via the JDBC driver. In this article we'll use a fictitious JDBC driver, aptly from the Acme Corporation, whose fully resolved class name is com.acme.jdbc.AcmeDriver. While database servers are in general explicitly designed to allow interprocess communication, the JVM, for obvious security reasons, is not. Therefore, the driver of another Java application also runs inside the same JVM as the Java database application and must provide this extra functionality.

Finding a Driver

Before it can be used, the JVM must "know" about the JDBC driver, which consists of two separate steps: loading and instantiation. The Java object that manages all of the JDBC drivers for a JVM is the DriverManager class. This single class is responsible for shouldering the burden of managing the pool of JDBC drivers that are available within a given JVM. The loading step can be done in two different fashions: static or dynamic loading. The first technique, which is rarely discussed in the popular literature, is static loading during the JVM initialization. This is accomplished by specifying the fully resolved driver (or drivers if multiple JDBC drivers are required) class to the JVM via the jdbc.drivers property:

```
java
-Djdbc.drivers=com.acme.jdbc.AcmeDriver
Test.java
```

One of the benefits of this approach is that the code doesn't need to be tied explicitly to a particular JDBC driver, allowing for changes in the actual driver used to be made by the system administrator (who ostensibly would be starting the JVM as a server process) without recompiling (and redistributing class files). Multiple-driver classes can be loaded in this fashion by separating them with colons, which could be important if the Java applications running inside the same JVM need to communicate to different databases.

The alternative approach is to dynamically load the JDBC driver within the actual Java application, which is done using the Java Reflection mechanism:

```
Class.forName(com.acme.jdbc.AcmeDriver) ;
```

This approach allows an application to dynamically load the requested driver (which can be specified at runtime). Of course, this requires that the driver class is actually in the CLASSPATH of the running JVM. This last point is one of the leading stumbling blocks for JDBC novices, as they will receive a ClassNotFoundException if the JVM can't find the requested class. If multiple drivers are loaded, any drivers listed in the jdbc.drivers property are registered first, followed by any dynamically loaded drivers.

Once a JDBC driver has been loaded into the JVM, it must be instantiated. The recommended method of designing drivers is to force them to be automatically instantiated during the loading process: that means that an application developer is able to load and instantiate the JDBC driver in one line of code. This is accomplished via a static initializer that creates a new instance of the driver class and registers the new driver object with the JDBC DriverManager object. As a result, the single line of code above is translated into the following steps:

1. JVM locates the requested driver class, which must be in the current CLASSPATH.
2. The JVM loads the driver class into memory.
3. The JVM executes the static initializer section of the driver class, which will create a new instance of the driver class and register this new instance with the DriverManager class.

This whole process should seem rather straightforward, so what cause is there for any concern? As a popular Java colloquialism states, "Write once, debug everywhere." The problem is that your Java code will need to run in a multitude of JVM implementations, all of which can have subtle variations. For example, some Microsoft virtual machines include a performance trick that loads Java classes differently than many other virtual machines. Unfortunately, this trick prevents the static initializer section from being executed until the JVM feels the class is actually needed. As a result, the driver is neither instantiated nor registered with the DriverManager object during the reflection process. To circumvent this "feature," the developer is required to perform these steps manually:

```
Class.forName(com.acme.jdbc.AcmeDriv-
er).newInstance() ;
```

This is not only a "Microsoft problem," as even the Sun JDK 1.1 reference implementation does not work properly due to a race condition (see the JDBC FAQ for more information) that prevented the static initializer section from being processed. The workaround for this bug is for the user to explicitly create and register the driver class:

```
java.sql.DriverManager.registerDriv-
er(new com.acme.jdbc.AcmeDriver()) ;
```

Fortunately, with the maturation of both Java and the JDBC driver implementations, these subtle variations are becoming less common (particularly if JDBC is confined to the server). This implies that the following code snippet is all that is required to explicitly instantiate and register a JDBC driver:

```
try {

Class.forName("com.acme.jdbc.Acme-
Driver") ;
}catch(java.lang.ClassNotFoundExcep-
tion e) {
  System.err.print("ClassNotFoundEx-
ception: ") ;
  System.err.println(e.getMessage())
;
}
```

Making the Connection

Once the driver class has been initialized, a Java application can request a connection from the DriverManager class. The DriverManager object selects the appropriate JDBC driver from its pool of registered drivers based on the specific JDBC URL passed to the DriverManager getConnection method. The DriverManager object tests each registered driver, using the provided URLs in the order in which they were registered. The first one that recognizes (i.e., establishes a connection) the JDBC URL is used to provide the actual database connection. Interestingly enough, this process is actually layered on top of the original antiquated method for establishing a database connection, which is occasionally still prominently mentioned in the documentation of certain JDBC drivers.

```
Driver driver = new com.acme.Jdbc-
Driver() ;
Connection con = driver.connect(url)
;
```

A JDBC URL follows the standard URL syntax (i.e., Web addresses), which includes the database name, the database server and optional additional parameters such as username and password. Instead of the more familiar protocols such as HTTP or FTP, the JDBC protocol is used. This is followed by a subprotocol which is designated by the individual driver vendors and registered with Sun to prevent confusing duplications. The last part of the URL is a subname that provides a mapping to the actual database of interest. The subname section of a URL can vary significantly from vendor to vendor; for example, the following are all valid JDBC URLs:

```
// An ODBC registered data source
String url = "jdbc:odbc:DataSource-
Name" ;

// An Oracle 8i database with thin
client connection
String url = "jdbc:oracle:thin:@serv-
er.acme.com:1521:jdbc" ;

// A Microsoft SQL Server database
String url =
"jdbc:JTurbo://server.acme.com:1433/j
dbc" ;

// A mSQL database
String url = "jdbc:msql://local-
host:1114/jdbc" ;
```

Once the URL is known, obtaining the connection object is straightforward:

```
try {
  Connection con = DriverManager.get-
Connection(url);
  // Utilize the connection to make a
query
  con.close();
}catch(SQLException e) {
  System.err.println("SQLException: "
+ e.getMessage());
}
```

Of course, the getConnection method has two additional signatures that allow the developer to pass additional information, such as a username and password, to the database.

The Evolution to Data Sources

While it might seem rather odd at first glance, the JDBC 2.0 specification introduced a new technique for establishing connections to a database, the DataSource object. In fact, the DriverManager class may be deprecated in future versions of Java. The main reason for this abrupt change is the emergence of server-side Java applications within the Enterprise framework. While a DriverManager object is limited in the functionality it can provide (i.e., it essentially serves as a JDBC driver pool), a DataSource object can represent an arbitrary data source, work with JNDI (Java Naming and Directory Interface), provide connection pooling (simplifies tracking license and database cursor limitations) and provide support for distributed transactions (which is important for Enterprise JavaBeans). Each of these new features provides powerful new functionality that exceeds the scope of this article.

While the future of Java Database Connectivity clearly lies with DataSource objects, the traditional DriverManager connection techniques will not disappear. Hopefully this article has helped to illuminate some of the finer points involved in connecting your Java application to a database. Once connected, the rest is SQL.

rjbrunner@pacbell.net

AUTHOR BIO

Robert Brunner is a member of the research staff at the California Institute of Technology, where he focuses on very large (multiterabyte) databases, particularly on KDD (Knowledge Discovery in Databases) and advanced indexing techniques. He has used Java and databases for more than three years, and has been the Java database instructor for the Java Programming Certificate at California State University Pomona for the past two years.

WRITING CUSTOM JSP TAG LIBRARIES

WRITTEN BY ADAM CHACE

Learn how to abstract complex business logic from UI design

Server-side Java continues to gain ground as the technology of choice for powering dynamic Web sites, but the goal of using Java to separate presentation from business logic has been a tough one to achieve.

JavaServer Pages 1.1 addresses that goal with the introduction of custom JSP tag libraries. Java developers can now embed complex logic into middle-tier objects while exposing only simple, easy-to-use tags to the presentation layer. This frees Java developers to do what they do best while enabling presentation developers to focus on building good UIs.

If you're already using JavaBeans and the bean tags introduced in JSP 1.0, you know that these tags still require tag users to have some basic understanding of Java. Custom JSP libraries can abstract the implementation away completely so page designers don't even know what language is used behind the scenes. Even better, unlike bean tags, custom tags can inspect and modify the content within the tag's body. For example, a custom JSP tag could be used to translate content from HTML to WML or to apply formatting to some text.

Getting Started

Custom tags are made up of two components: the Tag Handler class and an XML file called a Tag Library Descriptor. The Tag Handler class contains the actual Java code executed during a page request. The Tag Library Descriptor (.tld file) contains the attributes for all tags in a particular tag library. The JSP engine uses these attributes to decide how to handle the tags at runtime.

Tags can come in two basic types: those with a body and those without. For my first tag I'll build a basic tag that has no body and simply prints a line of text onto the page. The first step is to create the Tag Handler class. To do this I define a new class that will implement the javax.servlet.jsp.tagext.Tag interface. To make things easier, you can also extend a class called TagSupport, which defines default methods for the Tag interface.

Listing 1 shows that I've implemented only one method of Tag, doStartTag(). This method is called whenever the JSP engine encounters an occurrence of my custom tag. The first thing I did in this method was to get an instance of JspWriter, which is a specialized version of java.io.Writer that can be used to write content to the page. The JspWriter is retrieved from pageContext, which is an instance of the PageContext class. PageContext is an abstract class implemented by the JSP engine vendor and provides access to all the namespaces and attributes of a particular JSP page. For now, I just need to use it to get the JspWriter to write out to the page.

You'll note that doStartTag must return an int value. The JSP engine uses this return value to determine how to process the remainder of the page. For tags like SimpleTag that implement the Tag interface, only two return values are valid:

- **SKIP_BODY:** Instructs the JSP engine to ignore the body (if one exists) of this tag and not return it to the client
- **EVAL_BODY_INCLUDE:** Instructs the JSP engine to evaluate and include the content of this tag's body and return it to the client

Since SimpleTag won't have a body at all, its doStartTag method simply returns SKIP_BODY. If I wanted to allow users of the tag to place some content between its start and end tags (and have that content interpreted and sent to the client), I could return EVAL_BODY_INCLUDE instead.

That's it for the TagHandler class. Now I need to write a TLD for this tag that will provide the JSP engine with the information it needs to use it. For SimpleTag this file will include the standard heading for a tag library that looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "web-jsptaglib_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
```

This header is followed by the declaration of each individual tag. Here is the definition of our SimpleTag, with an explanation of what each attribute means:

```
<tag>
  <name>simple</name>
```

This is the name that the tag will be referred to as in a JSP.

```
  <tagclass>com.jdj.SimpleTag</tagclass>
```

This is the fully qualified class name of our SimpleTag class.

```
 <bodycontent>empty</bodycontent>
```

This is where we indicate what type of body the tag will have. A value of empty means the tag will always appear as <tag/> (without a body). A value of JSP indicates the body can be interpreted as JSP. A third (and less used) possibility here is tagdependent, which means the tag will interpret the body entirely itself.

```
</tag>
```

Don't forget to close the tag library tag with </taglib>. I'll save the file as "jdj.tld".

Using My Tag

I now have a TagHandler class and a tld file and I'm ready to use the tag. The last step in the process is to write a JSP that will use SimpleTag. I'll write the JSP just like any other, with two additions: at the top I need to provide a Taglib Directive that makes the tag library available within the page. For my library this looks like:

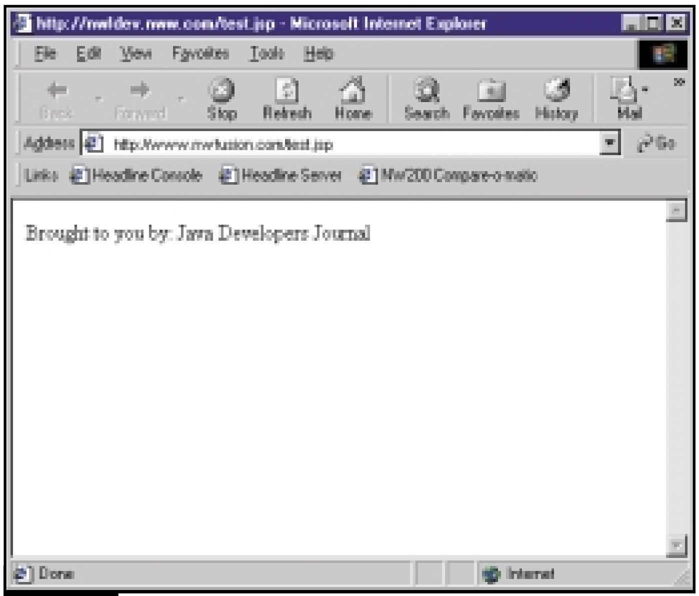
```
<%@ taglib uri="jdj.tld" prefix="jdj" %>
```

The prefix lets me refer to individual tags within a library with the syntax <prefix:tagname>. The rest of my JSP looks like this:

```
<html>
  <body>
    Brought to you by: <jdj:simple/>
  </body>
</html>
```

Now I need to deploy my new JSP file, jdj.tld, and SimpleTag.class to my Web server. For WebLogic, which I'm using, I've JARred all my classes and placed that JAR in the class path. I also need to copy the JSP file to /weblogic/myserver/public_html and the tld file to /weblogic/myserver/public_html/WEB_INF. Consult your JSP engine's documentation for details on deployment locations.

Once deployed, I simply navigate to the page with my browser (see Figure 1).


Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://www.mwfusion.com/test.jsp Go
Links Headline Console Headline Server MW200 Compare-o-matic
Done
Internet

```
Brought to you by: Java Developers Journal
```

FIGURE 1 SimpleTag in action

Enhancing SimpleTag

I've discussed the components of a custom JSP tag and the syntax for its use in a JSP. Now let's look at some more advanced tags to begin learning how to build really useful libraries.

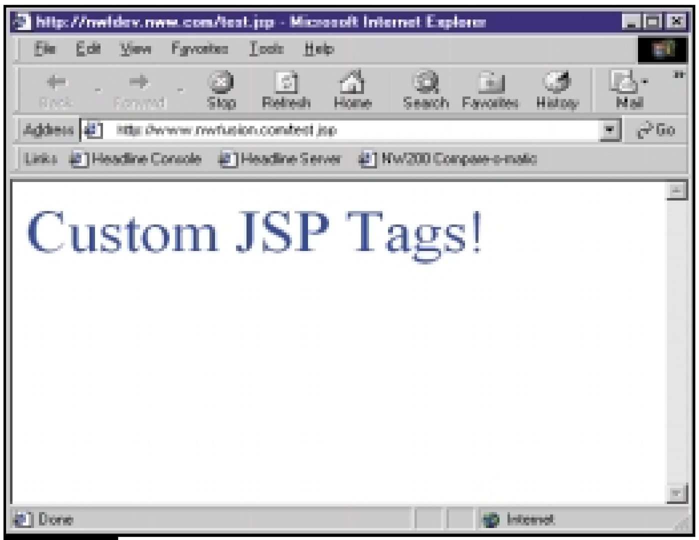
SimpleTag didn't use a body, but suppose I wanted to have a tag that could output some text at the beginning and end of an arbitrary piece of content. I can easily extend SimpleTag to do this by implementing the doEndTag() method. This method works just like the doStartTag in that it obtains a JspWriter and prints content to it, but (as its name implies) the JSP engine always calls it when it encounters the close tag for a custom tag. Like doStartTag(), the doEndTag() method must also return an int. The possible values it can return are:

- **SKIP_PAGE:** Instructs the JSP engine to ignore anything on the page that follows this tag
- **EVAL_PAGE:** Instructs the JSP engine to continue processing the rest of the page as normal

I'll modify the SimpleTag to produce some simple font-formatting tags that we can surround some text with. The doStartTag method will output font and formatting information and the doEndTag will close our font tag. Now any text that is embedded in the JSP between <jdj:simple> and </jdj:simple> will take on those attributes.

Note that I've changed doStartTag to return EVAL_BODY_INCLUDE so the engine will evaluate and return any text between the tags (see Listing 2). EVAL_PAGE is returned by doEndTag so the rest of the page after my close tag is interpreted as well. Since I'm changing the content type of my body – that is, my body will no longer be empty – I also need to change the tld file. I do this by changing the value of <bodycontent> in the tld from "empty" to "JSP". This tells the JSP engine that my tag may contain standard JSP that should be treated as such (see Listing 2).

For my last step I just need to modify the JSP to put some text between the tags (see Figure 2 and Listing 3).


http://nwldev.nww.com/test.jsp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://www.nwfusion.com/test.jsp
Go
Links Headline Console Headline Server NW200 Compare-o-matic
Done
Internet

Custom JSP Tags!

FIGURE 2 Enhanced SimpleTag

To make tags truly useful, they often need to be flexible enough to offer settable attributes to the tag user that help modify their behavior. For instance, with SimpleTag I may want to allow the tag user to decide what color the font is (see Listing 4). To support an attribute, I need to modify both my class file and deployment descriptor. First I need to add a method to the tag class called setXXX (where "XXX" is the attribute name). In this example the method is called setFontcolor (the first letter of any attribute is always capitalized when calling a "set" property). This method will set a private field within the tag that I'll use within doStartTag to set the color of the font.

The next step is including the attribute in jdj.tld. To do so, I add an attribute to the end of SimpleTag's <tag> definition.

```
<attribute>
  <name>fontcolor</name>
  <required>false</name>
</attribute>
```

The "name" of my attribute is the name that users of the tag will refer to it by. For example, indicating "required," as you might assume, determines whether this attribute must be provided by the tag's user. Now I simply add the attribute to my tag within the JSP.

```
<jdj:simple fontcolor="green">It ain't easy being…</jdj:simple>
```

An attribute can even be the result of a runtime Java evaluation. To support this ability, I add the following line within the attribute definition:

```
<rtexprvalue>true</rtexprvalue>
```

A value of true here indicates to the JSP engine that it should evaluate any JSP appearing in the attribute and pass its result to the tag. So I could pass a value of red to my tag this way:

```
<jdj:simple name=<%= new String("red").toUppercase() %>/>
```

Tags That Modify Their Body

So far, my tag has always just ignored its body or included it verbatim. However, there are times when a tag might want to inspect and/or modify the contents of its body. For a tag to have this ability it must implement a different interface than Tag: BodyTag. Like TagSupport, BodyTag also has a convenience class that defines default methods for the interface called BodyTagSupport. A tag that implements BodyTag can parse the contents of its tag body and output anything based on that content. One example might be a tag that does simple conversion from HDML to WML or formats a block of text into paragraphs. For my example I'll create a simple body-modifying tag that will change any content in its body to uppercase. To do so I'll introduce a new method here that's useful only on BodyTag classes, called doAfterBody. This method gets called after doStartTag and before doEndTag, giving the tag author a place to inspect the tag body and either print it verbatim, ignore it or change it. My doAfterBody method looks like this:

```
public int doAfterBody() {
  try {
    BodyContent body = getBodyContent();
    JspWriter writer = body.getEnclosingWriter();
    writer.print(body.getString().toUppercase());
  } catch (Exception x) {
  return(SKIP_BODY);
}
```

For this tag I don't need a doStartTag or a doEndTag since I'm only concerned with changing the tag's body. You'll notice that the way to get the contents of the body is through a method called getBodyContent, which is defined in BodyTagSupport. From it I can get the actual String containing the contents of the body by calling getString(). Note also that while in doAfterBody, the JspWriter must be retrieved by calling getEnclosingWriter() from the BodyContent class instead of from the pageContext. The full listing for the class is in Listing 5.

The tld for the UppercaseTag looks like:

```
<tag>
  <name>upper</name>
  <tagclass>com.jdj.UpperCaseTag</tagclass>
  <bodycontent>JSP</bodycontent>
</tag>
```

Any content included between the <jdj:upper> and </jdj:upper> tags will now be forced to uppercase.

Like both doStartTag and doEndTag, doAfterBody must return an int value. UpperCaseTag returns a value of SKIP_BODY from doAfterBody. A value of SKIP_BODY causes the engine to continue processing the rest of the page and not include the body in its original form. Instead, the body is skipped and the contents that have been written to the JspWriter are included instead. The only other possible return value for doAfterBody is EVAL_BODY_TAG. A tag can return EVAL_BODY_TAG to cause the JSP engine to call doAfterBody another time. Use of EVAL_BODY_TAG is common for tags that perform some looping – where a looping variable is checked at each call to doAfterBody() and SKIP_BODY is returned at the end of the loop.

Nested Tags

Another useful feature of JSP tags is the ability to have them work cooperatively by nesting them. When nested, outer tags can make methods and variables available to tags that are contained within their body. For example, I might want to define an outer tag that performs a JDBC query. I could then make the query's ResultSet available to inner tags that output individual column values. In this scenario the outer tag will have attributes that specify the driver to use, the JDBC connect String, and the SQL to execute. To make the results of the query available to inner tags, I need to define a public method that I'll call getDataValue(), which takes the column name and returns the String value for the current row and the specified column. The full definition of the QueryTag can be found in Listing 6. Note that with this tag I repeat the body contents by returning EVAL_BODY_TAG until I reach the end of the ResultSet, where I return SKIP_BODY.

Next I'll define an inner tag, which I'll call a DataValueTag. This tag will have a single attribute to specify which column to write the value of. Since DataValueTag will be nested within a QueryTag, I need to get the instance of the parent QueryTag and call getDataValue on it. Inner tags get access to their enclosing tags by calling findAncestorWithClass and passing it the inner tag's current instance (this) and the class type of the outer tag:

```
QueryTag qt = (QueryTag) findAncestorWithClass(this,
QueryTag.class);
```

If qt is null, I throw an exception since my DataValueTag must appear within a QueryTag. If it isn't null, I do the following:

```
String val = qt.getDataValue( columnName );
out.print(val);
```

The code for the DataValueTag and tld for these tags can be found in Listings 7 and 8, respectively. By combining these two tags with, for example, an HTML table, I can easily create a JSP page that displays only the desired columns within each row in a particular SQL query.

JSP Life Cycle

Once you begin building custom tags and the JSP pages they're contained in, you may find yourself confused (as I was) about what's actually getting executed when. Without a clear picture of the life cycle and execution order of a page, your JSPs can give you unexpected results. Luckily, there's a simple flow of events and a couple of ground rules that can help clear this all up.

The first (and rather intuitive) ground rule is that page elements, whether lines of Java code or JSP Tags, are evaluated from top to bottom in the order in which they appear. When JSP code is encountered, it's simply executed; when a tag is encountered, its appropriate methods are called one at a time depending on the tag type. The following is the flow of methods that are called on a tag when it's encountered in a page:

1. Two methods, setParent() and setPageContext(), are called on the TagHandler class. These methods are handled automatically by BodyTagSupport and TagSupport so you don't need to implement them explicitly if your tag extends either one of them.
2. Any set methods for attributes on this tag are called.
3. doStartTag() is called. If you haven't implemented this method, the flow continues. Otherwise you must return one of the following:
   - **SKIP_BODY:** Instructs the engine to ignore the body for this tag if one exists
   - **EVAL_BODY_TAG:** Instructs the engine to evaluate the body and call the Tag's doInitBody() method (relevant only for tags that implement the BodyTag interface; tags that extend from BodyTagSupport implement this interface)
   - **EVAL_BODY_INCLUDE:** Instructs the engine to evaluate and include anything in the tag body; engine proceeds to step 7 (relevant only for tags that implement the BodyTag interface; tags that extend from BodyTagSupport implement this interface)
4. setBodyContent() is called on the tag. This allows classes that extend BodyTagSupport to evaluate, manipulate and modify the body of the tag.
5. doInitBody is called. Any initialization necessary before doAfterBody is called can be done here (setting up Connections, setting variables in the pageContext, etc.).
6. doAfterBody is called. A return value of SKIP_BODY here will result in doEndTag() being called. Returning EVAL_BODY_TAG will produce another call to doAfterBody.
7. doEndTag() is called. A return value of EVAL_PAGE here will result in the rest of the JSP page being evaluated by the engine. Returning SKIP_PAGE will tell the engine to ignore the rest of the page.

It's sometimes useful to look at the source code for the servlet your JSP engine produces from a particular JSP to help understand when things occur in a page.

Sharing Variables

One last topic I'll cover is variable scope. What if I want to share data back and forth between tags and the JSP within a page? Can I pass variables back and forth? Well, I've already shown that tag attributes can be used to pass variables to a tag. But what about the inverse, making variables available to the JSP scope from within a tag? Can this be done? The answer, of course, is Yes, but the approach isn't quite intuitive. At first you might assume that you could just write a tag whose output is a line of Java code that defines a variable. Something like:

```
public class DefinesAVarTag extends BodyTagSupport {

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.println("<% String magazineTitle = \"JDJ\";%>");
    } catch ( Exception e ) {
       System.out.println( e );
    }
  }
}
```

Then use the tag like this:

```
<jdj:definesAVar/>
<% if (magazineTitle.equalsIgnoreCase("JDJ") ) { %>
    I love Java Developers Journal
```

But try to load this page and you'll see that there's a problem. The JSP returned by DefinesAVarTag isn't executed – it's actually just returned to the client as text. Meanwhile, the "if" statement is executed as standard JSP, which makes the compiler complain that the variable "magazineTitle" isn't defined.

This behavior exemplifies one of the ground rules for using custom tags: a JSP tag can't return Java code itself. Any output from a custom tag is sent as is to the client browser instead of being interpreted as Java. Though this might seem somewhat restrictive, the limitation actually makes sense from a performance standpoint. It allows a JSP page to be compiled into a Servlet class once, when first requested. Subsequent requests just run through the now-compiled Servlet class. If tags were permitted to return JSP themselves, the compiler would have to compile the page for every single request since the Java code output from a tag could differ from call to call.

Despite this limitation I can create a tag that defines and exposes variables as long as I decide the types and number of those variables up front. Once I've settled on what types (the actual Java classes) and how many variables a particular tag will put into scope, I incorporate that information into a class that extends the TagExtraInfo class. This class has only one method, which JSP engine calls to learn about the variables a particular tag defines. The method, called getVariableInfo, returns an array of VariableInfo objects that define the name, type and scope of our variables. Here's an example:

