TUNING UP FOR BETTER PERFORMANCE WITH WLS

# BEA WebLogic DEVELOPER'S JOURNAL

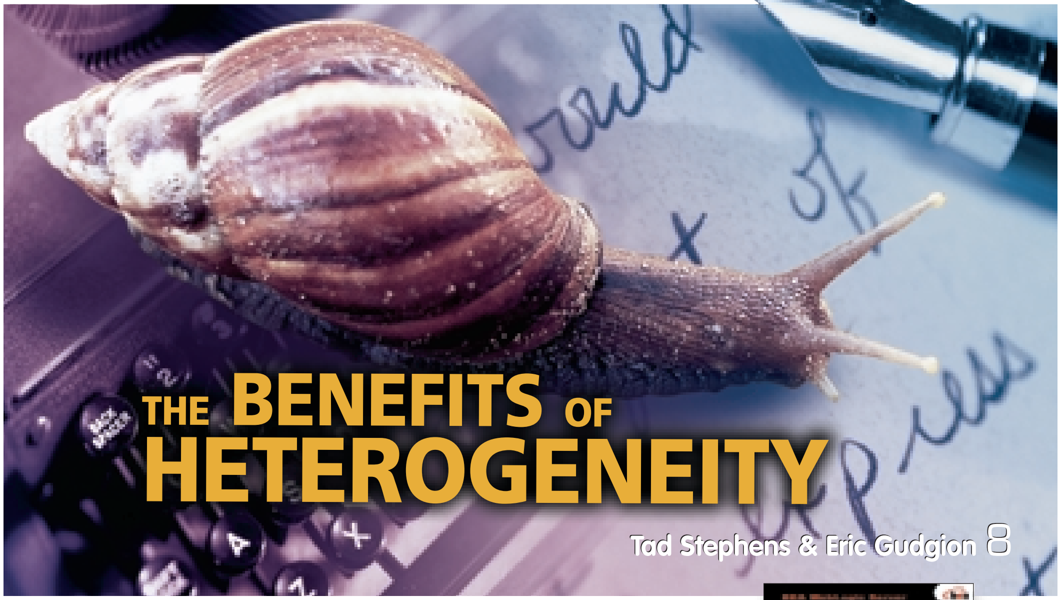
JULY 2002 - Volume:1 Issue:7

weblogicdevelopersjournal.com

BEA eworld 2002
THE 5TH ANNUAL BEA EUROPEAN E-BUSINESS CONFERENCE
Palais des Congres
June 25-26, 2002
Paris, France

RETAILERS PLEASE DISPLAY UNTIL AUGUST 31, 2002
$15.00US $16.00CAN

SYS-CON MEDIA



BEA

http://developer.bea.com

BEA

http://developer.bea.com

Wily Technology

www.wilytech.com

BEA WebLogic DEVELOPER'S JOURNAL

**EDITORIAL ADVISORY BOARD**
TYLER JEWELL, FLOYD MARINESCU, SEAN RHODY

**FOUNDING EDITOR**
PETER ZADROZNY

**EDITOR-IN-CHIEF**
JASON WESTRA

**EDITORIAL DIRECTOR**
JEREMY GEELAN

**EXECUTIVE EDITOR**
GAIL SCHULTZ

**MANAGING EDITOR**
CHERYL VAN SISE

**SENIOR EDITOR**
M'LOU PINKHAM

**EDITOR**
NANCY VALENTINE

**ASSOCIATE EDITOR**
JAMIE MATUSOW, JEAN CASSIDY

**ASSISTANT EDITOR**
JENNIFER STILLEY

**WRITERS IN THIS ISSUE**
PHILIP ASTON, DAVE COOKE, JOAKIM DAHLSTEDT, ERIK GUDGION, ARUNABH HAZARIKA, STEVE MUELLER, TOM MULVEHILL, JOHN MUSSER, SAM PULLARA, TAD W. STEPHENS, SRIKANT SUBRAMANIAM, SCOT WEBER, JASON WESTRA, PETER ZADROZNY

**SUBSCRIPTIONS**
For subscriptions and requests for Bulk Orders, please send your letters to Subscription Department.
SUBSCRIPTION HOTLINE:
888-303-5282
Cover Price: $15/Issue
Domestic: $149/YR (12 Issues)
Canada/Mexico: $169/YR
Overseas: $179/YR
(U.S. Banks or Money Orders)

**PUBLISHER, PRESIDENT AND CEO**
FUAT A. KIRCAALI

**CHIEF OPERATING OFFICER**
MARK HARABEDIAN

**CHIEF FINANCIAL OFFICER**
BRUCE KANNER

**VP, BUSINESS DEVELOPMENT**
GRISHA DAVIDA

**SENIOR VP, SALES & MARKETING**
CARMEN GONZALEZ

**VP, PRODUCTION & DESIGN**
JIM MORGAN

**ART DIRECTOR**
ALEX BOTERO

**ASSOCIATE ART DIRECTORS**
AARATHI VENKATARAMAN
LOUIS CUFFARI • CATHRYN BURAK
RICHARD SILVERBERG

**ASSISTANT ART DIRECTOR**
TAMI BEATTY

**VP, SALES & MARKETING**
MILES SILVERMAN

**ADVERTISING SALES DIRECTOR**
ROBYN FORMA

**ADVERTISING ACCOUNT MANAGER**
MEGAN RING

**ASSOCIATE SALES MANAGERS**
CARRIE GEBERT • ALISA CATALANO
KRISTIN KUHNLE • LEAH HITTMAN

**VP, SYS-CON EVENTS**
CATHY WALTERS

**CONFERENCE MANAGER**
MICHAEL LYNCH

**REGIONAL SALES MANAGERS**
MICHAEL PESICK • RICHARD ANDERSON

**ASSISTANT CONTROLLER**
JUDITH CALNAN

**ACCOUNTS PAYABLE**
JOAN LAROSE

**ACCOUNTS RECEIVABLE**
JAN BRAIDECH

**ACCOUNTING CLERK**
BETTY WHITE

**WEBMASTER**
ROBERT DIAMOND

**WEB DESIGNERS**
STEPHEN KILMURRAY • CHRISTOPHER CROCE
CATALIN STANCESCU

**ONLINE EDITOR**
LIN GOETZ

**JDJ STORE.COM**
ANTHONY D. SPITZER

**CUSTOMER SERVICE REPRESENTATIVE**
MARGIE DOWNS

**EDITORIAL OFFICES**
SYS-CON Publications, Inc.
135 Chestnut Ridge Road, Montvale, NJ 07645
Telephone: 201 802-3000 Fax: 201 782-9637
SUBSCRIBE@SYS-CON.COM
BEA WebLogic Developer's Journal (ISSN# 1535-9581) is published monthly (12 times a year)
Postmaster: Send Address Changes to
BEA WEBLOGIC DEVELOPER'S JOURNAL,
SYS-CON Publications, Inc.
135 Chestnut Ridge Road, Montvale, NJ 07645

SYS-CON MEDIA

# Scalability for the Masses

BY JASON WESTRA
EDITOR-IN-CHIEF

If you asked me what the theme for this month's ***WLDJ*** is, I'd have to say "performance and scalability." I was once asked, "What is the most scalable way to build a J2EE application?" "Let's just find the holy grail while we're at it!" I thought. The question is quite common among J2EE developers but not an easy one to answer, even with a stack of ECPerfs up your sleeve. So I did what every good football coach does when it's fourth and long. I punted. Actually, I answered with a few questions, "By scalable, do you mean an application that can increase its users without a significant degradation in performance? Or do you mean, 'How many developers can you throw at this thing to get it done more quickly?'" My associate wanted to know the former, but I sparked enough interest to discuss the latter as well.

I use WebLogic because it's scalable and performs well, but that's not the only reason. I use it because it's easy to develop with and easy to administer once development is finished. It scales well to a large development group for several reasons. First, the install is simple and easy to duplicate across development machines, which allows each developer to have the same environment.

WebLogic is (and always has been) configured through files, whether they are Java properties files or XML files. This feature allows easy customization of server environments with or without tools (I'm thinking Notepad here). Also, files are easily stored under source-code control and can be customized per developer and in an automated fashion. On the other hand, WebSphere utilizes a DB2 database for configuration information and is a bear to configure, with or without its administration console. I know what you're thinking, and yes, I've worked with WebSphere on more than one occasion. Each experience was a reaffirmation that its technology is ages behind WebLogic's.

WebLogic has also improved its administration features tremendously from the 5.1 to the current 7.0 release. It is by far the most highly supported application server by third-party vendors in the application monitoring space. Its management and monitoring backbone is based on JMX (Java Management Extension). This is the key to allowing WebLogic to scale beyond a product to a platform that integrates seamlessly with add-on tools and products from other vendors as well as BEA.

I've mentioned numerous ways to think about WebLogic's scalability from the perspective of development and administration. It's food for thought; however, this issue also has articles from experts on WebLogic performance and scalability of the kind you are most familiar with – raw speed! Peter Zadrozny, chief technologist for BEA Systems, Europe, joins us this month with a modified excerpt from his book, *J2EE Performance Testing*. In it, he focuses on the performance results his team gathered during their study of WebLogic Server's JMS implementation. Philip Aston looks at "The Grinder: Load Testing for Everyone," an open-source performance testing tool. Also, we have an article on WebLogic performance tuning from Srikant Subramaniam and Arunabh Hazarika that covers many of the knobs and buttons you can twist and turn to make WebLogic hum, whether the knob is on an EJB deployment descriptor, the Web container, or a JVM argument. This month's ***WLDJ*** will show you how to make WebLogic scale to meet the masses, whether in development or production.

I'd also like to introduce a new ***WLDJ*** column, "In the Admin Corner," which features information that will make life easier for system administrators of WebLogic applications. As more applications are put into production, this is becoming a hot topic.

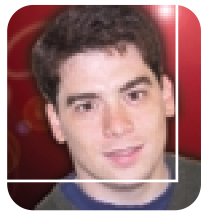
AUTHOR BIO...
Jason Westra is the editor-in-chief of *WLDJ* and the CTO of Evolution Hosting, a J2EE Web-hosting firm. Jason has vast experience with the BEA WebLogic Server Platform and was a columnist for *Java Developer's Journal* for two years, where he shared his WebLogic experiences with readers.

CONTACT: jason@sys-con.com

# CMP 2.0, EJBGen, and Builder Make EJBs Easy!

## WITH NEW TOOLS THE PROCESS IS PAINLESS AND EFFICIENT

BY SAM PULLARA



EJBs have always been the best way to ensure that your applications were portable and would leverage all the optimizations of the J2EE server. Now they are also easy to build. With the release of WebLogic Platform 7.0, you can create EJBs in record time. At the center of this revolution is Container Managed Persistence 2.0, which allows WebLogic to build tools that remove the layers from EJB development. Relations, the standard EJB query language, and JavaBean-like properties all contribute to this new age. Since EJB 1.0 was introduced it's had a rap as a difficult, complex technology to implement – probably for good reason. This release of WebLogic is targeted at changing that. Going forward there will be more and more tools to help you develop EJBs painlessly and efficiently.

The biggest tool for the developer is EJBGen. EJBGen gives you the power to create an entire EJB, including home and remote (and/or local) interfaces and deployment descriptors, by creating a single implementation file marked up with special JavaDoc tags. This drastically increases the maintainability of the system and makes it simple for developers to quickly generate the EJBs needed for a project. Listing 1 demonstrates this for you.

The relatively small amount of source code and description translates directly into all the pieces that make up an EJB component JAR. To generate the code, simply use EJBGen, which runs as a JavaDoc doclet:

```
javadoc -doclet weblogic.tools.ejbgen.EJBGen -d targetDir FooEJB.java
```

In the targetDir you'll find the Java source and the deployment descriptors for all the components that make up your EJB. Compile and JAR the files to deploy to the server. It's really as simple as that.

WebLogic Builder is the biggest time-saving tool if you've already written your EJBs and just need to configure and deploy them. The Builder tool will crack open .jar, .ear, and .war files or work on open directory structures to configure the deployment descriptors for all your components. Not only that, using an internal tool called DDInit (we provide Ant tasks to run it outside the Builder), you can take loose classes and turn them into components.

Let's say you don't want to use EJBGen or are given a set of interfaces for the EJB. Simply write the implementation beans compile, and point the Builder at your directory. Not only will it discover what kinds of beans you have, it will also generate defaults for all the deployment descriptors, including all the relations between beans. It's that simple. With a bean that is mostly for data, you can get away with writing as little as a couple of interfaces and one abstract implementation class with no methods!

There are also a couple of tools in the works that can save you time. Expect to see SQL2EJB and ReverseEJBGen in a follow-on release. SQL2EJB does what you might imagine: given a database with a reasonable schema (it's best if it has all of its foreign and primary keys properly marked), it will generate a set of EJBGen source classes that can then be EJBGen'd, compiled, and deployed to a server. Even better, since WebLogic supports dynamic table creation, you can deploy it to another database that doesn't already have the schema for testing or development purposes. This is the type of tool that has only become reasonable with the introduction of relations to the EJB specification. ReverseEJBGen does what you might imagine: given an EJB component, or set of components, it will generate an EJBGen source file that includes all the information found in the deployment descriptors. It will consolidate all those disparate files down to one file that is the final word on an EJB. Also, it lets you use the Builder tool to edit a deployed EJBGen'd component, change the deployment descriptors, and then convert back to the original format.

Expect to see these types of tools integrated with products like WebLogic Builder to make developing products for the WebLogic Platform as simple as possible.

**Listing 1**

```
/**
 * @ejbgen:entity    ejb-name = foo
 *                    data-source-name = info
 *                    table-name = foos
 *                    prim-key-class = java.lang.String
 * @ejbgen:jndi-name local = FooHome
 * @ejbgen:finder    signature = "Collection findAllFoos()"
 *                    ejb-ql = "SELECT b FROM foos AS b"
 */
public abstract class FooEJB extends GenericEntityBean {
  /**
   * @ejbgen:cmp-field column = foo_name
   * @ejbgen:primkey-field
   */
  public abstract String getFooName();
  public abstract void setFooName(String fooName);

  public String ejbCreate(String fooName) throws CreateException {
    setFooName(fooName);
    return null;
  }
  public void ejbPostCreate(String fooName) { }
}
```

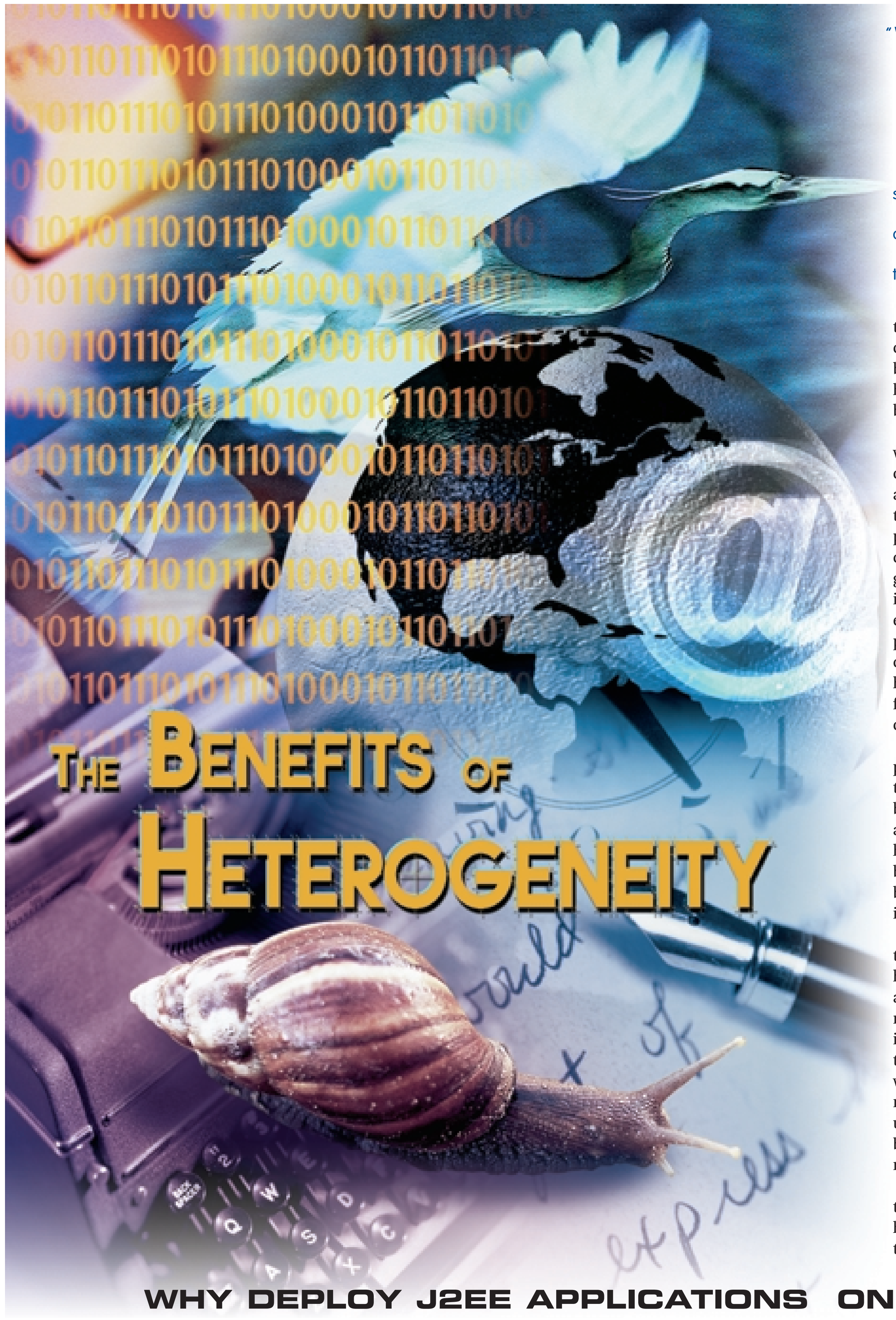
**AUTHOR BIO...**

Sam Pullara has been a software engineer at WebLogic since 1996 and has contributed to the architecture, design, and implementation of many aspects of the application server.

**CONTACT:** sam@sampullara.com

Panacya

www.panacya.com



# The Benefits of Heterogeneity

## Why Deploy J2EE Applications on the Mainframe?

BY
TAD STEPHENS &
ERIC GUDGION

AUTHOR BIO...

Tad Stephens is a systems engineer for BEA based in Atlanta, GA. Tad came to BEA from WebLogic and has over 10 years of distributed computing experience covering a broad range of technologies including J2EE, Tuxedo, CORBA, DCE, and the Encina transaction system.

Eric Gudgion is a principal system engineer with the Technical Solutions Group. His background on mainframe systems comes from years of work within the field as a system programmer and system engineer for various Mainframe vendors.

CONTACT...

tad@bea.com
eric.gudgion@bea.com

"WebLogic Server is supported on the mainframe." I read the internal announcement and thought "Huh?" Why would someone want to deploy distributed Java applications on the big iron? What about training Java developers on the underlying mainframe systems?

However, the more I thought about it the more the strategy made sense. Why wouldn't I want to combine the industry's most reliable, most scalable, most mature application server with the hardware platform that best provides the same benefits?

Heterogeneity is a reality in today's computing world. IT organizations are responsible for applications in production on Windows, Unix, AS/400, and the mainframe. Critical data is stored in relational databases from multiple vendors on multiple platforms as well as in the form of structured data on the mainframe. While demand for aggregated data and customer information has increased, islands of applications and systems evolved that solve tactical business problems but provide little or no integration. Industry standards have partially addressed this problem; however, what is required is an application platform designed to bridge these gaps and offer the capability to aggregate business applications.

A key aspect of heterogeneity is platform independence. Traditional systems deployed solely on the mainframe have been extended to include a blended environment of PCs and Unix servers in addition to the mainframe. The mainframe itself has undergone numerous changes and is today based on IBM's S/390 hardware architecture, 64-bit machines that can run a variety of workloads, including Linux.

