

# **Performance Management for J2EE**

---

Understanding and managing performance  
bottlenecks while implementing enterprise systems  
based on J2EE™ technologies

**A Borland White Paper**

*By Java Business Unit*

January 2003

---

**Borland®**

## **Contents**

<b>Overview .....</b>	<b>3</b>
<b>J2EE™ implementation .....</b>	<b>4</b>
<b>Managing performance of J2EE systems.....</b>	<b>5</b>
Planning performance goals.....	6
Designing and developing for performance .....	6
Testing performance prior to deployment.....	6
<b>Performance and reliability requirements for J2EE systems ....</b>	<b>8</b>
Performance considerations for Servlet and JSP™ .....	8
Performance considerations for JDBC® .....	8
Performance considerations for EJB™ .....	9
Performance considerations for JNDI.....	10
Performance considerations for JMS .....	10
<b>Quality assurance .....</b>	<b>10</b>
<b>Performance analysis tool selection .....</b>	<b>11</b>
Nonintrusiveness operation.....	12
Ease of use for QA and test teams.....	12
Effectiveness .....	12
Smarter tool with predictive analysis .....	12
Closing-the-loop with developers .....	12
<b>Conclusion.....</b>	<b>13</b>

## Overview

Java™ is becoming the language of choice for developing and deploying enterprise-class server-side applications. Java™ 2 Platform, Enterprise Edition (J2EE™) technology standardizes the operating environment for server-side Java applications. The J2EE platform provides solid baseline standards on various functional components or containers for presentation and business logic with communication links to client-side presentation, as well as back-end database and legacy systems (see Figure 1). It also offers communication links to other remote J2EE systems.

Server-side Java applications require application servers, which run on a wide variety of server hardware and operating systems. J2EE technology and the associated application servers allow Java applications to be deployed in a wide range of configurations with varying performance and scalability needs. As the adoption of J2EE technology for enterprise applications increases, so increases the demand to address scalability and reliability concerns and to improve the performance and availability of J2EE systems. Due to the size and complexity of most enterprise applications, performance management strategies must include the use of tools and automation for locating and analyzing J2EE performance bottlenecks.

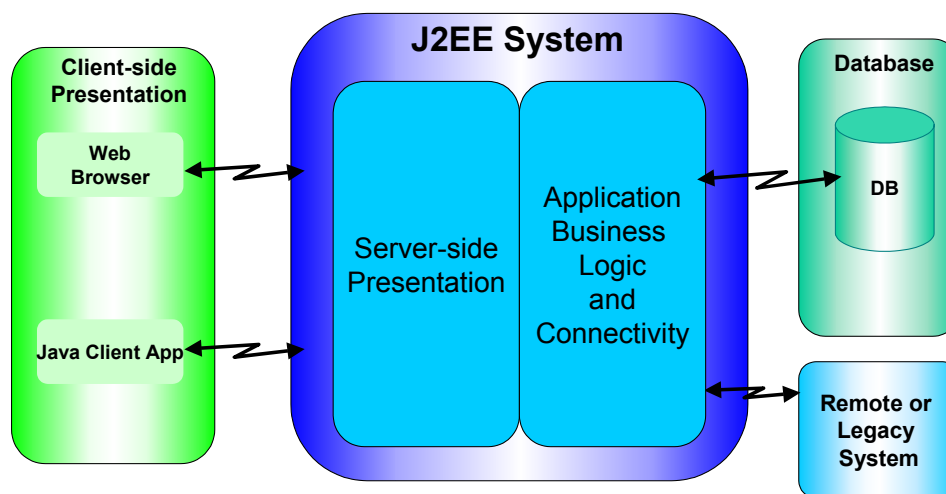


Figure 1: A typical J2EE system

## **J2EE implementation**

The J2EE application architecture makes use of well-defined components and containers for carrying out specific types of operations, such as database transactions or session management. These distinct components and containers benefit from the sophisticated services inherent in the platform.