```
public class JDJExtraInfo extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[]
            {
                new VariableInfo("username",
                                "String",
                                true,
                                VariableInfo.NESTED)
    };
    }}
```

In this example the JDJExtraInfo class indicates that any tag associated with it will define one variable of type String called "username". The third parameter in the VariableInfo constructor will always be true for Java, while the fourth indicates what scope the variable should have. The possible scopes are:

- **VariableInfo.NESTED:** Means the variable will be in scope for any JSP found between the open and close tags
- **VariableInfo.AT_BEGIN:** Means that the variable will be in scope after the start tag and for the remainder of the page
- **VariableInfo.AT_END:** Means that the variable won't be in scope until after the end tag and will remain in scope through the end of the page

Once I've created a TagExtraInfo class, I need to associate the tag with it. This is done with a new line in my tld file:

```
<tag>
 <name>membership</name>
<tagclass>com.jdj.MemberShipTag</tagclass>
<teiclass>com.jdj.JDJExtraInfo</teiclass>
</tag>
```

Next I'll modify the tag to actually publish this variable. I do it by simply setting the variable in the pageContext from within any of the methods in my TagHandler. For example, the doStartTag might include the following line:

```
pageContext.setAttribute( "username", theUser );
```

where theUser is some String variable retrieved from elsewhere, like the session or the EJB layer.

Any variable exposed this way must match exactly the name and Java type of one of the VariableInfo objects in my TagExtraInfo class. Now I can use my membership tag within a JSP and refer to username like any other variable:

```
<jdj:membership>

  Welcome back <%= username%>

</jdj:membership>
```

Conclusion

Custom JSP tag libraries are a valuable addition to the J2EE standard and offer several advantages over previous methods of separating logic and presentation within Java. By using the tactics introduced in this article, developers can begin building rich tag libraries that can abstract complex business logic from user interface design.

AUTHOR BIO

Adam Chace is a senior Web software engineer at Network World, Inc. (an IDG company), where he contributes to site architecture and development for Fusion (www.nwfusion.com) a leading site for networking professionals. He has over two years of experience developing wired and wireless Internet applications using server-side Java.

achace@nww.com

Listing 1

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class SimpleTag extends TagSupport {

public int doStartTag() {
  try {
    JspWriter out = pageContext.getOut();
    out.println("Java Developers Journal");
  } catch (IOException ioe) {
    System.out.println("Error writing to out: " + ioe);
  }
    return(SKIP_BODY);
  }
}
```

Listing 2

```
public class SimpleTag extends TagSupport {

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.println("<font face=geneva size=8  color=blue>");
    } catch (IOException ioe) {
      System.out.println("Error writing to out: " + ioe);
    }
      return(EVAL_BODY_INCLUDE);
    }

  public int doEndTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.println("</font>");
    } catch (IOException ioe) {
      System.out.println("Error writing to out: " + ioe);
    }
    return(EVAL_PAGE);
  }
}
```

Listing 3

```
<%@ taglib uri="jdj.tld" prefix="jdj" %>

<html>
  <body>
    <jdj:simple>
     Custom JSP Tags!
    </jdj:simple>
  </body>
</html>
```

Listing 4

```
private String color;
```

```
//Set via the tag attribute fontcolor
public void setFontcolor(String value ) {
  color = value;
}

public int doStartTag() {
  try {
    JspWriter out = pageContext.getOut();
    out.println("<font face=geneva size=8 color="+color+">");
 } catch (IOException ioe) {
    System.out.println("Error writing to out: " + ioe);
  }
    return(EVAL_BODY_INCLUDE);
  }
```

Listing 5

```
package com.jdj;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class UpperCaseTag extends BodyTagSupport {

public int doAfterBody() {
  try {
    BodyContent body = getBodyContent();
    JspWriter out = body.getEnclosingWriter();
    out.print( body.getString().toUpperCase() );
  } catch (IOException ioe) {
    System.out.println("Error writing to out: " + ioe);
  }
    return(SKIP_BODY);
  }

}
```

Listing 6

```
package com.jdj;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.sql.*;

public class QueryTag extends BodyTagSupport {

  private ResultSet rs = null;
  private Statement st = null;
  private Connection con = null;

  //Attributes
  private String connectString;
  private String driverClass;
  private String sql;
  private String user;
  private String password;


  public int doStartTag() {
    loadData();
    return( EVAL_BODY_TAG );
  }

  public int doEndTag() {
    closeConnections();
    return( EVAL_PAGE );
  }

  private void closeConnections() {
    try { rs.close(); } catch ( Exception e ) {};
    try { st.close(); } catch ( Exception e ) {};
    try { con.close(); } catch ( Exception e ) {};
  }

  private void loadData() {
    try {
      Class.forName( driverClass );
      con = DriverManager.getConnection( connectString, user,
password );
      st = con.createStatement();
      rs = st.executeQuery( sql );
      rs.next();
    } catch ( Exception e ) {
      System.out.println("Error loading data: " + e );
    }
  }

  public int doAfterBody() {
    try {
      BodyContent body = getBodyContent();
      JspWriter out = body.getEnclosingWriter();
      out.println( body.getString() );
      //Clear the body (in case we loop again)
      body.clearBody();
      if ( rs.next()  ) {
         //There is another row so evaluate the body again
        return( EVAL_BODY_TAG );
      } else {
        //Last row so don't evaluate the body anymore
        return( SKIP_BODY );
      }
    } catch ( Exception e ) {
      System.out.println( "Error in doAfterBody: " + e );
      return( SKIP_BODY );
    }
  }

  //This is the method our nested tags will call to get the
  //value of a particular column in the current row
  public String getDataValue( String columnName ) throws
SQLException {
    return( rs.getString( columnName ) );
  }

  public void setConnectString( String value ) {
    connectString = value;
  }

  public void setDriver( String value ) {
    driverClass = value;
  }

  public void setQuery( String value ) {
    sql = value;
  }

  public void setUser( String value ) {
    user = value;
  }

  public void setPass( String value ) {
    password = value;
  }

}
```

Listing 7

```
package com.jdj;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;
import java.io.*;

public class DataValueTag extends TagSupport {
  //Attributes
  private String columnName = "";

  public DataValueTag() {
  }

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      //Now try to get the parent
      QueryTag qt = (QueryTag)findAncestorWithClass(this,
QueryTag.class);
      if ( qt == null ) {
       out.print( "Must be enclosed in a query tag!");
      } else {
          try {
            String val = qt.getDataValue( columnName );
            out.print( val );
        } catch ( Exception e ) {
          System.out.println(" An error occurred in DataVal-
ueTag " + e );
          out.println("An error occurred: " + e );
        }
      }
    } catch ( Exception ioe ) {
      System.out.println( "Error in doStarTag" + ioe );
    }
    return( SKIP_BODY );
  }


  public void setColname( String value ) {
    columnName = value;
  }
}
```

Listing 8

```
<tag>
  <name>query</name>
  <tagclass>com.jdj.QueryTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>sql</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>driver</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>connectString</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>user</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>pass</name>
    <required>true</required>
  </attribute>
</tag>

<tag>
  <name>data_value</name>
  <tagclass>com.jdj.DataValueTag</tagclass>
  <bodycontent>EMPTY</bodycontent>
  <attribute>
    <name>colname</name>
    <required>true</required>
  </attribute>
</tag>
```

Download the Code!

The code listing for this article can also be located at

www.JavaDevelopersJournal.com

—continued from page 7

So what does ProSyst have at stake in the battle for protocol dominance? Simply, everything. Like the different parts of the body, ProSyst's embedded server acts as the brain that drives, orchestrates and governs the different parts, allowing them to function in perfect synchronicity. The new gateway technology, with all the new revenue streams and services it represents, is too good and too promising to adopt a wait-and-see attitude. We believe the first one to hit the market and do it right will score, but it's a big world with lots of devices and a lot more to come. "There is room in the market for everyone," says George Reel. "I guess the message I'd like to send out is, Go ahead and start building your gateways. You don't have to hedge your bets that they will be covered."

Further movement to counterpunch the polarized protocol dilemma is mounting; it's finding a place with developers who need something now that can be easily implemented at little or no cost. To answer this call is the protocol Salutation. It's an open protocol with no royalties, and it provides service discovery and service management. It has strong support from the IBM and Bluetooth camps and is making a serious bid to fill the promises of other protocols that have not yet delivered the goods. It's posturing itself to be the Linux of protocols and is especially fitted for low-cost device applications. The embedded world is playing with it enough so it's quickly becoming one of the sharpest tools in the shed.

Silently, in backrooms and golf courses, deals are being penned to bring these capabilities to bear. Service channels are being developed and smart homes are cropping up everywhere. And, while no one owns a crystal ball, the writing is on the wall. Historically, it's during times like this, that a smaller, more agile player seizes the opportunity to capitalize on market uncertainty and devises a universal solution that slays the sleeping giants.

If ProSyst feels it has found the answer in being platform independent, others are still too politically motivated by the big names to act. So what's causing the indecisiveness? Perhaps they're still waiting for their other projects to be completed. Perhaps they don't feel the real world needs or is ready to pay for a computer in the refrigerator, or they're afraid of alienating the campus bully. Either way, it's safe to say that progress never sleeps and never retreats. The clash of the titans will rage on, and there's no way to address the unforeseen. It's just good to know that there are options and pioneers who believe that it's better to work together than defeat one another.

Using JNI for Safer Java Servers Under UNIX

Create a test Java application

WRITTEN BY
MICAH SILVERMAN

For those of you not too familiar with the UNIX way of life, here's a brief overview. There are really two categories of accounts under UNIX: the superuser (named *root*) and everything else. Being root on a UNIX machine gives you the keys to the kingdom. You can remove files created by any other user, for instance. You can stop running processes started by root or any other user. A UNIX system can be entirely compromised if an unauthorized person or process gains root access.

One of the more common ways to do this is to exploit bugs in server processes that are running as the superuser. Generally, if no extra code has been put in place to prevent it, subprocesses spawned by a process running as root will also be running as root.

So Why Run Anything As Root?

As a protection against unauthorized common users hijacking a UNIX machine, processes that need to bind to low ports (0-1024) must be run as root. Web servers, for instance, run on port 80 by default. Without this protection any old user could start up a process that listened on port 80.

One of the common ways to avoid the risk of running as root is to switch to a different userid after binding to a low port. Remember, the superuser can do just about anything, including assuming the identity of another user. UNIX has a system call, setuid(), that allows the userid of a process to be changed to another one. So, typically, a process would start as root, bind to the appropriate port, and then call setuid()with the appropriate userid to switch to a less privileged user. If the process is compromised in this scenario, damage is limited to those processes and files that the less privileged user has access to.

What Does Java Know from UID?

Java is platform independent. Java applications run inside the Java Virtual Machine. The JVM has no internal representation for the platform-specific concept of UID.

The creators of Java understood that sometimes platform-specific operations would need to take place. This is where JNI (Java Native Interface) comes in.

Nuts and Bolts of JNI

I had a situation where we needed to ensure that the Java process could run as an unprivileged user but still bind to a low port – in this case the standard ftp port (21). Since the platform is UNIX (Linux, in particular), a BindException would be thrown when the thread running in the JVM attempts to create a ServerSocket bound to port 21 if the JVM is running as an unprivileged user.

I was able to use JNI to call the native setuid() and switch to the unprivileged user after the ServerSocket had been created and bound to port 21. Further, since my development environment is on Windows NT, and in order to maintain the cross-platform capability of Java, I created native stub code for the Windows platform as well. Following are the steps involved.

1. Create Java code with native methods (see Listing 1).
2. Compile Java code with javac (i.e., javac UID.Java).
3. Generate header file with javah (i.e., javah-jni micah.util.UID; see Listing 2).
4. Create native C module to implement the methods declared in the Java code (see Listings 3, 4).
5. Compile the native code.
6. Create a test Java app (see Listing 5).

Create Java Code with Native Methods

Referencing native code in Java is very straightforward (see Listing 1): simply include the keyword native in the signature of an empty method declaration terminated with a semicolon:

```
public static native int setuid(int
uid);
```

In my example these methods are static because I don't need to maintain an object in order to use these methods.

Notice the lines:

```
static {
 System.loadLibrary("uid");
}
```

One of the nice things about JNI is that the actual loading of the native library is handled in a platform-independent way. The name of the library and its location is what is platform dependent. In the case of UNIX a file named libuid.so is expected to be in the LD_LIBRARY_PATH. In the case of Windows a file named uid.dll is expected to be in the system path.

This is declared as a static initializer to ensure that the library is loaded before methods of the class are referenced.

Generate Header File with Javah

Java comes with a utility called javah that is used to generate a .h file for use with C programs (see Listing 2). The nice thing when building an app that uses JNI is that you don't have to memorize the JNI calling conventions, structure names and return types. javah generates a file that has the function definitions declared that you'll need to implement in your C program. javah works similarly to Java in terms of classpath and package arrangement. Listing 2 was created using the command:

```
javah -jni micah.util.UID
```

Here is an example of one of the function definitions from the generated file:

```
JNIEXPORT jint JNICALL
Java_micah_util_UID_setuid
  (JNIEnv *, jclass, jint);
```

Notice that the calling convention is directly related to the package arrangement of your Java class. If that changes for some reason, you'll need to rerun javah to get the proper include file.

Create Native C Module to Implement the Methods Declared

Because I want to maintain my ability to develop on Windows NT and deploy on UNIX, I have created two platform-dependent implementation files: unix_uid.c and win_uid.c (see Listings 3 and 4). These files will be compiled to libuid.so and uid.dll, respectively.

Let us look at the UNIX code first. Notice the first two lines of the file:

```
#include <jni.h>
#include "micah_util_UID.h"
```

These references are required. The first file is found in the include directory of the Java distribution. The second is the file generated by javah.

There are a number of UID and GID manipulation system calls on UNIX. An in-depth discussion of these is outside the scope of this article. Suffice it to say that each UNIX call has been represented in my Java class (see Listing 1). Each of these calls follows a similar pattern. I set up the function just as outlined in the generated .h file (see Listing 2):

```
JNIEXPORT jint JNICALL
Java_micah_util_UID_setuid (JNIEnv *
jnienv,
jclass j, jint uid)
{
    return((jint)setuid((uid_t)uid));
}
```

The only thing different that I need to do from an ordinary call to setuid is to properly cast the parameter ((uid_t)uid) and the return value ((jint)) based on the signature generated by javah for the function.

As far as the Windows code goes, all of the functions return zero, which is the return value for a successful completion of the system call under UNIX.

"Being root on a UNIX machine gives you the keys to the kingdom"

Compile the Native Code

I used the following command to compile the code under UNIX (Linux, in this case):

```
gcc \
-I/usr/local/java/include \
-I/usr/local/java/include/genunix \
-shared unix_uid.c -o libuid.so
```

I used the following command to compile the code under Windows NT:

```
cl -Ie:\jdk1.1.8\include -
Ie:\jdk1.1.8\include\win32
-LD win_uid.c -Feuid.dll
```

Create a Test Java App

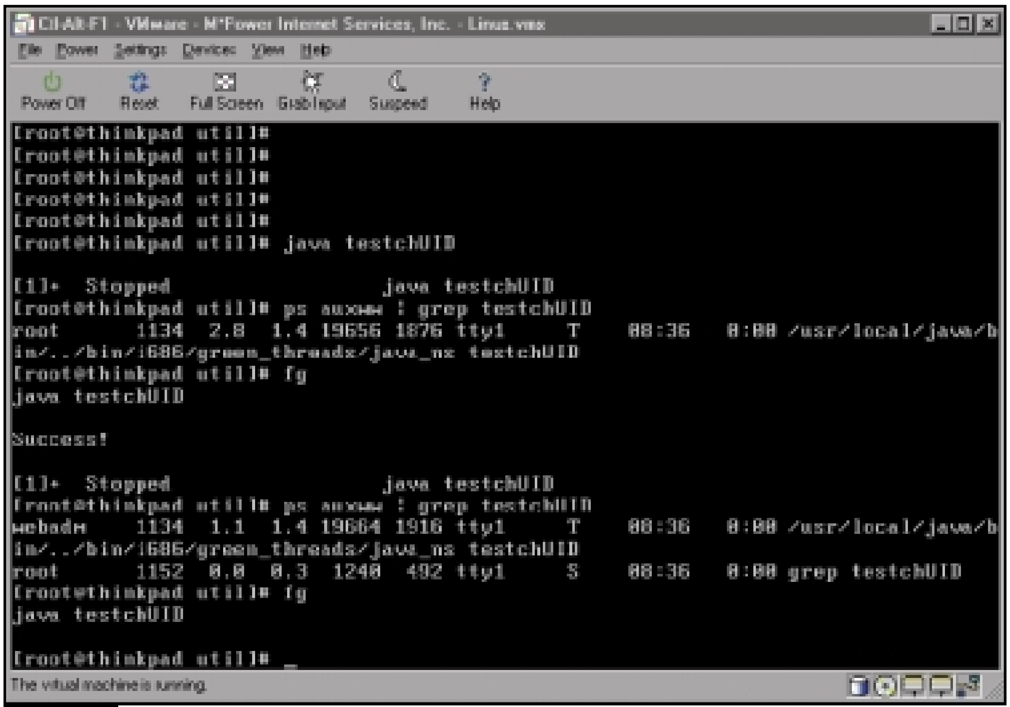
The test app shown in Listing 5 simply waits for input from stdin (so we can see what user the process is running as), calls UID.setuid(1010) (remember, it's static so we don't need an instantiated object), prints out a success message and does another read from stdin (so we can verify that the user has been changed). Figure 1 shows this in action. Notice that after the first time the Java application is stopped, it is running as root. After the second time the Java application is stopped, it is running as webadm because the native code has been called.

For more information check out Sun's JNI tutorial at http://java.sun.com/docs/books/tutorial/native1.1/index.html.

micah.silverman@appliedtechnologygroup.cc

AUTHOR BIO

Micah Silverman took an interest in UNIX internals and networking during the course of his computer science studies in the late '80s. He has been developing Java applications since its release. He is a cofounder of the Applied Technology Group, an Internet development and consulting firm (www.appliedtechnologygroup.cc).


Ctrl-Alt-F1 - VMware - M*Power Internet Services, Inc. - Linux.vmx
File Power Settings Devices View Help
Power Off
Reset
Full Screen
Grab Input
Suspend
Help
The virtual machine is running.

```
[root@thinkpad util]#
[root@thinkpad util]#
[root@thinkpad util]#
[root@thinkpad util]#
[root@thinkpad util]#
[root@thinkpad util]# java testchUID

[1]+  Stopped                 java testchUID
[root@thinkpad util]# ps auxww | grep testchUID
root      1134  2.8  1.4 19656 1876 tty1    T    08:36   0:00 /usr/local/java/b
in/../bin/i686/green_threads/java_ns testchUID
[root@thinkpad util]# fg
java testchUID

Success!

[1]+  Stopped                 java testchUID
[root@thinkpad util]# ps auxww | grep testchUID
webadm    1134  1.1  1.4 19664 1916 tty1    T    08:36   0:00 /usr/local/java/b
in/../bin/i686/green_threads/java_ns testchUID
root      1152  0.0  0.3  1240  492 tty1    S    08:36   0:00 grep testchUID
[root@thinkpad util]# fg
java testchUID

[root@thinkpad util]# _
```

FIGURE 1 testchUID running as root and then as webadm

Listing 1

```
micah/util/UID.java

package micah.util;

public class UID {

    public static final int SUCCESS = 0;
    public static final int FAILURE = -1;

    public static native int setuid(int uid);
    public static native int seteuid(int uid);
    public static native int setgid(int gid);
    public static native int setegid(int gid);

    static {
 System.loadLibrary("uid");
    }
}
```

Listing 2

```
micah/util/micah_util_UID.h

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class micah_util_UID */

#ifndef _Included_micah_util_UID
#define _Included_micah_util_UID
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     micah_util_UID
 * Method:    setuid
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_micah_util_UID_setuid
  (JNIEnv *, jclass, jint);

/*
 * Class:     micah_util_UID
 * Method:    seteuid
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_micah_util_UID_seteuid
  (JNIEnv *, jclass, jint);

/*
 * Class:     micah_util_UID
 * Method:    setgid
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_micah_util_UID_setgid
  (JNIEnv *, jclass, jint);

/*
 * Class:     micah_util_UID
 * Method:    setegid
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_micah_util_UID_setegid
  (JNIEnv *, jclass, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 3

```
micah/util/unix_uid.c

#include <jni.h>
#include "micah_util_UID.h"
#include <sys/types.h>
#include <unistd.h>

JNIEXPORT jint JNICALL
Java_micah_util_UID_setuid (JNIEnv * jnienv,
jclass j, jint uid)
{
    return((jint)setuid((uid_t)uid));
}
```

```
JNIEXPORT jint JNICALL
Java_micah_util_UID_seteuid (JNIEnv * jnienv,
jclass j, jint uid)
{
    return((jint)seteuid((uid_t)uid));
}

JNIEXPORT jint JNICALL
Java_micah_util_UID_setgid (JNIEnv * jnienv,
jclass j, jint gid)
{
    return((jint)setgid((uid_t)gid));
}

JNIEXPORT jint JNICALL
Java_micah_util_UID_setegid (JNIEnv * jnienv,
jclass j, jint gid)
{
    return((jint)setegid((uid_t)gid));
}
```

Listing 4

```
micah/util/win_uid.c

#include <jni.h>
#include "micah_util_UID.h"

JNIEXPORT jint JNICALL
Java_micah_util_UID_setuid (JNIEnv * jn, jclass j,
jint uid)
{
 return(0);
}

JNIEXPORT jint JNICALL
Java_micah_util_UID_seteuid (JNIEnv * jn, jclass j,
jint uid)
{
 return(0);
}

JNIEXPORT jint JNICALL
Java_micah_util_UID_setgid (JNIEnv * jn, jclass j,
jint gid)
{
 return(0);
}

JNIEXPORT jint JNICALL
Java_micah_util_UID_setegid (JNIEnv * jn, jclass j,
jint gid)
{
 return(0);
}
```

Listing 5

```
testchUID.java

import java.io.*;
import micah.util.*;

class testchUID {
 public static void main(String[] args) {
  try {
   System.in.read();
  }
  catch (IOException ioe) {}
  int result=UID.setuid(1010);
  if (result == UID.SUCCESS) {
      System.out.println("Success!");
  }
  else if (result == UID.FAILURE) {
      System.out.println("Failure!");
  }
  try {
   System.in.read();
  }
  catch (IOException ioe) {}
 }
}
```

WRITTEN BY P.G. RAMACHANDRAN

Java & Security

Java 1.2 Security Manager facilitates fine-grained security policies

There are many concerns surrounding the security of Java applets and applications downloaded from the Internet. But because Java developers placed a lot of importance on security from the start, Java is the preferred technology for use in networked environments. When Java's security features are implemented properly, Java programs are safe and can be downloaded to your computer without any security risk.

There are a number of ways to implement security in Java. Security features can be applied to applets running in browsers as well as to Java applications. This article discusses the features of the Security Manager class.

Java Security Manager

The Java Security Manager arbitrates access to many of the operating system features such as files, network sockets and printers. It provides fine-grained control over which operations can be performed by code running within the Java Virtual Machine (JVM). The relationship between the Security Manager and other components is illustrated in Figure 1.

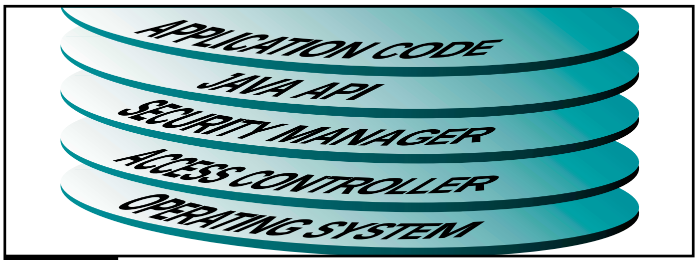


FIGURE 1 Relationship between the Security Manager and other components

The application code uses Java API methods to access the Security Manager, which in turn uses an access controller to implement the security policies. However, under special circumstances the Security Manager can bypass the access controller to implement security policies.

The role of the Security Manager is to grant access to each Java class based on the amount of trust the user has in that class. The Java program must get permission from the Security Manager before it can connect to a particular machine on the network or access the file system. Whenever a Java program performs a restricted operation, it checks with the Security Manager to determine if that operation can be performed. If access is denied, the program throws a SecurityException. Nothing inherent in the Security Manager requires security to be enforced as an all-or-nothing proposition for each class.

The Security Manager can be written so that only classes loaded from the CLASSPATH are prohibited from performing certain operations normally permitted to classes loaded from the file system. The Security Manager has powerful features that can enforce a very detailed complex policy if necessary.

The most significant difference in Java 1.2 is that it's much easier to implement fine-grained security policies. The Security Manager in versions prior to Java 1.2 relied on internal logic to determine what policies should be in effect. Changing the policy required changing the Security Manager itself. The Java 1.2 Security Manager uses an access controller to enforce its protections. A Security Manager isn't necessary for every Java application, and applications by default have no Security Manager. Plenty of Java implementations are available and they don't have a standard Security Manager implementation. The methods in the Security

Manager can be broadly classified into the following groups:

- ***Methods protecting file access***: These methods protect the file system from the user classes.
- ***Methods protecting network access***: These methods check the security details about the sockets and other network aspects.
- ***Methods protecting program threads:*** These methods protect the manipulation of threads by other classes.
- ***Methods protecting the JVM***: These methods protect the integrity of the JVM.
- ***Methods protecting system resources:*** These methods protect the system resources, such as printers and system properties.
- ***Methods protecting Java security aspects***: These methods protect security aspects of Java itself.

SecurityManager and Applets

The classes that constitute the applet are generally loaded from the network and in general are considered untrusted (for the sake of simplicity signed applets are not discussed here). An applet cannot set the Security Manager, so it will have to live with the established security policy of the browser it's running on. The implementation of browser SecurityManager prevents applets loaded over the network from reading or writing files from the client file system. Applets aren't allowed to get direct access to the underlying computer. Some other restrictions associated with applets are:

- Applets can only create network connections back to the originating host of the applet.
- Classes loaded over the network can't load libraries or consist of native methods. (This doesn't say applets can't call native methods – only that native methods can't be part of the classes loaded over the network.)
- Applets can't fork off new system processes or see system properties that would expose the username or working directories.

How to Create a Security Manager

The SecurityManager class itself isn't intended to be used directly in your Java program (each of the checks defaults to throwing a security exception), but instead is intended to be inherited and installed as the System Security Manager. The subclassed Security Manager can be used to implement the desired security policy. The Security Manager is an abstract class and you need to extend this class to create your own Security Manager. The code for creating your own Security Manager is shown in Listing 1.

The Security Manager class has more than 25 checkXXX() methods, and these methods can be overridden to grant or deny access to a user. No methods in the Security Manager class need to be overridden. By default the Security Manager class throws an exception to all implementations of checkXXX()method, meaning that access is denied. To grant access to a certain method you must override it. One thing you should keep in mind before overriding the methods is that the methods should return if access is granted; otherwise, the methods should throw a SecurityException.

The checkRead() and checkWrite() methods are being overwritten in the Security Manager in this program. These methods grant or deny access to a particular file in the file system. The Security Manager provides three versions of checkRead() and two versions of checkWrite(). The signature of the methods that check whether a program is allowed to read the given file is:

```
public void checkRead(FileDescriptor fd)
public void checkRead (String file)
public void checkRead(String file, Object context)
```

The signature of the methods that check whether a program is allowed to write the given file is:

```
public void checkWrite(FileDescriptor fd)
public void checkWrite(String file)
```

The methods that are overridden in the program are checkRead(String s) and checkWrite(String s) methods in the SecurityManager class. All the checkXXX() methods throw SecurityException. This exception is a subclass of RuntimeException so it doesn't need to be within a try/catch block. This utilizes the property of the Java language in which exception handling is free as long as no exceptions are thrown. It's up to the Java API to call the checkXXX() methods at the appropriate time so you don't have to call the methods explicitly before a file read or write operation.

Next we'll install the newly created SecurityManager in our JVM. The code for installing and testing "MySecurityManager.java" can be found in Listing 2.

The two methods in the System class used to work with the Security Manager are:

1. ***Public static SecurityManager getSecurityManager():*** Returns a reference to the currently installed Security Manager object (or null if no Security Manager is in place). This method can be used to test various security policies.
2. ***Public static void setSecurityManager(SecurityManager sm)***: Sets the Security Manager for the object. It can be called only once and can't be removed once the Security Manager is installed. The Security Manager will be used by other classes that run in that particular virtual machine.

To execute and test the program:

- Create a file called input.txt in the current directory and type some text in the file.
- Compile "MyTest.java" and "MySecurityManager.java" and run the program. The contents of the file input.txt will be transferred to the file output.txt. The output of the program is:

```
Successfully opened the files for read/write
Successfully performed the read/write operation
```

- Change the value of the flag passed to the constructor of MySecurityManager in MyTest.java.
- Compile both Java files and run the program. The program throws an exception, indicating that the security policy we implemented doesn't allow the user to read the file in the JVM that we're running the program on. The output of the program if the flag is set to false is:

```
Exception in thread "main" java.lang.SecurityException: checkRead
        at MySecurityManager.checkRead(MySecurityManager.java:14)
        at java.io.FileInputStream.<init>(FileInputStream.java:65)
        at java.io.FileReader.<init>(FileReader.java:35)
        at MyTest.testFunc(MyTest.java, Compiled Code)
        at MyTest.main(MyTest.java:20)
```

Conclusion

The publicity given to the security holes in various widely used software applications put increased pressure on developers and companies. Both are realizing the importance of implementing tight security policies in their applications. Java security has undergone a lot of changes in each version of the programming language.

This article provides an overview of the Security Manager class in Java 1.2. The listings show an implementation of a Security Manager that grants or denies access to files in the file system. Having a basic understanding of the Security Manager will help you create the complex security policies required by applications to prevent misuse.

References

Java tutorial: www.javasoft.com
Oaks, S. (1998). *Java Security*. O'Reilly.

AUTHOR BIO
P.G. Ramachandran is a senior software engineer at Tivoli Systems, Indianapolis, Indiana.

p_g_ramachandran@tivoli.com

Listing 1

```
//CodeMySecurityManager.java
public class MySecurityManager extends SecurityManager
{
    private boolean flag;
    public MySecurityManager(boolean flag)
    {
        this.flag = flag;
    }

    // Override checkRead function
    public void checkRead (String s)
    {
        if(!flag)
        {
            throw new SecurityException("checkRead");
        }
    }

    // Override checkWrite function
    public void checkWrite (String s)
    {
        if(!flag)
        {
            throw new SecurityException("checkWrite");
        }
    }
}
```

Listing 2

```
// Code MyTest.java
import java.io.*;

public class MyTest
{

  public static void main (String[] args)
  {
    try
    {
      MySecurityManager secMgr = new MySecurityManager(true);
        // Set the security manager
      System.setSecurityManager(secMgr);
    }
    catch (SecurityException excp)
    {
      System.out.println("Can't Change the SecurityManager!");
    }
    // Construct an Object
    MyTest test = new MyTest();
    test.testFunc();

  }


  public void testFunc()
  {
    try
    {

      BufferedReader is = new BufferedReader(
                              new FileReader("input.txt"));
      DataOutputStream os = new DataOutputStream(
                                new FileOutputStream("out-
                                               put.txt"));

      System.out.println("Successfully opened the files for
                        read/write ");

      String readString;

      while ((readString = is.readLine()) != null)
      {
          os.writeBytes(readString);
          os.writeByte('\n');
      }
      System.out.println("Successfully performed the
      read/write operation");
      is.close();
      os.close();
    }
    catch (IOException excp)
    {
      System.err.println("IOException.");
    }
  }

}
```

Download the Code!

The code listing for this article can also be located at

www.JavaDevelopersJournal.com

Getting All Your Beans from One Bag

A scenario worth imagining

WRITTEN BY
NATHAN OJKA

Imagine this scenario: you've written all the appropriate interfaces and implementations for an EJB and now it's time to use it in client code. First you get a bean reference. Everything is simple enough: use JNDI to get the home interface, call a create method on it and catch all the possible exceptions. Voilà, a usable EJB reference. No big deal. However, after creating the bean and looking at the number of beans you want to use, you realize you'll be doing the same thing over and over again. You shake your head and say, "There has to be a better way to create these objects."

Fortunately, there is. Polymorphism and the reflection API provide a powerful and flexible mechanism for obtaining EJB references.

Without using reflection or polymorphism, you'd obtain references something like this:

```
try {
   // Use a helper method to get the
JNDI context to use for lookups...
   //
   InitialContext context = getJNDI-
Context();

   // Then lookup the bean home
interface and create the bean
   //
   String jndiName =
"com/zefer/util/MyBeanHome";
   MyBeanHome myHome = (MyBeanHome)
context.lookup(jndiName);
   IMyBean myBean = (myBean)
myHome.create();

} catch (...){
   // Catch all the exceptions…
}
```

Granted, this isn't the most complicated code in the world, but it can lead to problems with code management. What happens when you want references for several different beans? If you use casting, as in the example above, your code would be different for each bean since you have to hard-code the objects you're casting to. And what does the code look like if you want to use several different create methods? Again, you'd need separate code segments for each situation. The upshot is, you'd have similar-looking code. No matter how simple the code is, large amounts of duplicate code can be extremely difficult to manage.

This particular situation rings a bell for OO analysts – it's similar to a specialization relationship or creational operation that's most likely covered by a design pattern. Indeed, several design patterns directly address this situation: Strategy, Builder and Factory, to name a few. Some patterns, such as the Builder pattern, appear to be overkill as the differences in code are so slight and the number of situations so varied that the application of this pattern implies a large number of trivial classes. This large number effectively changes the nature of the problem to one of object management rather than code management, so it doesn't really improve the situation. Other techniques, such as delegation, provide an easier way to manage the code. Using delegation, every single procedure that gets a reference would be hard-coded in; therefore it's not very flexible in its application.

Even a strict application of the Factory pattern may leave a lot to be desired in terms of reducing the complexity of the code and the design. Thus the straightforward application of classic designs needs to be rethought in order to streamline the design and code for obtaining EJB references.

Luckily, the power of polymorphism and the Java reflection API come to the rescue. The latter provides the ability to invoke arbitrary method calls on arbitrary objects (as long as that method exists for that object, of course!). In our situation we want to invoke arbitrary create methods for enterprise beans. The concept of polymorphism comes into play in the client code after we've created EJB objects and want to cast them to an appropriate bean type. With these two items in hand it's possible to write a single class with a few methods that can handle every single creation scenario for any bean.

For this article I've constructed a class named EJBFactory using Netscape Application Server 4.0 and its EJB 1.0 implementation. The responsibility of the EJBFactory class is to return EJBObject references (see Listing 1). This class contains three fairly straightforward methods:

1. ***setJNDIFinder():*** Sets a reference to an object that provides a wrapper around JNDI contexts and lookup services
2. ***getHomeInterface():*** Gets a reference to a particular home interface for a bean
3. ***createBean():*** Creates an EJBObject reference

The first two methods are simple, containing little (if anything) that's surprising. The third method, createBean(), does the bulk of the work despite its deceptively diminutive size.

The createBean() method creates beans using the same process outlined above, namely, the method first obtains the home interface from JNDI, then calls the appropriate create() method on the home interface. For this method the first parameter to the method specifies the particular home interface to retrieve from JNDI. The appropriate create method is defined as the one that matches the signature of the objects in the Vector parameter to the method. To get the proper create method for the EJB home interface, the createBean() method first reflects the Vector parameter to get the classes of the object it contains, then reflects the home interface returned from JNDI to get a reference to the proper create method. After getting this reference, invoking the method to create the bean is a trivial task.

Using reflection in this manner allows the createBean() method to create any type of bean using any type of create method. After creating the bean, the method still has to return it. To work with disparate types of beans, the createBean() method relies on the fact that all the remote stubs for EJBs inherit from the EJBObject class. This allows the client code to use the object normally after either downcasting the returned object (for EJB 1.0) or calling the javax.rmi.PortableRemoteObject.narrow(...) method (per section 5.9 of the EJB 1.1 specification) on the object. Thus, by exploiting polymorphism, the client code can access the returned bean just as if it had been created using a more verbose method.

There are three main objections with this approach:

1. Reflection may be an expensive operation, so we have the classic flexibility versus performance trade-off. This trade-off is something that may be answered only on a situational basis.
2. The createBean() method throws a number of exceptions, thereby begging the question of whether this method of creating beans reduces duplicate code. However, the creation of a small wrapper class to handle these exceptions and throw application-defined exceptions solves this problem (see Listing 2).
3. Creating beans in this manner doesn't handle inheritance in the bean-create calls. In other words, if you try to use a create method that has a parent class as its signature, with a child class as the actual parameter, the bean won't be created appropriately. Unfortunately, I haven't found an elegant way around this final objection as it appears to be a limitation in the reflection API.

In spite of these objections, the EJBFactory class brings a number of benefits. It greatly reduces code duplication, making code more compact and easier to maintain, extend and debug. Creating beans is extraordinarily easy since the code consists of a single method call and an associated catch block (see Listing 3). Furthermore, the flexibility of the EJBFactory allows you to easily extend an EJB-based system to include new beans without having to modify any code. Now that's a scenario worth imagining!

ncuka@acm.org

AUTHOR BIO
Nathan Cuka is a senior software engineer for Zefer Corp.

Listing 1

```
micah/util/UID.java

import java.util.*;
import java.lang.reflect.*;
import javax.ejb.*;
import javax.naming.*;

/**
* A general purpose EJB factory.  If you wanted to
* use the abstract factory design pattern here, you could
* have this class implement a generic interface to provide
* a higher degree of abstraction.  That level of abstraction
* was not really needed for this example.
*/

public class EJBFactory {
   private JNDIFinder _jndiFinder = null;

   /** This constructor sets the JNDIFinder to use.  The
JNDIFinder class is
   * a wrapper around JNDI functionality such as object
lookups.
   */
   public EJBFactory(JNDIFinder finder) {
      setJNDIFinder(finder);
   }

   /** Method to get the home interface for an EJB with
   * the specified JNDI name.  The method it uses for lookup
   * is through a helper JNDIFinder object.
   *
   * @param jndiName  The name of the EJB object to get
   * @return An Object that is the home interface
   * @throws javax.naming.NameNotFoundException if the
   *         name does not exists in JNDI.  Also throws
   *         javax.naming.NamingException if there is
   *         a problem in looking up the name.
   *
   */
   public Object getHomeInterface(String jndiName) throws
      javax.naming.NameNotFoundException,
      javax.naming.NamingException
   {
      Object obj = null;
      obj = _jndiFinder.lookup(jndiName);
      return obj;
   }

   /** Method to create an enterprise bean.  Right now the
   * Vector of params is getting reflected to determine their
   * class type.
   * This might be an expensive operation and so we might
   * want to use something more efficient in the future to
   * speed things up.  The important thing to remember is
   * that ORDER IS IMPORTANT for these parameters.  Otherwise
   * if you have a create method that takes multiple parame-
   * ters of the same type, then you will get the method
   * invocation wrong.
   *
   * @param jndiName   The JNDI name of the EJB
   * @param params     A vector containing the arguments to
   * the create method.
   * @return An EJBObject that references the remote inter
   * face for the specified EJB runtime instance.  It is safe
   * to down cast this reference to a specific remote inter
   * face type.
   * @throws InvocationTargetException or IllegalAccessExcep-
   * tion if the method has trouble invoking the create
   * method.  See the java.lang.reflect.Method class for
   * details on these exceptions.  Also throws javax.naming.*
   * exceptions if there is a lookup failure of the home
   * interface of the specified EJB.   A RemoteException is
   * thrown per standard EJB rules.  Throw a generic excep-
   * tion so that any subclasses (e.g. wrappers) can  throw
   * their own exceptions.
   */
   public EJBObject createBean( String jndiName, Object
                                 []params)
      throws InvocationTargetException,
      IllegalAccessException,
      NoSuchMethodException,
      java.rmi.RemoteException,
      javax.naming.NamingException,
      javax.naming.NameNotFoundException,
    Exception {

    EJBObject bean = null;

    // Get the home interface and its associated Class
    // object.  We need the Class object for reflection...
    //
    Object homeInterface = getHomeInterface(jndiName);
    Class beanClass = homeInterface.getClass();

    // Need a class array for the method lookup...
    //
    Class[] signature = getSignature(params);

    // Look up the method--throw exception if method not
    // there.
    // All create methods in the home interface are named
    // "create" so search for the create method with the
    // appropriate  signature
    //
    Method createMethod = beanClass.getDeclaredMethod("cre-
     ate",  signature);

    // This is the key.  Here homeInterface is a reference
    // to a REAL remote object -- i.e. the skeleton class
    // on the server.  The invocation is done on this spe-
    // cific instance of the remote object created by the
    // EJB container on the server.
    //
    bean = (EJBObject) createMethod.invoke(homeInterface,
            params);

    return bean;
}
```

```
    /** Method to construct an array of Class objects repre-
    * senting a method signature
    *
    * @param parameters  A vector whose elements will be
    * reflected or their specified Classes.
    * @return A Class array
    *
    */
    private Class[] getSignature(Object [] parameters) {
        Class sig[] = new Class[parameters.length];

        for(int i =0; i<sig.length; i++) {
            sig[i] = parameters[i].getClass();
        }
        return sig;
    }

    /** Method to set the internal JDNIFinder variable */
    private void setJNDIFinder(JNDIFinder finder) {
        jndiFinder = finder;
    }
}
```

Listing 2

```
package com.myapp.util;

import java.util.*;
import javax.naming.*;

/*
* Class to wrap the EJBFactory to handle the number of excep-
tions
* that the factory throws.  By wrapping the factory, it is
```

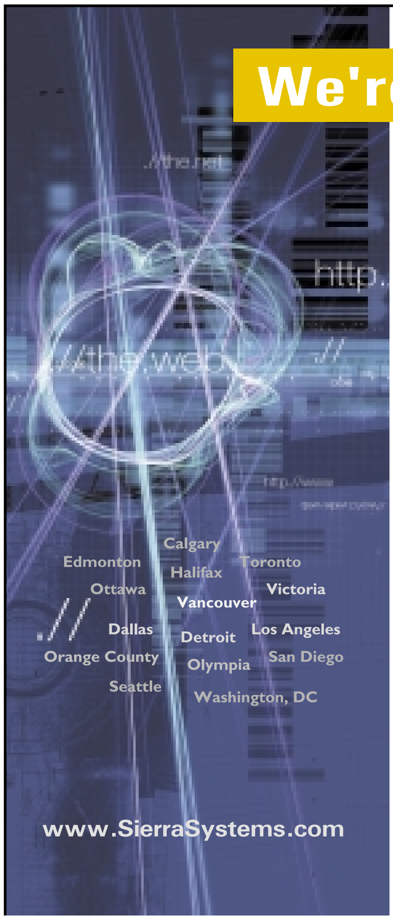
```
easier to use
* its functionality in application code.
*/
public class FactoryWrapper extends EJBFactory {

/* Method to create an enterprise bean. */
public EJBObject createBean( String jndiName, Object []params)
    throws  AppException
    {
        EJBObject bean = null;

        try {
            bean = super.createBean(jndiName, params);

        } catch (InvocationTargetException e) {
            throw new AppException(e.getMessage());
        } catch (NoSuchMethodException e) {
            throw new AppException(e.getMessage());
        } catch (IllegalAccessException e) {
            throw new AppException(e.getMessage());
        } catch (javax.naming.NameNotFoundException e) {
            throw new AppException(e.getMessage());
        } catch (javax.naming.NamingException e) {
            throw new AppException(e.getMessage());
        } catch (java.rmi.RemoteException e) {
            throw new AppException(e.getMessage());
        } catch (Exception e) {
            throw new AppException(e.getMessage());
        }

        return bean;
        }
}
```

Listing 3

```
/* Code example to create a bean using a create method
 * that has a signature of create(String).  This example
 * uses the FactoryWrapper class from Code Example 2.
 *
 * The JNDIFinder class mentioned is a simple wrapper
 * around a JNDI context.  The code is not given here
 * for the sake of brevity.
 */

 ...
 try {
    FactoryWrapper factory = new FactoryWrapper();

    // The finder variable is created outside of this scope
    //
    factory.setJNDIFinder(finder);

    String username = "Foo";
    String jndiName = "com/foo/entity/myAppBean";
    Object [] createArgs = { username };

    myAppBean bean = (myAppBean) factory.createBean(jndiName,
                         createArgs);
    bean.doWhatever();