Historically, each hardware platform has dictated a different programming model, with little hope for application or component reuse. One advantage of the Java programming model is its natural platform independence; however, rewriting existing applications is not practical unless there is a corresponding business benefit that warrants such an effort. Tools that can solve business problems without a dependence on the underlying deployment hardware increase flexibility for deployment, decrease cost of development, and minimize technology risks.

BEA's strategy is to address the requirements of today's business environments through a comprehensive suite of products that address business-to-consumer, business-to-business, and business-to-employee scenarios. The BEA product line leverages a common platform to cover the Web interface through to back-office and existing systems. The fundamental component of this platform is WebLogic Server and its support of hardware spanning the needs of today's business environment, including Windows, a wide range of Unix and Linux systems, and even the mainframe via z/OS and z/Linux operating systems. Figure 1 illustrates BEA's platform support strategy.

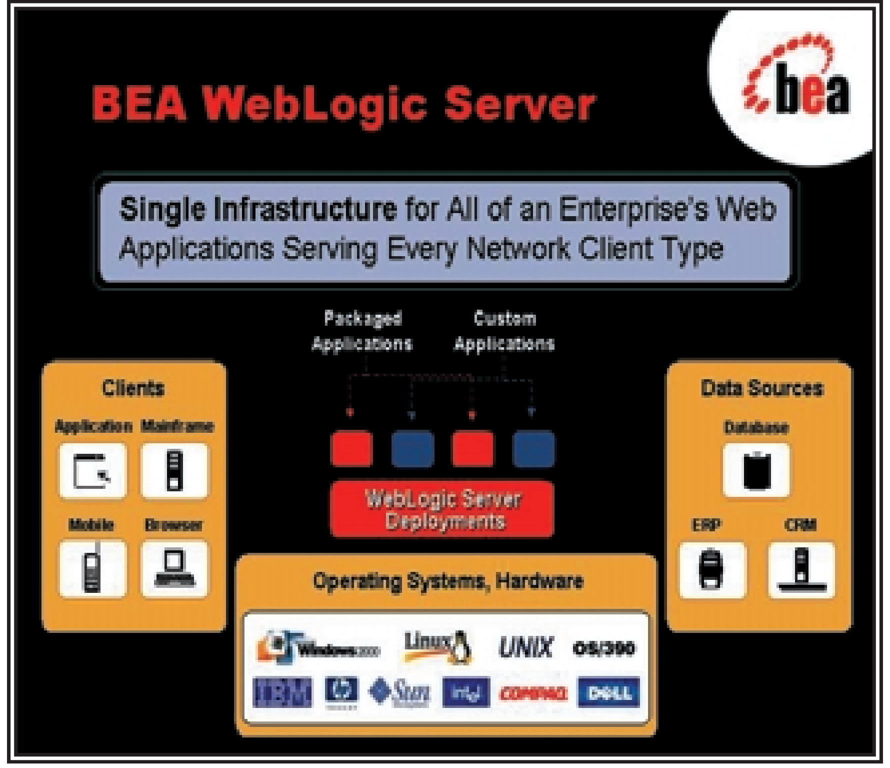
This article, the first in a series, will present the business and technical advantages gained when using WebLogic Server for mainframe J2EE development and deployment. The second article will cover the specifics of deploying J2EE applications to mainframe operating systems. The final article will address development and testing strategies and some lessons learned from BEA customers who are running production WebLogic–based applications on the mainframe.

## Business Drivers for WebLogic Deployment on the Mainframe

There are a number of business benefits for deploying J2EE applications on the mainframe. Deploying with WebLogic Server enhances these benefits in a number of key ways, including:

- Rewriting existing mainframe applications in Java for higher programmer productivity, flexibility, and elimination of dependence on a single vendor
- Consolidating Unix and Windows servers to z/Linux to lower total cost of ownership

FIGURE 1



BEA's platform support strategy

- Deploying new applications on existing mainframes (z/Linux and z/OS) for better resource utilization
- Leveraging business-contingency benefits through the mainframe qualities of service and operational properties to ensure that J2EE applications are always available
- Lowering costs by extending existing systems and applications when rewriting isn't practical or feasible

Let's consider each of these advantages in more detail.

### JAVA FOR PROGRAMMER PRODUCTIVITY

Corporate IT managers are always on the lookout for ways to lower costs and do more with less. On the mainframe this goal presents unique challenges. In most cases mainframe applications run critical business functions and reduced development costs can represent significant savings. However, the tools and techniques in use are measured in decades. Many of the efficiencies and advances in software development have not made it to the mainframe, which is inflexible, proprietary rather than based on open standards, and often ill-suited to the rapid changes required in today's IT environment.

**"Heterogeneity is a reality in today's computing world"**

For many years software and tool vendors have been touting various products and solutions to make developing applications on the mainframe more efficient, easier, and less costly. As a programming language, Java combines platform independence with a number of advantages to address this requirement: it's easy to learn, there are a number of training services readily available, and there are a broad range of tools and aids for the developer. Java also lays the foundation for Java 2 Enterprise Edition (J2EE), a suite of services and APIs for building applications that leverage Web access, database and transaction support, and a robust component model. The ideal model is one that provides the power, productivity, and platform of Java and J2EE in a mainframe deployment model. WebLogic Server to the rescue!

Another key advantage of running J2EE with WebLogic Server on the mainframe is the removal of any reliance on proprietary software packages. A traditional weakness of mainframe computing is the use of vendor-specific software languages, applications, and tools. Few programming environments on the mainframe support the notion of objects or components. The costs and risks associated with replacing a vendor's product can include major rewrites of existing applications or implementing totally new applications. Deploying on WebLogic Server alleviates this problem by delivering a standards-based, component foundation more akin to the "plug-and-play" models popular today.

### PLATFORM CONSOLIDATION

For as long as most of us can remember, folks have been looking for ways to move applications off the mainframe – minis, Unix servers, Intel servers running Windows, etc. However, a funny thing happened in the past few years as corporations struggled with the "sunk cost" tied to the big iron. Organizations found that it makes sense to consolidate applications onto the mainframe instead of the other way around. IBM has made great strides in modernizing the mainframe with support for Java and Linux, and the mainframe delivers an unmatched level of 24X7X365 availability. Many businesses find they can dramatically reduce their total cost of ownership by relocating distributed applications onto mainframe hardware servers.

Why has this happened? The primary reason is that the work done by multiple distributed servers can be consolidated onto a single mainframe. The mainframe supports applications that are vital to continuing business operations; thus, the operating costs associated with maintaining the mainframe are relatively fixed. On the other hand, adding more distributed Unix or Windows servers to a cluster or server farm usually requires adding more administrators to manage them. To expand an existing application or deploy a new application, the IT executive can either 1) buy more equipment and hire new people to manage it, or 2) deploy to the existing mainframe and leverage existing system management personnel. Although there often are other factors to consider, one trend that is gaining momentum is consolidation.

### BETTER RESOURCE UTILIZATION

Another key cost reduction driving application deployment on the mainframe is to more efficiently utilize the resources we already have available to us. We've discussed the reality of operational costs associated with maintaining the big iron. In addition, those costs are relatively fixed even if the hardware isn't running at maximum capacity. In many environments today the mainframe has underutilized resources – CPU cycles, memory, or disk – yet the organization's costs don't reflect this. More efficiently utilizing these resources lowers operational costs by doing more with less – the CIO's main objective.

In many production environments distributed resources are often underutilized. For example, a recent review of a production cluster of Unix

Sitraka

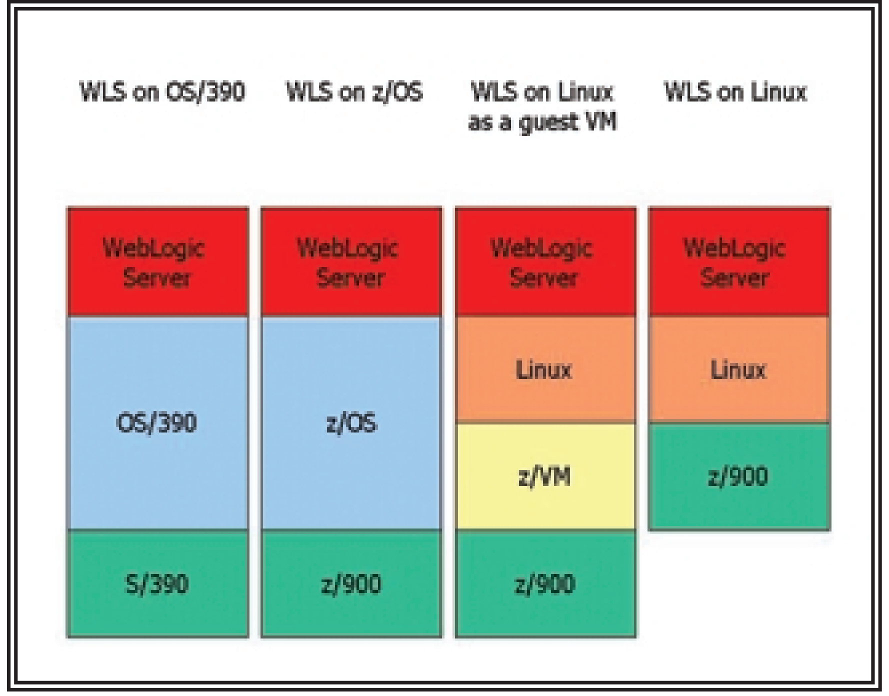
www.sitraka.com/performance/wldj

servers found that the servers were operating at 15–20%. If mainframe resources are available it's often easier to consolidate these under-utilized servers onto a logical partition under z/OS or z/Linux. While delivering the same overall performance, consolidating these servers can result in savings that include lower administrative costs; a simpler, less complex deployment environment; and high quality-of-service and operational properties.

## CLOSER ACCESS TO CRITICAL BUSINESS SYSTEMS

Contrary to some wishful thinking, the predominant hardware platform for many critical business systems in use today is still the mainframe. With the fears and hype surrounding Y2K behind us, there may be little incentive to replace existing applications as long as they deliver business value at minimal incremental cost. However, more and more personnel, systems, and customers require information stored in these legacy systems. Providing this access in a way that is secure, and transactional, without introducing unnecessary complexity, is key to delivering real business value.

FIGURE 2



Deployment models

WebLogic Server provides a number of key advantages that make it easier to access legacy systems and data from Web and distributed solutions. WebLogic Server makes J2EE, including Enterprise JavaBeans, available on the mainframe. Through J2EE, mainframe services can be exposed without requiring distribution of mainframe client APIs. Access is extended to Java applications via a stateless session EJB or message-driven bean. Web services permits access to these same EJB components from any distributed program via a SOAP call using an interface defined in WSDL. When deployed on the mainframe, Java components have "local" access to mainframe applications, without expensive network latencies or the complexity of heterogeneous hardware deployments.

In addition, the basic services of WebLogic Server on the mainframe can be extended with WebLogic Integration, providing standards-based application access using the J2EE Connector Architecture, business process management, data translation and transformation, and business-to-business messaging. WebLogic Integration is built on WebLogic Server and leverages the underlying J2EE services such as Enterprise JavaBeans and Java Messaging Service. WebLogic Integration is certified for deployment on the mainframe as well as on various Unix, Linux, and Windows systems.

## WEBLOGIC SERVER FOR THE MAINFRAME

WebLogic Server provides a complete suite of the J2EE standard for distributed Java applications. Included in the server is support for Java Server Pages and servlets; Enterprise JavaBeans; RMI for communication, transactions, naming and directory services; Java Messaging Service (JMS); and other J2EE facilities. The J2EE model is extended with key features that make it uniquely suitable for large-scale production deployments, including clustering, in-memory session and object persistence, entity bean caching, support for the J2EE security model, and a comprehensive administrative foundation including integration with third-party solutions.

WebLogic Server for the mainframe offers this same suite of services on a Java Virtual Machine running natively in either z/OS or z/Linux systems. Specifically, WebLogic Server supports four deployment models for the mainframe:

- Running directly on a virtual machine
- Running in a Logical Partition (LPAR) directly on the hardware
- Running in a Virtual Machine guest under Linux
- Running in a Unix or Windows NT/2000 environment with the mainframe as a data server

Figure 2 illustrates each of these deployment models.

An often asked question is why would someone want to run products from BEA on the mainframe? What does BEA know about parallel sysplex, about z/OS and z/Linux, about data sets and copy books, about VTAM and NCP gens, about using the Workload Manager? While BEA is the leader in distributed transaction processing, is there a compelling reason to run WebLogic Server on the mainframe? What about other companies and their application servers, particularly ones that have some history with the mainframe?

These questions actually point out many of the reasons BEA is the best choice for J2EE on the mainframe. WebLogic Server offers the following advantages for mainframe J2EE solutions:

- Common code base across heterogeneous hardware platforms
- Application performance, including tools and utilities to aid in tuning
- Extensible architecture to ease integration with existing applications and databases

Let's consider each of these advantages.

A key advantage of WebLogic Server is that the product running on the mainframe is from the same code base as other environments. Java was designed to offer platform independence and WebLogic Server builds on this. Applications developed for WebLogic Server in a distributed environment can be deployed directly on the mainframe without requiring any code modifications or detailed knowledge of mainframe systems. Developers familiar with current Java development tools and strategies can build applications for deployment directly to the host. Existing design patterns and best practices can be implemented regardless of the underlying operating system and hardware. Since the code base for WebLogic Server is the same across all platforms, the product's maturity and stability can be leveraged to ensure high availability, reliability, and scalability. The result is lower cost of development and training, and lower risk associated with the corresponding deployment.

Another advantage is in the area of application performance. Benchmarks have shown that applications that require a large and relatively complex deployment on distributed servers can be run on a few mainframe partitions, leveraging existing hardware and system management resources. WebLogic Server can be configured to use the Workload Manager (WLM) on the mainframe to efficiently manage the deployment for maximum performance. Third-party tools, such as Wily Technologies' Introscope, Candle, CA Unicenter, BMC, and Velocity Software can be used to tune the application's performance, identify bottlenecks, and diagnose application problems.

One of the benefits of using WebLogic Server for J2EE applications is the clustering support. Multiple replicas of an application can be clustered together for redundancy, failover, and scalability. Application objects deployed on the mainframe can be clustered with other servers, including heterogeneous hardware types. Within the same cluster an application can be deployed on the mainframe using z/Linux, Solaris, HP-UX, Windows 2000, and others. WebLogic Server also provides an administrative domain in which all the distributed resources can be managed as a single system image.

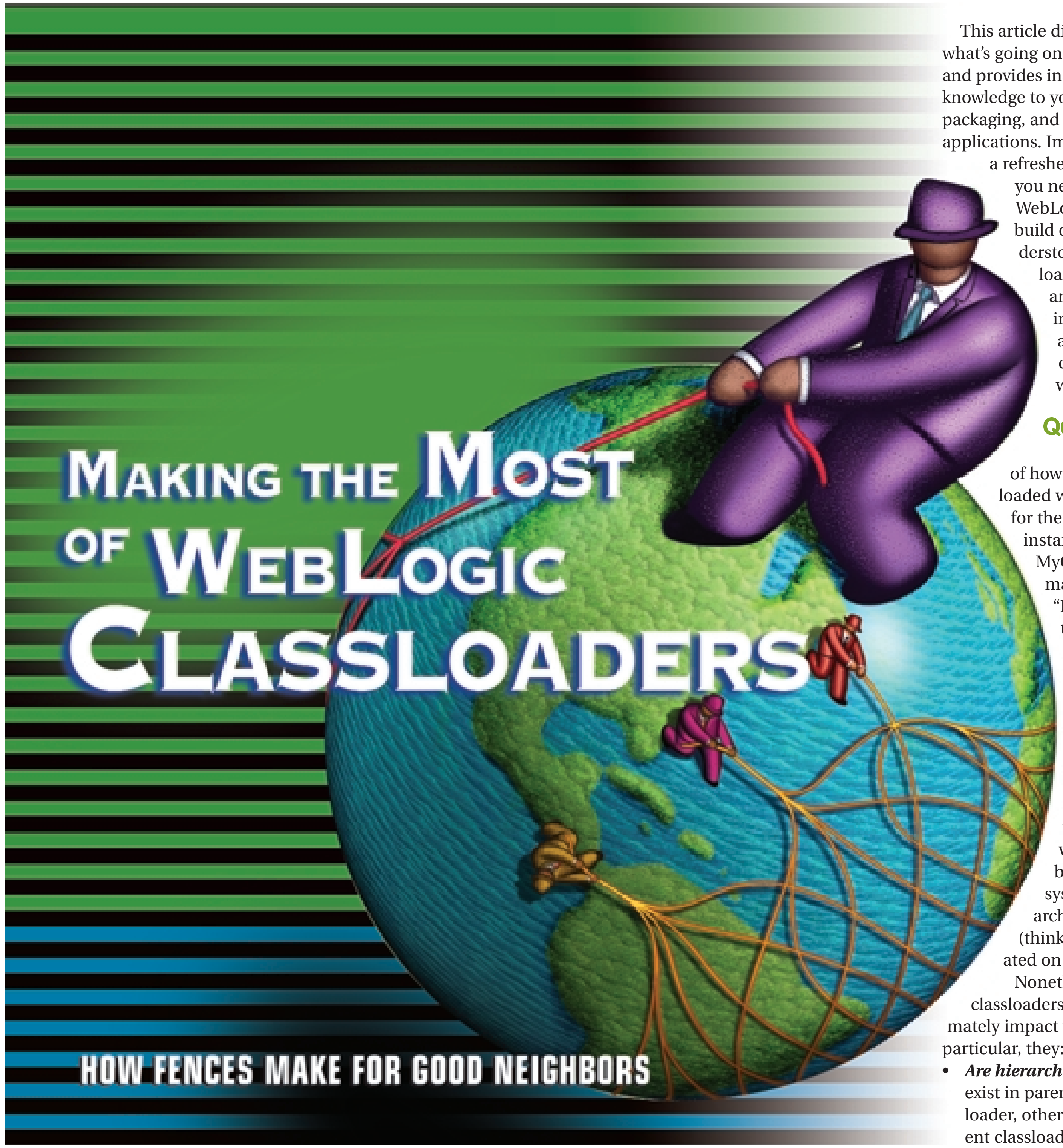
A third advantage is the ease with which it can be integrated with the existing resources and applications. WebLogic Server can be integrated with the RACF security environment to offer consistent services and can be configured through the Tivoli Policy Director. It integrates with MQSeries by including the MQ Java APIs and coding directly to the MQ_* verbs in the application code. In addition, WebLogic Server can include MQSeries in distributed transactions with full two-phase commit support. It coordinates these transactions using the industry-standard XA protocol. WebLogic Server can be integrated with mainframe databases using the vendor-provided JDBC drivers or using products such as the CrossAccess eXadas Data Integrator or Neon Systems' ShadowDirect. Finally, WebLogic Integration extends the WebLogic foundation for adapters based on the J2EE Connector Architecture, business process management, data translation and transformation, and business-to-business message exchanges.

**"Historically, each hardware platform has dictated a different programming model, with little hope for application or component reuse"**

## Summary

We've examined some of the business benefits of deploying J2EE applications on the mainframe, including the use of Java tools and patterns for developer productivity and lower cost, server consolidation to lower total cost of ownership, consolidation of distributed servers onto a mainframe with available resources, and shortening the access to critical data and business applications. We've briefly covered WebLogic Server on the mainframe and some of the reasons to select BEA for mainframe deployment of J2EE-based applications.

The next article in this series will detail the specifics of deploying WebLogic Server in mainframe environments, including z/OS and z/Linux, and detail the useful tools and services available. The third article will address some of the production applications using WebLogic Server on the mainframe, as well as some tips and helpful hints.



# Making the Most of WebLogic Classloaders

How fences make for good neighbors

BY
JOHN MUSSER

AUTHOR BIO...
John Musser is a consultant who over the past 20 years has built software ranging from Wall Street trading systems to games for Electronic Arts. He is currently lead architect on an e-logistics system and teaches software development at Columbia University.

CONTACT...
jrmusser@hotmail.com

As a Java developer, have you ever found yourself running into what might be politely called 'issues' related to the CLASSPATH and class loading? If you haven't, you're one of the few. This is one of the most notorious sources of developer aggravation in Java development. Now J2EE has added a new set of wrinkles to this phenomenon.

This article dispels some of the mystery of what's going on behind the classloader curtain and provides insights into how you can use this knowledge to your advantage when designing, packaging, and deploying your WebLogic Server applications. Important items we'll cover include a refresher on class-loading fundamentals you need to be aware of; how WebLogic's own custom classloaders build on these basics; the often misunderstood relationship between classloader hierarchies and EARs, EJBs, and WARs; techniques for packaging and deploying utility classes; and a simple but handy tool for diagnosing how classes are loaded within your applications.

## Quick Classloader Review

Let's begin with a short review of how classloaders work: Classes are loaded when new bytecode is executed for the first time, either via a new class instance as in "MyClass x = new MyClass();" or by a first reference made to a static class, "MyStatic.staticMethod();". When this occurs, the JVM relies on classloaders – either the standard Java classloaders that ship with the runtime or custom classloaders – to load the bytecode in memory (all classloaders are subclasses of java.lang.ClassLoader). While there's a standard format for how the class is loaded in memory, where a classloader loads the bytecode from, be it from the file system (your typical .class files), an archive (JARs), a network socket (think applets and RMI), or even generated on the fly, is left to each classloader.

Nonetheless, there are certain rules all classloaders follow, many of which can ultimately impact your WebLogic applications. In particular, they:

- ***Are hierarchical:*** As of JDK 1.2 all classloaders exist in parent-child relationships. Each classloader, other than the initial loader, has a parent classloader. The root parent classloader, built into the virtual machine, is called the "bootstrap" (or "primordial") classloader.
- ***Can have siblings:*** Given that any classloader may have multiple child classloaders, it's possible for these children to have sibling classloaders at their same level in the hierarchy.
- ***Delegate load requests:*** Before attempting to load classes themselves, children delegate all load requests to their parent. And each parent will delegate to its parent until reaching the root of the hierarchy. Only if any parent does not first resolve a request does a child attempt to load the requested class.
- ***Have visibility constraints:*** Child classloaders, and the classes loaded by these loaders, have access to (or can "see") classes loaded by their parents, but parents cannot access classes loaded by their children nor can children see classes loaded by sibling classloaders (as with delegation, this only goes up the hierarchy, not down).
- ***Cannot unload classes:*** Classes cannot be unloaded directly by their classloader, but classloaders themselves can be unloaded. This effectively unloads any classes loaded by that classloader. More about this soon.

## Classloaders in a WebLogic Application

In order to support the J2EE standard and useful features such as hot deployment, WebLogic Server defines and initializes a number of its own custom classloaders. These classloaders work under the covers within the application server to manage loading and unloading your classes, EARs, EJBs, and WARs (and if you're interested in writing your own specialized classloader, you might want to take a look at Philip Aston's excellent article on this in April's ***WLDJ*** *[****Vol. 1, issue 4****]*).

Always keep in mind that class loading and deployment are inseparable. When you deploy your modules to WebLogic, each is typically loaded in its own dynamic classloader. If you're deploying a single EJB JAR file, it will get its own classloader. When you deploy a single WAR file, it gets one too. And when deploying an EAR file, WebLogic will create, depending on what's in your EAR file, an entire hierarchy of classloaders.

How does this work? What rules does WebLogic follow? The principal concept is that of application deployment units such as an EAR, an EJB, or a WAR. When any of these are deployed separately they are loaded in separate sibling classloaders. If EJBs and WARs are grouped into an EAR, they get a hierarchy of classloaders: at the top is the application classloader, which loads the EJBs and JARs referenced in its manifest file's classpath (more on this later), and then classloaders for each WAR file.

Figure 1 illustrates a sample WebLogic Server deployment. Each shadowed box in the WebLogic section represents an independent classloader instance. At the top level we have two EARs, one EJB JAR, and one WAR. Each of these is loaded by a sibling classloader. Within the first EAR are three EJB JARs (each with potentially multiple EJBs) and manifest-referenced utility files that share a single classloader and two Web applications that each get another classloader. Because of the established classloader hierarchy, the JSPs, servlets, and utility classes in both WARs can directly use any of those EJBs. In the second EAR, a simpler hierarchy allows the same interactions as the first, but these components are isolated from those in the first EAR.

In this arrangement, if either of the independently deployed modules, the EJB or the WAR, needs to access the other, this must be done via remote interfaces. This means that the EJB home and remote interfaces must be packaged into the calling modules and all communication may require more overhead than if they were packaged together (in which case WebLogic can perform various call optimizations such as using call-by-reference as it does for new EJB 2.0 local interfaces).

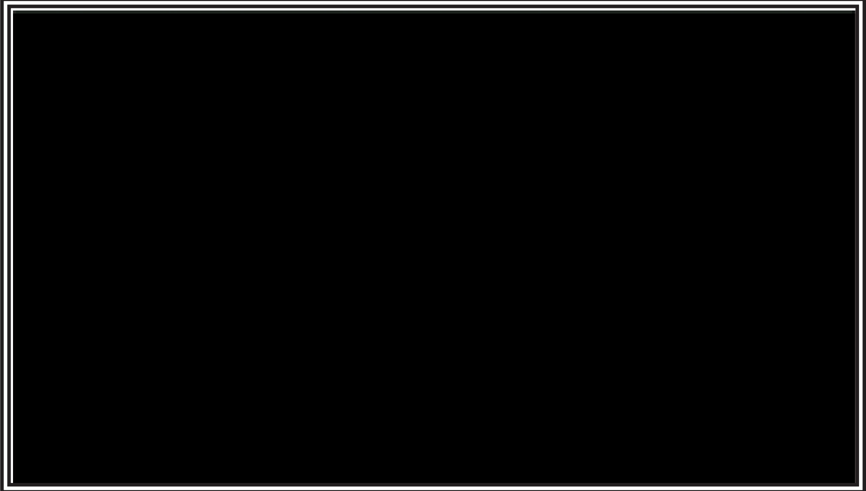
Table 1 lists the primary classloaders in a typical WebLogic EAR application. These are listed in order of creation from the primordial bootstrap classloader through various child classloaders down to the Web classloader. A couple of notes here: each of these classloaders may or may not be a distinct ClassLoader subtype or may be multiple instances of the same class configured differently. WebLogic may internally use additional classloaders (the details of which don't matter to us here); the native JVM classloaders are not set up to be reloadable but WebLogic's are (thus the familiar problem of having to reboot the application server if classes loaded by the non-reloadable classloaders have changed).

## What This Means

The consequences of this classloading scheme include:

- ***Isolation***: The positive, deliberate side effect of the lack of sibling classloader visibility is isolation – classes loaded by different classloaders at the same level do not, and cannot, directly access one another without explicitly including the other's remote interfaces. You can see an advantage of this sand boxing by looking at how WebLogic handles EAR files: each is run in its own classloader instance and can be deployed and redeployed independently. This means that when you and I both deploy our separate applications to Server A, we can do so in parallel without worrying about a host of issues that might occur if we shared the same classloader. As the old adage goes, fences make for good neighbors.
- ***Visibility:*** A key benefit of the hierarchical approach to classloader visibility is that JSPs, servlets, and other Web application components have direct access to all EJBs within the larger application. This both simplifies deployment and gives the application server more opportunities to optimize the performance of calls between components (covered in more detail below).
- ***Namespaces:*** Class instances within the JVM are identified by the combination of the full classname and the classloader that loaded it. This namespace identification can come into play in your WebLogic applications. For example, think about how common it is for a Web application to start from a JSP or servlet with a standard name like index.jsp. That's fine. But what happens when you have two Web apps (two WAR files) within the same EAR and both of those apps have that innocuously named JSP or servlet? Without the benefit of classloader isolation, after the compiler finishes turning those source files into compiled servlets, a namespace or class-loading issue occurs because they both resolve to the same name, resulting in only one being loaded. Regardless of what happens in that scenario, it won't happen in WebLogic because each WAR is loaded in its own classloader and even if the JSPs or servlets have the same name, each exists independently as far as the classloaders are concerned.
- ***Performance:*** Keep in mind that class loading, or more generally, packaging, impacts performance. Components deployed together in an EAR allow the WebLogic Server to optimize performance by reducing RMI call overhead as well as allowing you to use the new EJB 2.0 local interfaces. And it's through the default

FIGURE 1



Classloaders in WebLogic

TABLE 1

| CLASSLOADER | CREATED BY | RELOADABLE | DESCRIPTION |
|---|---|---|---|
| Bootstrap | JVM | No | Loads JDK internal classes, java.* packages (as defined in the sun.boot.class.path system property – typically loads rt.jar and i18n.jar). |
| Extensions | JVM | No | Loads JAR files from JDK extensions directory (as defined in the java.ext.dirs system property – usually lib/ext directory of the JRE). Allows extending JDK without adding to the classpath. |
| System | JVM | No | Loads classes from system classpath (as defined by the java.class.path property, which is set by the CLASSPATH environment variable or -classpath or -cp command line options). |
| Application | WebLogic | Yes | Loads EAR, EJB, or WAR JAR files depending on the hierarchy (as per Figure). |
| Web | WebLogic | Yes | Loads WAR JAR files within EARs. |

The classloader hierarchy

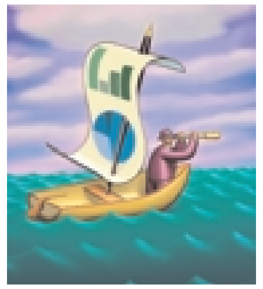
# Chart Your Course to Success in IT...

...order your copy of **Java Trends: 2003**

**Available July 1***

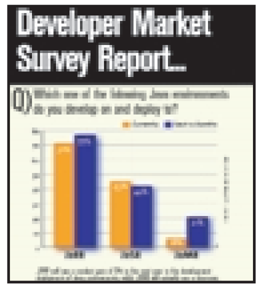
EnginData's research is invaluable to leading IT vendors, integrators, Fortune 500 companies, trade-show organizers, publishers, and e-business organizations worldwide.

Don't go astray. In the vast sea of Internet technology, market conditions change constantly. Will Java remain the hot platform it is today? Will C# rapidly gain ground? What are the world's foremost Java developers aiming their sights toward? Which companies come to mind when planning their next project? How can their thinking direct your efforts?



**EnginData Research** has studied the IT industry extensively, spotting many key trends and alerting industry leaders along the way. In the first quarter of 2002, they embarked on their most challenging mission ever: the most in-depth study ever done of the Java development marketplace.

After months of analyzing and cross-referencing more than 10,500 comprehensive questionnaires, the results are now available.



**Here's just a sample of the critical data points the Java report delivers...**

- ✔ Current and projected usage of languages and platforms
- ✔ Multiple rankings of hundreds of software vendors and tools
- ✔ Types of applications being developed
- ✔ Databases being used
- ✔ Purchasing patterns
- ✔ Company size
- ✔ Development and purchasing timelines
- ✔ Perceptions and usage of Web services
- ✔ Preferred Java IDE
- ✔ J2EE, J2ME, J2SE usage comparisons

As an IT specialist, **EnginData Research** focuses exclusively on three leading drivers in IT today – Java, Web services, and wireless development. Coming soon, our Web services survey will deliver such critical knowledge as:

- ✔ Time frame for developing Web services – enabled apps
- ✔ Percentage of apps with Web services today
- ✔ Sourcing and hosting Web services
- ✔ Perceived leaders in Web services tools and solutions

Navigate your company through the challenging IT marketplace with the trusted knowledge and intelligence of **EnginData Research**. Order your copy of the 2002–2003 Java market study by calling Margie Downs at 201-802-3082, or visiting our Web site.

engindata RESEARCH

**www.engindata.com**

*A limited number of preview copies will be available at our booth (#624) at JDJEdge International Java Developer's Conference & Expo, June 24–27, at the Jacob Javits Convention Center in New York.

class-loading behavior (whereby loading is first deferred to parents as opposed to having each classloader load its own copy of the same class) that overall memory consumption is reduced.

- ***Singletons:*** If you use the singleton design pattern in your application you need to be aware that classloaders matter because singletons (at least the run-of-the-mill, GoF design pattern variety such as those accessed with a getInstance-type method) exist not just within a single JVM but within a single classloader. Thus, if the singleton is loaded at the EAR level (such as from a utility library included at that level and not at the system-wide classpath level) by EAR 1 and EAR 2, then each application will get a different copy of that singleton. And there are of course other caveats to be aware of with singletons: clustering can throw a monkey wrench into the works, and in a world of distributed objects other patterns may be necessary, such as using JNDI to manage a singleton EJB (see some active discussions on this topic at www.theserverside.com).
- ***Deployment:*** Regardless of how you deploy your application module – by using the console, by WebLogic's automatic hot deployment mode (quite useful during development), or through command-line utilities – the server will create a new classloader instance for loading this application. If you choose to deploy EJBs and WARs separately and you want your Web application to use those EJBs, you'll need to include the EJB home and remote interfaces in your WAR file. These are loaded by sibling classloaders and you'll need to make remote calls across them. Similarly, if you want to reference EJBs across EAR files you'll need to package the home and remote interfaces from the one EAR into the other (or you could try the approach of generating client JARs for your EJBs by using the ejb-client-jar tag in ejb-jar.xml and add references to those JARs in the manifest classpath – see below for more about the manifest).

## Handling Utility Classes

Nearly every application, large or small, needs utility classes, either your own or from one or more third parties. Given what we've seen in looking at classloaders and packaging, where can and should these go? Table 2 outlines some of the choices you have as well as their attendant pros and cons.

To elaborate a bit on the last item in Table 2: because they are JAR files, all J2EE modules have a MANIFEST.MF file in their META-INF folder. The J2EE specification allows classes in one JAR – such as an EJB or WAR archive – to access other JARs by explicitly adding their names to a special "Class-Path:" entry in the manifest file (this is built upon the "extensions mechanism" introduced in JDK 1.2). By using this entry you can tailor EAR class-loading behavior and can modularly deal with many of the classpath issues mentioned earlier.

In order to make this work, first create a text file specifying a list of JAR filenames as relative URLs separated by spaces and then include this file when building your deployment. Here's what a typical entry might look like:

```
Manifest-Version: 1.0
Class-Path: myparser.jar vendorlib.jar
```

And here's an example of how you might add it when creating an EJB JAR (presuming you have the above line in a file called mymanifest.txt):

```
jar cmd mymanifest.txt myejb.jar com META-INF
```

Now the classes in myejb.jar have access to myparser.jar and vendorlib.jar because they've been loaded with this EJB JAR into its same classloader. Within the context of an EAR file, the JARs referenced within EJBs and WARs are loaded at the same level as EJBs and are available to all beans and Web modules within the application. (Note what happens: WebLogic takes the transitive closure of all manifest classpath entries specified in the EJB and WAR archives for a given EAR

TABLE 2

| UTILITY CLASS LOCATION | PROS | CONS |
|---|---|---|
| WEB-INF/classes | Useful if 'loose' individual classes are needed only by this Web application (providing isolation). Typical location for servlets. | Cannot be shared by multiple Web modules or by EJBs and other classes within application. |
| WEB-INF/lib | Same as above with benefit of slightly easier manageability of JARs over individual class files. | Same as above. |
| Class files within EJB JAR | Fine-grained deployment option. | Visible to all modules in application (possible namespace conflicts). Individual files somewhat harder to manage than JARs. |
| Class files on global classpath | Visible to all applications. Does not need multiple copies deployed. Updating in one location updates for all applications. | Must reboot server when classes change. Less isolation – visible to applications that may not use them (or inadvertently use them). Classes loaded globally cannot see classes in subsequently loaded modules. Must be set up on each server separately. |
| JAR on global classpath | Visible to all applications. Keeps classes grouped within their own JAR. | As above but slightly more manageable because they're packaged as a unit. All applications will share the same version (which can cause problems during upgrades). |
| JAR in EAR file | Keeps classes grouped within their own JAR. Visible to all modules within a single application. Provides isolation between applications. | Must be included in each application that wants to use those utilities (duplication). |
| JAR referenced via Manifest.mf | Modular and flexible. Allows explicit references to JARs required for this specific module. Can resolve circular references. | Must be manually added to the file and explicitly added to the appropriate JAR's manifest. Lack of tool support for editing and maintaining this entry. Supports references to JARs but not directories. |

Utility class placement options

and loads them at the same top level within that EAR.)

Regardless of which classloaders and packaging technique you use, there's one other caveat worth mentioning and that's the issue of class-loading order. Based upon the interdependencies among your utilities, JARs, EJBs, etc., you may find yourself needing to tailor what's placed where solely to satisfy load-time ordering. For example, utility classes needed by WebLogic itself, such as JDBC drivers, must be specified on the system classpath so that they're available when it starts. Complications can also arise in the case of startup classes and their dependencies (this is covered in more detail on BEA's site at http://edocs.bea.com/wls/docs61/programming/packaging.html).

Overall, here are a few general guidelines on utility classes: packaging in JAR files is generally more manageable than using individual class files. If the utilities are needed only by a single Web module then put them in a jar in its WEB-INF/lib folder; if the utilities are to be shared by multiple components or modules within a single application then keep them as a JAR at the application level; and finally, if you need to use the utility classes across applications, first try to use the manifest classpath approach and if that's not feasible, then use the global classpath. You can find a good discussion of this as well as J2EE packaging in Tyler Jewell's articles on this topic at www.onjava.com/pub/a/onjava/2001/06/26/ejb.html.

## Debugging Class-Loading Issues

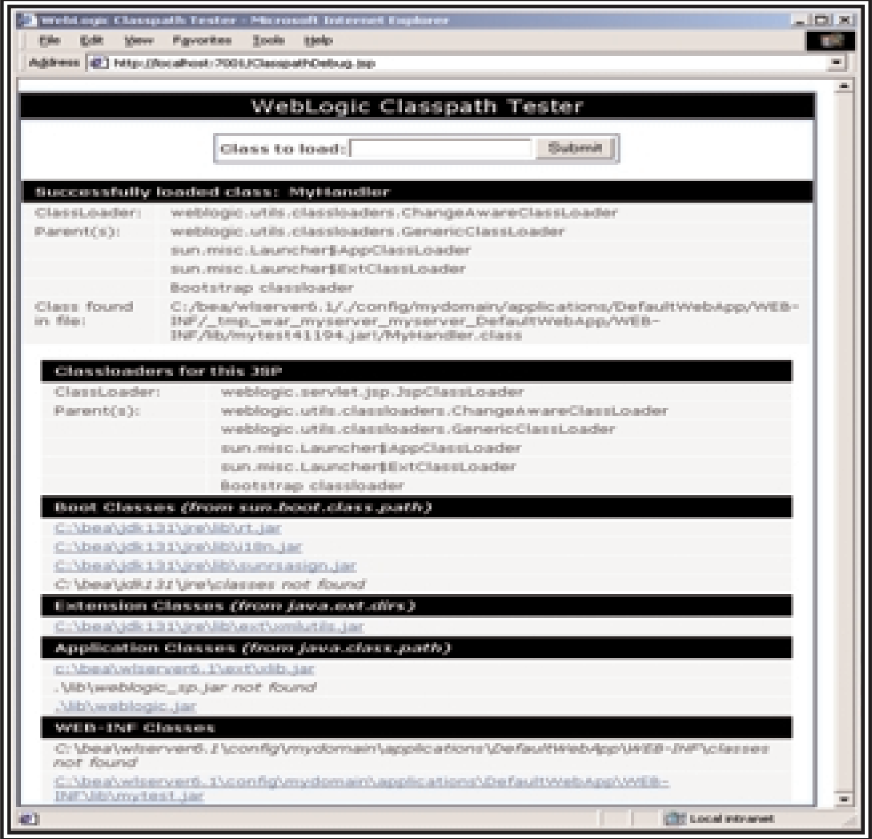
As with any other development issue, class loading at times needs to be debugged. Here are a few suggestions:

- ***Plan:*** Invest some up-front effort in considering how many JARs to build (or WARs, EJB JARs, and EARs as the case may be), their names, their location, and which classes go into which archives.
- ***Refactor:*** The layout and structure of this runtime packaging, like all software, can be refactored. In this case, refactoring means modifying and refining your implementation of the aforementioned packaging plan. Yes, this can take some effort, but so does any sort of refactoring (and because debugging class-loading issues are inherently "not fun" this serves as a deterrent to treating this step too lightly).
- ***Diagnose:***
  - Walk through your shell environment, startup scripts, and configuration files to get a handle on what's being set where. Often it makes sense to consolidate setting file locations and paths in a single place such as one startup script (which itself does not assume any environment variables have been set or explicitly overrides them).
  - Given that some things won't be apparent until runtime, have your run script print out to the console all settings prior to server launch.
  - Write a simple Java debug routine that displays relevant values (including system properties such as "sun.boot.class.path").
  - Use a tool like the JSP page described below.