    } catch (ApplicationException e) {
    // ... Error handling code ...
    //
    }
...
```

Download the Code!

The code listing for this article can also be located at www.JavaDevelopersJournal.com



We're looking for technical architects

You've been in the technology business long enough to know

it's not just about technology.

It's about **you** – people who are **passionate** about their work and where it can take them.

And we'd like to hear from you.

For over 30 years, Sierra Systems' people have been providing **leading** technology **solutions** to a worldwide clientele. We use leading edge technologies for **eBusiness** and **Web-based** projects.

We need **6 Technical Architects** to join our delivery teams and contribute to technology leadership across the **U.S. and Canada**. Based in one of our offices, you will provide mentoring and **leadership** locally, and also contribute on the **bigger stage**, likely involving some travel.

As a Sierra Systems Technical Architect, you will **develop** the overall structure of a system while considering **business and technology** constraints. You will **establish** the procedural **framework** for developing the architecture and building the system.

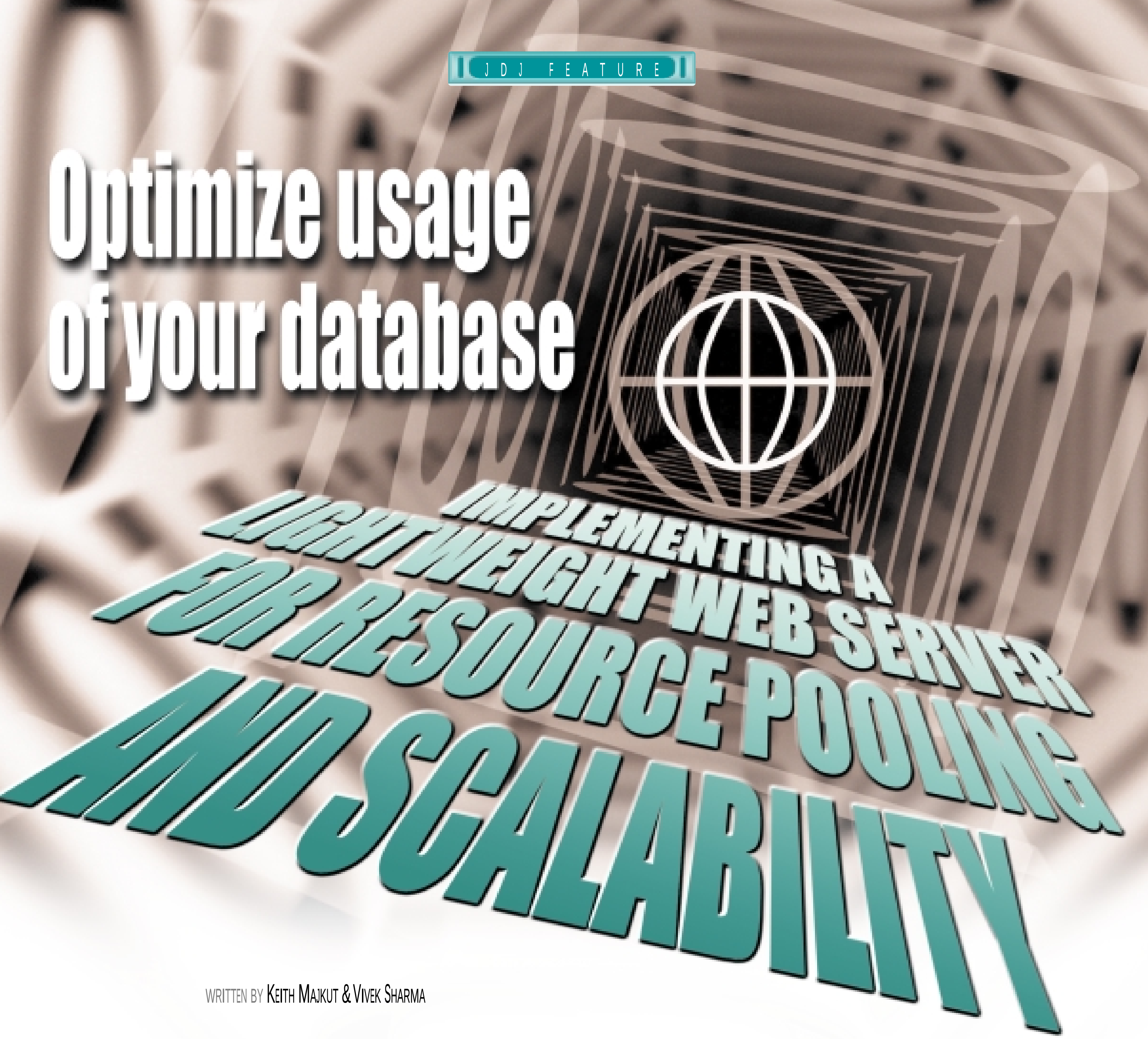
We are looking for a minimum 7 to 10 years' broad development **experience** with a demonstrated understanding of component development methodologies, **Java 2EE or Windows DNA** application architectures, data architecture and systems **security**.

We can offer you a technical **career path** and **recognition** for your efforts. **Opportunity** for equity. **Motivating** projects. **Exceptional** education, health and other **benefits**. Our **success** depends on how well we recognize and **reward your technical excellence**.

Sierra Systems – Technology based. People driven.

Sierra
New thinking.

Send your resume to
SheilaPage@SierraSystems.com



Optimize usage of your database

Implementing a Lightweight Web Server for Resource Pooling and Scalability

WRITTEN BY KEITH MAJKUT & VIVEK SHARMA

On the Web it's about three things – speed, reliability and scalability. Does your Web site respond quickly? Does your Web site always respond quickly? Does your Web site always respond quickly when it's being used by tens or hundreds of thousands of users?

Hardware is part of the answer, but software may arguably be more of the answer. Poor use of limited resources can affect performance as much, or more, as too little hardware.

Over time your Web system may have become a combination of everything from static files to database applications, from CGI to servlets, from C to Perl to Java and on and on and on. At some point it's good to step back and examine each of these pieces and the resources they use.

Even if you do, you may only tweak each component independent of the rest. You may think the CGI and the servlet don't affect each other, or the database applications don't affect (the serving of) the static files, but they do. Just like any large piece of software, every piece of a Web system affects every other. It would be good to step back and examine each piece in regard to the system as a whole. Examining what system resources each piece is using can prove very beneficial.

Most Web applications depend on a database back end. A database is a common resource shared by different applications. An examination of database resource usage should answer a number of questions. How many connections are needed? How many SQL statements are being executed? And how many round-trips (from the Web server to the database) are being made? The answer, probably, is too many.

Just because a large system can support a huge amount of resource usage doesn't mean that having one will improve speed, reliability or scalability. Quite the opposite could be true – it may be causing undue bottlenecks. This is exactly what we found while developing large Web systems for Oracle. A resource bottleneck is one of the main reasons we implemented a lightweight Web server.

In this article we discuss our implementation of a lightweight server that pools database resources for disparate pieces of a Web system. We describe our initial system problems as well as the first and second (improved) versions of the server. After that we encourage you to examine your system and its limited resources to see if you need a server like this to drastically improve performance.

The Initial System

We begin with a brief description of our Web system and some of the applications that are part of the architecture. As with many Web systems, ours includes a membership database. After login, members are authorized to access (static) documents in the file system. The Apache Web server is used to serve static pages, and an Apache module, written in C, controls access. The Apache authentication module receives the username and password, queries the database and (dis)allows access as necessary. The module makes multiple database queries to extract different pieces of authorization information from different tables, which causes multiple round-trips to the database.

We also have a Java servlet (the level servlet) that's used to set membership levels. After payment is received (via another system), the servlet receives the username and updates the database. The servlet executes only one database update and causes one round-trip.

Finally another Java servlet receives a username and password, queries the database, sets cookies and (dis)allows access as necessary. The module makes multiple database queries and causes multiple round-trips. This servlet performs the same duties as the Apache module, but is used for application login, not file (system) protection.

Each of these three components makes one or more connections to the database. The authentication module is written in C and loaded with the Apache process. The level and authentication servlets are written in Java and basically operate stand-alone. The authentication module and the authentication servlet execute the same SQL statements; the level servlet executes a different statement. These three components are a tangled web of software, whose pieces only partially overlap (see Figure 1).

The Resulting Problem

The resource problems did not occur all at once, nor were they initially obvious. This was because each module was written independently of each other and implemented over a long period of time. As system usage grew, the fact that each component made an unrestrained number of database connections and round-trips caused the Web system in general to slow down. This is because opening and closing connections is a relatively expensive process. At peak times the system was being overloaded by connections and round-trips.

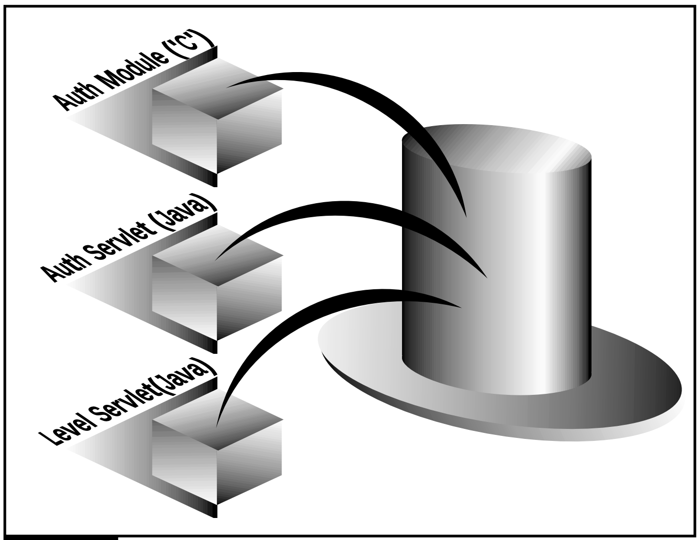


FIGURE 1 The problem

The first thought was to maintain connection pools in each of the components. This would definitely save the time that was spent opening connections. However, determining the optimal number of connections for each component was difficult as each component had different usage patterns (peak times). The authorization module is used constantly (with peaks); the authorization servlet has regular peaks and valleys; and the level servlet is used sparingly. Maintaining a fixed number of connections in each component would solve part of the problem, but not all of it. To illustrate this, let's say the authorization servlet needs to maintain 50 connections for peak usage, which occurs at 6 a.m. Pacific time. And let's say the authorization module needs 70 at 5 p.m. If each of them maintained their own connection pool, we'd have to establish 120 (70 + 50) connections throughout the day. However, realistically, we don't need that many connections around all the time (after all, established connections themselves are a drain on the system). It would be better if we could keep 90 (or some number greater than 70) open connections to serve peak usage for both the servlet and the module.

And the Solution

A centralized connection pool was needed. Each component could then retrieve connections from the pool, execute SQL and return connections to the pool. This immediately suggested an RMI- or CORBA-based system. The first problem is that Connection objects can't be transmitted over an RMI system.

This meant the core logic that performs SQL operations had to be coupled with the connection pool inside a server. This server can accept basic information, such as usernames, passwords and some action indicator, and perform tasks for different client applications. This again seems like a task that can be accomplished by an RMI/CORBA server. However, we didn't use RMI or CORBA due to a number of reasons:

1. Distributed systems using RMI/CORBA are beneficial when objects need to be exchanged between applications. However, in our case several components just needed to transfer string data, so the overhead of RMI/CORBA was not justified.
2. With RMI and CORBA you need to run special compilers to generate skeleton and stub files, then you must distribute them. This maintenance overhead was not desirable.
3. The final disadvantage, unique to RMI, is that it can only work with components written in Java. In our Web system that would mean we can combine only two out of three components. The Apache authorization module would be excluded and it was the largest resource user.

Although the investigation into RMI and CORBA didn't provide a solution, it did help us conceptually. It reintroduced us to one of the underlying mechanisms – sockets. Both Java and C components can communicate over sockets.

What we needed now was a lightweight server that could maintain a database connection pool and efficiently execute SQL statements on behalf of multiple applications (see Figure 2). The usage of sockets makes it similar to a Web server because it can reside on any machine in the system.

A Simple (Lightweight) Protocol

To make this work, we needed to define a protocol for communication between the server and the different applications. The two things this protocol had to accomplish were:

1. The server should be able to distinguish which client application is connected to it so it can execute the corresponding method that services the request.

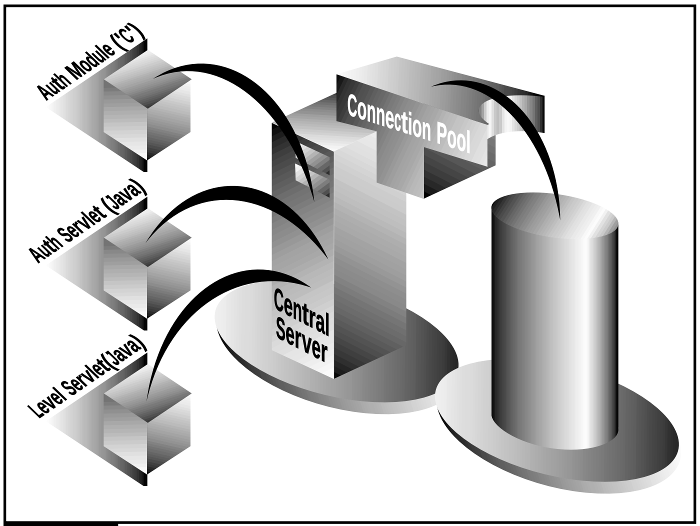


FIGURE 2 The solution

2. The server and the client should be able to exchange multiple pieces of data during one request.

The first exchange would be from the application to the server. It would send a string, which would contain a unique value that identifies the client component. Optionally, the application would also send any data required by the server to process the request. The Authentication application would send a username/password that it wants validated and the level servlet would send a username/password/level of a user for update.

Since multiple pieces of data need to be sent in one string, a predefined separator is required. This separator is a unique string of characters that doesn't normally exist in the kind of data exchanged between the client and the server. So the client-to-server portion of the protocol is:

```
[clientIdentifier][separator][data1][separator][data2][separa-
tor].........
```

After receiving the information the server would first identify which client application is connected to it. This information would be present in the first data element of the string sent by the client. Based on this, the server would invoke a method that serves requests for this client application. This method would process the remaining data and prepare a result string. For instance, the method that handles authentication requests would parse the input string, retrieve the username/password, validate it against the database and prepare a result string that contains a Success or Failure code. The level servlet might pass only a username/password and a new level. This protocol would be like:

```
          [data1][separator][data2][separator].......
```

Here's an example of this communication:

The authentication module would establish a connection with the server. It would then write the following string on the data stream:

```
AuthModule[SEP]username1[SEP]password1
```

Here [SEP] is the separator string we want to use in the communication. The server would read this string and parse it. The first element in the string would indicate that the request came from the authorization module. From this the server would figure out that there should be a username and password in the string. It would then validate this username/password combination and send back a result such as:

```
username1[SEP]authorized
```

System Overview

Several pieces are required to build such a system. First we need a server that can listen on a port for requests from client applications. The server then needs to spawn a thread that can handle each request. The thread should be able to read the information sent by the client application, parse it and invoke the appropriate method. The method that it invokes would be specific to the client application.

Also, we need a connection pool that's maintained by the server. Each method that serves client applications should be able to request a connection from this pool. It should also be able to return the connection back to the pool so other methods could use it. A description of the classes follows.

The Connection Pool

TheConnPool

This is a class for pooling database connections. It contains a vector that can store instances of the ConnectionObject class shown in Listing 1. ConnectionObject is an abstraction around a JDBC connection. Each ConnectionObject instance can contain a JDBC Connection object. It can also contain other connection-related information such as whether it's valid and/or open.

TheConnPool class itself is shown in Listing 2. It contains a method, createConnectionObject(), that creates and returns a ConnectionObject.

The constructor of TheConnPool class creates a number of ConnectionObjects and stores them in the vector. This class also contains a variable called *totalConnections* that keeps count of the number of ConnectionObjects created.

TheConnPool contains a getConnection() and a putConnection() method in it. Whenever a method of the server needs a database connection, it calls the getConnection() method. This method removes the first object (ConnectionObject) in the vector and returns it. When the server method no longer needs the connection, it can return the ConnectionObject back to the pool using the putConnection() method.

Note that both getConnection() and putConnection() are synchronized methods. This ensures that no two methods of the server can be putting/getting connections from the pool simultaneously, saving them from possible memory corruption.

Lightweight (Java) Server

Listing 3 shows the implementation of our lightweight server. LightweightServer first creates a TheConnPool object, thereby causing MAX_CONNECTIONS number of connections to be established with the database.

It then creates an instance of the ServerSocket class (in java.net) that listens for client connections on port 10101. (Of course, this can be easily changed so that all parameters like port, max connections, etc., are read from a configuration file.)

The ServerSocket class has an accept() method in it. This method causes the server to wait till a client tries to establish a connection to the machine/port on which the server is running. Since we want to serve all client requests, we start "accept()"ing calls inside a while(true) loop. Also, we don't want to make clients wait until all previous clients' requests have been fully serviced. So we create a thread for handling each client's request and start it. Once the thread has been started, control will come back to the accept() method so the server can process other client requests waiting in the queue. Meanwhile, the thread can service the client for which it was started.

WorkThread class is the heart of our server – it reads data sent by the calling application, processes it and returns results. This thread figures out the client application that called it based on the first element defined in our protocol. It then calls the appropriate method to handle that client's request. TheConnPool object created in LightweightServer is passed to each thread. Note that the same instance of this class is being passed to each thread. By doing so we're ensuring that there's synchronization among threads that try to get/put connections in the pool object.

Client Connections

We've now seen how to develop a server that can use a common database connection pool for serving requests from one or more client applications. Next we'll see how applications written in C (like our authentication module) and those written in Java (like our level servlet) can exchange information with this server. Listing 4 shows a Java snippet for connecting to the server. Listing 5 shows a C program for doing the same. In both cases we assume the server is running on port 10101 on the machine "dummy.server.com".

Optimization

Monitoring the Size of the Pool

We now have an architecture in place that can provide centralized processing and database resource pooling for multiple applications. The next step is to optimize the usage of this pool. This can be achieved with a monitor thread that periodically checks the number of connections in the pool and compares it with the number actually being used by one of the server threads. The monitor thread can then add/remove connections from the pool depending on usage. In times of heavy usage, the monitor can create new ConnectionObjects and add them to the pool. At low traffic times it can remove some ConnectionObjects to free up database resources.

Also, we need to maintain the concept of minimum and maximum connections. When the server starts up it should bring up a few connections. As usage grows, the number of connections in the pool can increase. However, we want some restrictions to prevent a process from hogging the database. This can be done by restricting the number of connections to a prespecified hard limit.

To do this we need to add a couple of methods to TheConnPool class: addConnectionObjects() and removeConnectionObjects(). Also, the constructor of TheConnPool needs to start the monitoring thread. See Listing 6 for an outline of these changes.

For the sake of illustration let's assume that the site using this monitor expects traffic that varies from very low, to medium, to very high. In the first case we'd want no more than five connections. In the second we'd like to keep open connections at 10. Third, no matter how high the traffic goes, we don't want connections to go beyond 20. So, we have a SOFT_LIMIT of five, MID_LIMIT of 10 and a HARD_LIMIT of 20.

The MonitorThread class shown in Listing 7 attempts to optimize the number of open connections. It wakes up every minute and checks the total number of connections and compares it with the number of connections being used. If too many are being used, it reduces the number to MID_LIMIT or SOFT_LIMIT, whichever applies. Similarly, it increases the connections if it figures that more are required due to high traffic. *Note:* You can add a further level of granularity by adding limits between the SOFT, MID and HARD limits.

Logging Connection Usage Information

The next big question is how to determine good values for SOFT_LIMIT, MID_LIMIT and HARD_LIMIT. One easy way is by adding some code to the MonitorThread so it writes to a log file whenever it increases or decreases connections in the pool. You can put in time stamps that tell you how often the thread is activated. If the thread is activated too often, then the selected numbers aren't good. Another way is to write a LogThread along the same lines of the MonitorThread. This thread could also wake up every few minutes and find out the total number of connections and the connections being used. It could then write this information along with a time stamp to a log file.

If the log file is written in a Comma Separated Value (CSV) format, you can use something like Excel to see traffic patterns on your site graphically. So, if total connections are equal to the connections being used for most of the day, then you might consider raising your HARD_LIMIT. On the other hand, if the number of connections being used never reaches the total connections, then you can reduce the HARD_LIMIT number and save some database resources.

Other Improvements

Stored Procedures

We save time by preopening connections to the database in TheConnPool class. Making round-trips to the database is another expensive operation. If you're executing a large number of SQL statements, each causing a round-trip to the database, you'll see performance degradation. You should try to reduce these round-trips. One way is by creating stored procedures. Instead of having a number of JDBC calls executing one SQL statement at a time in your logic, you could bunch the statements together and create a stored procedure in the database.

Different vendors provide different ways to do this. With an Oracle database you can create a stored procedure using PL/SQL. This way you'd make just one call to the database over the connection, greatly improving your overall performance.

Statement Pooling

The ConnectionObject above is a wrapper class that contains a JDBC Connection object. We're not storing Connection objects directly in the vector that's maintained by TheConnPool because we want to store information about the connection, such as whether it should be discarded. Another piece that can be stored here is a set of precreated Statement objects. This is beneficial for statements that need to be executed repeatedly, since creating a PreparedStatement and keeping it open saves time.

Since we know which statements the server threads can execute, we can precreate the corresponding Statement objects. This can be done with minimal changes. The createConnectionObject() method in TheConnPool class can be enhanced so that after creating a connection, it creates a CallableStatement (for stored procedures) or a PreparedStatement object, gives it a name and stores it in the stmts hashtable. A method in the server thread can then retrieve a connection and the actual statement it needs to execute.

In Listing 8 we've slightly modified the ConnectionObject class by adding a hashtable called *stmts.*

We modify TheConnPool class so it creates the required statements and stores them in the ConnectionObject. In Listing 9 we're creating one CallableStatement and storing it with the name LEVEL_STMT, and creating a PreparedStatement and assigning AUTH_STMT as its name. Before we had statement pooling, the method that served authentication requests would extract a connection out of the ConnectionObject and create a statement from it. Now it can simply request the required Statement object and execute it. Listing 10 illustrates how the new method would look.

Making the Server More Generic (Version 2)

The lightweight server discussed above is fairly generic. You can easily add new methods to the WorkThread class and make it capable of handling requests from new applications. However, the drawback is that for every new application you'd have to recompile (and restart the server). In addition, the server code would start looking clumsy and become difficult to maintain as new applications are added to the system. To overcome this we decided to decouple the request handling functionality from the WorkThread class. So, instead of having the functionality inside methods defined in WorkThread, they would be contained in their own classes. By slightly changing our protocol, we can invoke handlers for any class without ever having to recompile the lightweight server.

To do this we defined an interface called *ServerWorker*. This interface has a method called *execute* that returns a string. The input parameters of this method are a ConnectionObject and the string that's communicated from the client application to the server (containing the username, password, etc.). The ServerWorker is shown below.

```
public interface ServerWorker
{
    public String execute(ConnectionObject connObj, String inp);
}
```

Every client handler needs to reside in a class that implements this interface. An example implementation is shown in Listing 11.

To make the server independent of client applications we need to make two changes:

1. The protocol needs to be changed. Instead of passing a client identifier to the server exchange as the first element in the client, it should pass the name of a class that can handle the application's requests.
2. After examining the first element in the protocol, the server should instantiate an object of the class specified in the exchange instead of calling a method in WorkThread.

As long as this class implements the ServerWorker interface, the server can invoke its execute method and have it process the client request. The benefit is that now new applications can be added to the system without changing the server. The changed portions of the server are shown in Listing 12.

To use this server, the following string is sent from the client to the server:

```
AuthWorker[SEP]username1[SEP]password1
```

Now look at Listing 12 to see how our server would respond to such a request. First it'll parse this string and determine that the class that handles requests for this client is named AuthWorker. It will use the forName method of class to load the class of this name (AuthWorker). It will then create an instance of this class using the newInstance() method of Class. newInstance() returns an object. However, our server deals only with handler classes that implement the ServerWorker interface. We can cast the object that results from newInstance() to a ServerWorker object. Since each class that implements ServerWorker needs to have an execute() method in it, our server will simply invoke this method and pass it a ConnectionObject and the string it received as input from the client. Now the execute() method in the AuthWorker class can parse the input string, figure out the username/password, use the ConnectionObject to authorize the user and send results back to the server.

Summary

Speed is a crucial factor in the success of any Web site. One of the common factors that impedes speed is improper utilization of resources. And database connections are a scarce and expensive resource that should be used very carefully. In this article we've looked at a centralized server that can help optimize usage of your database. This server can maintain a pool of connections that can be shared by any application.

This concept isn't limited to database connections; ideally, any resource that can affect performance should be moved into an architecture like this. By doing proper logging and monitoring you can figure out an optimal configuration and improve the response time of your applications.

Author Bios

Keith Majkut, a software development manager at Oracle Corporation, leads a team responsible for Web infrastructure projects. He's been at Oracle for over 10 years and has followed the technology from DOS to Web and from C to Java.

Vivek Sharma is a software developer at Oracle Corporation. He has over seven years of industry experience. His areas of experience and interest include Web-based research and development. He also coauthored the book Developing e-Commerce Sites: An Integrated Approach *Addison-Wesley.*

kmajkut@us.oracle.com

vivek_sharma_99@yahoo.com

Listing 1

```
public class ConnectionObject
{
    Connection conn;
    boolean connectionOk;
    ConnectionObject(Connection c)
    {
          conn = c;
    }
    public Connection getConnection()
    {
         return conn;
    }
}
```

Listing 2

```
public class TheConnPool
{
  int totalConnections = 0;
  Vector pool;
  TheConnPool(int max)
  {
    pool = new Vector();
    for(int i=0; i<max; i++)
    {
      ConnectionObject connObj = createConnectionObject();
                            pool.addElement(connObj);
      // Add connection to pool
                            totalConnections++;
                  }
           }
           public synchronized Connection getConnection()
           {
                    if(pool.size() == 0)  /* No more available
                   connections in pool */
                          return null;
                    ConnectionObject connObj =
                       (ConnectionObject)pool.elementAt(0);
                    pool.removeElementAt(0);
                    return conn;
           }
           public synchronized void putConnection(
        ConnectionObject connObj)
           {
                      pool.addElement(connObj);
           }
           public ConnectionObject createConnectionObject()
           {
                Connection c = .... /* Create a JDBC connection
                                      using the DriverManager
                                      class as you normally do */
   return new ConnectionObject(c);
  }
}
```

Listing 3

```
public class LightweightServer
{
          public static void main(String[] args)
          {
                      ........
                  TheConnPool connPool =
     new TheConnPool(MAX_CONNECTIONS);
                   ServerSocket serverSocket =
     new ServerSocket(10101);
                   while( true )
                   {
                         Socket clientSocket =
        serverSocket.accept();
                         WorkThread wt =
        new WorkThread(clientSocket, connPool);
                         wt.start();
                   }
                        .........
            }
}
class WorkThread extends Thread
{
                Socket clientSocket = null;
                TheConnPool connPool;
                WorkThread(Socket cs, TheConnPool cp)
                {
                        clientSocket = cs;
                        connPool = cp;
                }
                public synchronized void run()
                {
                      // Get connection object from pool
                      ConnectionObject connObj =
                         connPool.getConnectionObject();
                      // Establish input/output streams for
                      // communication with client
                      InputStream is = new
                         BufferedInputStream(
        clientSocket.getInputStream());
                      java.io.PrintStream ps = new
                        java.io.PrintStream(
     clientSocket.getOutputStream());

                      // Read data sent by client
                      byte[] buffer = new byte[1024];
                      is.read(buffer, 0, 1024);
                      String inputString = new
                                            String(buffer);

                   ...//Parse the input string and examine
                      //the first element
                      // According to the protocol this
                      // element should tell us which
                      // application has made the request

                      if(firstElement.equals("AUTH"))
                   /*    Called by the Apache or the Servlet
                         authentication client */
                         processedResult =
                         executeAuthenticate(connObj,input-
                         String);
                      else
                      if(firstElement.equals("LEVEL"))
                         processedResult =
                         executeLevelUpdate(connObj,input-
                         String);
                        // Called by the level servlet
            // More else ifs for  handling other client types

connPool.putConnection(conn); // Return connection to pool

                        // Send result to the client
                       ps.print(processedResult);
                }
               private synchronized String executeAuthenticate(
                         ConnectionObject connObj, String i)
               {
// Extract Connection from connObj and create required Statement
                      Connection conn = connObj.getConnection();

                        /* Extract username/password
                           Verify this against database
                           using username/password and
                           Connection c Prepare result
                           string and send it back
                                */
               }
               private synchronized String executeLevelUpdate(
                         Connectiononnobj connObj, String i)
               {
                                // Extract Connection from
                       connObj and create required Statement
                      Connection conn = connObj.getConnection();
                             /* Extract username and level
                                Connect to database and
                             set new level for this member
                           Send a Success  or Failure code
   */
               }
}
```

Listing 4

```
String HOST = "dummy.server.com";
                int PORT = 10101;
                String sep = "[SEP]";

                Socket clientSocket = new Socket(HOST, PORT);
                os = new PrintWriter(
                                      new OutputStreamWriter(
                                        clientSocket.getOutputStream()));
                          is = new BufferedReader(
                                       new InputStreamReader(
                                        clientSocket.getInputStream()));
                          String sendVal = "LEVEL" + sep + username +
                                          sep + level;
                          os.println(sendVal);
                          os.flush();
                          String recVal = is.readLine();
                          os.close();
                          is.close();
                          clientSocket.close();
```

Listing 5

```
#include <arpa/inet.h>
#include <sys/socket.h>

 main()
 {
        struct hostent *host;
        unsigned long address;
        unsigned short port = 10101;
        char *hostName = "dummy.server.com";

        int sock;

        char buffer[60];
        int bufferlen;

        char reslt[2048];
        int RESLT_LEN = 2047;
        struct sockaddr_in addr;

        /* Lookup the host, create a socket and establish
connection on port 10101 */
    host = gethostbyname(hostName);
    address = ((struct in_addr *)host->h_addr_list[0])-
>s_addr;
    addr.sin_addr.s_addr = address;
    addr.sin_port = htons(port);
    addr.sin_family = AF_INET;
    sock=socket(AF_INET, SOCK_STREAM, 0);
    connect(sock, (struct sockaddr *)&addr, sizeof(struct
sockaddr_in));

        /* Put data to be sent to server in variable 'buffer'
*/
    strcpy(buffer, "AUTH[SEP]username[SEP]password";

        /* Write 'buffer' to the output stream */
    bufferlen=strlen(buffer);
    write(sock, buffer,bufferlen);

        /* Read result sent by the server  */
   read(sock, reslt, RESLT_LEN);
}
```

Listing 6