As an example, say you have a utility class, MyHandler, which for some reason (deliberate or accidental) appears more than once in your environment; it could be that the version loaded at runtime may not be the one you expect. If in this scenario your servlet (in a WAR file) instantiates MyHandler and the only physical location of MyHandler.class is under the WEB-INF/classes directory of that WAR, then this will be the one that gets loaded. But, if you have another version of MyHandler.class somewhere on the system classpath, it will be this one that gets loaded, regardless of the fact that there's one right there in the same archive as your program.

Why does this happen? Because of classloader delegation. As noted earlier, all classloaders first delegate load requests to their parent, recursively up the tree, allowing each parent the opportunity to handle the request. In this case, because the system classloader was able to resolve this class reference, the child WAR classloader never had the chance to resolve and load this class itself. The nuances of such behaviors can appear to be quite idiosyncratic even if they're not and cer-

FIGURE 2


WebLogic Classpath Tester - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:7001/ClasspathDebug.jsp
Class to load:
Submit
Local intranet

WebLogic Classpath Tester

Successfully loaded class: MyHandler

```
ClassLoader:   weblogic.utils.classloaders.ChangeAwareClassLoader
Parent(s):     weblogic.utils.classloaders.GenericClassLoader
               sun.misc.Launcher$AppClassLoader
               sun.misc.Launcher$ExtClassLoader
               Bootstrap classloader
Class found    C:/bea/wlserver6.1/./config/mydomain/applications/DefaultWebApp/WEB-
in file:       INF/_tmp_war_myserver_myserver_DefaultWebApp/WEB-
               INF/lib/mytest41194.jar!/MyHandler.class
```

Classloaders for this JSP

```
ClassLoader:   weblogic.servlet.jsp.JspClassLoader
Parent(s):     weblogic.utils.classloaders.ChangeAwareClassLoader
               weblogic.utils.classloaders.GenericClassLoader
               sun.misc.Launcher$AppClassLoader
               sun.misc.Launcher$ExtClassLoader
               Bootstrap classloader
```

Boot Classes *(from sun.boot.class.path)*

```
C:\bea\jdk131\jre\lib\rt.jar
C:\bea\jdk131\jre\lib\i18n.jar
C:\bea\jdk131\jre\lib\sunrsasign.jar
C:\bea\jdk131\jre\classes not found
```

Extension Classes *(from java.ext.dirs)*

```
C:\bea\jdk131\jre\lib\ext\vmutils.jar
```

Application Classes *(from java.class.path)*

```
c:\bea\wlserver6.1\ext\xlib.jar
.\lib\weblogic_sp.jar not found
.\lib\weblogic.jar
```

WEB-INF Classes

```
C:\bea\wlserver6.1\config\mydomain\applications\DefaultWebApp\WEB-INF\classes
not found
C:\bea\wlserver6.1\config\mydomain\applications\DefaultWebApp\WEB-
INF\lib\mytest.jar
```

Class-loading diagnostic tool

tainly vary across application servers and server versions. If you have multiple versions of the same libraries (such as might happen with common utilities such as XML parsers), be prepared to do a little investigating.

Also keep in mind that because standard classloaders (including WebLogic's) delegate all load requests to their parents before their own searches, anything deployed on the system classpath can't be redeployed. As just shown, these classes will always be loaded by the JDK system classloader, which does not support reloading. Thus, anything you want to be redeployable without having to restart the server must only be contained within one of the dynamically loadable units such as EARs, EJB-JARs, and WARs.

One last debugging note: Classes with no visibility modifier before the class declaration are deemed to be "package level" or "package private". In order for another class to access these classes, they must be in the same package and loaded by the same classloader. If this isn't the case, the caller will get an IllegalAccessException.

## J2EE Class-Loading Diagnostic Tool

In order to diagnose and debug the subtleties of class-loading issues, I've developed a small JSP utility page. Just drop this JSP into your default Web app directory or include it with a specific application to run some quick tests. It has two primary functions: the first is to allow you to specify a class to load (typically a utility class, but it could be any class that can be instantiated with a direct newInstance() call) and show you the classloader that loaded your class, the classloader hierarchy of ancestors to that classloader, and the file from which your class was loaded.

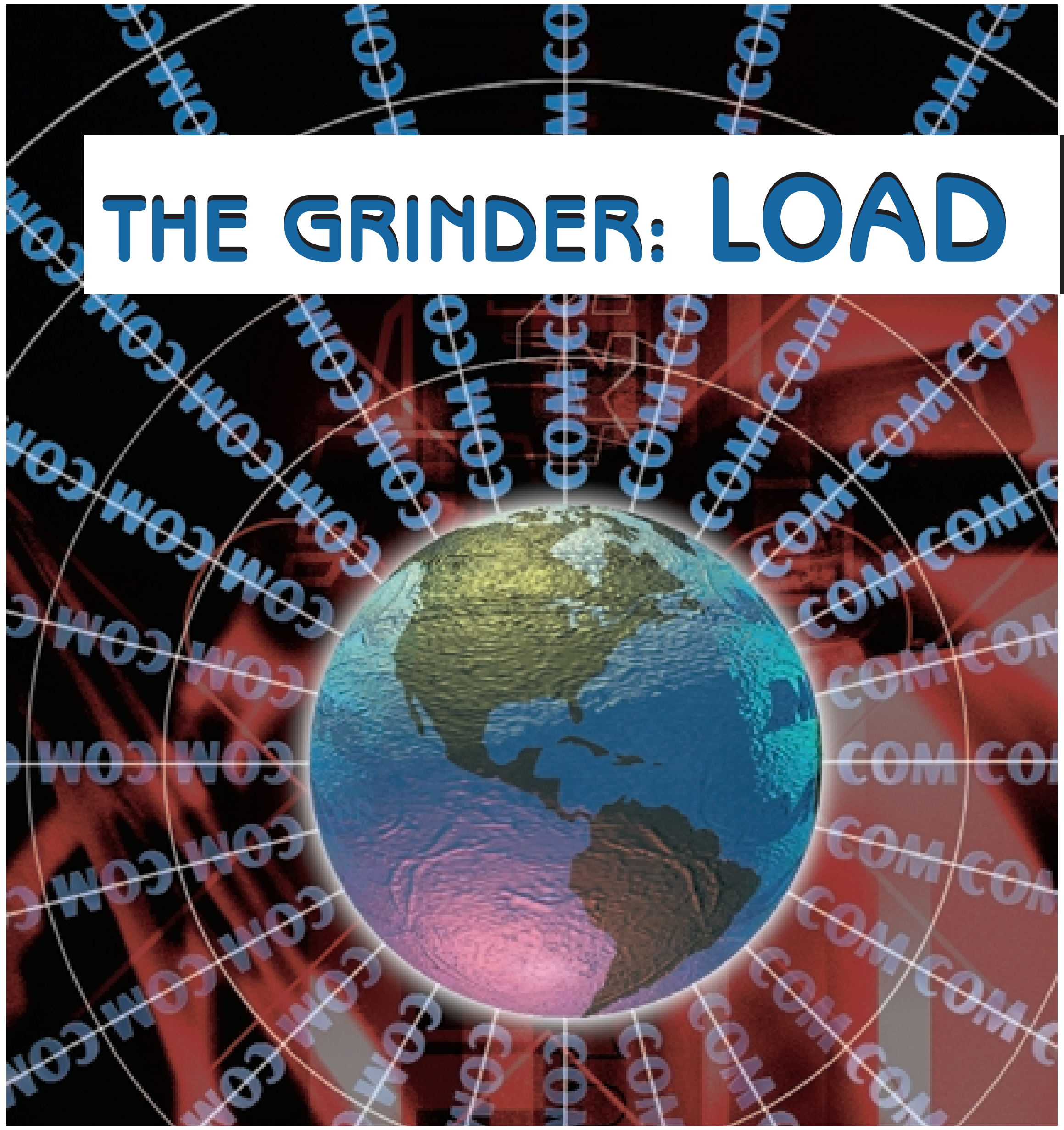
The second function is to outline in a table a number of the standard locations searched by the various classloaders. This includes the locations searched by the standard Java classloaders (such as the CLASSPATH or, more specifically, the locations identified in the system properties). If you are running the server on your local machine and can click on the names of any JARs, you might want to open them and view the contents.

You can get a sense of what this tool provides from Figure 2. In this example, the page has been reloaded after I specified that I wanted my utility class, MyHandler, to be loaded. At the top it shows how and where it found this class, and below it shows an outline of each set of files and directories searched by various Java and WebLogic classloaders. [Two notes: 1) the ClassLoader method getParent() will return null to represent the bootstrap classloader, which explains why a specific classname isn't given in this case; and 2) this utility doesn't try to dig into things like the manifest files or other bean references so it may not display all references in the lower table].

When the tool is unable to locate one of the standard locations (such as WEB-INF/lib), it will print a "not found" message. In this case, the missing directory is perfectly acceptable given that those locations are optional. On the other hand, for items derived from your CLASSPATH, this can indicate a problem such as a JAR file that's specified in the environment but doesn't really exist at that location. This happens as JARs are moved around and can be a handy way to catch these errors (in the example pictured, under Application Classes the file Weblogic_sp.jar is not found – the right location for this should be determined or it should be removed from the classpath).

## Conclusion

Keep in mind that the class-loading mechanisms described here are applicable to WebLogic Server 6.1 (and likely the final 7.0 release). These behaviors have changed over time as both WebLogic and the J2EE specification have evolved. Additionally, each application server vendor may implement its class-loading mechanisms differently as long as it conforms to the requirements outlined in the J2EE 1.3 specification. This means that other servers, while they may conform to the behavioral requirements defined by the J2EE spec, employ class-loading schemes that vary from that used by WebLogic.



# THE GRINDER: LOAD TESTING FOR EVERYONE

## OPEN-SOURCE PERFORMANCE-TESTING TOOL

BY
**PHILIP ASTON**

**AUTHOR BIO...**
Philip Aston is a senior consultant for BEA Professional Services, specializing in WebLogic Server. He also maintains The Grinder, an open-source load-testing tool at http://grinder.sourceforge.net.

**CONTACT...**
paston@bea.com

The Grinder is an easy-to-use Java-based load generation and performance measurement tool that adapts to a wide range of J2EE applications. If you have a J2EE performance measurement requirement, The Grinder will probably fit the bill.

Paco Gómez developed the original version of The Grinder for *Professional Java 2 Enterprise Edition with BEA WebLogic Server* (Wrox Press, 2000). I took ownership of the source code at the end of 2000 and began The Grinder 2 stream of development. The Grinder is freely available under a BSD-style license.

This article will introduce only the basic features of The Grinder. I encourage you to download the tool and try it out. The recently published *J2EE Performance Testing* by Peter Zadrozny, Ted Osborne, and me (Expert Press, 2002) contains much more information about The Grinder.

## Where to Obtain The Grinder

You can download The Grinder distribution from The Grinder home page at http://grinder.sourceforge.net. The examples in this article were run using The Grinder 2.8.3.

There are some mailing lists that you can join to become a part of The Grinder community:

- ***grinder-announce:*** Low-volume notifications of new releases
- ***grinder-use:*** The place to ask for help
- ***grinder-development:*** For those interested in developing The Grinder

## So, What Is The Grinder?

In short, The Grinder is a framework for generating load by simulating client requests to your application, and for measuring how your application copes with that load.

Typically, you will have begged, bought, or borrowed a number of test-client machines with which to test your application. You can use The Grinder console to control many processes across your test-client machines, each running many threads of control. The Grinder is a pure Java application, so there's a wide variety of platforms that you can use.

Three types of processes make up The Grinder:

- ***Agent processes:*** A single agent process runs on each test-client machine and is responsible for managing the worker processes on that machine.
- ***Worker processes:*** Created by The Grinder agent processes, they are responsible for performing the tests.
- ***The console:*** Coordinates the other processes and collates statistics.

Each of these processes is a Java Virtual Machine (JVM) and can be run on any computer with a suitable version of Java installed.

To run a given set of tests, an agent process is started on each test-client machine. This process is responsible for creating a number of worker processes. Each loads a plug-in component that determines the type of tests to run and then starts a number of worker threads. For example, with the provided HTTP plug-in each test corresponds to an HTTP request to a URL. Each of the worker threads uses the plug-in to execute tests.

The *grinder.properties* file is a configuration file that is read by the agent and worker processes, and the plug-in. This file contains all the information necessary to run a particular set of tests, such as the number of worker processes, the number of worker threads, and the plug-in to use. For most plug-ins, the file also specifies the tests to run and can be thought of as the "test script." For example, when using the HTTP plug-in, the grinder.properties file contains the URL for each test.

The agent process and the worker processes read their configuration from grinder.properties when they are started (see Figure 1). I usually put the grinder.properties file on a shared network drive so I don't have to copy it to each of the test-client machines.

The net effect of this scheme is to allow the easy configuration of many separate client contexts, each of which will run the same set of tests against your server or servers. Each context simulates an active user session. The number of contexts is given by the following formula:

(Number of agent processes) x (Number of worker processes) x (Number of worker threads)

## The Console

The Grinder console (see Figure 2) provides an easy way to control multiple test-client machines, display test results, and control test runs.

The console is used to coordinate the actions of the worker processes by sending them start, reset, and stop commands. IP multicast is used to broadcast the commands simultaneously to processes running on many machines. The worker processes send statistics reports to the console, which combines these reports to produce graphs and tables showing test activity. The results of a particular test run can be saved for further analysis.

The console also calculates and displays derived statistics. A key derived statistic that the console can calculate, but the individual worker processes cannot, is a combined transactions per second (TPS) figure for all the worker processes. This is because a rate, such as TPS, can't be calculated without a shared notion of the beginning and the end of the timing period. The console performs the required timekeeping function.

## Getting Started

Have I whetted your appetite? Let's try running The Grinder. In this example, we'll start both the console and an agent process on a single machine.

Having expanded The Grinder distribution and set up your CLASSPATH appropriately (see the README file provided with The Grinder for details), you can start the console with the following command:

```
$ java net.grinder.Console.
```

The console window should appear. Now change to a directory to hold the output of the worker processes and create a grinder.properties file:

```
grinder.processes=2
grinder.threads=5

grinder.plugin=net.grinder.plugin.http.HttpPlugin

grinder.test0.parameter.url=\
 http://e-docs.bea.com/wlsdocs70/index.html
grinder.test1.parameter.url=\
 http://e-
docs.bea.com/wlsdocs70/images/bealogo.gif
```

This particular file specifies that there will be two worker processes with five worker threads each, and that the HTTP plug-in will be used. It also defines two tests that involve accessing resources from the BEA e-docs site.

Start the agent process in the same directory as the grinder.properties file :

```
$ java net.grinder.Grinder
```

The console display will update to show the two tests. To instruct the worker processes to start the test run, select *Start processes* from the *Action* menu. After a short delay, the console display will show graphs of the incoming reports.

Individual graphs will show the TPS for each test, and a full graph will show the total TPS. Alongside each graph, the mean transaction time, mean transactions per second, peak transactions per second, number of transactions, and number of errors recorded for each test are shown. The colors of the individual test graphs vary from yellow to red to indicate the tests that have the longest mean transaction times. The more red a test graph is, the longer the transactions for that test are taking.

Try selecting the Results tab to see the results in a tabular form. You can also select the Sample tab to show the sum of all reports received during the current console sample interval.

*Note:* If this example doesn't work the first time, it's usually something straightforward. Have a look though the documentation that comes with The Grinder, and if that doesn't help you, e-mail grinder-use@lists.sf.net.

## Recording Test Scripts

It's quite feasible to have HTTP plug-in grinder.properties test scripts containing hundreds or thousands of individual tests. The Grinder lets you specify the timing of each test. Additionally, the HTTP plug-in provides support for setting cookies, authentication information, dynamically generated requests, HTTPS, and other HTTP features. All of these are configured using properties in the grinder.properties file.

Writing such test scripts by hand quickly becomes impractical. The Grinder is shipped with a tool, the *TCP Sniffer,* that can automatically capture test-script entries corresponding to the HTTP requests a user makes using a browser, and generate corresponding test-script entries. The TCP Sniffer is configured to sit between the user's browser and the target server and capture all the requests the browser makes before proxying the requests on to the server. (Technically the TCP Sniffer is a proxy and not a sniffer at all, but it's very useful despite being misnamed!) The responses the TCP Sniffer receives from the server are returned to the browser.

You can start the TCP Sniffer in a special mode in which it outputs a recording of the requests you make with the browser as a full grinder.properties test script. You can then take this test script and replay it using The Grinder.

## More Than HTTP

While the HTTP plug-in is the most commonly used, The Grinder can also be used in contexts other than Web and Web-services testing. Two other example plug-ins are shipped with The Grinder, a JUnit plug-in that allows you to repeatedly exercise a JUnit test case using many threads, and a raw socket plug-in.

It's also easy to write your own plug-in – you just provide a Java class that conforms to a simple interface. I often do this to test J2EE applications with EJB or a JMS interface.

The Grinder is already a powerful tool, but it can be improved. One of the key limitations is that each worker process executes the tests in the test script sequentially, in a fixed order. The Grinder 3 will address this by allowing tests to be specified using a variety of scripting languages, including Visual Basic, Jython, and JavaScript. Test scripts will allow arbitrary branching and looping, perhaps using the scripting languages' support for random variables. That's if I can find the hacking time.

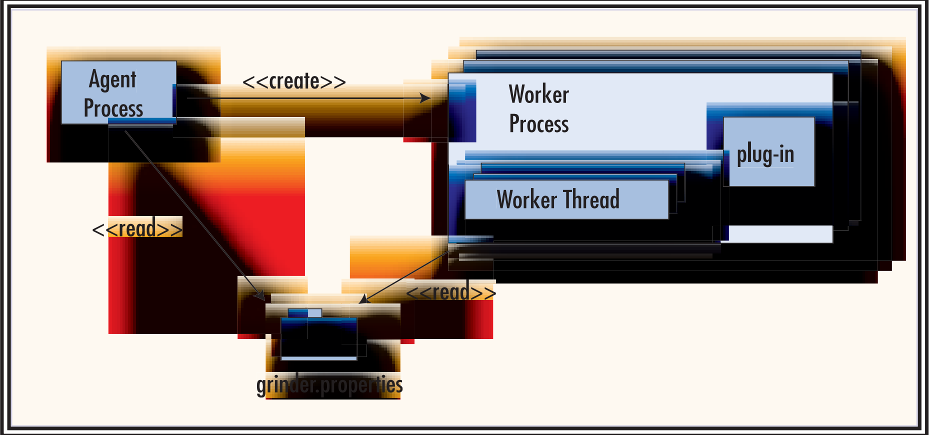
Happy grinding!

## Acknowledgements

I am grateful to Tony Davis and the Expert Press team for their permission to use material from *J2EE Performance Testing.* As well as full coverage of The Grinder, this book contains much practical information about J2EE performance and application benchmarking.