```
TheConnPool(int max)
   {
         ....
         MonitorThread mt = new MonitorThread(this);
         mt.start();
   }
   public synchrnonized void addConnectionObjects(int n)
   {
         for(int i=0; i<n; i++)
         {
           ConnectionObject connObj = createConnectionObject();
             pool.addElement(connObj); // Add connection to pool
              totalConnections++;
         }
   }
   public synchronized void removeConnectionObjects(int n)
   {
         for(int i=0; i<n; i++)
         {
             ConnectionObject connObj = pool.elementAt(0);
             try{
                  Connection conn = connObj.getConnection();
                   conn.close();
             }catch(Exception ex){}
```

```
                pool.removeElementAt(0);
                totalConnections--;
            }
    }
```

Listing 7

```
class MonitorThread extends Thread
{
    int SOFT_LIMIT=5;
    int MID_LIMIT=10;
    int HARD_LIMIT=20;
    int SLEEP_TIME = 60000; /* One minute */
    TheConnPool tc;
    MonitorThread(TheConnPool t)
    {
         tc = t;
    }
    public void run()
    {
       while(true)
       {
          int connsAvailable = tc.pool.size();
          int totalConns = tc.totalConnections;
          int connsBeingUsed = totalConns - connsAvailable;
          boolean increaseConns = false, reduceConns = false;
          int increaseBy = 0, reduceBy = 0;
          if(connsBeingUsed < SOFT_LIMIT && totalConns > SOFT_LIMIT)
          {
                reduceConns = true;
                reducyBy = totalConns - SOFT_LIMIT;
          }
          else
          if(connsBeingUsed < MID_LIMIT && totalConns > MID_LIMIT)
          {
                reduceConns = true;
                reducyBy = totalConns - MID_LIMIT;
          }
          else
          if(connsAvailable == 0 && totalConns < MID_LIMIT)
          {
                increaseConns = true;
                increaseBy = MID_LIMIT - totalConns;
          }
          else
          if(connsAvailable == 0 && totalConns < HARD_LIMIT)
          {
                increaseConns = true;
                increaseBy = HARD_LIMIT - totalConns;
          }
          if(reduceConns)
tc.removeConnectionObjects(reduceBy);
          else
          if(increaseConns)
               tc.increaseConnectionObjects(increaseBy);
          try{
                sleep(SLEEP_TIME);
          }catch(InterruptedException ie){}
      }
    }
}
```

Listing 8

```
public class ConnectionObject
{
    Connection conn;
    boolean connectionOk;
    Hashtable stmts;
    public setStatements(Hashtable h)
    {
         stmts = h;
    }
    public Statement getStatement(String nm)
    {
         return (Statement)stmts.get(nm);
    }
    ......
```

Listing 9

```
public class TheConnPool
{
           public ConnectionObject createConnectionObject()
           {
                    Connection c = .... /* Create a JDBC
connection using the

DriverManager class as you normally do */
                    ConnectionObject connObj = new Connec-
tionObject(c);
                    Hashtable stmts = new Hashtable();

                    PreparedStatement pstmt =
                                   c.prepareStatement(
  "UPDATE member_table " +
                                      " SET level = ?
WHERE username = ?");
                    stmts.put("LEVEL_STMT", pstmt);
                    CallableStatement cstmt =
                          c.prepareCall(
      "{call Auth_Api.authenticate(?,?,?,?)}");
                    cstmt.registerOutParameter(3, Types.VARCHAR);
                    cstmt.registerOutParameter(4, Types.VARCHAR);

                    stmts.put("LEVEL_STMT", pstmt);
                    stmts.put("AUTH_STMT", cstmt);

                    connObj.setStatements(stmts);
                    return connObj;
           }
```

Listing 10

```
               private synchronized String executeAuthenticate(
                                   ConnectionObject connObj,
   String i)
               {
                    // Extract required Statement connObj
CallableStatement cstmt =
(CallableStatement)connObj.getStatement(
"AUTH_STMT");
        // Execute the query using this pre-created statement
               }
```

Listing 11

```
public class AuthWorker implements ServerWorker
{
    public String execute(ConnectionObject connObj, String inp)
    {
           String result = "";
           .... // Parse the username/password
           try{
                 PreparedStatement pstmt =
connObj.getStatement("AUTH_STMT");
                 .....

           }catch(Exception ex){}
           return result;
    }
}
```

Listing 12

```
                public synchronized void run()
                {
                      // Get connection object from pool

                 ConnectionObject connObj =
                         connPool.getConnectionObject();
                      ....
                      // First element contains the name of
                      // the class that can handle the request.

                      Class c = Class.forName(firstElement);
// Create an instance of this class and invoke the execute method

                      ServerWorker sw =
(ServerWorker)c.newInstance();
                      String result = sw.execute(connObj,
inputString);
                      ....
```

PRODUCT REVIEW

eWave Studio and ServletExec

by Unify

REVIEWED BY JIM MILBERY

AUTHOR BIO
Jim Milbery is a software consultant with Kuromaku Partners LLC (www.kuromaku.com), based in Easton, Pennsylvania. He has over 16 years of experience in application development and relational databases.

jmilbery@kuromaku.com

JAVA DEVELOPER'S JOURNAL JDJ ★★★★★★★★ WORLD CLASS AWARD

Unify Corporation
100 Century Center Court
Suite 302
San Jose, CA 95112
Web: www.unify.com
Phone: 408 451-2000
Fax: 408 451-2007
E-mail: info@unify.com

Test Environment
Client/server: Dell 410 Precision, 18MB RAM, 14GB disk drive, Windows NT 4.0 (Service Pack 4)

Over the past year all the major database vendors and many of the classic client/server tools vendors have turned their attention to the application server market. The venerable database and tools vendor, Unify, is no exception. Unify has released a new version of its eWave Studio and eWave application servers into the fray. It considers itself an endorser of the J2EE platform, but is not yet an official licensee of the J2EE. I recently looked at this latest release with an eye on its Servlets and JSP capabilities.

Unify's Application Server Products

Unify offers four different products in their application server family: Unify eWave Studio, eWave ServletExec, eWave Commerce and eWave Engine. The eWave ServletExec product was obtained by Unify in its acquisition of New Atlanta Communications LLC in April of this year. eWave Studio and eWave ServletExec are designed for creating Servlets and JavaServer Pages applications. The eWave Engine and Commerce products are oriented toward the creation of EJB applications. Unify offers the eWave ServletExec as a separate product. In addition, it's bundled into the EJB application server, eWave Engine. Since the two server engines can be evaluated separately at this point, you can work with whichever engine best suits your purpose. If you're primarily interested in building Servlet and JSP applications, then eWave Studio and eWave ServletExec are all you need. Conversely, if you're looking to build EJB-based applications or applications that combine EJB technology with Servlets and JSPs, then eWave Engine is probably the better choice.

Installing and Configuring eWave Products

Both the eWave Engine and Studio can be downloaded as two installation kits with a combined size of over 120MB. This makes them unwieldy to download without a high-speed Internet connection.

eWave ServletExec comes in two flavors:

1. The "in-process" version works directly with a variety of popular Web servers (but not Apache).
2. The "out-of-process" version supports Apache and provides some additional flexibility for stopping and starting the server exclusive of the Web server, and for plugging in alternative JVMs. (The ServletExec downloads are just over 2MB in size and much quicker to download.)

To download a trial version of any Unify product you'll need a license key that unlocks the software for the duration of the trial period. Unify sends the license keys for each product via e-mail. Unify has packaged a lot of technology into its various servers, and each one has enough features to be a complete topic in its own right. I chose to focus my attention on the eWave Studio and eWave ServletExec combination for several reasons.

First, while Unify's eWave engine is positioned to compete with the larger enterprise players such as iPlanet, IBM, BEA and Oracle, Unify has not yet licensed the J2EE. Second, Servlets and JSPs are hot commodities at the moment, and customers are more likely to consider Servlet/JSP engines from vendors that have not yet fully committed to the J2EE. Furthermore, I liked the concept of the eWave Studio product and was excited about the chance to get my hands on it. After downloading the two product kits, I managed to get them installed after some initial fumbling. (I started off using the JDK 1.3 release, which caused some problems with Unify.) eWave Studio installs relatively quickly, but the ServletExec engine comes equipped with a lengthy installation and configuration guide. Integrating the engine with IIS, iPlanet WebServer or Apache is not a simple drag-and-drop process, but it's not rocket science either. As usual, I attempted to get everything installed without carefully reading through the directions. After running into the JDK 1.3 problem, I hopped onto the Unify news server and got all my configuration questions answered by scanning through some existing posts. All in all the installation went quite smoothly.

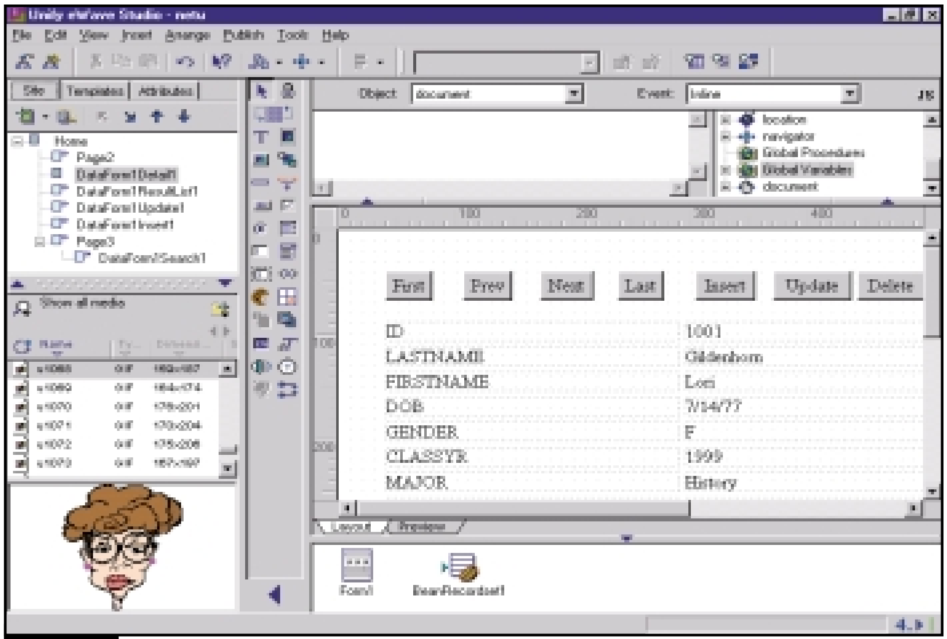
eWave Studio

JavaServer Pages are a relatively simple and elegant solution to building dynamic, data-based Web applications. This doesn't necessarily mean that JSPs are simple to work with, especially if your application is somewhat complicated. After all, a hammer is a relatively simple tool, but it would be overly complex to try and build a house using just a hammer. I like to think of eWave Studio as a sort of toolbox for JSPs and Servlets. There's a bit of a learning curve, just as there would be with any sophisticated toolbox. However, given the fact that eWave Studio is built on the Servlet/JSP foundation, any investment you make in learning eWave is also an investment in the core technology itself. The core interface of eWave Studio is shown in Figure 1.

To get to the main eWave panel you'll be channeled through a wizard interface that sets up the basics of your site for you. This generation wizard covers a lot of ground, and I'd expect novice users to be confused by some of the parameters. From my experience you can change most of the settings after you've completed the interface. The best advice I can give you is to accept the defaults when you first launch eWave Studio. The upper left-hand corner of the development interface is a tabbed panel that allows you to switch among three separate views. The site panel displays a hierarchical layout of the various pages that are available within your site. You're free to manually create pages, but eWave Studio generates lots of content for you, and both types of pages appear within this list. This panel is ideal for moving around the various parts of your application.

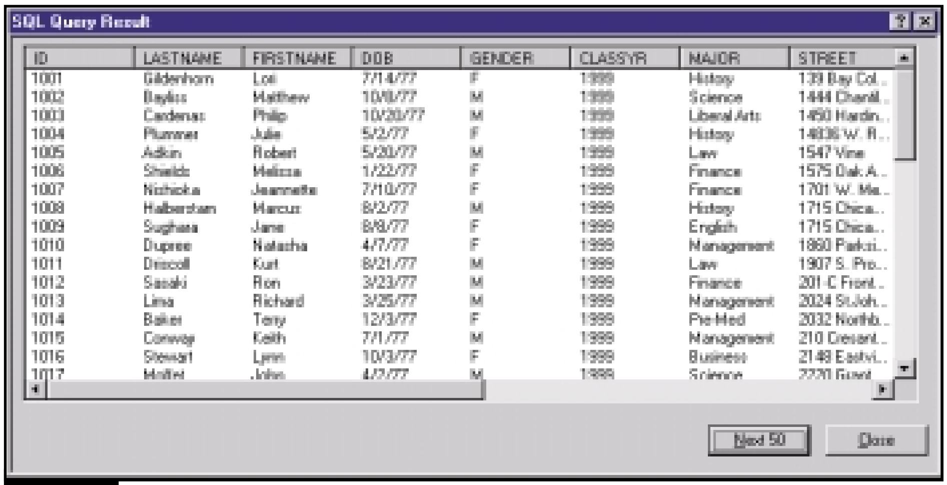
I generated a number of forms from my Oracle8*i* database tables and was then able to drag and drop the resulting objects into different page flows from the site panel. If you wish to provide standard formatting for the various pages in your application, you can apply templates using the templates tabbed panel. The display palette just below the site panel shows all the media elements that have been defined as part of your application, including images and sound files. I could quickly add new directories of media files to the panel and then browse through these images in the viewer as shown in Figure 1. Adding the class pictures for my familiar NetU database was easy and painless. I dragged one of the class pictures onto a page and eWave Studio successfully copied the graphic to my site when I published the page later on.

The large panel in the middle of the studio interface is the page builder. It's a simple process to create Web pages by dragging and dropping elements onto the page development area. By default the component palette has 20 different elements that can be dropped onto a page (including common HTML elements such as tables, text and images). However, you can also access an extensive element library of code that can be added to the palette. I was able to locate an SMTPMailer element and quickly add it to my sample page. The Studio interface uses a paneled design, but I found it easy to hide sections I wasn't using and to expand sections as needed. (This is especially helpful when you're designing a JSP page visually.)



ID	1001
LASTNAME	Gildenhorn
FIRSTNAME	Lori
DOB	7/14/77
GENDER	F
CLASSYR	1999
MAJOR	History

FIGURE 1 eWave Studio



ID	LASTNAME	FIRSTNAME	DOB	GENDER	CLASSYR	MAJOR	STREET
1001	Gildenhorn	Lori	7/14/77	F	1999	History	139 Bay Col...
1002	Bayliss	Matthew	10/8/77	M	1999	Science	1444 Chantil...
1003	Cardenas	Philip	10/20/77	M	1999	Liberal Arts	1450 Hardin...
1004	Plummer	Julie	5/2/77	F	1999	History	14836 W. R...
1005	Adkin	Robert	5/20/77	M	1999	Law	1547 Vine
1006	Shields	Melissa	1/22/77	F	1999	Finance	1575 Oak A...
1007	Nichicka	Jeannette	7/10/77	F	1999	Finance	1701 W. Ma...
1008	Halberstam	Marcus	8/2/77	M	1999	History	1715 Chica...
1009	Sughaea	Jane	8/8/77	F	1999	English	1715 Chica...
1010	Dupree	Natasha	4/7/77	F	1999	Management	1860 Parksi...
1011	Driscoll	Kurt	8/21/77	M	1999	Law	1907 S. Pro...
1012	Sasaki	Ron	3/23/77	M	1999	Finance	201-C Front...
1013	Lima	Richard	3/25/77	M	1999	Management	2024 St.Joh...
1014	Baker	Terry	12/3/77	F	1999	Pre-Med	2032 Northb...
1015	Conway	Keith	7/1/77	M	1999	Management	210 Cresent...
1016	Stewart	Lynn	10/3/77	F	1999	Business	2148 Eastvi...
1017	Moffet	John	4/2/77	M	1999	Science	2220 Guest

FIGURE 2 DataForm SQL query

Adding Database Access

The real power of Servlets and JSPs is the ability to interact with data from your database. This is where the real fun begins with eWave Studio. Unify provides a DataForm wizard that walks you through the process of creating JSP pages that interact with your data. The deployment version of eWave Studio relies on ODBC and Microsoft's Data Objects as the data source for the wizard, but the deployed version of the pages uses JDBC. The ServletExec engine provides familiar scalability features such as connection pooling, and you're free to make use of native JDBC 2.0 drivers as you see fit. Within the DataForm wizard you can define connections and create SQL queries – the output is displayed directly in the wizard as shown in Figure 2.

The DataForm wizard creates multiple pages from a single database query, including insert, update, delete, results lists and search panels. You can select fields on which to link detail pages, and each of the page types can display different fields from the query. The pages themselves can be edited in the layout panel (but the wizard doesn't support two-way editing). I was able to create pages for the UGRADS table (as shown in the layout panel in Figure 2) in no time at all. If you've installed the eWave ServletExec properly, you can switch from the layout to the preview view and see the results of your query interactively. This makes it easy to work with JSP pages without the overhead of having to completely deploy your application each time you want to test out a page.

SmartPages and Debugging

Unify provides two additional features with the eWave product line that impressed me. eWave's SmartPages lets you develop JSP pages that adapt to the browser at runtime. Unify packages JSP code within the page that adapts the output to Internet Explorer or Netscape Navigator. Although I didn't test this capability myself, it appears to work as advertised. This is a nice feature if you plan on deploying your applications to a wide audience with varying browsers and versions. Unify also provides a debugger version of the ServletExec engine, which makes it easier to solve problems before they're deployed into production.

Summary

Unify is a solid company with a good reputation, but it remains to be seen whether or not they can compete in the larger J2EE arena with eWave Engine. In the meantime, eWave Studio and ServletExec provide an excellent suite of tools for building Servlet and JSP applications and I'd encourage you to take a look at these tools for dynamic Web application development.

A PopupButton Component

A simple component that provides an intuitive interface to the popup menu's availability

WRITTEN BY
PAT PATERNOSTRO

Popup menus, the extremely functional components available to the Java developer, allow developers to provide menu capability without the inclusion of a full-blown menu system (i.e., MenuBar, Menus). From a user interface perspective, however, they're not intuitively accessible. The popup menu is usually triggered by pressing the right mouse button, but users may not be aware of its availability.

This article details a PopupButton component that ties the popup menu to a button component and allows users to display the menu via a click of the PopupButton.

Class Design

The PopupButton component is made up of one abstract class, PopupButton, located in PopupButton.java (see Listing 1). This class extends the java.awt.Button class and implements the java.awt.event.ActionListener event listener interface. Since the class is abstract, the java.awt.event.ActionListener interface method, actionPerformed(), doesn't need to be implemented; however, any nonabstract (concrete) direct descendant class is required to implement the method (more on this later).

Two overloaded constructors are provided for class construction, each taking a different number of arguments. The two-argument constructor simply calls the three-argument constructor via the this() method, which provides a convenient mechanism for one constructor to call another constructor, allowing you to localize construction code inside a single constructor.

The three-argument constructor requires three parameters:

- A **java.lang.String** reference that represents the button's label
- A **java.lang.String array** reference that represents the popup menu item labels
- A **java.awt.Container** reference that represents the popup menu's container

The constructor first calls a superclass constructor, passing in the button's label concatenated with the " " character (Alt-0164 on your keyboard). This arbitrarily chosen character (you can choose any nonalphanumeric character you like) acts as a visual clue to the user that the button will display a popup menu when pressed. (I decided to use a character versus a "down arrow" image file [JPEG or GIF] for the visual clue to minimize resource requirements. However, if you wish to use an image file, you'll need to add code to read the image in the constructor and override the PopupButton component's paint() method to draw the image.) Next, a popup menu is created and its reference is saved to a private instance variable. Storage is allocated for an array of java.awt.MenuItem components whose array size is based on the size of the String array constructor parameter. A for() loop constructs the menu items, adds an action listener to each menu item, then adds the menu items to the popup menu. Finally, the popup menu is added to the container, and an action listener is added to the PopupButton component. This action listener, implemented via an anonymous inner class, is needed in order to respond to button clicks for the purpose of displaying the popup menu. It has no relation to the menu items' action listener.

Implementation

To use the class, extend it and provide a constructor that calls one of the superclass constructors. Since the abstract superclass doesn't provide an implementation for the java.awt.event.ActionListener interface method, actionPerformed(), a nonabstract direct descendant class, is "contractually" obligated to provide method implementations for any interfaces the abstract superclass implements. In the case of the PopupButton component this is desired, as the response to popup menu item selections should be provided in the descendant class.

I've provided a sample application (see Figure 1) that uses the PopupButton component. The sample application is made up of three classes located in PopupButtonTest.java (see Listing 2):

- PopupButtonTest
- PopupButtonTestFrame
- MyPopupButton

The PopupButtonTest class simply contains a main() method and instantiates the PopupButtonTestFrame class. The PopupButtonTestFrame class extends the java.awt.Frame class and is the container for my implementation of the PopupButton class – MyPopupButton.

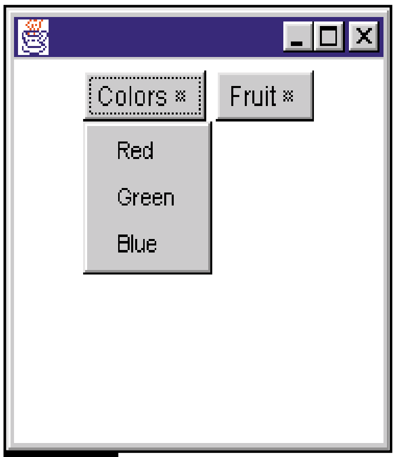


FIGURE 1 Sample PopupButton component

The MyPopupButton class contains a three-argument constructor that calls the superclass three-argument constructor and provides an implementation for the java.awt.event.ActionListener interface method actionPerformed(). If the actionPerformed() method is left out of the class definition, the compiler will generate the following error: class MyPopupButton must be declared abstract. It doesn't define void actionPerformed(java.awt.event.ActionEvent) from class PopupButton. The actionPerformed() method retrieves the "action command" (via the java.awt.event.ActionEvent class's getActionCommand() method) associated with the component that triggered the event. By default, the "action command" is the label of either a java.awt.Button or a java.awt.MenuItem component. You can use this label to perform a string comparison against the java.awt.MenuItem labels passed in the constructor to determine your course of action. For demonstration purposes I simply display the label in a console message when the PopupButton menu item is selected.

Summary

The PopupButton component is a simple yet functional component that provides the user with an intuitive interface to the popup menu's availability. This aids greatly in the usability of any application that provides popup menus.

AUTHOR BIO

Pat Paternostro is an associate partner with Tri-Com Consulting Group, Rocky Hill, Connecticut, which provides programming services for a wide variety of development tasks.

ppaternostro@tricomgroup.com

Listing 1

```
import java.awt.*;
import java.awt.event.*;

public abstract class PopupButton extends Button implements ActionLis-
tener
{
 private PopupMenu popup;

 protected PopupButton(String[] items, Container parent)
 {
  this("",items,parent);
 }

 protected PopupButton(String label, String[] items, Container parent)
 {
  super(label + "   ");

  popup = new PopupMenu();

  MenuItem menuItems[] = new MenuItem[items.length];

  for(int i = 0; i < items.length; i++)
  {
   menuItems[i] = new MenuItem(items[i]);
   menuItems[i].addActionListener(this);
   popup.add(menuItems[i]);
  }

  parent.add(popup);

  addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {

popup.show(PopupButton.this,0,PopupButton.this.getSize().height);}});
 }
}
```

Listing 2

```
import java.awt.event.*;
import java.awt.*;

public class PopupButtonTest {
 public static void main(String args[]) {
  new PopupButtonTestFrame();
 }
}

class PopupButtonTestFrame extends Frame {
 Panel panel = new Panel();
 MyPopupButton mpb1 = new MyPopupButton("Colors",new
String[]{"Red","Green","Blue"},this);
 MyPopupButton mpb2 = new MyPopupButton("Fruit",new
String[]{"Apples","-","Oranges","-","Bannanas"},this);

 PopupButtonTestFrame() {
  super();

  /* Add the window listener */
  addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent evt) {
    dispose(); System.exit(0);}});

  /* Size the frame */
  setSize(200,200);

  /* Center the frame */
  Dimension screenDim = Toolkit.getDefaultToolkit().getScreenSize();
  Rectangle frameDim = getBounds();
  setLocation((screenDim.width - frameDim.width) / 2,(screenDim.height
- frameDim.height) / 2);

  panel.add(mpb1);
  panel.add(mpb2);
  add(BorderLayout.NORTH,panel);

  /* Show the frame */
  setVisible(true);
 }
}

class MyPopupButton extends PopupButton
{
 public MyPopupButton(String label, String[] items, Container parent)
 {
  super(label,items,parent);
 }

 public void actionPerformed(ActionEvent evt)
 {
  String command = evt.getActionCommand();