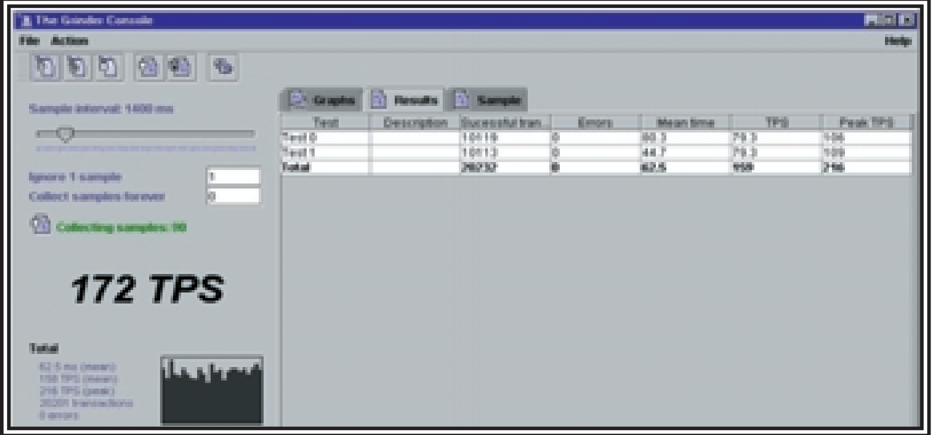
I also wish to express gratitude to VA Software for the SourceForge site (http://sourceforge.net/). SourceForge is without doubt a great resource for the open source community and is responsible for the continued success of The Grinder and many other open-source projects.

FIGURE 1



The agent process and the worker processes read their configuration from grinder.properties

FIGURE 2



Sample interval: 1400 ms

Ignore 1 sample: 1

Collect samples forever: 0

Collecting samples: 90

172 TPS

Total
62.5 ms (mean)
158 TPS (mean)
216 TPS (peak)
20201 transactions
0 errors

| Test | Description | Sucessful tran. | Errors | Mean time | TPS | Peak TPS |
|---|---|---|---|---|---|---|
| Test 0 | | 10119 | 0 | 80.3 | 79.3 | 106 |
| Test 1 | | 10113 | 0 | 44.7 | 79.3 | 109 |
| Total | | 20232 | 0 | 62.5 | 158 | 216 |

The Grinder console

# Performant

www.performant.com/weblogic1

# HTTP Session Object vs Stateful EJB

**THE CHOICE MAY NOT BE AS SIMPLE AS YOU THINK**

BY **PETER ZADROZNY**

**AUTHOR BIO...**

Peter Zadrozny is the founding editor of *WebLogic Developer's Journal* and works as BEA's chief technologist for Europe, Africa, and the Middle East.

**CONTACT...**

peter@sys-con.com

One of the big controversies of session handling concerns the performance difference between storing session state in an HTTP session object and using a stateful session bean. My colleagues and I expected that it would be more efficient to store data in an HTTP session object, as we were under the impression that there is more overhead involved with the infrastructure of session beans in the EJB container. Therefore, we were interested in measuring the performance of each method, to prove or disprove our initial notion.

To test this out, we created a small application that we used to store a specified amount of randomly generated content in either an HTTP session object or in a stateful session bean (*size=number_of_bytes_to_store*). Our application consisted of a single class, the SessionServlet, which we used both as a servlet and as a session listener. The servlet methods were responsible for handling requests and storing data in the associated session, using a specified storage method. When the servlet was requested using the argument type=0, it stored data in an HttpSession object, but when run with the argument type=1, it stored the data in a stateful session bean. When we used the session bean, we still needed the HTTP session object to store the bean's handle, in order to associate the bean instance with the client.

In addition to being a servlet, the SessionServlet extended the HttpSessionListener interface. We did this to ensure that all session beans would be removed from the container when the associated session was invalidated. If we had not explicitly removed the session beans – by calling the ejbRemove() method, as you will see later – they would have been passivated by the bean container, resulting in a dramatic impact on performance.

The test environment was based on the WebLogic server 6.1 SP2 running out of the box (15 execute threads). We used JDK 1.3.1-b24 HotSpot Server and the only parameter we defined was the heap space of 128MB (-ms and -mx were the same). The computer was a Sun Ultra 60 with dual Ultra SPARC II 450MHz, 512MB of memory, running Solaris 2.7. All of the tests were conducted on a dedicated 100Mbps network in which the only traffic present was generated by the tests themselves.

Once the SessionServlet had been deployed, we used The Grinder (http://grinder.sourceforge.net) to generate a test load in which each simulated user executed a test script (see Listing 1).

As you can see, every request stores a different amount of bytes. The total number of bytes stored per session is 3,200. Also note that there is no think time between requests. We did this deliberately to maximize the stress on the system.

To make sure that the stateful beans were removed at the end of every HTTP session, we set the HTTP session timeouts in the WebLogic server to 5 seconds and forced the test script to sleep for 6 seconds before starting a new HTTP session.

We ran the tests using 100, 200, 300, 400, and 500 simultaneous active users, each executing the test script in a sequential fashion for the duration of the test runs. The sample size was 10 minutes after ignoring the first 3 minutes of execution of the test. The results for the average response time can be seen in Figure 1.

The figure presents the aggregate average response time, which is the average of all the individual average response times for each of the 10 requests that make up the test script. We also present the average response time for the first request, which we expect to be a little more expensive than the other requests since this one has to establish the HTTP connection and create the HTTP session object (as well as the stateful session bean when type=1).

Notice how the first request becomes less expensive than the aggregate value of the response time as the load increases. This shows that under high loads the manipulation of the HTTP session object is more expensive than the HTTP handshake and the creation of the HTTP session object.

Looking at Figure 2, the total transactional rate, we notice that we have not yet reached the full capacity of the application server, as the curve has not stabilized.

The network utilization varied from an average of less than 1% for the case of 100 users all the way up to about 4% for 500 users (see Figure 3).

A similar occurrence was observed with the CPU usage of the computer running the application, which varied from an average of 20% for 100 users to about 90% for 500 users (see Figure 4).

The next set of tests uses the stateful session bean by specifying type=1 as an argument to the servlet. To our amazement, the results were basically the same as the comparison seen in Figure 5.

In the case of the transactional rate, the biggest difference was on the order of 1%, which can be considered negligible. We did observe that the network and CPU utilization of this set of tests was basically the same as the ones for the tests using the HTTP session object.

We had to acknowledge that the stateful session bean did not use the security features offered by the EJB container. Nevertheless, it was interesting to find out that under the test conditions *the comparative costs of storing data in an HTTP session object are roughly the same as storing the same data in a stateful session bean***.** To say the very least, this came as a surprise.

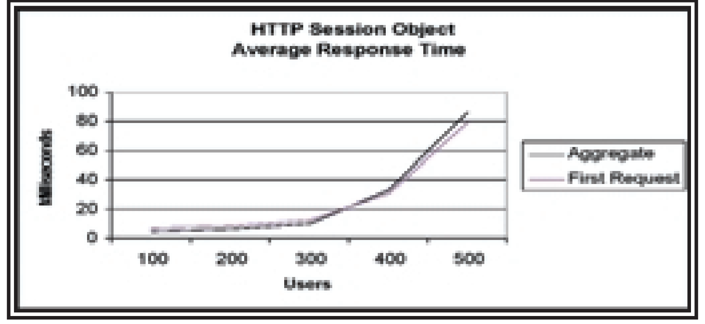
I strongly encourage you to test your situation, taking particular care of the think times you use in the test scripts. Although I'm sure you'll see different raw performance numbers, I expect that the comparative costs between the two models will be roughly the same.

The think times can have a very big impact on the results you obtain. We did not use think times purposely to observe behavior in a high-stress situation. You must use the real think times that apply to the normal utilization of your application.

## Using ejbRemove

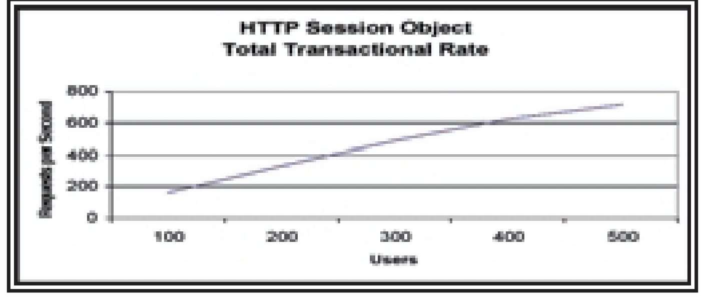
One of the most common programming mistakes in J2EE is to forget to *explicitly* destroy or remove EJBs once they have been used. This usually happens when you call EJBs from servlets. We mentioned earlier that we took a lot of care in our previous tests to make sure that the EJBs were removed. We did this by implementing a session listener, which made sure that before a session was

**FIGURE 1**



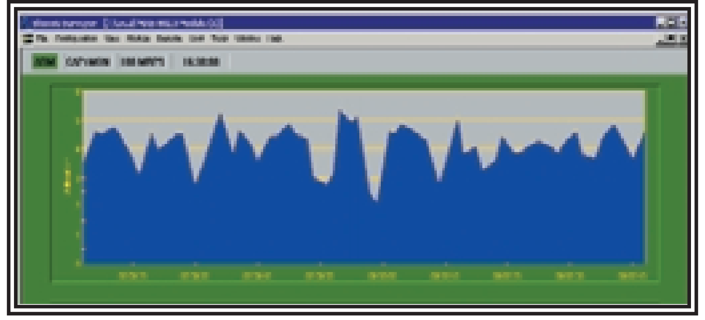
Average response time

**FIGURE 2**



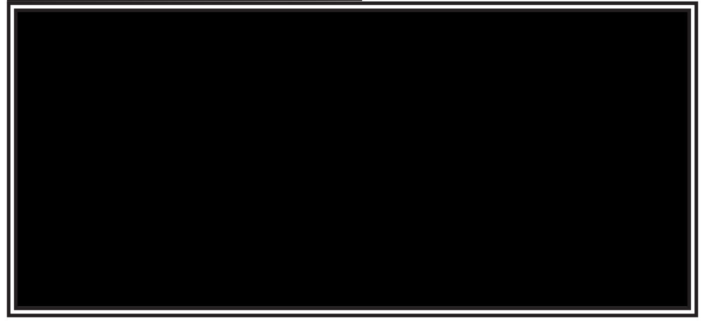
Total transactional rate

**FIGURE 3**



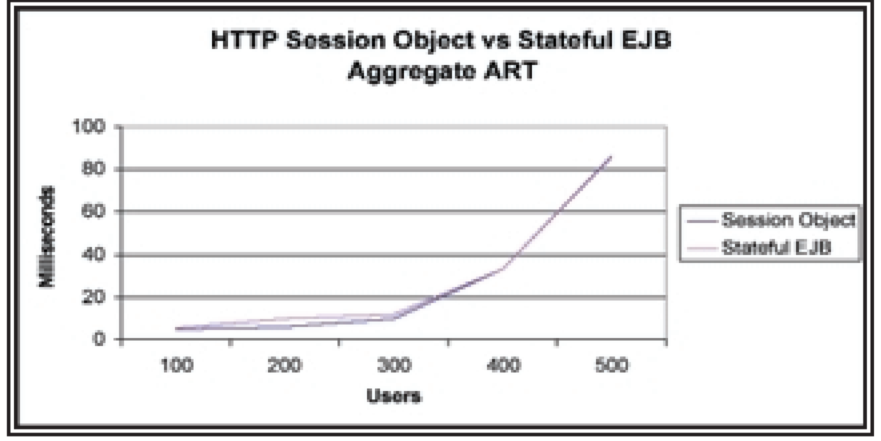
Network utilization variations

**FIGURE 4**



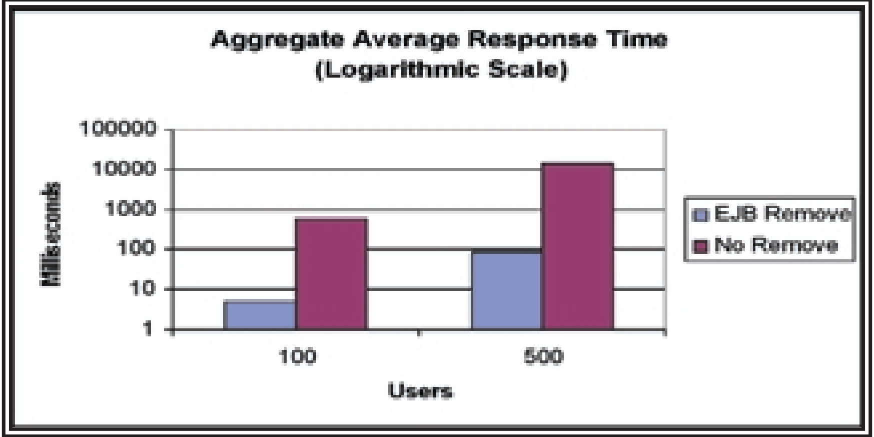
CPU usage by computer running the application

FIGURE 5



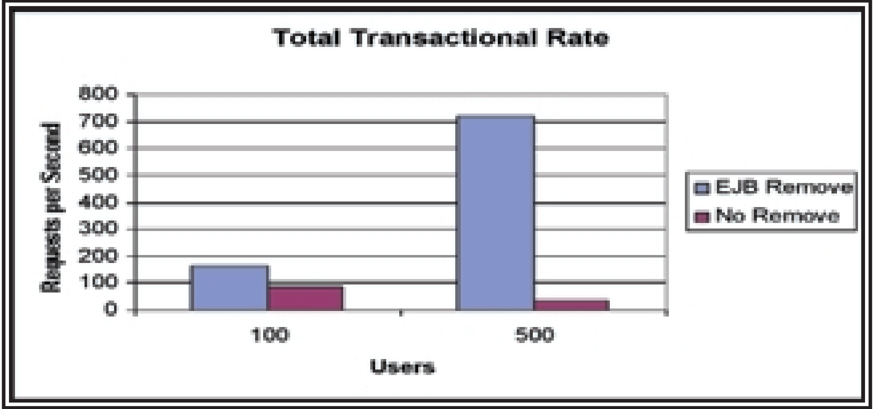
Stateful session bean specifying type=1

FIGURE 6



The price of passivation

FIGURE 7



Passivation in the transactional rate

terminated, all the beans it may have been using would be terminated. In addition, we made sure that our test script waited until the HTTP session timed out, giving the listener time to remove the EJBs.

Failure to remove an EJB that should have been removed carries a very high price from the performance perspective. Basically, what happens is that the EJB will be passivated, a rather silly way of removing an EJB from the container. As you probably know, passivation is a very expensive operation, as it first serializes the bean, and then writes it to disk.

To clearly illustrate the expense of passivation, we modified the servlet used for the previous tests. We did this by adding a type=2 test that will not remove the stateful session bean when the HTTP session is terminated. The differences in performance are so big we had to use a logarithmic scale for the chart in Figure 6. The picture for the transactional rate is even worse (see Figure 7).

If we remove the EJB while the number of users increases, the throughput also increases; if we don't remove it, the throughput actually decreases as the number is users increases.

## Conclusion

It's amazing to find out that the cost of storing data in an HTTP session object is basically the same as using a stateful session bean, assuming the bean is removed in a proper way at the time the session terminates. Not doing so will have a negative impact on the performance of the application. But knowing that the beans must be removed is one thing – actually getting it done in time is another. In fact, it takes a considerable programming effort to ensure that this is done correctly. In our case, we used the session listener mechanism to monitor the session lifecycle and then to cut in moments before the beans are passivated. For your own applications, you can use this method or any other you find more viable. In any case, always make sure to properly test and analyze the system before making any final decisions.

## Acknowledgements

This article is an extract from the book *J2EE Performance Testing* by Peter Zadrozny (Expert Press, June 2002). Special thanks to Bjarki Hólm and Gareth Chapman for their help in preparing the application used for these tests.

**Listing 1**

```
http://wls_host:7001/servlet/SessionServlet?type=0&size=512
http://wls_host:7001/servlet/SessionServlet?type=0&size=512
http://wls_host:7001/servlet/SessionServlet?type=0&size=256
http://wls_host:7001/servlet/SessionServlet?type=0&size=128
http://wls_host:7001/servlet/SessionServlet?type=0&size=512
http://wls_host:7001/servlet/SessionServlet?type=0&size=256
http://wls_host:7001/servlet/SessionServlet?type=0&size=128
http://wls_host:7001/servlet/SessionServlet?type=0&size=512
http://wls_host:7001/servlet/SessionServlet?type=0&size=256
http://wls_host:7001/servlet/SessionServlet?type=0&size=128
```

Mongoose Technology

www.portalstudio.com

Welcome to this edition of "In the Admin Corner," a new monthly column devoted to the administration, configuration, management, and deployment aspects of WebLogic Server.

# Removing Performance Bottlenecks Through JSP Precompilation

## COMMAND-LINE UTILITY IS RELIABLE AND FLEXIBLE

BY **STEVE MUELLER & SCOT WEBER**

**AUTHOR BIOS...**

Steve Mueller is a principal consultant for BEA Systems, where he specializes in the design, development, and administration of enterprise systems running on WebLogic Server.

Scot Weber is an independent consultant focusing on the construction, administration, and performance issues of J2EE production environments, with emphasis on WebLogic Server.

**CONTACT...**

**steve.mueller@slackwerks.com**
**scot.weber@slackwerks.com**

The goal of this column is to provide you with a closer look at the nondevelopment issues of J2EE that are commonly encountered when working with WLS. Developers and administrators alike will find this column to be of value, as the material applies to both the development and production ends of the application spectrum. Furthermore, this feature draws heavily upon experiences from both the field and the engineering lab, providing detailed answers to real-world problems that go beyond the hand-waving solutions of corridor conversations and whiteboard sessions.

## The Need for JSP Precompilation

This month's article looks at removing a potential system performance bottleneck by addressing one of the most common problems that plague nearly all J2EE development projects – the overhead of JavaServer Page (JSP) compilation at server runtime. Although JSPs are the ideal choice for presenting dynamic HTML views within J2EE applications, they do impact performance in a way that, while more annoying than detrimental, initially creates the perception that the application is slow.

According to the J2EE specification, JSPs are primarily HTML files in which embedded Java code is utilized to interact with other system components and present information dynamically. The specification states that all J2EE-compliant application servers supporting JSP, upon a client request for a given JSP, will

- Convert the JSP from its HTML format into a Java class (in Java source format) of servlet type, substituting out any shorthand JSP notation for fully qualified Java syntax
- Compile the newly generated Java source into its .class bytecode format
- Execute the appropriate interface method on the newly compiled class and return the response to the requesting client

While this is an excellent approach – from a development perspective – for managing dynamic HTML generation within the presentation layer, its impact on the server runtime environment requires a JSP to be parsed, converted to Java code, and compiled before it is executed to handle a given client request. The obvious impact on the end user is that a response will, at minimum, be delayed by the time it takes for the JSP compilation phase to occur for one given JSP file. Take into account the possibility that a given user request may hit two or more JSP files within the same request, and suddenly the time required for the compilation phase is multiplied that many more times.

For the end user who first hits a given JSP and thus forces the initial compilation of the requested file, the perception is that the application is slow and unresponsive. Although such a perception may exist, the compilation process for a given JSP file is generally done once in the lifespan of a given app server VM instance. Therefore, its overall impact on performance is considered to be a nuisance, rather than a critical roadblock to the overall response time of the application. Nonetheless, production systems that aim to deliver a JSP-based J2EE application in a production environment must overcome the pitfalls of JSP and make compilation transparent to the end user.

So how can a production environment benefit from the use of JSP files, yet escape the performance hit of runtime compilation? The answer is simple – implement a process commonly referred to as JSP precompilation. With JSP precompilation, JSP files and their compiled equivalents are deployed into a production environment, having already been precompiled in an offline environment. If precompilation and deployment of the resultant class files are done correctly, the application server will execute the previously compiled classes for the JSP files, and will not force a given request to recompile the JSP at runtime. This creates a situation in which the application operates without the unnecessary overhead of compilation, allowing the administrator to remove a perceived bottleneck to the overall performance of the system.

## Different Methodologies and Approaches

There's no doubt that the promises of JSP precompilation sound exciting. However, in order to fulfill such promises, you must first understand the different approaches that can be taken to implement the technique, and the advantages and disadvantages of each.

### EXERCISING THE APPLICATION TO FORCE PRECOMPILATION

The most obvious approach used to implement JSP precompilation is to exercise a given site application by requesting all possible JSP pages in the application before the production release, so compilation is done before end users access the site. This can be done either by manually browsing the entire site for the first time or by launching automated requests from test suite applications or other scripted clients (such as LoadRunner or SilkPerformer). While this approach appears to have the downside of being the simplest of all possible JSP precompilation methods, its disadvantages quickly become apparent. Perhaps the biggest disadvantage here is that it's difficult to implement across clustered environments, where the number of requests once required by this approach for a single node instance are now multiplied by the number of nodes within the cluster. Furthermore, it's even harder to ensure that each server instance within a clustered environment undergoes JSP precompilation when the cluster is proxied by one or more Web servers or hardware load balancers, since there is generally no way of knowing to which app server the proxy will initially forward the request. Additionally, this method must be implemented every time the application server is recycled, making it a painful task that cannot be accomplished for all but the smallest of sites. Therefore, we don't recommend this approach to JSP precompilation.

### USING COMPILATION TOOLS TO IMPLEMENT PRECOMPILATION

Since manually exercising a site application to force JSP precompilation has significant disadvantages in real-world production environments, the alternative of compiling the JSPs into servlets during preproduction runtime becomes much more enticing. Fortunately, WLS offers two methods to do this. The first way performs precompilation at server startup during deployment of a given Web application (declarative precompilation), while the second offers a command-line Java tool (weblogic.jspc) to allow the process to be handled completely offline (programmatic precompilation). While both have their advantages, programmatic precompilation is the more flexible option of the two, and offers more compelling reasons to be used.

### DECLARATIVE PRECOMPILATION

For declarative precompilation under WLS, a given Web application (standalone or as part of an EAR) can be configured so all of its JSPs are precompiled during application deployment (at server startup) and redeployment (at runtime). The necessary configuration changes are made to the WEB-INF/weblogic.xml deployment descriptor, which uses the precompile <jsp-param/> directive as follows:

```
<weblogic-web-app>
      ...
<jsp-descriptor>
<jsp-param>
<param-name>precompile</param-name>
<param-value>true</param-value>
</jsp-param>
</jsp-descriptor>
      ...
</weblogic-web-app>
```

Upon deployment (or redeployment) of a given Web application, WLS will attempt to precompile all JSP files within the WAR if the above parameter is set to true, recursively working its way down from the root directory of the Web application in the process (and skipping over WEB-INF). Files with either a .jsp or .JSP extension become targets for compilation. Compiled class files are then placed underneath the temporary working directory of the Web application (by default a subdirectory of WEB-INF, unless explicitly specified within weblogic.xml) in the appropriate package directory structure.

While this method is by far the most convenient approach to JSP precompilation (the "flick-a-switch" approach), it has a number of disadvantages that render it almost useless. If an error occurs during compilation of a JSP at the time of deployment (or redeployment), precompilation of the Web application will halt at the point of the exception. Additionally, in situations where there are a large number of JSP files within a given Web application, declarative precompilation significantly impacts deployment time, blocking the deployment until all of the files have been compiled. For large applications, such deployment times tend to be on the order of minutes (10 to 15 minutes in some cases, potentially even longer in others) when hundreds of JSP files are present and declarative precompilation is implemented. Imagine starting a server instance in which a given Web application cycles into the deployment phase with declarative precompilation enabled. If there are a large number of JSP files within the app, and the deployment, nearing completion and having already taken a significant amount of time, suddenly fails because an exception is thrown during compilation, frustration will surely ensue. While convenient at first look, declarative compilation poses a significant risk to production system management and should be utilized only with great consideration.

## PROGRAMMATIC PRECOMPILATION

The most reliable way to precompile JSP files under WLS is to use the Java command-line utility, weblogic.jspc, located in the weblogic.jar file under the lib directory of the WLS installation. This tool allows a developer to compile the desired JSP files during the development phase and iron out compile-time issues before deployment. It also provides an administrator with the ability to implement JSP precompilation for production systems. The major benefits of this utility are:

- Files can be precompiled once and then deployed multiple times. (This isn't affected by the recycling of a server instance.)
- Compile-time exceptions can be worked out in advance without affecting deployment.
- Classes can be deployed across a cluster.

The drawbacks are that using weblogic.jspc requires manual intervention and that it must be rerun when JSP files become out-of-date in development. However, given the issues with the other two methods discussed above, we hardly find this inconvenience to be a disadvantage, and recommend it as the most reliable and flexible mechanism to implement JSP precompilation.

## EXECUTING WEBLOGIC.JSPC

In order to use weblogic.jspc effectively, you must first understand its usage and syntax. For this article, we will utilize the features of the tool from WLS 6.1 SP2. *Note:* The syntax and best practices given below should apply to all versions of WLS 6.1 and in part to the new WLS 7.0.

To invoke the command-line JSP compiler (weblogic.jspc), you must ensure the following:

- The PATH environment variable must include the binary directory of the J2SE 1.3. package installed on your machine for JVM runtime support (e.g., /opt/j2se/1.3.1/sdk/bin or c:\sunsoft\j2se\1.3.1\sdk\bin). If you plan on using javac as your Java compiler for JSP compilation, be sure your PATH includes the binary directory of the full Java 1.3 software development kit (SDK), and not simply the Java Runtime Engine (JRE), as no compiler is shipped with the JRE. If you plan on using a compiler other than javac (such as Jikes), be sure to include the appropriate directories within the PATH for that compiler as well.
- Set the Java system classpath to include the weblogic.jar file from the WLS 6.1 SP2 installation directory, by default found under the product library directory (e.g., /opt/bea/-wlserver6.1/lib/weblogic.jar, or c:\bea\wlserver6.1\lib\weblogic.jar). Additionally, be sure to reference any libraries (JAR or class files) that you might need from the classpath during the JSP compilation stage.

Before executing weblogic.jspc for the first time, it's a good idea to quickly test your command-line configuration as set above. This can be achieved by simply running a version check of WLS with the command, "java weblogic.version", which should return the following:

```
WebLogic Server 6.1 SP2 12/18/2001 11:13:46
#154529
WebLogic XML Module 6.1 SP2 12/18/2001
11:28:02 #154529
```

If your output isn't similar to the above (appropriate to the version you are running), be sure to revisit the PATH and classpath variables set within your current command-line environment before proceeding with JSP precompilation.

The general syntax of weblogic.jspc is given below:

```
java weblogic.jspc [options] <jsp files>...
```

The JSP compiler can by default compile a single JSP file or a set of JSPs in a single invocation of the compiler, and can be configured in a number of different ways via command-line options. A working example is provided:

```
java
weblogic.jspc
-webapp mywebapp
-compiler javac
-compileFlags "-g"
-classpath /u/apps/dist/src/lib.jar
-d .
-package com.slackwerks.mywebapp.jsp
-commentary
-keepgenerated
-k
mywebapp\index.jsp
```

This article shows  a single example, but so you may better understand how weblogic.jspc can be used and managed in your environment, we provide the complete set of working options, implications of use, and associated issues at www.slackwerks.com/wldj.

## Conclusion

While the argument for JSP precompilation can easily be made, there are a number of approaches that can be taken to implement it. However, given the advantages and disadvantages of those shown above, it should be readily apparent that programmatic precompilation via weblogic.jspc provides the most flexible option for overcoming the pitfalls inherent to JSP. Becoming familiar with the tool early in the development cycle will improve the administration and performance aspects of the application during production.

BEA eWorld Europe
www.bea-eworld.com

# Top Five Challenges for J2EE Application Development and Deployment

## KEY PERFORMANCE MANAGEMENT CONSIDERATIONS

BY TOM MULVEHILL

According to Gartner, Java has penetrated as many enterprises as Visual Basic. The implications of this evolution in Java adoption highlight some cruel realities. Java, and by extension J2EE, are no longer niche technologies. The benefits of Java – combining code reuse and scalability – are well understood and validated. Enterprises worldwide have been investing billions of dollars in Web-based applications for years, and it's time to demonstrate a return on investment. For IT organizations to best leverage their investment in J2EE technologies, there are five important application development and deployment challenges to consider. How your organization addresses these challenges will impact the performance and success of your applications.

## Visibility Challenge

As you progress from development to QA and into production, you lose visibility into your J2EE applications. The J2EE applications run inside the Java Virtual Machine (JVM), which runs on a J2EE application server, which runs on the operating system optimized for the hardware platform. There isn't an easy way to monitor these components to understand what each is contributing to performance and how interactions among them compound performance problems.

### VISIBILITY PERFORMANCE CONSIDERATIONS

A mechanism for understanding what Java elements are contributing to performance overhead is needed. For example, organizations need to understand how the presentation layer communicates with the business logic layer. An understanding of how JSPs, servlets, EJBs, and JDBC calls are interacting in the context of application execution is needed. Visibility is important, and correlation of the total application context is required to isolate the root cause of performance bottlenecks.

## Knowledge Gap Challenge

One of the greatest frustrations IT organizations face is finding qualified J2EE developers. No one will equate Visual Basic development with Java development, yet Java development is becoming equally pervasive. A common solution is to outsource J2EE development to specialized consultants and integrators. However, outsourced application development introduces an additional challenge: organizations are scrambling to find the resources necessary to develop and deploy J2EE applications.

### KNOWLEDGE GAP PERFORMANCE CONSIDERATIONS

Developing J2EE applications is only the first step; the applications also need to be tested and supported in production. One of the biggest oversights impacting the performance of J2EE applications is the lack of application knowledge in QA and production. This is extremely problematic in production environments where fast problem identification is required. What's needed is a way for less-experienced professionals to quickly identify problems, drill down and isolate the bottlenecks, and accurately recommend necessary changes. Automated correlation, analysis, and interpretation are necessary to bridge the application knowledge gap.

## Performance Management Challenge

The most eloquent application architecture doesn't benefit an organization if the application doesn't meet service-level expectations. The real challenge in J2EE performance management is measurement. The tools and probes used in development are excellent developmental aids but can't be used in QA under load or in production. These development tools are too obtrusive and introduce too much performance overhead to be considered a viable solution for measuring application performance.

### PERFORMANCE MANAGEMENT CONSIDERATIONS

The best solution for measuring the performance of your J2EE applications is a tool that introduces very little overhead while profiling the performance of your application. More important, you should use a solution that has been designed for QA under load and production environments. Efficient data collection and correlation of the important performance metrics will reveal a true picture of application performance.

## Communication Challenge

Have you developed the perfect application yet? What happens when things go wrong? One of the biggest challenges organizations face is the "blamestorm." The blamestorm is the inevitable finger-pointing that takes place when there are problems with the application. "It's not the application, it must be the database!" "It's not the database, it must be the network!" The one constant in the communication challenge is the lack of actionable information. Problem identification (e.g., the application is too slow) is not the challenge; the challenge is isolating the problem and identifying the responsible group to take corrective measures.

### COMMUNICATION CONSIDERATIONS

Multiple constituencies are involved in solving J2EE application performance problems. Two key factors contribute to resolving problems; the first is actionable information. The QA and production staffs need a solution that clearly highlights and isolates the performance problem. These groups need to provide detailed information to development. The second factor is easy information exchange. Performance management solutions that promote sharing performance information help address communication challenges best. With today's geographically distributed development and operations staffs, the ability of all parties to simultaneously view the same data and root-cause analysis facilitates rapid problem resolution.

## Complexity Challenge

It's not uncommon for organizations to have five, six, or even seven tiers in their application architecture. The more complex the infrastructure, the more difficult it is to isolate performance overhead and bottlenecks. Two common elements in these complex Web-based applications are the J2EE application server and the database. Research has shown that many application bottlenecks occur in either the J2EE application or the database.

### COMPLEXITY CONSIDERATIONS

The best way to address performance management bottlenecks in complex architectures is to identify where the bottleneck exists. The ideal solution will provide the ability to look at performance from an end-to-end perspective. The requirement is to understand what each application tier is contributing to overall performance overhead. Once the problematic tier has been identified, the next step should be an in-depth analysis to isolate the root cause of the problem. To fully understand the performance overhead, there should also be some correlation between the application tiers. For example, poor performance on the J2EE tier may actually be only a symptom; the root cause could be in the database. Only a solution that tracks, measures, and correlates performance across a complex application architecture will provide optimal application performance management.

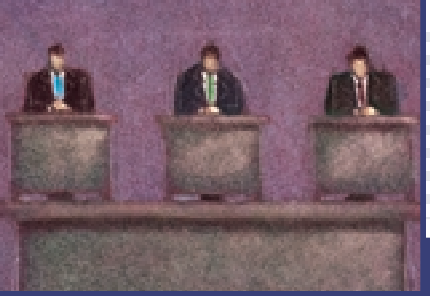
## Conclusion

J2EE application performance management should be proactive. It's infinitely better to find performance problems in development and QA before the application reaches production. However, given the complexity of J2EE applications, performance problems will arise. The degree to which organizations can address the challenges of visibility, knowledge gap, performance, communication, and complexity will determine the efficiency of application performance. Finding the right performance management solution will help organizations in any stage of development or deployment better meet the service level goals of their applications.

**AUTHOR BIO...**

Tom Mulvehill is a senior product marketing manager responsible for Java and J2EE-based solutions for Precise Software Solutions. Leveraging 17 years of engineering, product management, and product marketing experience, he helps to bridge the gap between technologies and solutions.

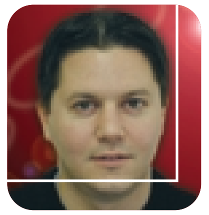
**CONTACT:** tmulvehill@precise.com



This month, we'll study for the EJB portion of the BEA WebLogic Server 6.0 Certification Test.

# EJB
## WHAT YOU NEED TO KNOW ON THE MOST WIDELY COVERED TEST TOPIC

BY DAVE COOKE



AUTHOR BIO

David Cooke is an experienced software developer currently working for Ness Technologies, Inc. (www.ness-usa.com), a consulting firm located in Dulles, VA. In his current position, he utilizes Java and BEA WebLogic Server 6.0 to build J2EE-compliant e-commerce systems for a variety of clients. Dave maintains Microsoft, Java, and BEA developer certifications.

CONTACT...

dave.cooke@ness-usa.com

Now, I know I was also supposed to talk about JDBC this month, but I just didn't have enough room to squeeze it in. EJB is by far the most widely covered topic on the test, and I didn't want to take any attention away from it. I'll be sure to cover JDBC next month.

Let me say it again: EJB is the biggest topic of the test. With that said, it should be the topic that you know best. I'll try to summarize the portions of EJB that you should be familiar with. We'll start with the general interfaces you need to build EJBs.

For the purposes of the test, there are four primary interfaces to know when building Enterprise JavaBeans. These interfaces are the Remote interface, the Home interface, the MessageListener interface, and the implementation class interface.

To implement the Remote interface, you need to implement the EJBObject interface. The aptly named EJBHome interface must be implemented to create the Home interface. The MessageListener interface is used only for message-driven beans. It informs the bean that a message has been received from a JMS Queue or Topic. Since each type of EJB performs a different job, there are different types of interfaces for the implementation class interface. More details about how all of these interfaces work for each type of EJB are discussed below.

### Entity Beans

You'll create the Remote interface by extending EJBObject. The Remote interface should primarily reflect your business methods in the implementation class. All Remote interface methods must be defined as throwing RemoteException.

Create the Home interface by extending EJBHome. Please note: you must provide a findByPrimaryKey() method and may provide zero or more createXXX() or findXXX() methods (where XXX is anything).

In order to create the implementation class, you must implement the EntityBean interface. You must make sure the class is public – not marked as static, final, or abstract – and does not contain a finalizer method. The class must provide an implementation of every method listed in the Remote interface and an ejbFindByPrimaryKey() method that corresponds to the findByPrimaryKey() method in the Home interface. In addition, it must provide an implementation of every creation and finder method listed in the Home interface. Be sure to review Table 1, which lists some entity bean methods of note.

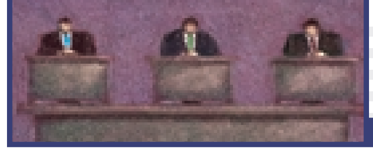
### Stateless Session Beans

As with entity beans, create the Remote interface by extending EJBObject. The Remote interface should primarily reflect your business methods and the methods must be defined as throwing RemoteException.

Create the Home interface by extending

Rational

www.rational.com/ruc



EJBHome. Please note: you must provide a create() method. This method must not contain any input parameters and cannot be overloaded.

To create the implementation class, you must implement the SessionBean interface and make sure your class is public – not marked as static, final, or abstract – and doesn't contain a finalizer method. The class must provide an implementation of every method listed in the Remote interface and an ejbCreate() method that corresponds to the create() method in the Home interface. Table 2 lists some stateless session bean methods of note.

TABLE 1

| METHOD NAME | DESCRIPTION |
|---|---|
| ejbFindByPrimaryKey() | Required. |
| ejbFindXXX() | Must be provided for every findXXX() in Home interface |
| ejbCreateXXX() | Must be provided for every createXXX() in Home interface. |
| ejbPostCreateXXX() | Must be provided for every createXXX() in Home interface |
| ejbActivate() | Invoked when bean is activated. |
| ejbPassivate() | Invoked when the bean is deactivated. |
| ejbLoad() | Instruct bean to load from underlying database. When using Bean-Managed Persistence, this function actually should load from the persistent store. |
| ejbStore() | Instruct bean to store to underlying database. When using Bean-Managed Persistence, this function actually should store to the persistent store. |
| ejbRemove() | Called when the bean is removed. |
| setEntityContext() | The container invokes this method on an instance after the instance has been created. |
| unsetEntityContext() | Last method called before the instance is removed. |

Entity bean methods

TABLE 2

| METHOD NAME | DESCRIPTION |
|---|---|
| ejbCreate() | Required. Must contain no input arguments. |
| ejbRemove() | Called when the bean is removed. |
| ejbActivate() | Invoked when bean is activated. |
| ejbPassivate() | Invoked when the bean is deactivated. |
| setSessionContext() | After constructor, before ejbCreate() – passes SessionContext. |

Stateless session bean methods

TABLE 3

| METHOD NAME | DESCRIPTION |
|---|---|
| ejbCreateXXX() | Must provide a matching ejbCreate() for each create() in Home interface |
| ejbPassivate() | Called when the bean is removed. |
| ejbActivate() | Invoked when bean is activated. |
| ejbRemove() | Invoked when the bean is deactivated. |
| setSessionContext() | After constructor, before ejbCreate() – passes SessionContext. |

Stateful session bean methods

TABLE 4

| METHOD NAME | DESCRIPTION |
|---|---|
| ejbCreate() | Invoked when a new bean instance is added to pool. |
| ejbRemove() | Invoked when a new bean instance is removed from the pool. |
| setMessageDrivenBeanContext() | Invoked before a new bean instance is removed from the pool. |

Message-driven bean methods

## Stateful Session Beans

The Remote interface of a stateful session bean is constructed the same as a stateless session bean, but the Home interface is slightly different. You create the Home interface by extending EJBHome. Please note: you must provide at least one create() method; you may provide additional create methods, using the createXXX() syntax (where XXX is anything).

In order to create the implementation class, you must implement the SessionBean interface. Again, make sure the class is public, not marked as static, final or abstract. It should not contain a finalizer method and should provide an implementation of every business method listed in the Remote interface. The class should also contain an ejbCreateXXX() method that corresponds to each createXXX() method in the Home interface. Table 3 lists some stateful session bean methods you should know for the test.

## Message-Driven Beans

If you're unfamiliar with message-driven beans, you just need to know that they're similar in function to a stateless session bean, yet the processing is done asynchronously. The business methods are executed based on messages received from a single JMS Queue or Topic. For more information on message-driven beans, see http://e-docs.bea.com/wls/docs60/ejb/message_beans.html.

Also, unlike the other EJBs, a message-driven bean doesn't have a Remote or Home interface, because it's never invoked directly by the client. Given this, the methods of a message-driven bean do not have return values or propagate exceptions back to the client.

The MessageListener interface provides the ability for the bean to realize that it has consumed a message. The onMessage() method takes an input of type Message, which can be a BytesMessage, ObjectMessage, TextMessage, StreamMessage, or MapMessage.