  System.out.println("You selected the " + command + " popup menu
item!");
 }
}
```

Interview...with Paul Chambers PART 2

CTO of GemStone Systems (Europe), A Brokat company

An interview by Jason Westra

***JDJ:* Paul, what do you think about wireless and the state of the industry in five years?**

Chambers: There's certainly going to be a lot of change. The telecommunications industry is really going to go through a revolution over the next five years. Two markets we really need to look at are the North American market and the rest of the world, which really complies with another set of standards. Ultimately they're going to converge in about four years.

The sort of standards we've got in Europe – it's packet switched so it can just do data, but not fantastically. It's got a limited bandwidth, currently about 14 kilobits per second, if you've got good reception, etc. Where that's going is, by the end of the year there'll be several networks up and several devices. There's actually several devices available now through a technology called GPRS [General Packet Radio Service]. GPRS gets a bandwidth up to 115 kilobits per second. Now that's pretty nice for applications, and you can start doing real stuff. The other aspect of it, though, is that it's packet switched rather than circuit switched. This is the first step into real TCP/IP packet-switched information so that we're running voice over IP and gateway and back end, too. For all these things the existing networks are based on PSTN [Public Switched Telephone Network], the existing circuit-switched networking protocols. The big change is that, as well as the bandwidth – it's always on, like any TCP/IP, LAN-based networking packet switch.

This introduces the concept of being able to have push models as well as pull models in the mix. Another thing that bandwidth will introduce, though, is that J2ME is going to become very important. So the idea of having smarter applications running on mobile devices, based on Java and messaging back the services and technologies like JMS. JMS is being rolled out there in terms of lightweight JMS, just as J2ME is a lightweight Java form of platform. So you're going to start to see smarter applications as well as the wireless application protocol (WAP), which is essentially a markup language, albeit a binary one, which is more efficient for the narrow bandwidth.

You'll get around some current limitations of the applications. I've used them back in Europe – I've sat on the train trying to send e-mail and what happens? A coffee man comes along and interrupts you. You put your device down and go through a tunnel. You lose the signal and when you come back to your application – lo and behold, it's gone! All that you did…

In an ideal world what you want is an asynchronous model – you want a smart Java application. Possibly you want an embedded database – and nice caching locally – and asynchronous messaging back to the server so you can have asynchronous tasks, asynchronous messaging, which will get you over the problem of interruption. You know, it's not a continuous service when you're mobile. So there's going to be some nice, interesting applications once you've got this GPRS. That's happening this year. It'll really start making an impact next year in terms of the mobile Internet.

In the U.S., what you've got currently is Code Division Multiple Access (CDMA). I talked recently with Sprint. They're running out of a lot of WAP applications in America. So you guys aren't too far behind. I know the U.S. predictions have been a bit behind, but not that far behind. There's a lot of work going on. There's not a lot of clients throwing out WAP applications, but Sprint got to the point where they've got national coverage now with their network, which means they've got huge coverage and it's based on CDMA. They're going to take out CDMA next year and put in CDMA 2000. Now CDMA 2000 is like GPRS; many of the technologies are the same. Qualcomm actually owns a lot of the technologies that GPSR and CDMA 2000 are based on. Again, I guess the bandwidth is packet switched. It's going to be really nice for mobile applications and data, etc. That's what's happening over the next 18 months.

***JDJ:* With the way chips are coming out with Java embedded in them, a number of companies have had some great announcements of their products. How do you see that coinciding with J2ME? Are they complementary? Or do the actual embedded Java chips in the hardware replace J2ME? Does J2ME's role decrease?**

Chambers: I think that remains to be seen. There are three sort-of solutions I've seen out there, which may compete, or may coexist. So you're going to see chips that are designed to run bi-code instructions on these mobile devices. You're also going to see that some companies are trying to take the standard JDK – the standard J2SE platform – and not reduce functionality but squeeze it into small enough footprints so the device is just above the very small things like smart phones. They're all competing for their own space, along with J2ME, but J2ME does have some limitations, and ideally you wouldn't want to do that. You don't want to reduce the functionality of the application, etc. The only reason you're doing it is because of the limitations of the devices. The other thing is the capabilities – the bang for the buck. Obviously, it goes up. All these things are going to converge, and we'll just have to see how over the next few years.

***JDJ:* It sounds like JMS is making headway into both wireless and the J2EE platform. How do you see that evolving into this lightweight form that's going to be supported in the devices?**

Chambers: Obviously, devices need to communicate, applications need to communicate, and there's going to be a number of mechanisms to do that. JMS is an interesting mechanism for doing that because it has a more loosely coupled messaging capability.

Take a mobile application talking to a J2EE application server. As I said before, you get interrupted, you lose your signal, and so on. JMS is really good – guaranteed delivery, batched queues. It sits there when the connection comes back up, and your application will reconnect and synchronize the messaging, get the message back to the application. Of course, you've got the smart application back there in terms of the J2EE application, so it's going to play a key role. I think you're going to see things like CORBA running over the air, so it's going to be there as well. You're going to see I/O type communications as well. As the bandwidth goes up, downloading servlets, running these different object protocols and so on over the bigger bandwidth is going to be possible – it's going to happen.

***JDJ:* Are these some of the things you're seeing GemStone's partners doing with JMS and integrating GemStone as well?**

Chambers: Two companies we're working with are working within this space (probably more, but these are two I can say). SoftWired, JMS vendors from Switzerland and Zurich, has mobile, lightweight JMS for mobile devices, so that's currently available. They're starting to do wireless applications with them. Similarly, SpiritSoft, based in London, is a JMS vendor. They're working on lightweight JMS as well, and doing mobile applications in the financial industry. This is the real thing. It's really happening.

***JDJ:* You mentioned your partners. How exactly are you using the wireless? How are you building wireless applications with GemStone/J?**

Chambers: It's this mutual architecture, this aggregator. It's an EAI engine – it aggregates information at the back end and now it's aggregating at the front end as well. What you want to do is write your application once, protect that investment from technology change and from all the different channels you have to deliver front-end technology to. What we're seeing is that people are relying on GemStone behind these sorts of applications – these wireless applications – for robustness 24/7, for smartness in terms of caching. When you lose a connection it comes back. GemStone's got all that. The user state is currently cached – we make sure you don't lose the work, so really, it's just a robust, scalable thing at the back end. Do it once, distribute it to any channels. It's a channel-neutral platform so it's doing no more than you'd expect from a J2EE server, eh? Scalable, robust, lots of transactions, lots of uses.

jwestra@vergecorp.com

Developing Web Applications Using VisualAge for Java and WebSphere Studio

Build the client side of an application Part 2

WRITTEN BY
ANITA HUANG &
TIM DEBOER

n the September *Java Developer's Journal* (Vol. 5, issue 9) we discussed the tools available in VisualAge for Java and WebSphere Studio for building and debugging Web applications. This month we demonstrate how to use these tools to build a simple Web site that allows users to access their bank accounts using an Enterprise JavaBean.

We'll build on the EJB created in the June ***JDJ*** article "Building Enterprise Beans with VisualAge for Java" (Vol. 5, issue 6).

The Tutorial

In that article you learned how to create an EJB bean (i.e., how to build part of the server-side piece of the Web application). Here we'll show you how to generate an access bean from the EJB bean previously created. Then we'll show you how to import that bean into WebSphere Studio to generate a JavaServer Page (JSP) file, HTML file and servlet to build the client side of the application.

Step 1: *Prepare for the tutorial.*

To follow this tutorial you'll need the following products installed:

- IBM VisualAge for Java, Enterprise Edition, v3.02
- IBM WebSphere Studio, v3.0
- DB2 v5.2 with fixpack 11 or higher, DB2 v6.1 with fixpack 2 or higher or Oracle 8.05

Make sure you've applied the appropriate fixpacks for DB2. They can be downloaded from www.ibm.com/software/data/db2/udb/support.html.

Building the Server Application

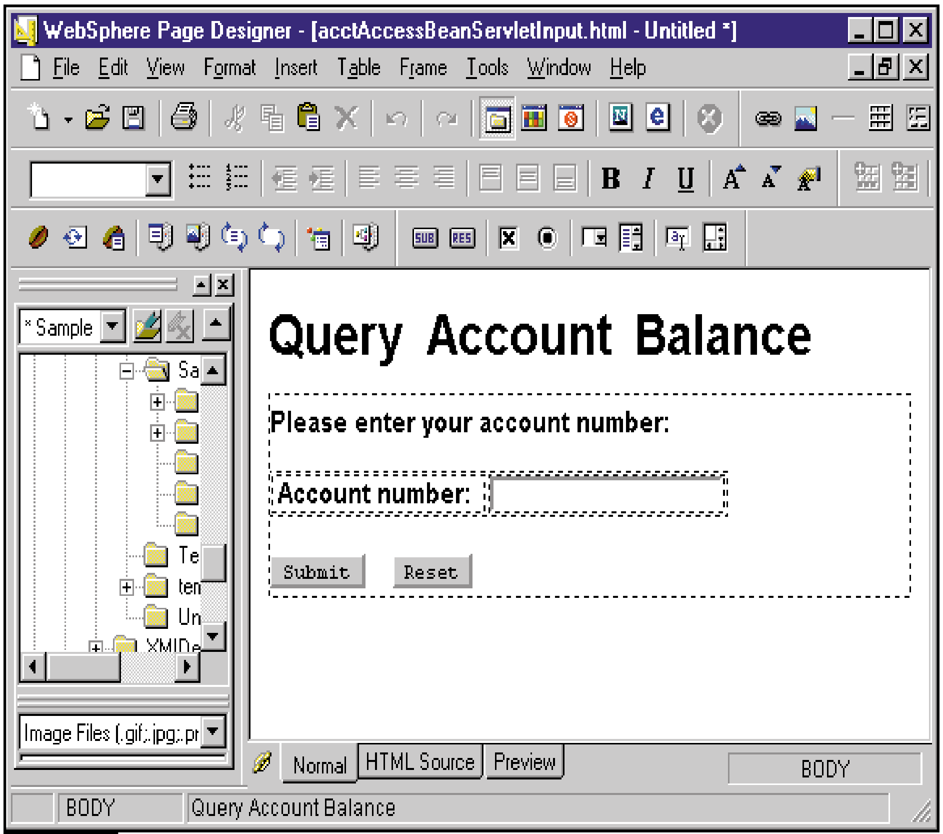
You can use the EJB Development Environment of VisualAge for Java to develop and test enterprise beans that conform to the distributed component architecture defined in Sun Microsystems' EJB specification.

Now we'll generate an access bean from the EJB bean you created from the June tutorial.

Step 2: *Create the access bean.*

In the EJB Development Environment an access bean is a JavaBean wrapper for EJB beans; an access bean is typically used by client programs, such as JSP files, servlets and sometimes other enterprise beans. Access beans adapt enterprise beans to the JavaBeans programming model and hide the complexities involved when programming directly to the EJB home and remote interfaces. Access beans can provide fast access to enterprise beans by maintaining a local cache of their attributes. They also make it possible to consume an enterprise bean in much the same way that you'd consume a JavaBean.

1. In the Enterprise Beans pane of the EJB Development Environment, right-click the Account EJB. This is the bean you created using the earlier tutorial.
2. Select Add > Access Bean. The Create Access Bean SmartGuide opens.
3. In the EJB group field make sure that BANK is entered.
4. In the Enterprise Bean field make sure that Account is entered.
5. In the Access bean type field select Copy Helper for an entity bean from the drop-down menu. (Create a copy helper access bean when you want the EJB bean attributes to be used in the creation of JSP files. Copy helpers cache data so that only one request is made to the EJB server.)
6. Click Next.
7. In the Select home method for zero argument constructor field, select findByPrimaryKey(AcctKey).
8. Make sure that initKey_primaryKey is the initial property, and that the converter is indicated as None.
9. Click Next.
10. Make sure that balance is the enterprise bean, that copy helper is checked and that the converter is indicated as None.
11. Click Finish to generate the access bean.
12. In the Enterprise Beans pane right-click Account and select Generate > Deployed Code to update the EJB bean's deployed code.
13. In the Projects workspace double-click the AccountAccessBean class. The AccountAccessBean class opens in a VisualAge for Java browser.
14. Select the BeanInfo tab. In the Features pane right-click and select Generate BeanInfo class from the popup menu. Close the AccountAccessBean browser. The AccountAccessBean-


WebSphere Page Designer - [acctAccessBeanServletInput.html - Untitled *]
File Edit View Format Insert Table Frame Tools Window Help
Sample
Image Files (.gif;.jpg;.pn
Normal HTML Source Preview
BODY
Query Account Balance

Query Account Balance

Please enter your account number:

Account number:

Submit Reset

FIGURE 1 Modified HTML page

BeanInfo class now appears in your workspace. You'll need to have access to these properties when using WebSphere Studio to generate a JSP file from the access bean.

Building the Client Application

You've created an access bean to wrap the EJB bean, so now we can create the client part of the Web application (the JSP and HTML files and a servlet that uses the access bean) using WebSphere Studio.

Step 3: *Export the access bean to WebSphere Studio.*

First you must export the access bean from VisualAge for Java to WebSphere Studio, using VisualAge for Java's EJB Development Environment to export it as an EJB client JAR file.

1. In the EJB Development Environment of VisualAge for Java, right-click Account in the Enterprise Beans pane. Select Export > Client JAR. The Export to an EJB Client JAR File SmartGuide opens.
2. In the JAR file field enter the following fully qualified path:

```
<X:\WebSphere\Studio>\projects\SimpleEJB\-
servlet\<AccessBean.jar>
```

where X:\WebSphere\Studio is the directory in which you installed IBM WebSphere Studio and AccessBean.jar is the client JAR file that you want to export.

3. Make sure that beans and .class are selected. (Deselect .java and resource if they are selected.)
4. Click Finish.

Step 4: *Import the client JAR file into WebSphere Studio.*

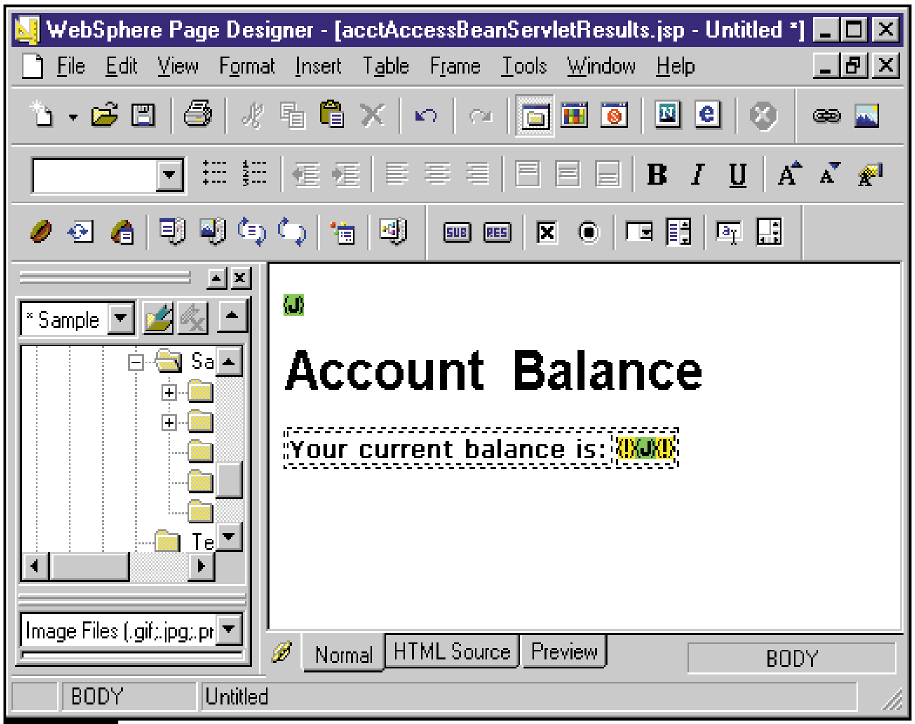
In WebSphere Studio you must create a project to contain the generated servlet, JSP files and HTML files. Once you've created the project, you can import the JAR file into it.

1. Place the following JAR file in WebSphere Studio's class path or on the Windows NT system class path:

```
<X:\IBMVJava>\eab\runtime30\ivjejb302.jar
```

where X:\IBMVJava is the directory in which VisualAge for Java is installed.

2. Start WebSphere Studio.
3. In the Welcome to IBM WebSphere Studio dialog, select Create a New Project. In the New Project dialog enter SimpleEJB in the Project Name field and click OK.
4. The SimpleEJB project is now open.
5. Right-click the servlet folder under SimpleEJB and select Insert > File. The Insert File dialog opens.
6. Select the Use Existing tab, then click Browse to locate the AccessBean.jar file. Click OK. The servlet folder now contains the AccessBean.jar file.

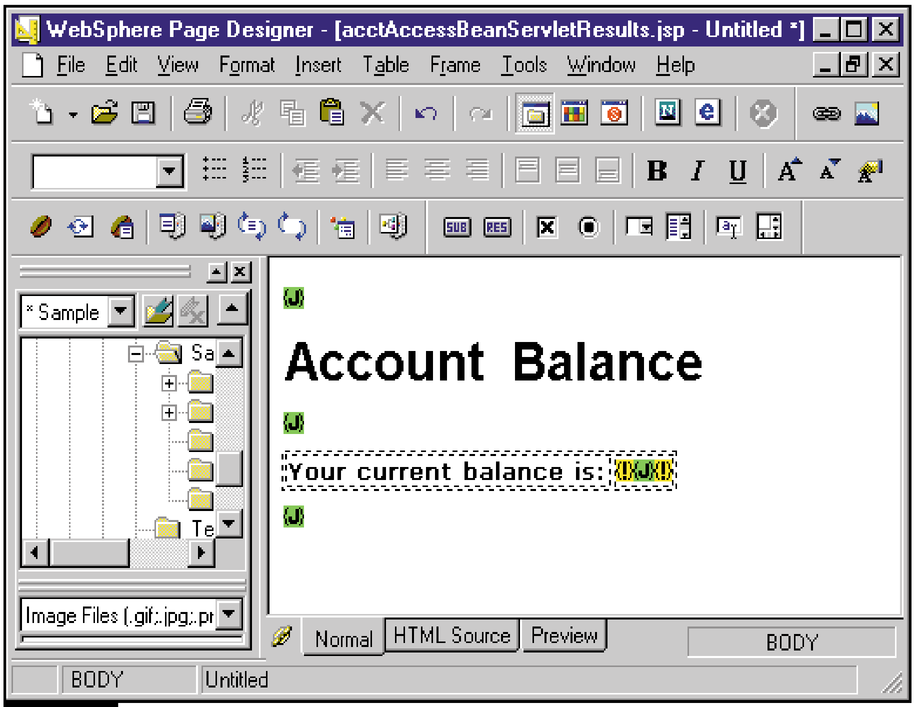


WebSphere Page Designer - [acctAccessBeanServletResults.jsp - Untitled *]

Account Balance

Your current balance is:

FIGURE 2 Modified JSP page



WebSphere Page Designer - [acctAccessBeanServletResults.jsp - Untitled *]

Account Balance

Your current balance is:

FIGURE 3 Completed JSP page

Step 5: *Use WebSphere Studio to generate an HTML file, JSP file and servlet from the access bean.*

Now you can use WebSphere Studio's JavaBean Wizard to generate an HTML file, JSP file and servlet from the access bean.

1. Select the SimpleEJB project in WebSphere Studio and choose Tools > Wizards > JavaBean Wizard. The JavaBean Wizard opens.
2. Make sure that netbank.AccountAccessBean appears in the JavaBean pane. Click Next. The Web Pages page of the JavaBean Wizard opens.
3. Make sure that both Create an input page and Create a results page are selected. Click Next. The Input Page page of the JavaBean Wizard opens.
4. Select the initKey_primaryKey property. Select the entire initKey_primaryKey line and click Change. The Change Details dialog opens. In the Caption field enter Account number. Click OK. Click Next. The Results Page page of the JavaBean Wizard opens.
5. Select the balance property. Select the entire balance line and click Change. The Change Details dialog opens. In the Caption field enter Current balance. Click OK. Click Next. The Methods page of the JavaBean Wizard opens. Don't select anything. Click Next. The Session page of the JavaBean Wizard opens.
6. Select Yes and store it in the user's session. Click Next. The Finish page of the JavaBean Wizard opens.
7. Click Rename. In the Rename dialog type netbank in the Package Name field, and type accessBeanServlet in the Prefix field. Click OK.
8. In the JavaBean Wizard click Finish. A customized HTML file, JSP file and servlet are generated from the Account access bean.

Step 6: *Customize the HTML file.*

You can use the Page Designer in WebSphere Studio to customize the HTML and JSP files.

1. In WebSphere Studio right-click accessBeanServletInput.html and select Edit with > Page Designer. The WebSphere Page Designer launches open to accessBeanServletInput.html.
2. Modify the HTML page to look like Figure 1.
3. Save the revised HTML page.

Step 7: *Customize the JSP file.*

Page Designer's advanced HTML editor also allows easy insertion of JSP tagging. We'll add error handling to the generated JSP page.

1. In WebSphere Studio right-click accessBeanServletResults.jsp and select Edit > Page Designer. Page Designer opens to accessBeanServletResults.jsp.
2. Modify the JSP page to look like Figure 2.
3. We need to add a try/catch block in the JSP page since access bean methods can fire RemoteExceptions. In the JSP page position the cursor directly beneath the Account Balance text. Select Insert > JSP Tags > Scriptlet. The Script editor opens.
4. In the empty pane add the code:

```
try {
```

5. Click OK.
6. In the JSP page position the cursor in the HTML space directly beneath the HTML table containing the text Your current balance is: Select Insert > JSP Tags > Scriptlet. The Script editor opens.
7. In the empty pane add the code:

```
} catch (Exception e) {
   out.println("Not available at this
time");
}
```

8. The completed JSP page should look like Figure 3.
9. Save accessBeanServletResults.jsp and close WebSphere Page Designer.

Step 8: *Export the servlet source (.java) to VisualAge for Java.*

To modify the generated servlet source, export the source to VisualAge for Java and start the Remote Access to Tool API. Then, from WebSphere Studio, send the source to VisualAge for Java.

1. In VisualAge for Java select Window > Options > Remote Access to Tool API. Click Start Remote Access to Tool API, then click OK.
2. Make sure that the netbank package in the SimpleEJB project exists in VisualAge for Java as an open edition (otherwise the class can't be transferred correctly from WebSphere Studio). In the VisualAge for Java project workspace right-click the netbank package and select Manage > Create Open Edition.
3. In WebSphere Studio highlight accessBeanServlet.java (this file is contained in the servlet folder).
4. Select Project > VisualAge for Java > Send to VisualAge. The Send to VisualAge dialog opens.
5. Select the SimpleEJB project. Click OK.

You've now finished building an end-to-end Web application using VisualAge for Java and WebSphere Studio.

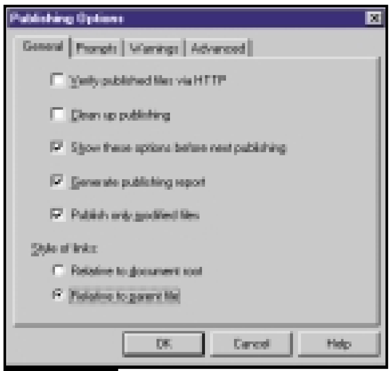
Running / Testing the Sample Application

To run and test the sample Web application, first publish the Web application using WebSphere Studio, then start the Web and EJB servers with VisualAge for Java to run the application. Finally, test the application using your Web browser.

Step 9: *Publish the sample Web application.*

Use WebSphere Studio to publish the sample so you can test the application in VisualAge for Java. Publishing allows your code to be used by other components. Complete the following:

1. Check all of the files in WebSphere Studio. Right-click the SimpleEJB project and select Check In.
2. In WebSphere Studio select Tools > Publishing Options, then click the Advanced tab and select Default Publishing Targets.
3. Set the HTML path to that of your Web resources path (i.e., <X:\IBMVJava>\Ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web, where X:\IBMVJava is the directory in which VisualAge for Java is installed).
4. Click OK when completed.
5. Publish the SimpleEJB project:
   • Select View > Publishing. (If you don't see http://localhost in the Publishing view, right-click Test and select Insert > Server. The Insert Server dialog opens. In the Server name field enter http://localhost and click OK.)
   • In the right pane of WebSphere Studio right-click Test and select Publish Whole Project. The Publishing Options dialog opens.
   • Make sure the options in Figure 4 are set.
   • Click OK to publish only the modified files. Accept all defaults by clicking Yes after each prompt.


Publishing Options

General | Prompts | Warnings | Advanced

☐ Verify published files via HTTP
☐ Clean up publishing
☑ Show these options before next publishing
☑ Generate publishing report
☑ Publish only modified files

Style of links:
○ Relative to document root
◉ Relative to parent file

FIGURE 4 Publishing options

Step 10: *Prepare to run sample Web application.*

Use VisualAge for Java to run the Web application. Complete the following:

1. Start the WebSphere Test Environment by choosing Workspace > Tools > Launch WebSphere Test Environment. Check the Console window to make sure the WebSphere Test Environment is launched.
2. Start VisualAge for Java's Persistent Name Server and EJB Server to prepare to run the EJB application:
   • In the Workbench select the EJB tab.
   • Right-click the BANK EJB group and select Add To > Server Configuration. The EJB Server Configuration dialog opens.
   • Right-click Persistent Name Server and select Properties.
   • In the Properties for Persistent Name Server dialog enter the database URL for your database in the Data Source field. (If you've used DB2's First Steps, enter jdbc:db2:sample to use the SAMPLE database.)
   • In the Connection Type field select the JDBC driver for your database. (If you're using DB2, select COM.ibm.db2.jdbc.app.DB2Driver.)
   • Click OK.
   • In the Servers pane of the EJB Server Configuration dialog right-click Persistent Name Server and select Start Server.
   • In the Console window look for the open for business message that indicates that the Persistent Name Server is running.
   • In the EJB Server Configuration dialog right-click EJB Server and select Properties.
   • In the Properties for EJB Server dialog enter your database URL in the Data Source field. (Once again, enter jdbc:db2:sample if you're using the DB2 SAMPLE database.)
   • In the Connection Type field select the JDBC driver for your database. (If you're using DB2, select COM.ibm.db2.jdbc.app.DB2Driver.)
   • Click OK.
   • In the Servers pane of the EJB Server Configuration dialog right-click EJB Server and select Start Server.
   • In the Console window look for the open for business message that indicates that the EJB Server is running.

Step 11: *Use your Web browser.*

Launch your Web application in your Web browser.

1. Launch your Web browser.
2. Specify the following URL: http://localhost:8080/accessBeanServerInput.html.
3. Enter a valid account number that you created earlier using the Test Client (see the June ***JDJ*** article). Click Submit. The JSP results page will display the account's current balance.

Congratulations! You now have a running end-to-end Web application. When you want to deploy it, use the WebSphere Application Server. (See the WebSphere Application Server help documentation for details.)

Conclusion

WebSphere Studio and VisualAge for Java contain all the tools required to build and debug a complete end-to-end Web application. By using these development products, which support a role-based development model, developers have the tools required to carry out their specific responsibilities.

AUTHOR BIOS

Anita Huang is currently working on IBM's WebSphere Developer Domain site, providing in-depth samples and tutorials that incorporate the WebSphere software platform for e-business. Previously, she worked on the VisualAge for Java Information Development team, focusing primarily on componentry to build enterprise applications.

Tim deBoer currently develops tools to build applications that run on WebSphere Application Server. He previously worked with the VisualAge for Java Technical Support group, providing support to enterprise developers working with VisualAge for Java.

anitah@ca.ibm.com
deboer@ca.ibm.com

Enterprise Java: Oracle8*i* JVM

A Java engine for e-business development and deployment platforms

J D J F E A T U R E

WRITTEN BY KUASSI MENSAH

*The Oracle Internet Platform embeds the Oracle8*i *JVM within the Oracle8*i *database and Oracle Internet Application Server (iAS) as the enterprise Java engine for Oracle. This article explains Oracle8*i *JVM's base architecture, its support for J2EE APIs and its latest performance and architecture enhancements.*

E-Business Platform Base Architecture

The base architecture, the foundation on which a Java platform's building blocks are implemented, is typically the VM architecture: memory management, execution environment and so on. As an e-business platform, it should provide robustness, reliability and scalability across all system components. One key question: What makes the Oracle8*i* JVM different from other VMs?

Session-Based Architecture

In a break from the design of most client Java Virtual Machines, the Oracle8*i* JVM has promoted a session-based architecture in which each user executes within a session. From the user's perspective, each session executes Java with its own dedicated JVM that has its own containers and its own Java global variables and garbage collector. Under the covers, sessions are scheduled and run on a dynamically allocated set of processes (or threads on Windows/NT) – typically less than 10% of the number of connected users – using Oracle's multithreaded server (MTS). The MTS architecture provides robustness, as a replacement process is automatically started by the Oracle 8*i* runtime in the event that process fails. By isolating sessions, this architecture allows parallel and independent processing, memory allocation and garbage collection (see Figure 1). On the other hand, sharing of code, metadata, immutable statics and read-only objects across sessions reduces session memory footprint and activation time.

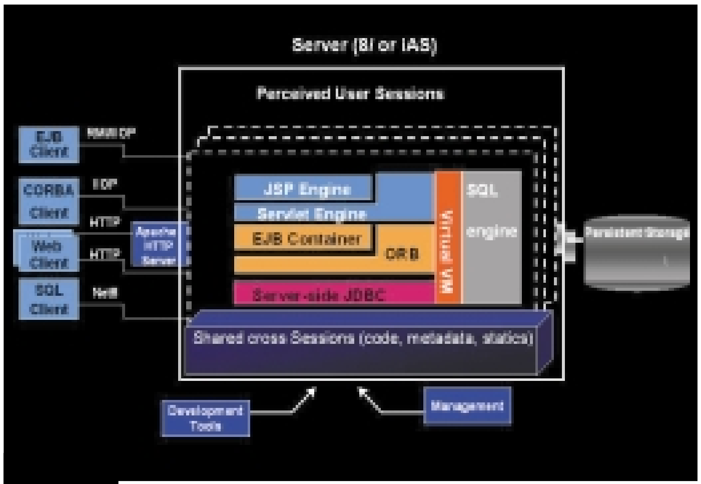


FIGURE 1 Oracle8: JVM architecture

Session Isolation vs Sharing

Java objects held in static variables between calls are, by default, completely private to the users session. The 8*i* JVM maintains the complete context of each session so that isolation is an implicit part of the system, not one that's enforced by the programming model. Session isolation doesn't allow the sharing of user "state" or contexts. Oracle8*i* JVM sessions can share read-only objects across sessions through JNDI. This feature is used internally for now to maximize sharing of metadata and will be exposed in the future for use by user programs as well.

Connectivity

The total number of MTS processes required to run tens of thousands of connected users sessions is dynamically adjustable and approximately equals the number of concurrently active sessions (approximately less than 10% of the number of connected users).

Session IIOP

Session IIOP provides Oracle JVM with the ability to create and access objects in multiple sessions from a single client and services that handle multiple server objects in multiple sessions. This allows the JVM to distinguish objects based on the session in which they are activated. The standard CORBA doesn't have the notion of "sessions." Oracle8*i* JVM does. It achieves this by extending the IIOP protocol, using a Session IIOP component tag within the object reference (the OMG assigned component tag ID for session IIOP is 0x4f524100). Session IIOP still looks the same, on the wire, as regular IIOP. URLs have the following form: sess_iiop://<host>:<port>:<service>/:session/name.

Storing Java in the Database

Java sources, classes and resources are stored within the database in internal structures called *libunits*. Libunits are managed by the Oracle8*i* runtime in the same way Oracle's PL/SQL packages are managed, providing LRU tradeout of Java classes and keeping track of class dependencies during development and updating of Java applications. Users can place Java code in the repository graphically using Oracle JDeveloper or by a command-line utility (loadjava). Classes can be resolved and updated dynamically without interrupting or recycling the 8*i* JVM.

Consistent JDK Across Platforms

This base architecture, currently based on Sun's JDK 1.2.1, is identical on every platform where Oracle8*i* or iAS is ported, providing a consistent Java deployment environment across more than 60 platforms.

E-Business Base Infrastructure Services

The Object Management Group has defined one of the most exhaustive standards-based infrastructure services via CORBA services. Several key services in this collection have been respecified as is or redesigned by Sun Microsystems as part of Enterprise Java infrastructure services. Among them are Naming, Transaction, Security and Persistence. These services form the foundation for higher-level services required by e-business applications. Oracle8*i* JVM incorporates these services.

Naming: Built-in JNDI

As a central naming repository, Oracle8*i* JVM implements JNDI 1.2 API using database tables for secure, robust and scalable storage of namespace mappings. This JNDI implementation provides a server-side SQL-based driver (SQL SPI) for accessing JNDI directly from within the server tier and a client-side RMI/IIOP-based driver (RMI/IIOP SPI) for accessing the JNDI service from Java clients. It's used for binding or registering servlets and JSPs, CORBA/EJB components and non-Java CORBA Objects. It also allows the sharing of read-only objects across sessions and the registration of services at database start-up, typically IIOP and HTTP. The CORBA CosNaming interface has been implemented on top of the JNDI namespace for pure CORBA users.

RMI/IIOP

For robustness, scalability and ease of programming (no IDL), Oracle8*i* JVM has implemented RMI over IIOP based on Visigenics Caffeine; it also provides support for method overloading and Java exception handling through serialization.

Java Transaction API `

Oracle8*i* JVM implements the Java Transaction API (JTA) 1.0 standard, including client- and server-side transaction demarcation, two-phase coordination, multitier transaction context propagation, single-phase commit optimization, and support for JDBC as well as CORBA/EJB transaction managers. It also provides support for JDBC, HTTP, RMI/IIOP clients and mixed client types.

Security

Oracle8*i* JVM security is based on:

- A built-in Oracle8*i* database security: execution rights required on stored Java classes as well as on SQL execution
- Java 2 Security, including granting or revoking fine-grain or collections of Java permissions to roles
- Execute rights required on published objects in JNDI
- SSL (SSL 3.0) authentication and non-SSL login authentication based on a challenge response protocol using DES 48 bit encryption

Query and Persistence

Object-relational mapping is achieved by:

- ***Either*** direct access to Oracle8*i*'s rich database object types that are SQL Types, Object Types, Variable Arrays (varrays), Nested Tables, XML documents and SQL Types exposed to Java (JPublisher) through JDBC/SQLJ or Java Stored Procedures
- ***Or*** by using transparent persistence mechanisms like the Persistence Service Interface for CMP EJB in conjunction with persistence managers that implement such an interface (PSI Reference Implementation, Oracle Business Components Java (BC4J) – O/R mapping and third party O/R mapping) (see Figure 2.)

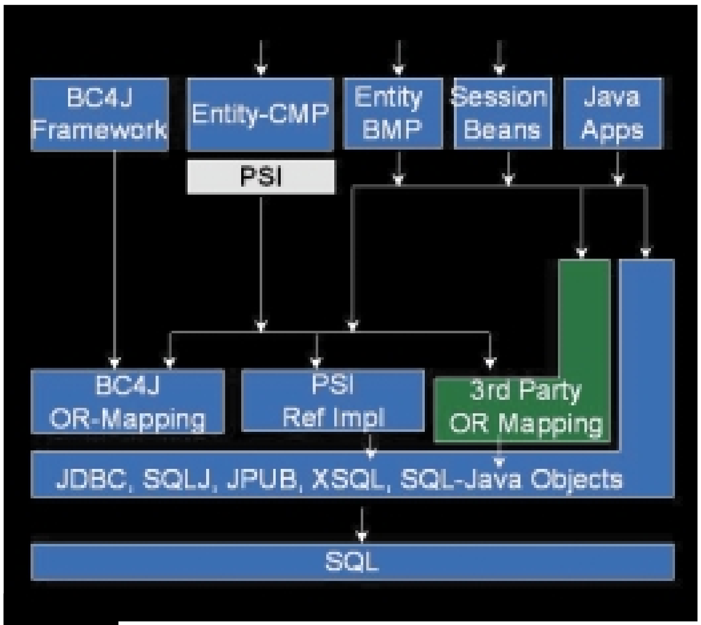


FIGURE 2 Persistency mechanisms

Component-Based Applications Architecture

The main characteristic of a component-based application is the loose coupling between layers, allowing reuse, partitioning and minimal change impact. Typically there are four logical layers to a partitioned application: View, Application-Model, Domain-Model and Database (see Figure 3).

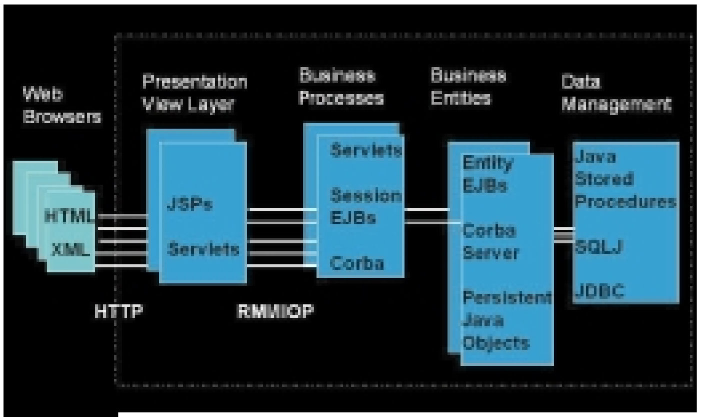


FIGURE 3 Application architecture and J2EE programming model

View Layer

Components that implement the user interface handle the GUI screens and Web pages using GCI scripts, static HTML, Perl scripts, ActiveX, Visual Basic, Java Client Application, Java Applet, Servlets and JavaServer Pages (JSPs).

Servlets are server-side Java classes that execute independently or call back-end components (EJBs, CORBA Server Objects, Java/PLSQL Stored Procedures, other Servlets) and generate coarse-grained dynamic content, typically used as a UI controller or application controller, in conjunction with JSP (serving as the View). Servlets can be chained or pipelined to extend Web server functionality.

JSPs are used to dynamically generate the actual UI or Web pages by assembling coarse-grained dynamic content from servlets and back-end components (EJBs, CORBA Server Objects, Java/PLSQL Stored Procedures) with fine-grained content from Java Scriptlets and static HTML.

Oracle Servlet Engine

Oracle8*i* JVM embeds a Servlet 2.2-compliant engine suitable primarily for stateful servlets. All servlets activated by one client are in the same session, along with EJBs, CORBA Server Objects, Java Stored Procedures, the default server-side JDBC connection and the SQL engine. The first servlet request creates a session for the client and eventually (on request) an HTTPsession object per stateful servlet context; further requests are routed to the same session using cookies or URL rewrites. OSE supports authentication and access control as specified by Servlet 2.2 through Realms and URL security mappings in the JNDI namespace.

Servletclasses are loaded, stored as library units (libunits) in the database repository, then resolved and published in the JNDI namespace where they can be looked up and invoked through HTTP. For example: http://cavist.com:8080/cellar/welcome.html<path_info>.

```
public class Hello extends HttpServlet {
 public void doGet(HttpServletRequest rq, HttpServletResponse rsp){
  rsp.setContentType("text/html");
  PrintWriter out = rsp.getWriter();
  out.println("<HTML><HEAD><TITLE>Welcome</TITLE></HEAD>");
  out.println("<BODY><H3>Welcome!</H3>");
  out.println("<P>Today is "+ new java.util.Date()+
  ".</P>");
  out.println("</BODY></HTML>");
 }
}
```

OracleJSP Engine

Oracle8*i* JVM embeds a JSP engine (JSP 1.1- compliant) on top of the OSE, the Oracle HTTP server (powered by Apache Jserv) and all Web-enabled Oracle products (Oracle Portal, Portal-to-go, Oracle JDeveloper). The Oracle JSP engine supports SQLJ within JSP Scriplets, data access beans for connecting and querying an Oracle database, custom tag handlers and the ability to format results of a JSP using XSL. It emulates some Servlet 2.2 features on Servlet 2.0 engines (JServ) using a globals.jsa file:

```
<HTML><HEAD><TITLE>Welcome</TITLE></HEAD>
<BODY><H3>Welcome!</H3>
<P>Today is <%= new java.util.Date() %>.</P>
</BODY></HTML>
```

Oracle HTTP Server Powered by Apache and Extension Mods

Oracle8*i* JVM ships with an HTTP server (based on Apache) as the default HTTP listener for serving fine-grained static HTML and stateless servlets (JServ). Oracle extends Apache with extension modules (Mods):

- ***mod_ose:*** Dispatches HTTP requests for stateful, coarse-grained dynamic contents to the Oracle servlet engine
- ***mod_plsql:*** Dispatches HTTP requests for stateless PL/SQL and Java stored procedures; maintains database connections specified by DADs (database access descriptors)

Application Model Layer

This layer hosts components that implement stateful business processes (such as a shopping cart), use case controllers or view controllers. It also hosts stateless service components (e.g., rate engine, tax calculator). They can be implemented using servlets and stateful and/or stateless session EJBs, as well as CORBA server objects when infrastructure services are invoked programmatically. The componentization of this layer allows users with different interfaces to access the same components using different types of client interfaces.

Domain Objects or Data Component Layer

This layer hosts components that model and implement coarse-grained domain objects, also called *data* components or *business* components.

Coarse-grained business objects are designed as root objects and dependent objects using a design pattern (Façade). They can be implemented as simple persistent Java objects (when they don't require their own transaction and security services) but also as CORBA server objects (programmatic invocation of transaction and security services) or entity EJB (declarative transaction, security services and persistence calls by EJB container).

Oracle8i JVM EJB Architecture

The EJB component model allows the design and implementation of complex business components:

- ***Stateless business services:*** tax calculator, rate engine, for example
- ***Stateful business processes/use cases:*** shopping cart, for example
- ***Persistent, uniquely identifiable, coarse-grained, application-independent business entities:*** customer, portfolio, purchase order, for example

The EJB container in Release 3 of Oracle8*i* JVM is fully compliant with the EJB 1.1 specification, which mandates the support of entity EJB (bean-managed and container-managed persistency), including a standard and provider-specific XML deployment descriptors. The container is activated within the session address space along with all collocated Java components, including the server-side JDBC driver and the SQL engine. EJB's classes are stored within the shared libunits repository (the database), then published and looked up from the distributed objects subdomain of the JNDI namespace; they run as transactional Java objects when invoked locally from collocated components (like servlets) or as CORBA server objects when invoked remotely through RMI/IIOP.

Persistence Service Interface

For container-managed persistence EJB, Oracle has defined a persistence service interface (PSI) to handle the relationship between the Oracle8*i* JVM EJB container and the persistence manager (see Figure 2). The container handles the allocation and management of EJB objects, transparent service invocation (transaction and security) and bean deployment. The persistence manager handles the allocation and management of beans; their loading, storing and caching; and JTA synchronization for commits/rollbacks.

Oracle8*i* JVM ships with a PSI reference implementation (PSI-RI) that provides a simple attribute to column mapping: persistence manager. Oracle's JDeveloper and third-party vendors provide PSI-compliant, complex mapping persistence managers. By providing a common way for dealing with persistence managers (à la EJB 2.0), PSI allows their replacement with little impact.

Object Request Broker

Oracle8*i* JVM embeds a 100% Java CORBA 2.0-compliant ORB (Visibroker 3.4) that can be activated within MTS processes and shared by multiple sessions while preserving their context isolation. It extends the IIOP protocol and IORs with session ID (for routing), implements the CosNaming JNDI URL interface (for easy and consistent naming) and allows programmatic invocation of all infrastructure services.

Data Access and Management Layer

JDBC Drivers

Oracle8*i* JVM supports four main JDBC drivers:

1. ***Oracle call interface (OCI) client driver (type 2):*** Must be installed on the Client. Partly Java, it allows the use of Oracle's Advanced Networking options, Net8 tracing and logging; components are deployed on a middle tier but an applet can use this driver.
2. ***Thin client driver (type 4):*** All Java, and uses Net8/TTC database protocol over Java sockets; applets must use this driver, as can components deployed on a middle tier.
3. ***Thin server driver (type 4):*** For applications that need to access other databases through JDBC, from within Oracle8*i*.
4. ***Server-side driver ("kprb"):*** Runs inside the Session address space, incurring no network round-trip; all components, EJBs, Servlets and Java Stored Procedures deployed on an Oracle8*i*'s tier must use this driver.

An ultrathin driver ("proxy"), under development at the time of this writing, will act as a client-side proxy for the "kprb" driver, reducing memory footprint on the client side.

Object Types

Oracle JDBC drivers materialize database objects as instances of Java objects using either a default mapping (object as oracle.sql.STRUCT) or explicit customized mapping (SQLData interface). For example:

```
ResultSet rs = stmt.executeQuery ("select VALUE(p) from
CUSTOMER_TAB p");
while (rs.next ())
{
  // retrieve the STRUCT
  oracle.sql.STRUCT cust_struct = (STRUCT)rs.getObject(1);
  // list the attributes
  Object cust_attrs[] = cust_struct.getAttributes();
  // string attribute in Object
  String name = (String) cust_attrs[0];
  // embedded object
  oracle.sql.STRUCT address = (STRUCT)cust_attrs[1];
  //embedded array
  oracle.sql.ARRAY = (ARRAY)cust_attrs[2];
}
```

Enhanced Support for JDBC and SQLJ

Oracle8*i* JVMs' JDBC drivers have complete support for JDBC 2.0 including the XA Resource API, basic statement caching that minimizes cursor creation and tears down overhead, LONG functions on LOBs, JDBC/SQLJ support for VARCHAR functions on CLOBs and support for PL/SQL table of scalars.

In addition, numerous performance improvements have been made at the server-side JDBC driver level and for access to objects.

SQLJ

A higher level than JDBC API, which allows embedding static SQL in Java:

```
#sql {UPDATE emp Set sal = 3000 WHERE ename = 'SCOTT'};
```

SQLJ provides support for selecting a single row directly into Java variables, named and positional iterators for accessing the results set of a multirow query and connection to multiple schemas at the same time, although an SQL statement executes in a single connection context:

```
     #sqlj [ctx1] departments = {select dept_name from departments}
```

Oracle8*i* JVM enhances its SQLJ implementation by supporting all the JDBC 2.0 features as outlined in the forthcoming ISO standard, including structured types, scrollable iterators, datasources, batching, row prefetching and interoperability with JDBC 2.0 connection pooling. It also provides three different runtime versions: (1) a generic runtime that can be used with any Oracle JDBC driver, (2) a runtime optimized for JDK 1.1 environment, and (3) a runtime optimized for JDK 1.2 environment.

JDBC versus ...

```
String name;
int id=131341;
float salary=6000;
PreparedStatement pstmt =conn.preparedStatement
   ("select ename from emp where empno=? And sal>?");
pstmt.setInt(1,id);
pstmt.setFloat(2,salary);
ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
 name=rs.getString(1);
 System.out.println("Name is: " + name);}
rs.close();
pstmst.close();
```

... SQLJ

```
String name;
int id=131341;
float salary=6000;
#sql (select ename into :name from emp where empno=:id and
sal>:salary);
System.out.println("Name is: " + name);
```

Java Stored Procedures

Oracle also provides Java stored procedures. They implement compute-intensive procedures (as opposed to SQL-intensive procedures for which PL/SQL is more suitable). Java stored procedures can be used in the same ways as PLSQL: as stored functions, stored procedures, user-defined functions and database triggers.

Stored procedures are standards based, portable across vendor and secure (access control). They can be invoked through the JDBC by EJBs, servlets and JSPs or called by non-Java and legacy modules or servers (PLSQL packages, Forms, Reports, etc.). Using the same call specification or wrapper as PL/SQL, Java stored procedures allow transparent migration to Java (transparently replacing PL/SQL code with Java, when recommended).

Any public static methods of Java classes can be published and called or invoked as Java stored procedure.