To create the implementation class, you must implement the MessageDrivenBean interface. As with all the other implementation classes, make sure the class is public – not marked as static, final, or abstract – and doesn't contain a finalizer method. Table 4 lists message-driven bean methods of importance.

## EJB Exceptions

In studying for the test, you should know how different types of EJB exceptions are processed. The first thing to know is that application exceptions are propagated back to the caller, but the EJB container will intercept and handle a system exception. At this point, generally, the container will throw a RemoteException back to the caller. Table 5 lists the important exceptions used by EJB.

## EJB Utility Classes

Several utility classes that you need to be familiar with are associated with EJBs.

The SessionContext interface allows the session bean to access properties of the EJB container. Understanding that SessionContext is passed into a Session Bean at creation time and is maintained until the bean is destroyed is important. Similarly, the EntityContext interface allows the entity bean to access properties of the EJB container. The EntityContext is passed into an entity bean at creation time and is maintained until the bean is destroyed.

The last utility class to know for the test is the SessionSynchronization interface. This interface notifies a session bean of important transaction-related events. Table 6 summarizes the events.

## Deployment

You need to understand the EJB application file structure. The EJB application should contain the class (or java) files, the ejb-jar.xml file, and weblogic-ejb-jar.xml file.

The ejb-jar.xml and weblogic-ejb-jar.xml files must reside in the \META-INF subdirectory of the jar file. The class (or java) files should reside in the package-relative location. Table 7 shows the structure of an EJB application.

The ejb-jar.xml file contains a large number of deployment descriptors, which are important to know for the test. Some of these are listed in Table 8; a sample ejb-jar.xml file is shown in Listing 1.

The weblogic-ejb-jar.xml file contains more information about the EJBs being deployed. The information in this file is specific to WebLogic Server. Some important descriptors are listed in Table 9 and a sample file is shown in Listing 2.

Next month, I'll look at JDBC in addition to the transactions and clustering topics of the test. Before you take the sample test, though, I'd like to mention another resource to help you study for the test! Be sure to check out the *BEA WebLogic Server 6.1 Bible*; it can be used as a valuable study aid for the test.

Good luck on the test!

## Sample Test

1. Which JMS acknowledgement mode increases message throughput by responding to messages when processor time is available?
a) DUPS_AUTO_ACKNOWLEDGE
b) DUPS_NO_ACKNOWLEDGE
c) DUPS_OK_ACKNOWLEDGE
d) DUPS_IMMED_ACKNOWLEDGE

2. An EJB created using EJB version 2.0 must throw what type of exception?
a) RemoteException
b) EJBException
c) NetworkException
d) None of the Above

3. Which interface are you most likely to use to retrieve properties of a session bean's EJB container?
a) SessionSynchronization
b) SessionContext
c) EJBContext
d) EJBContainer

4. What statement illustrates a difference between creating an entity bean and a session bean?
a) A session bean Home interface does not have a create method.
b) An entity bean does not have a Home interface.
c) An entity bean Home interface does not have a create method.
d) A session bean Home interface does not have any finder methods.

5. Which subdirectory does the ejb-jar.xml file reside in?
a) WEB-INF
b) META-INF
c) Application Directory
d) Root Directory

6. What is a good reason to use a message-driven bean instead of a session bean?
a) Asynchronous Processing
b) Synchronous Processing
c) Exceptions are not propagated to the client
d) Can consume messages from multiple Queues and Topics

7. What interface do you extend to create the EJB's remote interface?
a) EJBObject
b) EJBRemote
c) EJBHome
d) BeanRemote

8. What interface do you extend to create a stateful session bean's implementation class?
a) StatefulSessionBean
b) EJBObject
c) EJBHome
d) SessionBean

TABLE 5

| EXCEPTION | DESCRIPTION |
|---|---|
| java.rmi.RemoteException | Thrown by an EJB method to report a system-level failure. |
| javax.ejb.EJBException | Thrown by an EJB implementation method to its container to report that the invoked method failed. The methods of an EJB implementation class are defined as throwing RemoteException. This is provided for Backward compatibility with EJB 1.0. EJB 1.1 should throw EJBException instead of RemoteException. EJB 2.0 must throw EJBException instead of RemoteException. |
| javax.ejb.CreateException | Thrown by EJBHome createXXX() methods. |
| javax.ejb.RemoveException | Thrown by EJBHome, remove() method. |
| javax.ejb.FinderException | Thrown by EJBHome, findXXX() method. |

EJB exceptions

TABLE 6

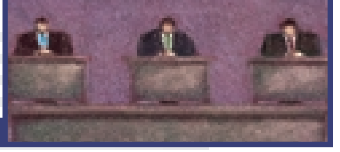
| METHOD NAME | DESCRIPTION |
|---|---|
| afterBegin() | Notifies the bean that a transaction has started. |
| afterCompletion() | Notifies the instance whether a transaction will be committed or rolled back. |
| beforeCompletion() | Notifies the bean that a transaction is about to commit. |

Events

TABLE 7

| FILE | DESCRIPTION |
|---|---|
| /package path/EJB.class or EJB.java | EJB code. As noted above, these files should be located relative to the package they reside in. |
| /META-INF/ejb-jar.xml | Contains the J2EE-specified deployment descriptors for beans. The majority of the defined deployment descriptors reside in this file. |
| /META-INF/weblogic-ejb-jar.xml | Contains WebLogic-specific deployment descriptors, such as the JDNI name of deployed beans. |

Structure of an EJB application



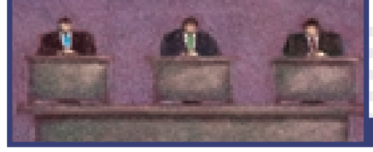


TABLE 8

| DESCRIPTOR | BEAN | VALID VALUES | DESCRIPTION |
|---|---|---|---|
| ejb-name | All | N/A | A name for the EJB. |
| home | All | N/A | The fully qualified name for the EJB's Home interface. |
| remote | All | N/A | The fully qualified name for the EJB's Remote interface. |
| ejb-class EJBs | All | N/A | The fully qualified name for the implementation class. |
| session-type | Session | Stateful or Stateless | Identifies the bean as a stateful session bean or a stateless session bean. |
| transaction-type | Session or message-driven | Bean or Container | Identifies this bean as using either Bean- or Container-managed transactions. |
| persistence-type | Entity | Bean or Container | Identifies this bean as using either Bean or Container managed persistence. |
| prim-key-class | Entity | N/A | Identifies the fully qualified name of the class that represents the primary key. |
| reentrant | Entity | True or False | Identifies the bean may be reentered by another bean in the same thread. |
| message-driven-destination | Message-driven | N/A | Identifies the kind of JMS listener being used by the bean. |
| destination-type | Message-driven | javax.jms.Queue or javax.jms.Topic | Identifies the type of communication the bean supports. |
| jms-acknowledge-mode | Message-driven | AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE | Identifies how the bean acknowledges receipt of a JMS message. AUTO_ACKNOWLEDGE forces the container to immediately acknowledge every message. DUPS_OK_ACKNOWLEDGE will lazily acknowledge all messages when processor time is available. |
| message-selector | Message-driven | N/A | Identifies a query for filtering out certain messages from a Topic or Queue. |

Deployment descriptors

TABLE 9

| DESCRIPTOR | BEAN | DESCRIPTION |
|---|---|---|
| ejb-name | All | A name for the EJB. This must match the corresponding ejb-name in ejb-jar.xml. |
| jndi-name | All | The name you wish to use to bind this object into the JNDI tree. |
| message-driven-descriptor | Message-driven | A container for associating properties to a bean. |
| entity-descriptor | Entity | A container for associating properties to a bean. |
| stateless-session-descriptor | Session | A container for associating properties to a bean. |
| stateful-session-descriptor | Session | A container for associating properties to a bean. |
| destination-jndi-name | Message-driven | Identifies the JNDI name of an actual JMS Queue or Topic available in WebLogic Server. |

WebLogic Server-specific descriptors

9. What interface do you extend to create a stateless session bean's implementation class?
a) StatelessSessionBean
b) EJBObject
c) EJBHome
d) SessionBean

10. What interfaces do you need to implement to build a message-driven bean?
a) ServerSessionPool and MessageBean
b) MessageDrivenBean and EJBObject
c) MessageListener and MessageDrivenBean
d) MessageListener and EJBObject

## Answer Key

1. a, 2. b, 3. b, 4. d, 5. b, 6. a, 7. a, 8. d, 9. d, 10. c

### Listing 1:

```
Sample Session Bean Descriptor
<enterprise-beans>
<session>
<ejb-name>MySessionBean</ejb-name>
  <home>com.beans.MySessionBeanHome</home>
  <remote>com.beans.MySessionBean</home>
  <ejb-class> com.beans.MySessionBeanBean</ejb-class>
  <session-type>Stateless </session-type>
  <transaction-type>Container</transaction-type>
</session>
</enterprise-beans>

Sample Entity Bean Descriptor
<enterprise-beans>
<entity>
<ejb-name>MyEntityBean</ejb-name>
  <home>com.beans.MyEntityBeanHome</home>
  <remote>com.beans.MyEntityBean</home>
  <ejb-class> com.beans.MyEntityBeanBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.String</prim-key-class>
  <reentrant>false</reentrant>
</entity>
</enterprise-beans>

Sample Message Driven Bean Descriptor
<enterprise-beans>
<message-driven>
<ejb-name>MyMsgBean</ejb-name>
  <ejb-class> com.beans.MyMsgBean</ejb-class>
<transaction-type>Container</transaction-type>
<jms-acknowledge-mode>AUTO_ACKNOWLEDGE</jms-acknowledge-mode>
<message-driven-destination>
  <destination-type>javax.jms.Queue</jms-destination-type>
</message-driven-destination>
</message-driven>
</enterprise-beans>
```

### Listing 2

```
Sample EJB Deployment Descriptor
<weblogic-enterprise-bean>
  <ejb-name>MyBean</ejb-name>
  <jndi-name>GenericBeanA</jndi-name>
</weblogic-enterprise-bean>
```

# WLDJ ADVERTISER INDEX

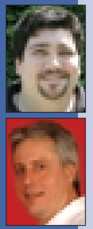
| ADVERTISER | URL | PHONE | PAGE |
|---|---|---|---|
| Altaworks | www.altaworks.com | 603-598-2582 | 47 |
| BEA | http://developer.bea.com | 408-570-8000 | 2,3 |
| BEA eWorld Europe | www.bea-eworld.com | 404-240-5506 | 33 |
| EnginData | www.engindata.com | | 17 |
| Intel | www.intel.com/ad/bea | 408-765-8080 | 21 |
| JDJ Store | www.jdjstore.com | 201-802-3012 | 35 |
| Mongoose Technology | www.portalstudio.com | 281-461-0099 | 29 |
| Panacya | www.panacya.com | 877-726-2292 ext. 3344 | 7 |
| Performant | www.performant.com/weblogic1 | 866-773-6268 ext.2 | 25 |
| Precise | www.precise.com/wldj | 800-310-4777 | 52 |
| Rational | www.rational.com/ruc | 888-889-0074 | 37 |
| Sitraka | www.sitraka.com/performance/wldj | 800-663-4723 | 11 |
| Sitraka | www.sitraka.com/jprobe/wldj | 800-663-4723 | 51 |
| SYS-CON Reprints | www.sys-con.com | 201-802-3026 | 41 |
| Web Services Journal | www.wsj2.com | 888-303-5282 | 49 |
| Wily Technology | www.wilytech.com | 888-GET-WILY | 4 |

Advertiser is fully responsible for all financial liability and terms of the contract executed by their agents or agencies who are acting on behalf of the advertiser.

## LOOK WHAT'S COMING NEXT MONTH

### A Talk with Adam Bosworth
BEA's VP of Engineering and CTO of the Frameworks Division talks about making building applications easier.

### Building Large Financial Applications and Services
Create reusable components that developers can leverage and integrate into the presentation tier.
*by Anwar Ludin*



### Advanced JMS Design Patterns for WLS Environments
What to do when heterogeneity, scalability, and availability requirements go beyond native support.
*by Hub Vandervoort & Jake Yara*

### JMS Performance Notes
Does your application look just like another? You still can't extrapolate performance results.
*by Peter Zadrozny*

### Implementing WebLogic QL
Cut down on how much code you have to write – yet make deployment easier.
*by Michael Gendelman*

### Capacity Planning Guidelines for WLS
Plan now for tomorrow. Make sure you have sufficient resources for current – and future – usage levels.
*by Arunabh Hazarika & Srikant Subramaniam*

BEA WebLogic DEVELOPER'S JOURNAL

# JAVA – A SLOW LANGUAGE?

## IT DEPENDS ON WHAT YOU'RE TALKING ABOUT

BY JOAKIM DAHLSTEDT

AUTHOR BIO...

Joakim Dahlstedt is the CTO of the Java Runtime Products Group at BEA Systems, Inc., where he is responsible for the future development directions for the JVM. He was the founder and one of the chief architects of JRockit, the JVM BEA acquired in early 2002. He has been working with Java and dynamic runtime optimizations since 1996.

CONTACT...

joakim.dahlstedt@appeal.se

### SYSTEM-LEVEL PERFORMANCE IS THE TRUE MEASURE

In the past six years or so the claim that Java is a slow language has regularly appeared in articles and news discussions. Most of the time I follow the debate, but after a while I get bored because the discussions remain at the micro-benchmark level. It continues to amaze me that there isn't more focus on system-level performance in discussions of language performance.

Having spent seven years actively interested in the field of runtime system optimization and the past four years designing a server-side JVM, JRockit, I am pretty convinced that micro-benchmark results cannot be extended to the system level, where performance really matters. Thus, I thought it would be interesting to give you a view of the performance issue from a JVM developer perspective. I'm arguing that Java is anything but a slow and inefficient language, and that the JVM is the key component ensuring that the system is as fast and easy to deploy and run as it was fast and easy to develop.

## Performance Is Not a Micro-Benchmark

This is 2002 and people are still arguing that Java is a slow language. The most common argument comes from developers who have written small benchmarks in Java and then rewritten them in C and are touting how much faster the C programs are. I'm not writing this article to claim that they are wrong.

Of course they can write micro-benchmarks in a couple lines of Java code and redo the same benchmarks in C/C++ and the C programs will probably run faster than the Java programs. That's not where I disagree with them. I disagree with their conclusion; the results of a small micro-benchmark test cannot be extended to say anything about a large-scale application. I believe most systems will be built faster by fewer developers and they will even run faster if they are written in Java instead of C. Java is a language of scale; C is a language of micro-level performance. The development of Java is a natural result of the evolution of system development.

## The System Development Evolution: From Punch Cards to Web Services

What has happened during the past 30 years or so is an explosion – the explosion of large-scale system development. There are so many developers in the world now, and they're building systems that are larger than what anyone imagined possible 30 years ago. In addition, application development today is no longer an I-build-every-part-of-the-application-myself process. People use standard libraries, frameworks, application servers, and most recently, Web services, to build their applications.

The components are getting bigger and bigger and more and more general. These building blocks, which were once looked upon with disbelief and mistrust, are now taken for granted. However, using these general building blocks has a price: performance. We have traded performance for significantly reduced development time and system complexity. At least that's what we think and what our early measurements indicate. What if we're wrong?

## Enter Java and JVMs

The beauty of Java from a large-scale application performance perspective is in two features that are very crucial to performance for large-scale systems: type information and a runtime system (the JVM). The type information is needed by the code generator to discern who calls who, and who points to who. As a result more information can be extracted from a Java program than from a C program, if the compiler is smart. The end result is, of course, higher performance. The JVM is the vehicle for dynamic optimization. The JVM can analyze the behavior of the application and adapt itself and the generated code to the application.

The beauty of this runtime optimization approach is that for the first time ever, the whole system can be optimized specifically for how it is used without having the source code. This is the opportunity. The framework developer can build a general framework that gets used by application developers and that gets optimized specifically for each application that uses the framework. The general building blocks are broken down and melt into each other at runtime. It is the responsibility of the JVM to do this melting. The JVMs, including JRockit, have not reached that goal…yet.

## When Dynamic Optimization Really Pays Off

Let me give you another example of the benefits of dynamic optimization. Look at Intel's new 64-bit platform, the Itanium Processor. Its characteristics are novel and enticing. Nevertheless, its EPIC architecture puts very high pressure on the compiler. EPIC means that the compiler has to choose which instructions should be executed in parallel. Most normal processors today do that selection automatically in their out-of-order execution engine. Because the EPIC-based CPUs need to make fewer decisions, they increase the computational efficiency. The caveat is that the compiler has to make intelligent decisions for the performance to increase.

Given all this pressure on the compiler, you might think that EPIC architecture is well-suited for static compilation and that it virtually eliminates the need for a dynamic runtime system. I believe we'll be able to show you in the coming years how totally opposite the reality is. The reason is that the devil is in the details: for the compiler to be able to parallelize code in a good way requires more than knowing the instructions of the program. The compiler has to know two more things: the flow of the program, and the values that are passed around and used in different contexts.

A static compiler doesn't have this runtime information; it cannot tell what parts of the program are executed frequently or what values are commonly passed to a function from a specific call site. A dynamic runtime system, e.g., a JVM, is the easiest kind of system to collect and take advantage of such information. Consequently, I expect to see exciting gains during the next two years for the Itanium-based JVMs. There is a possibility that they'll catch up with and supercede the profile-guided C compilers on EPIC platforms. Because it lacks type information and a runtime system, C is inferior to Java for EPIC architecture.

## The JVM – The Glue for the Building Blocks

BEA acquired JRockit in February of this year, and we are now working to integrate JRockit into the BEA product line, making sure that it works well with all of the different BEA products. Bringing the building blocks and glue under the same roof makes your system more manageable and scalable and makes it run faster. The JRockit JVM is becoming an important piece of the BEA WebLogic Platform.

In conclusion, I still expect to see a lot of micro-benchmarks out there comparing Java and C programs, but I hope I've convinced you that using runtime systems like Sun Microsystems' HotSpot or BEA WebLogic JRockit Java is not slow or inefficient, and that the performance of a large-scale system built with Java may be superior to that of the same system built using C.

# WebLogic Server Performance Tuning

Good coding practices alone are not sufficient

BY
**Arunabh Hazarika & Srikant Subramaniam**

AUTHOR BIOS...
Arunabh Hazarika and Srikant Subramaniam are engineers in the Performance Team at BEA's WebLogic Division.

CONTACT...
srikant@bea.com

Any product that does well in the market has to have good performance. Although many characteristics are necessary for a product to become as widely used as WebLogic Server is today, performance is definitely indispensable.

Good coding practices go a long way toward getting an application to run fast, but they alone are not sufficient. An application server has to be portable across a wide range of hardware and operating systems and has to be versatile in order to manage an even wider range of application types. This is why application servers provide a comprehensive set of tuning knobs that can be adjusted to suit the environment the server runs in as well as the application.

This article discusses some of the WebLogic-specific tuning parameters and is not an exhaustive list of the attributes that can be tuned. In addition, it is recommended that you try out any suggestions made here in an experimental setup before applying them to a production environment.

## Monitoring Performance and Finding Bottlenecks

The first step in performance tuning is isolating the hot spots. Performance bottlenecks can exist in any part of the entire system – the network, the database, the clients, or the app server. It's important to first determine which system component is contributing to the performance problem. Tuning the wrong component may even make the situation worse.

WebLogic Server provides a system administrator the ability to monitor system performance with the administration console as well as a command-line tool. The server has a set of what are called mbeans, which gather information like thread usage, resource availability, and cache hits or misses. This information can be retrieved from the server either through the console or with the command-line tool. The screenshot in Figure 1 shows statistics such as cache hits and misses in the EJB container and is one of the several options that the console provides to monitor performance.

Profilers are the other useful tools that help detect performance bottlenecks in the application code itself. There are some great profilers out there: Wily Introscope, JProbe, OptimizeIt.