```
public class Foo {
  public static String prependHello(String tail) {
 return "Hello " + tail;
  }
}
```

To create a PL/SQL wrapper to expose/publish/register the method and make it callable:

```
CREATE FUNCTION PREPENDHELLO (s VARCHAR2) RETURN VARCHAR2 AS
   LANGUAGE JAVA NAME 'Foo.prependHello(java.lang.String)
   return java.lang.String';
```

E-Business Deployment Platform

Runtime Performance: Bytecode Compilation

By default, Java applications run interpreted. A Java interpreter successively fetches, decodes and executes Java bytecodes. The fetching and decoding steps can be avoided by natively compiling the bytecode; also, the compilation process can apply some code optimization techniques. Oracle8*i* JVM offers native compilation and optimization.

Oracle8i JVM Accelerator

Oracle8*i* JVM provides a native bytecode accelerator that allows a deployed Java bytecode (.jar or .class) to be translated into platform-independent C code. The C code is then compiled using platform-specific compilers (optimized for their respective platforms), yielding fully optimized platform-dependent Oracle8*i* JVM-specific native shared libraries, and run as natively compiled code. Performance improvements range several orders of magnitude.

Deployment Schema

Most Java platforms on the market promote a three-tier deployment with all Java and Web components on a middle tier, typically an application server, and data access and management components on the database tier (see Figure 4).

Oracle Internet Platform Deployment Scenario

Being present in both Oracle8*i* and iAS servers, Oracle8*i* JVM allows flexible Web and Java components partitioning across tiers. The following four

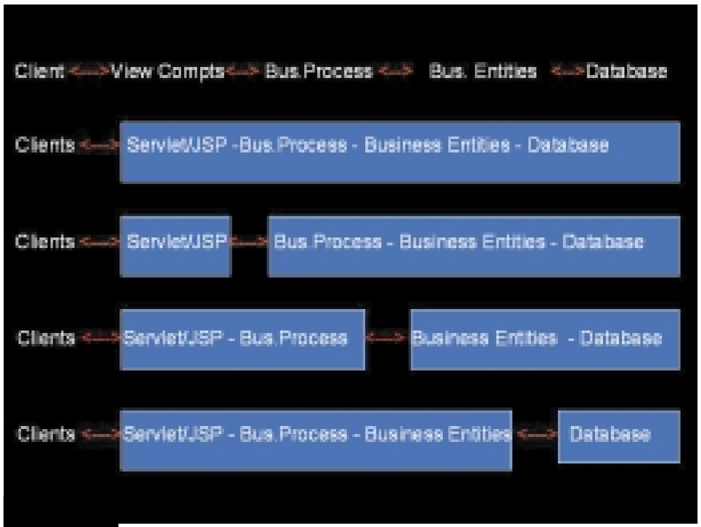

Client <—> View Compts <—> Bus.Process <—> Bus. Entities <—> Database
Clients <—> Servlet/JSP - Bus.Process - Business Entities - Database
Clients <—> Servlet/JSP <—> Bus Process - Business Entities - Database
Clients <—> Servlet/JSP - Bus.Process <—> Business Entities - Database
Clients <—> Servlet/JSP - Bus.Process - Business Entities <—> Database

FIGURE 4 Deployment over the 8*i* and iAS tiers

deployment scenarios are simply examples. Customers can make variations in the way components are deployed over the Oracle Internet Platform.

1. ***Single-tier deployment:*** In this schema all components are deployed on the Oracle8*i* tier. The benefits include local calls, hence reduced network traversal. This scenario scales to tens of thousands of concurrent users against a single Oracle8*i* instance using the high-connectivity capabilities of the shared server MTS architecture. Since Oracle8*i* JVM and its containers are the same on the iAS tiers, beyond this level of scalability offloading the Oracle8*i* tier on iAS tiers will provide additional scalability.

2. ***View layer on iAS; other layers on Oracle8i:*** A Web site activity consists primarily (75%) of static HTML and stateless servlet requests; this can be served by a farm of iAS/Apache HTTP servers. The remaining 25%, involving stateful servlets, stateless session EJBs, persistent EJBs or CORBA servers, and JDBC/SQLJ/Java stored procedures, can be served by an 8*i* tier.

3. ***View and application layers on iAS; domain layers and data access and manipulation on 8i:*** In this scenario static HTML, stateless and stateful servlets, and stateful and stateless session EJBs run on the iAS tier while entity EJBs or persistent CORBA server s objects reside on the 8*i* tier along with JDBC/SQLJ/Java stored procedures. Since domain components are data-oriented, running inside the database will provide efficient execution.

4. ***Traditional middle-tier deployment:*** In this scenario Web/EJB/-CORBA components are deployed on the iAS tier while data access and management modules (JDBC/SQLJ and Java stored procedures) reside on the Oracle8*i* tier. iAS tiers can be duplicated to support the requirements of hundreds of thousands of concurrent users.

Conclusion

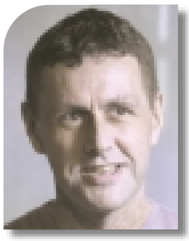
By extending its support to J2EE containers and programming models, and enhancing the architecture and performance of Oracle8*i* JVM, the Oracle Internet platform offers a scalable, secure and robust end-to-end Enterprise Java platform for developing and deploying e-business applications.

AUTHOR BIO
Kuassi Mensah is a senior product manager in Oracle's server technology division and leads the product management team of the Java products group.

kuassi.mensah@oracle.com

Working with Swing

Experiences with the Swing Library during development of a splash screen component



WRITTEN BY
PAUL ANDREWS

One of the strengths of Java is the abundance of standard APIs for doing everything from enterprise-level data access to manipulating data structures, sending and receiving e-mail and building GUIs. This broad sweep of APIs makes choosing how to implement the various parts of an application much simpler, but it also presents a problem: How do you best use the API?

Javadocs – the raw API documentation – are never enough for this purpose because they concentrate on each class or interface as a separate entity from the other classes. API specifications aren't much better because they deal with the minutiae of the API and with being a correct specification of how everything works. The most useful tool I've found for any newcomer to an API is the Java Tutorials (www.java.sun.com/docs/books/tutorial/index.html). These Tutorials guide you to correct and efficient use of an API rather than swamping you with detail.

This article extends the Tutorials and addresses some of the issues a developer might come across when writing an application that uses the Swing API. In it I will dive below the surface of the Swing and AWT toolkits to build a simple but useful component called a *splash screen*. I'm barely scratching the surface here, but you should learn some useful techniques and have a component that's useful in its own right.

What Is a Splash Screen?

A splash screen is the small window that appears while a program is loading. It usually has none of the decorations normally associated with a window, appears centrally positioned on the screen, and contains a graphic announcing the application and company and often some kind of progress indicator. Figure 1 provides an example.

My primary aim here is to produce a component that encapsulates most of the detail required to create such a window so it's easy for other people to include its functionality in their own applications. Essentially, it'll be an extension to the Swing Library.

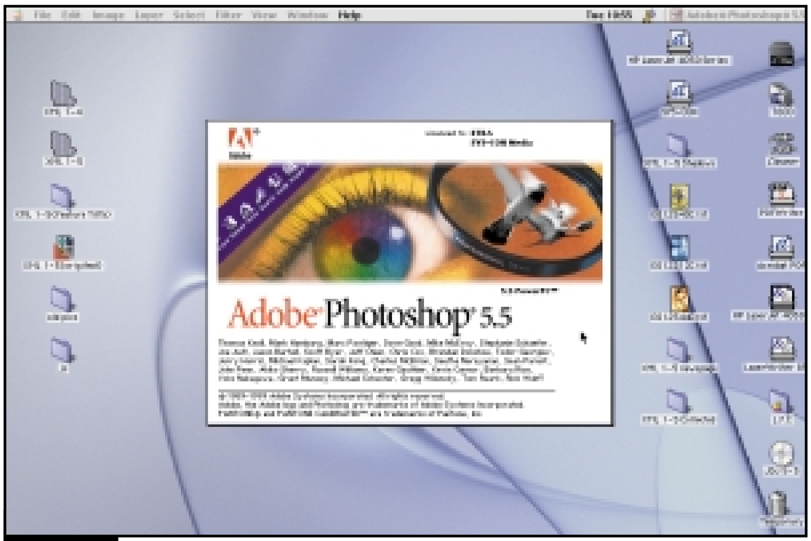


FIGURE 1 A splash screen

Getting Started

First I need to select a component that will produce a top-level window with no decorations. A quick trawl through the API documentation reveals that JWindow is the class I should be using. Creating one of these is simple enough, but I'm going to add specialized functionality later so I may as well subclass JWindow so I can encapsulate that functionality in one place:

```
import javax.swing.JWindow;
public class SplashScreen extends
JWindow {
…
}
```

Next I want to center the window on the screen. This involves a brief trip back into the AWT class libraries to use the Toolkit class. Ordinarily I wouldn't use this class directly, but it's the source of some useful information – in this case the screen dimensions. The Toolkit class itself is abstract, but can be used to locate an actual implementation. Once I have the implementation I can use it and information about the window size to position the window properly. To get the window size before it's visible, I just have to "pack" it. This code has to be executed after the window has been fully populated; the SplashScreen constructor would be a good place:

```
  pack();
  Dimension screenDim = Toolkit.get-
DefaultToolkit().getScreenSize();
```

```
  setLocation((screenDim.width - get-
Size().width) / 2,
    (screenDim.height -
getSize().height) / 2);
```

Creating and showing my window now is simple. In the following example I pass in the application's main window as a parent:

```
SplashScreen ss = new
SplashScreen(this);
ss.setVisible(true);
```

Once I'm finished with the window I'll need to remove it from the screen. I could do this with setVisible(false), but this will leave the window consuming resources (e.g., memory). As I'll never be showing it again, I may as well destroy it so that those resources are released. Merely removing my references to the window isn't sufficient as Swing has internal references that will prevent it from being garbage-collected. To remove Swing's references I need to call dispose() on the window, so I'll add a remove() function to my class that does this for me. This function will be made to perform other tasks later:

```
public void remove() {
  dispose();
}
```

Filling in the Window

So far, I have a window with no contents, so I need to decide what contents to add and how to lay them out. Earlier I suggested that a splash screen should contain a graphic and a progress indicator. I'd also like to add a title as a separate component and lay out these components from top to bottom in the window. All products that my company ships will use the same layout for their splash screens, providing a consistent look across the whole product range, so I'm free to enforce the policy by encapsulating the code that produces it inside my class. If I wanted a more flexible splash screen, I could expose the standard JWindow functions to allow users to specify a layout manager and to add components as they want. To prevent my applications having to implement boilerplate code to create the splash screen, I'll implement as much as possible inside my class. For this reason my constructor will accept two JComponents for the title and progress indicator (more on why later) and the name of a graphic file to use as the main graphic:

```
public SplashScreen (JFrame parent,
  String graphicName, JComponent
title,
  JComponent progress) {…}
```

Graphics is one area where the Java GUI libraries provide excellent facilities. My graphic file can be a JPEG, a GIF or even an animated GIF. All I have to do is create an ImageIcon, passing it the URL of a file containing the image:

```
ImageIcon i = new ImageIcon(url);
```

Resources

One issue: What location does this URL point to? I need a location that I can guarantee will exist wherever my application is installed. One of the best locations to fulfill this requirement is somewhere on the classpath. Even better would be to put the graphic in the same place as the rest of the classes that make up this specific application because then I can ship everything the application needs in one simple package.

This tells me where I should put the graphic, but I won't know where my application package will be installed. How can I discover what URL to use to actually load the graphic? Fortunately, the class loader will do this for me. Classloaders can be used to load anything off the classpath – not just classes. First I need to get a classloader – the one that was used to load me. This is easy enough to do. Every object has a method to get its class and hence its classloader. Then I use that classloader to obtain the URL of my graphic:

```
public URL getURL(String path) {
   ClassLoader cl =
      getClass().getClassLoader();
   return cl.getResource(path);
}
```

Now I pass that URL to the constructor of ImageIcon as in the previous example.

This is a generally useful facility, so in the interests of reuse I should really encapsulate all this code in a class. ImageIcon has a constructor that takes a string – which it assumes is a filename. I could simply subclass ImageIcon and make the same constructor look for a resource of that name. If that fails, it can default to assuming the string is a filename. If I implement the other ImageIcon constructors too, I can use my new class wherever ImageIcon has been used already without breaking that code – plus I get the benefit of being able to distribute my application, along with the resources it needs, as one JAR file.

Now adding the resulting image to my display is simple. I just create a JLabel to hold it, then add the JLabel to my splash screen. I'll do this in my constructor so the code will look like this:

```
i = new ResourceImageIcon(path);
image = new JLabel(i);
add(image);
```

Careful analysis of running code shows I have a problem. The resources used by the instance of ImageIcon aren't completely released when I dispose of the window because ImageIcon caches the image data so that subsequent requests for the same image are faster. Worse, if the image is an animated GIF, the thread that performs the animation continues to run. To prevent this, I need to call flush() on the actual image. I may as well add this code to the remove function, which now becomes:

```
public void remove() {
   i.getImage().flush();
   dispose();
}
```

Swing and Threads

What about that progress indicator? I need to update the contents of one component, and this is one reason I passed in the components themselves rather than just, say, strings. If I make the progress indicator a JLabel, I can hold a reference to it in the object that created the SplashScreen instance and update its contents by calling setText(). I could use any other JComponent that I can dynamically change the contents of, but this will do for my purposes.

The only problem is that the Swing library isn't thread-safe once a component has been realized. In other words, I can't call setText() from any thread other than the Swing thread once I've made the splash screen visible. This would normally be okay because I'd be updating the display in response to a Swing event that would automatically ensure I was in the Swing thread. However, it's likely that I'll update my progress indicator in response to external events (connecting to servers, reading data, etc.), so how do I ensure that I call setText() from the right thread? Fortunately, Swing provides two functions that allow me to queue up work for it: SwingUtilities.invokeLater() and SwingUtilities.invokeAndWait(). The functions take a Runnable as an argument that's responsible for actually performing the work. The Runnable needs to have access to any data it needs at the time it runs. In the following example I call invokeAndWait() from a member function of a class called MainWin. The MainWin object holds a reference to the JLabel I'm using to display the message and to the text the Runnable will need to change the contents of the JLabel. The JLabel reference is called *progress* and the message reference is called *status*.

```
SwingUtilities.invokeAndWait (new
Runnable() {
   public void run() {

progress.setText(MainWin.this.sta-
tus);
   }
});
```

I'll have to use this mechanism whenever I want to update the splash screen once it's been made visible. This includes the call to "remove," which will remove it from the screen. I could put this code inside any function of SplashScreen that I think might be called from outside the Swing event loop. This would allow users of SplashScreen to call its methods without having to think about whether they're inside or outside the event loop. If I did this, I'd have to be very careful that I didn't cause deadlocks and that the data I needed didn't change between the user calling my function and Swing invoking the Runnable.

Tidying Up

So far so good, but the window still looks a little messy. I'd like to center the title and the progress indicator and make sure that the colors coordinate properly with the graphic. The code to do this appears in Listing 1. I also think a border should be added to the original wish list for the appearance of the splash screen – I can do this using the BorderFactory class. Plus the image needs to be centered and the components need to be laid out in the right order, running from top to bottom: title, image and then the progress indicator. The code to do all this appears in Listing 2.

Keeping Busy

Finally, I'd like to display a watch cursor while the pointer is over the window and prevent the user from interacting with it. Adding such a cursor can be done simply by adding the following line of code to the constructor for the splash screen:

```
setCursor(Cursor.getPredefinedCur-
sor(Cursor.WAIT_CURSOR));
```

The easiest way to prevent user interaction with the splash screen is to make the glass pane visible. This pane, which is transparent, is a standard part of Swing windows. It's used to intercept events that would normally go to the window itself – for the splash screen I only need to intercept mouse events. The following code in the constructor of the splash screen will do it:

```
getGlassPane().addMouseListener(new
MouseAdapter() {});
getGlassPane().setVisible(true);
```

For windows that already have the keyboard focus, I'd also have to intercept key events, but as the splash screen will never receive the focus, this code is sufficient.

Summary

I'm now satisfied that I have a component that fulfills my criteria for a splash screen. This useful component can be used straight out of the box, but more important – for the purposes of this article – it explores some of the real-world problems that GUI developers encounter when writing a Swing application. In particular:

- Using the Toolkit to obtain information about the screen
- Using a classloader to locate resources
- Ensuring that all components are freed up when they're no longer needed
- Accessing Swing components from other threads
- Fine-tuning the appearance of a window
- Putting a window into a "busy" state

paul@jools.net

AUTHOR BIO

Paul Andrews has worked for over 20 years as a computer scientist in the IT industry and has been actively programming in Java since 1997. Paul specializes in the design of large distributed OO architectures for the implementation of secure e-commerce systems.

Listing 1

```
public void createSplashScreen()
{
  // Create a JLabel for the title. Ensure that the text is
  // centered within the JLabel and that the JLabel is cen-
  // tered within the window.
  JLabel _title = new JLabel("SplashScreen Test v3.1", JLa-
                             bel.CENTER);
  _title.setAlignmentX(Component.CENTER_ALIGNMENT);

  // Set the color of the text in the title. The background
  // will be that of the window itself.
  _title.setForeground(Color.green);

  // ditto for the progress indicator
  progressLabel = new JLabel("Connecting to server...", JLa-
                             bel.CENTER);
  progressLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
  progressLabel.setForeground(Color.green);

  // Create the SplashScreen
  splashScreen = new SplashScreen(
    this, "net/jools/test/SplashTest.jpg", _title, progressLa-
bel);

  // Set the background color of the SplashScreen
  splashScreen.getContentPane().setBackground(Color.white);

  // Show the SplashScreen
  splashScreen.setVisible(true);
}
```

Listing 2

```
public SplashScreen(…)
{
  JComponent pane = (JComponent)getContentPane();

  // Use a vertical BoxLayout to arrange the components
  // top to bottom.
  pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));

  i = new ResourceImageIcon(splashName);

  // To ensure that the image is centered horizontally, cen-
  // ter it in the JLabel and center the JLabel in the win-
dow.
  image = new JLabel(i, JLabel.CENTER);
  image.setAlignmentX(Component.CENTER_ALIGNMENT);

  // Add a border to the window
  pane.setBorder(BorderFactory.createCompoundBorder(
    BorderFactory.createRaisedBevelBorder(),
    BorderFactory.createLoweredBevelBorder()));

  // The order in which the components are now added to the
  // window is important. First the title, then the image,
  // then the footer.

  // To ensure that the title component spans the full width
  // of the window set its maximum sizes. The caller is
  // responsible for its alignment
  if (title != null) {
    title.setMaximumSize(
      new Dimension(Integer.MAX_VALUE, Integer.MAX_VALUE));
      pane.add(title);
  }

  pane.add(image);

  if (footer != null) {
    footer.setMaximumSize(
      new Dimension(Integer.MAX_VALUE, Integer.MAX_VALUE));
    pane.add(footer);
 }
  …
}
```

Interview...with Shawn Mitchell

President of CodeMarket

An interview by Israel Hilerio

CodeMarket is a global software development network where software developers and development managers can find and purchase freelance development work and ready-to-run Java components. It recently formed a partnership with ParaSoft, a provider of software error-prevention and error-detection solutions. ParaSoft's Jtest, a Java unit testing tool, will be the standard tool by which all components outsourced or purchased through CodeMarket will be tested.

JDJ: ***While your partnership with ParaSoft seems like a step in the right direction, how do you ensure the richness of the requirements you receive from customers? What methodology does CodeMarket use to ensure the accurate definition of requirements?***

Mitchell: That's an excellent question. One of CodeMarket's best features is that the buyer can select from a number of different bids. Each bid from a developer contains a use case that illustrates, from the client's point of view, how the objects behave and the top-level architecture of the proposed solution. So a buyer can compare not only the price and delivery date, but also the quality and intelligence of the proposed solution. This allows developers to showcase their object-oriented analysis and design skills and gives the stronger, more experienced ones pricing power in the marketplace.

JDJ: ***Software unit testing certainly enhances the quality of the software component. However, how does CodeMarket define quality?***

Mitchell: Quality is an ephemeral notion. It's difficult to put quality in a box. CodeMarket's goal is to allow developers to work on projects that interest them, to do the work offsite, and to protect them from feature-creep by establishing a fixed deliverable. For buyers, quality means that the code works as designed, handles unexpected inputs gracefully, and conforms to some stylistic conventions that encourage maintainability and readability, such as limiting method length and the number of return statements in a method. For developers, quality means a lot of things. It means being able to establish a reputation for producing excellent code and delivering it on time. It means knowing the ground rules won't change from project to project. And it means having the incentives and the structure in place to encourage a mentality of getting it right the first time – building quality code now rather than wading through spaghetti a week before release tracking down bugs identified in system- and user-acceptance testing.

CodeMarket uses ParaSoft's award-winning JTest for unit testing. JTest performs automated static, black-box, white-box and regression testing on all submitted code. Our partnership with ParaSoft allows us to offer the only service that provides an independent measure of development completion. CodeMarket pays the developer as soon as his or her code passes the JTest unit test, and that prevents the he-said/she-said, feature-creep and accounts-receivable collection headaches so common among freelancers today. That's the quality that every developer who is making the leap into the freelance economy appreciates most of all.

JDJ: ***What are the various test statistics and matrices provided by ParaSoft JTest? Do you deliver the test results, statistics and matrices output to the customer as part of the final deliverable? Is there an auditing process that ensures CodeMarket's accountability concerning the testing process?***

Mitchell: ParaSoft's JTest provides a comprehensive suite of analytics and test outputs. Information about JTest is available at www.parasoft.com. Clients receive all test reports with the delivered code. The delivery of the JTest reports establishes that testing has been completed and the code meets the specified requirements.

JDJ: ***Do the components outsourced to CodeMarket for development need to be self-contained, or can they have dependencies on the customer's own components?***

Mitchell: The issue of what makes a good CodeMarket project is an interesting one. Clearly some work is better suited to outsourcing than others. For instance, code that's proprietary or is part of the core competitive advantage of a company is probably not something you want to outsource over the Internet. Similarly, a project that requires access to specific large or difficult-to-duplicate resources (like a robotics controller for an assembly line) are not great candidates for outsourcing. CodeMarket doesn't expressly prevent someone from listing such a project but, as with any tool, CodeMarket works best on tasks for which it was designed. CodeMarket is the best way to develop fixed-cost, unit-tested loosely coupled Java components.

JDJ: ***As CodeMarket develops these components and ships them to its customers, what happens with component-integration testing? Is that the responsibility of the customer? What does CodeMarket do to ensure component-dependency resolution and interoperability with customer-developed components?***

Mitchell: After a registered freelance developer (www.codemarket.com/reg.jsp) submits code to us and we unit test it, the code is delivered to the buyer, who's responsible for application integration and further testing. Studies by the prestigious Software Engineering Institute indicate that unit testing reduces the cost of integration and system testing dramatically by eliminating structural faults at the unit level.

JDJ: ***How is the ParaSoft JTest tool going to help CodeMarket define standards of completeness and operability?***

Mitchell: JTest provides an objective standard for completeness and operability. It's that objectivity that's most valuable to buyers and developers. By establishing a standard level of quality and having that standard monitored and enforced by an independent third party, both developers and buyers are protected and empowered to produce excellent software.

JDJ: ***What are the additional types of testing services you provide as part of your component delivery contract?***

Mitchell: In addition to JTest testing, CodeMarket allows developers to attach guarantees to their code. If after delivery the buyer is dissatisfied, CodeMarket will arbitrate any dispute that might arise. This gives buyers another measure of confidence in the process and lets developers, who know their code is good, charge a premium for the buyers' added peace of mind.

"CodeMarket allows developers to attach guarantees to their code. If after delivery the buyer is dissatisfied, CodeMarket will arbitrate any dispute that might arise"

Interviewer Bio

Israel Hilerio is a member of a leading e-commerce firm in Dallas, Texas, focusing on Web-based e-commerce applications and new architectures.

israel@sys-con.com

JDJ NEWS

Quadbase Ships EspressChart 3.0

(*Santa Clara, CA*) – Quadbase Systems Inc. announced EspressChart 3.0 for creating and publishing dynamic graphic charts on the Web. Features include support for multiple data sources for a single chart, user definable legends, translucent chart elements, annotation for chart elements, 2D/3D scroll and zoom, and foreign language support.

www.quadbase.com

Tidestone Announces Formula One 8.0

(*Overland Park, KS*) – Tidestone Technologies, Inc., announced an upgrade to its Formula One for Java development tool. This upgrade will ease the integration of its spreadsheet engine with outside data sources when building Web-based reporting and analysis applications. The new version 8.0 will include the ability to bind spreadsheet cells to a much wider variety of input documents.

www.tidestone.com

RSW Software Launches EJB-test 2.2

(*Waltham, MA*) – RSW Software Inc. announced the availability of EJB-test 2.2 for testing the scalability and functionality of Enterprise JavaBeans (EJB) middle-tier applications. Significant enhancements to EJB-test include advanced session bean support, dynamic graphing capabilities and extended functional testing.

www.rswsoftware.com

Introducing Versant Developers Suite 6.0

(*Fremont, CA*) – Versant Corp. announced the beta release of Versant Developers Suite (VDS) 6.0. The new version extends the database's current query capabilities through the addition of multi-attribute queries, and incorporates support for the latest industry standards. The Java Developer's Interface (JVI) features advanced caching, synchronization and object mapping, and complies with Sun's Java Data Objects.

www.versant.com

Compuware Delivers DevPartner 2.0

(*Farmington Hills, MI*) – Compuware Corporation has begun shipping NuMega DevPartner 2.0 Java Edition.

DevPartner Java Edition supports a wide range of application servers, servlet engines and Web servers on Solaris, Linux and Windows 2000. It integrates with IBM VisualAge for Java, Symantec Visual Café, Borland JBuilder and Oracle JDeveloper.

www.compuware.com

META Group and Flashline.com Join Forces

(*Cleveland, OH*) – Flashline.com Inc. and META Group Inc. have joined forces to advance component-based development. Through a content-sharing agreement, the companies will provide organizations with resources to assist with the research, development and deployment of component-based software for business.

www.flashline.com
www.metagroup.com

WebGain Ships VisualCafe 4.0

(*Santa Clara, CA*) – WebGain Inc. announced VisualCafé Enterprise Edition V. 4.0. New features include productivity wizards that streamline the development and deployment of Enterprise JavaBeans, and provide an improved Java compiler and debugger. It's also tightly integrated with BEA's WebLogic family of application servers and will soon support the iPlanet application server.

www.webgain.com

SilverStream Acquires Excelnet

(*Billerica, MA*) – SilverStream Software, Inc., has acquired Excelnet Systems Limited, a UK-based e-business services company. The addition of Excelnet will expand SilverStream's ability to meet worldwide demand for comprehensive e-business solutions generated by the availability of the SilverStream eBusiness Platform.

The Excelnet group will supplement SilverStream's existing global customer service organization in Europe, the Middle East and Africa (EMEA).

www.silverstream.com

Brokat Closes Acquisition of GemStone

(*Stuttgart, Germany*) – Brokat AG, a leading provider of software platforms for e-business solutions, has concluded the acquisition of Beaverton, OR-based GemStone Systems, Inc. GemStone will operate as a wholly owned subsidiary of Brokat AG, and will continue to distribute its products and services under its existing brand name.

www.brokat.com

Instantiations Ships VA Assist Enterprise/J 1.5

(*Portland, OR*) – Instantiations, Inc., has begun shipping VA Assist Enterprise/J 1.5. New features include full compatibility with VisualAge for Java 3.5, enhanced repository management power tools, enhanced GUI building productivity tools and powerful new code development tools. The VA Assist Enterprise/J 1.5 update is available for download from the company's Web site.

www.instantiations.com/assist.

Czech Travel Site Chooses GemStone

(*Beaverton, OR and Praha, Czech Republic*) – GemStone Systems, Inc., announced that its high-performance, multitier application server software was chosen as the architectural platform for Fractal.cz, a fast-growing Czech B2C and B2B site for purchasing airline tickets and other travel services using the Internet and wireless application protocol (WAP).

www.FractalCorp.com
www.gemstone.com

DLJdirect Selects Quest's SharePlex for Oracle Software

(*Irvine, CA*) – Quest Software, Inc., announced that the DLJdirect online brokerage house chose Quest's SharePlex for Oracle database replication software to maintain 24/7 database availability for the online trading application.

SharePlex is an innovative log-based database replication product that enables users to replicate large volumes of database activity over local- or wide-area networks. The replicated copy can be used as a fully accessible Oracle instance, yet it doesn't consume high amounts of system and network resources.

www.quest.com

Second Edition of O'Reilly's *Java Network Programming* Released

(*Sebastopol, CA*) – *Java Network Programming* by Elliotte Rusty Harold is a complete introduction to developing network programs (both applets and applications) using Java. It covers everything from networking fundamentals to Remote Method Invocation (RMI). It includes chapters on TCP and UDP sockets, multicasting protocol and content handlers, and servlets. This second edition also includes coverage of Java 1.1, 1.2 and 1.3. New chapters cover multithreaded network programming, I/O, HTML parsing and display, the Java Mail API, the Java Secure Sockets Extension, and more.

www.oreilly.com