## The EJB Container

The most expensive operations that the EJB container executes are arguably database calls to load and store entity beans. The container therefore provides various parameters that help minimize database access. However, it's not possible to eliminate at least one load operation and one store operation per bean in a transaction, except in certain special cases. These special cases are:

1. The bean is read-only. In this case, the bean is loaded only once, when it is first accessed, and is never stored. However, the bean will be loaded again if the read-timeout-seconds expire.
2. The bean has a concurrency strategy of exclusive or optimistic and db-is-shared is false. This parameter has been renamed as cache-between-transactions in WebLogic Server 7.0. And the values for the parameter have opposite meanings, i.e., db-is-shared equals false is the same as cache-between-transactions equals true.
3. The bean has not been modified in the transaction. In this case, the container optimizes away the store operation.

If none of the above cases apply, each entity bean in the code path will be loaded and stored at least once in a transaction. Some of the features that further help reduce database calls or make such calls less expensive are caching, field groups, concurrency strategy, and eager relationship caching, some of which are new in WebLogic Server 7.0.

• ***Caching:*** The cache size for entity beans is defined in weblogic-ejb-jar.xml by the parameter *max-beans-in-cache*. The container loads a bean from the database the first time it is invoked in a transaction. The bean is also put in the cache. If the cache is too small, some of the beans are passivated to the database. So, regardless of whether the conditions for optimizations 1 and 2 mentioned above apply, these beans would have to be reloaded from the databse the next time they are called. Using a bean from the cache also means that calls to setEntityContext() are not made at that time. If the primary key is a compound field or is complex, it saves on setting that too.

• ***Field groups:*** Field groups specify which fields to load from the database for a finder method. If the entity bean has a large BLOB field (say an image) associated with it that is required only very rarely, a field group excluding that field could be associated with a finder method and that field would not be loaded. This feature is available only for EJB 2.0 beans

• ***Concurrency strategy:*** As of WebLogic Server 7.0, the container provides four concurrency-control mechanisms. They are Exclusive, Database, Optimistic, and ReadOnly. The concurrency strategy is also tied in with the isolation level the transaction is running with. Concurrency control is not really a performance-boosting feature. Its main purpose is to guarantee consistency of the data that the entity bean represents within the requirements imposed by the deployer of the bean. However, there are some mechanisms that allow the container to process requests faster than others, sometimes at the cost of data consistency.

The most restrictive concurrency strategy is Exclusive. Access to a bean with a particular primary key is serialized so only one transaction can access the bean at a time. This provides very good concurrency control within the container, but hurts performance. It is useful only if caching between transactions is allowed, which must not be done in a cluster, so load operations can be optimized away, which may make up for the loss in parallelism.

Database concurrency defers concurrency control to the database. Entity beans are not locked in the container, allowing multiple transactions to operate on the same bean concurrently, which results in faster performance. However, this may require a higher isolation level to guarantee data consistency.

Optimistic concurrency also defers concurrency control to the database, but the difference is that the check for data consistency is done at store time with a predicated update rather than by locking the row at load time. In cases where there isn't too much contention for the same bean in an application, this mechanism is faster than database concurrency, with both providing the same level of data-consistency protection. However, it does require the caller to retry the invocation in the event of a conflict. This feature can be used only for EJB 2.0 beans.

The ReadOnly strategy is used only for beans that are just that – read-only. The bean is loaded only the first time it is accessed in an application or when the read-timeout-seconds specified expires. The bean is never stored. There is also the related read-mostly pattern in which the bean is notified when the underlying data changes. This causes the bean to be reloaded.

- ***Eager relationship caching:*** If Bean A and Bean B are related by a CMR (Container Managed Relationship) and both are used in the same transaction, both can be loaded using a single database call. This is what is done in eager relationship caching. This again is a new feature in WebLogic Server 7.0 and is available only for EJB 2.0.

Besides the above features that help boost performance by optimizing the way the container accesses the database, there are also some parameters for session as well as entity beans that can be used to get better performance out of the EJB container.

Pooling and caching are the main features that the EJB container provides to boost performance for session and entity beans. However, not all of these are applicable to all types of beans. The downside is that memory requirements are higher, though this isn't a major issue. Pooling is available for stateless session beans (SLSB), message-driven beans (MDB), and entity beans. When a pool size is specified for SLSBs and MDBs, that many instances of the bean are created, their setSessionContext()/setMessageDrivenContext() methods are called, and they are put in the pool. The pool size for these types of beans need not be more than the number of execute threads (actually, less are required) configured. If the bean does anything expensive in the setSessionContext() method, where JNDI lookups are typically done, method invocations would be faster if a pooled instance is used. For entity beans, the pool is populated with anonymous instances of the bean (i.e., with no primary key) after the setEntityContext() method has been called. These instances are used by finders; the finders pick up an instance from the pool, assign a primary key, and load the bean from the database.

Caching is applicable to stateful session beans (SFSB) and entity beans. Entity beans have already been discussed above. For SFSBs, caching helps prevent serialization to disk. Serialization to disk is an expensive operation and should definitely be avoided. The cache size for SFSBs should be a little larger than the size needed, which is the number of clients that are concurrently connected to the server. This is because the container doesn't start to look for passivatable beans until the cache hits about 85% of its capacity. If the cache size is larger than actually needed, the container doesn't have to expend itself looking through the cache for beans to passivate.

The EJB container also provides two ways in which bean-to-bean and Web-tier-to-bean calls can be made: call-by-value and call-by-reference. By default, for beans that are part of the same application, call-by-reference is used, which is faster than using call-by-value. Unless there is a compelling reason to do so, call-by-reference should not be disabled. A better way to force call-by-reference is to use local interfaces. Another feature introduced in WebLogic Server 7.0 is the use of activation for stateful services. Though this hurts performance somewhat, it provides huge scalability improvements because of lowered memory requirements. If scalability is not a concern, activation may be turned off by passing the parameter –noObjectActivation to ejbc.

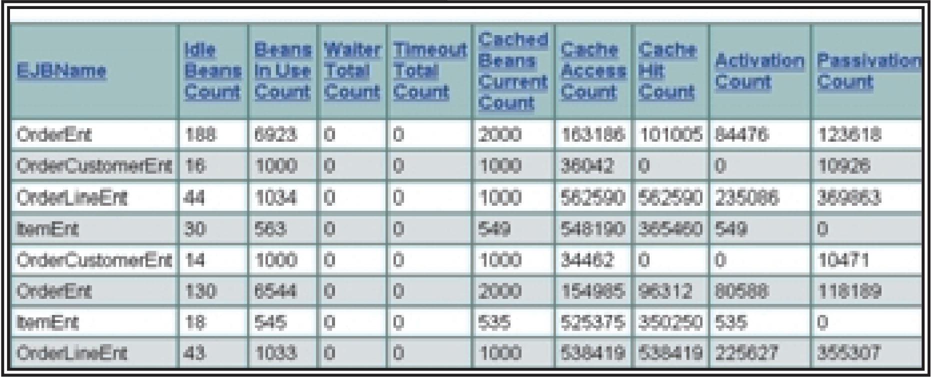
## JDBC

Tuning the JDBC layer is as important as tuning the EJB container for database accesses. Take, for instance, setting the right connection pool size – the connection pool size should be large enough that there aren't any threads waiting for a connection to become available. If all database access is done along threads in the default execute queue, the number of connections should be the thread count in the execute queue less the number of socket reader threads, which are threads in the default execute queue that read incoming requests. To avoid destruction and creation of connections at runtime, specify the same initial and maximum capacity of the connection pool. Also, ensure that TestConnectionsOnReserve is set to false (which is the default setting), if possible. If it's set to true, the connection is tested every time before allocating it to the caller, which requires an additional database round-trip.

The other significant parameter is the PreparedStatementCacheSize. Each connection has a static cache of these statements, the size of which is specified in the JDBC connection pool configuration. The cache is static and it is important to keep this in mind. This means that if the size of the cache is *n*, only the first *n* statements executed using that connection will be cached. One way to ensure that the most expensive SQL statements are cached is to have a startup class that seeds the cache with these statements. In spite of the big performance boost the cache provides, it should not be used blindly. If the database schema changes, there is no way to invalidate the cached statement or replace it with a new one without restarting the server. Also, cached statements may hold open cursors in the database.

In WebLogic Server 7.0, performance improvements in the jDriver have made it faster than the Oracle thin driver, especially for applications that do a significant number of SELECTs. This is evident from the two ECperf results submitted by HP using WebLogic Server 7.0 Beta (http://ecperf.theserverside.com/ecperf/index.jsp?page=results/top_ten_price_performance).

FIGURE 1



| EJBName | Idle Beans Count | Beans In Use Count | Waiter Total Count | Timeout Total Count | Cached Beans Current Count | Cache Access Count | Cache Hit Count | Activation Count | Passivation Count |
|---|---|---|---|---|---|---|---|---|---|
| OrderEnt | 188 | 6923 | 0 | 0 | 2000 | 163186 | 101005 | 84476 | 123618 |
| OrderCustomerEnt | 16 | 1000 | 0 | 0 | 1000 | 36042 | 0 | 0 | 10926 |
| OrderLineEnt | 44 | 1034 | 0 | 0 | 1000 | 562590 | 562590 | 235086 | 369863 |
| ItemEnt | 30 | 563 | 0 | 0 | 549 | 548190 | 365460 | 549 | 0 |
| OrderCustomerEnt | 14 | 1000 | 0 | 0 | 1000 | 34462 | 0 | 0 | 10471 |
| OrderEnt | 130 | 6544 | 0 | 0 | 2000 | 154985 | 96312 | 80588 | 118189 |
| ItemEnt | 18 | 545 | 0 | 0 | 535 | 525375 | 350250 | 535 | 0 |
| OrderLineEnt | 43 | 1033 | 0 | 0 | 1000 | 538419 | 538419 | 225627 | 355307 |

Monitoring system performance with the administration console

Altaworks

www.altaworks.com

## JMS

The JMS subsystem provides a host of tuning parameters. JMS messages are processed by a separate execute queue called the JMSDispatcher. Because of this, the JMS subsystem is not starved by other heavy-traffic applications running off the default or any other execute queues. Conversely, JMS does not starve other applications either. With JMS, there is a trade-off in quality of service for most tuning parameters. For example, when using file-persistent destinations, disabling synchronous writes (by setting the property -Dweblogic.JMSFileStore.SynchronousWritesEnabled=false) may result in dramatic performance improvement, but there is a risk of losing messages or of receiving duplicate messages. Similarly, using multicast to send messages will again boost performance but messages may be dropped.

The message acknowledgement interval should not be set too small – the larger the rate of sending acknowledgements, the slower message processing is likely to be. At the same time, setting it too large may mean messages will be lost or duplicate messages will be sent in the event of a system failure.

Typically, multiple JMS destinations should be configured on a single JMS server as opposed to spreading them out over multiple JMS servers, unless scalability has peaked.

Turning message paging off may improve performance, but it hurts scalability. With paging on, additional I/O is required for messages to be serialized to disk and read in again if required, but at the same time, memory requirements decrease.

Asynchronous consumers typically scale and perform better than synchronous consumers.

## Web Container

The Web tier in an application is mostly used to generate presentation logic. A widely used architecture uses servlets and JSPs to generate dynamic content from data retrieved from the application layer, which typically consists of EJBs. Servlets and JSPs in such architectures hold references to EJBs and, in cases where they talk directly to the database, to data sources. It's a good idea to cache these references. If the JSPs and servlets are not deployed on the same application server as the EJBs, JNDI lookups are expensive.

JSP cache tags can be used to cache data in a JSP page. These tags support output as well as input caching. Output caching refers to the content generated by the code within the tag. Input caching refers to the values to which variables are set by the code within the tag. If the Web tier is not expected to change very often, it's helpful to turn auto-reloading off by setting ServletReloadCheckSecs to –1. This way the server does not poll for changes in the Web tier and the effect is evident if there are a large number of JSPs and servlets.

It is also advisable not to store too much information in the HTTP session. If such information is needed, consider using a stateful session bean instead.

## JVM Tuning

Most JVMs these days are self-tuning in that they have the ability to detect hot spots in the code and optimize them. The tuning parameters that developers and deployers have to worry about most are the heap settings. There is no general rule for setting these. In JVMs that organize the heap into generational spaces, the new space or the nursery should typically be set to between a third and half of the total heap size. The total heap specified should not be too large for JVMs that do not support concurrent garbage collection (GC). In such VMs, full GC pauses could be as large as a minute or more if the heap is too large. Last, there is the caveat that these settings depend to a large extent on the memory usage patterns of the application deployed on the server. Additional information regarding tuning the JVM is available at http://edocs.bea.com/wls/docs70/perform/JVMTuning.html1104200.

## Server-Wide Tuning

Besides the tuning parameters provided by each subsystem, there are also some server-wide parameters that help boost performance. The most important of these is the thread count and execute queue configuration. Increasing the thread count does not help in every case – consider increasing it only if the desired throughput is not being achieved, the wait queue (which holds the requests on which processing hasn't started) is large, and the CPU has not maxed out. However, this may not necessarily result in improved performance. The CPU usage could be low because there is contention for some resource in the server, as would happen if, for example, there aren't enough JDBC connections. Factors like this should be kept in mind when changing the thread count.

In WebLogic Server 7.0, it's possible to configure multiple execute queues and have requests for a particular EJB or JSP/servlet processed using threads from deployer-defined execute queues. To do this, pass the flag –dispatchPolicy <queue-name> when running weblogic.ejbc on the bean. For JSPs/servlets, set the init-param wl-dispatch-policy in the weblogic-specific deployment descriptor for the Web app, with the name of the execute queue as the value. In applications in which operations on certain beans/JSPs appear to have larger response times than others, it could help to configure separate execute queues for them. The queue sizes for best performance will, however, have to be determined empirically.

The other big question is deciding under what circumstances the WebLogic performance packs (http://e-docs.bea.com/wls/docs70/perform/WLSTuning.html#1112119 should be used. If the number of sockets (there is one socket on the server for each client JVM for RMI connections) isn't large and if the incoming requests from the client are such that there is almost always data to be read from the socket, there might not be a significant advantage to using the performance packs. It's possible that not using the performance packs might result in similar or better performance, depending on the JVM's implementation of network I/O processing.

The socket reader threads are taken from the default execute queue. On Windows, there are two socket reader threads per CPU and on Solaris, there are three overall for native I/O. While using Java I/O, the number of reader threads is specified by the PercentSocketReaderThreads setting in config.xml. It defaults to 33% and there is a hard limit of 50%, which makes sense because there really isn't any point in having more reader threads if there aren't any threads to process the request. With Java I/O, the number of reader threads should be as close to the number of client connections as possible, because Java I/O blocks while waiting for requests. This is also the reason it doesn't scale well as the number of client connections increase.

## Conclusion

The above are only some of the various ways that the server can be tuned. Bear in mind, however, that a poorly designed, poorly written application will usually have poor performance, regardless of tuning. Performance must always be a key consideration throughout the stages of the application development life cycle – from design to deployment. It happens too often that performance takes a back seat to functionality, and problems are found later that are difficult to fix. Additional information regarding WebLogic Server performance tuning is available at http://e-docs.bea.com/wls/docs70/perform/index.html.

# News & Developments

## WebLogic Enterprise Platform Chosen by Cambrian

(San Jose, CA) – BEA Systems, Inc., has announced that Cambrian Communications, a wholesale broadband metropolitan area network provider, has selected the BEA WebLogic Enterprise Platform to standardize development and deployment of new business applications.

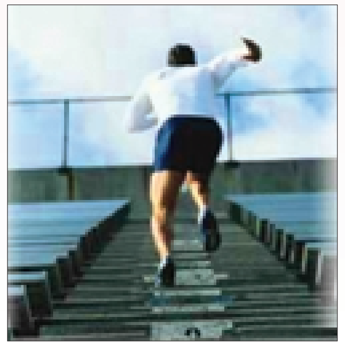
Cambrian will leverage WebLogic Enterprise Platform to provide end-to-end, all-optical network solutions for telecom carriers, emerging service providers, and large enterprises. The first applications Cambrian has deployed on WebLogic are procurement, ticket-tracking, and employee records systems. Plans are under way to use WebLogic as the infrastructure for inventory management.

The applications built on WebLogic are all Web-enabled and take advantage of its extensive Java messaging capabilities to integrate and share data efficiently. www.cambrian.net, www.bea.com

## Percussion Software Releases Rhythmyx Accelerator

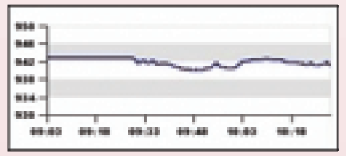
(Stoneham, MA) – Percussion Software, a content management vendor committed to reducing the total cost of ownership of enterprise-scale content management, has announced the availability of its Rhythmyx Accelerator for BEA WebLogic. Rhythmyx Accelerator works with the WebLogic platform to provide the ability to publish content and metadata to WebLogic applications and enable content contributors to preview content live in the application interface.

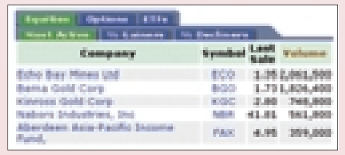
Percussion's Rhythmyx Content Manager 4.0 provides the lowest total cost of ownership for enterprise-scale content management solutions by allowing organizations to reduce implementation costs and control ongoing expenditures. The Rhythmyx Accelerator is available immediately at a listing price of $30,000. The Rhythmyx Accelerator for BEA WebLogic replaces the Rhythmyx Personalization Accelerator for BEA's WebLogic Personalization Server. All customers can immediately upgrade to the new Accelerator. www.percussion.com



## American Stock Exchange Builds for the Future

(San Jose, CA) – BEA has announced that the American Stock Exchange (Amex) is consolidating its three primary Internet sites into a single Web presence built on the BEA WebLogic Enterprise Platform.





| Company | Symbol | Last Sale | Volume |
|---|---|---|---|
| Echo Bay Mines Ltd | ECO | 1.35 | 2,061,500 |
| Bema Gold Corp | BGO | 1.73 | 1,826,400 |
| Kinross Gold Corp | KGC | 2.80 | 740,800 |
| Nabors Industries, Inc | NBR | 41.81 | 561,800 |
| Aberdeen Asia-Pacific Income Fund | FAX | 4.95 | 359,000 |

Amex is the second-largest options exchange in the U.S. and the only primary exchange that offers trading across a full range of equities and exchange-traded funds. Their Web sites provide investment research, background information regarding traded companies, and rules and regulations.

In making the decision to build on WebLogic, Amex cited it's superior scalability, performance, and ease of development as key criteria. www.amex.com

## Korean Customs Service Chooses WebLogic for e-Government

(San Jose, CA) – BEA announced that the Korean Customs Service has selected the WebLogic Enterprise Platform as the application infrastructure for its e-government program, which will migrate the agency's operations to the Internet over a three-year period.

The agency employs more than 3,800 full-time personnel, and in 2001 managed the importation of goods worth over $140 billion. The e-government program – spanning 60 customs offices at 42 locations – is designed to simplify procedures and paperwork, reduce training time for new employees, and accelerate workflow.

## Collaxa Announces Availability of Web Service Orchestration Server

(Redwood Shores, CA) – Collaxa, Inc., has announced that the company's flagship product, the Web Service Orchestration Server (WSOS) 1.0, for the BEA WebLogic application server is now available for public download at www.collaxa.com/developer.jsp. Collaxa's WSOS is the first JavaServer Page (JSP)–like abstraction for orchestrating Java Message Service (JMS) and XML Web services, enabling mainstream Java developers to solve complex business-process integration problems using emerging standards. Over 150 developers have successfully evaluated the solution through the company's preview program since it was unveiled at BEA's eWorld. www.collaxa.com

## FileNET Announces Strategy for ECM-Portal Integration

(Costa Mesa, CA) – FileNET Corp., a provider of Enterprise Content Management (ECM) solutions, has announced it will integrate its extended ECM solutions with major enterprise portal offerings. FileNET has an aggressive development plan in place to partner with major enterprise portal vendors to offer integrated solutions over the next six months.

Additionally, FileNET plans to offer out-of-the-box components that operate in BEA's portal frameworks. FileNET is developing a portlet to support integration between BEA's WebLogic portals, which will ship with the next release of FileNET's ECM software framework. www.filenet.com

Sitraka

www.sitraka.com/jprobe/wldj

Precise

www.precise.com/wldj