The J2EE standard specifies the use of servlet and JavaServer Pages™ (JSP™) technology to generate web pages, when a web browser is used as the client interface. The J2EE specification defines Enterprise JavaBeans™ (EJB™) to encapsulate business logic. These components can connect to the backend database tier through drivers based on Java Database Connectivity® (JDBC®) standards. Application servers for J2EE provide these services to Java applications transparently, allowing application developers to focus on business logic. In addition, where loose coupling to a remote business application system or a legacy system is required, message-driven bean (MDB) technology that makes use of the Java™ Message Service (JMS) standards can be used. J2EE also provides other standardized services for communication to external systems, for enterprise corporate data connectivity, and for using Internet technology.

Figure 2 illustrates the interactions between J2EE components that might result from an HTTP request originating in a customer's Web browser. For example, an HTTP (or HTTPS) request may cause the J2EE server to instantiate an EJB that in turn queries an SQL database before returning a response back to the customer. Alternatively, where the data from a legacy or remote system is required, an MDB might communicate via JMS to those external systems.

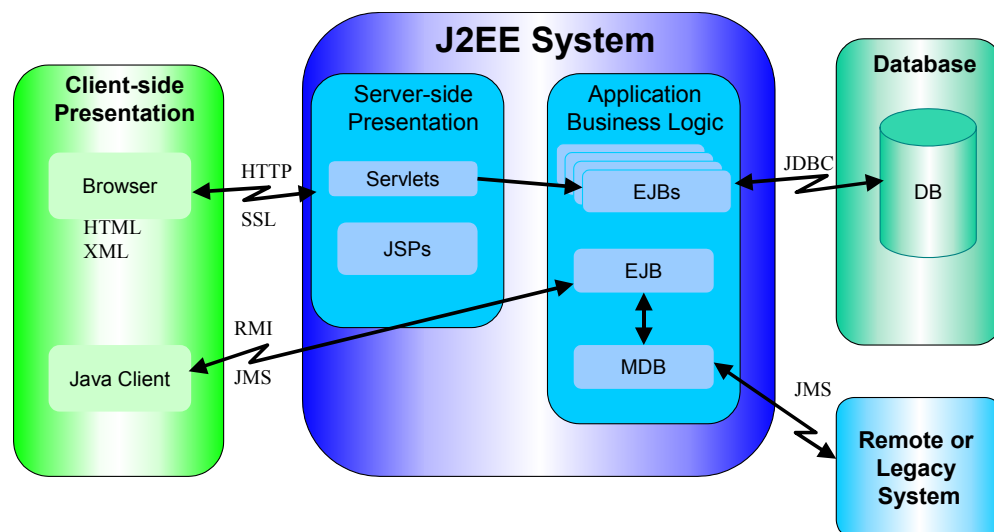


Figure 2: J2EE components and their interactions

## Managing performance of J2EE systems

Web architectures based on J2EE are multitiered and therefore designed to respond to high traffic intensity and user variability, both in terms of volume and transaction mix. Just as the underlying hardware architecture must scale to meet these demands, so too must the J2EE application software. Performance management of a Web-based application is challenging. For best results, performance management needs to start at the beginning of the development cycle, by defining performance expectations and metrics, and then carry through code development, system integration testing, load testing, and staging or pre-production. Incorporating performance management throughout the development cycle helps eliminate performance bottlenecks in the J2EE system before the application is deployed in a production environment.

*Note: Although J2EE application performance can be significantly affected by server clustering design, hardware configuration, and load balancing schemes, this paper focuses on the performance gains that can be realized by analyzing and tuning the application code itself.*

## **Planning performance goals**

Identifying the high-level performance goals for an application—such as the estimated number of concurrent users or the total response time to get specific customer-related data—sounds simple. But considering these goals in detail at the requirements and planning stage is critical for the success of the application in production. It is therefore necessary to interpret these high-level performance requirements at the J2EE components-level, before you start architecting and developing a J2EE system. Though challenging, the high-level performance goals must be translated into clear, measurable performance goals for each J2EE system component used in the application.

## **Designing and developing for performance**

Designing and developing high-performance Java code is challenging, despite the vast amount of information on the subject. Many performance-affecting features of Java development cannot be fully understood without testing. For example, an application architect may need to consider the tradeoff between using a SAX parser instead of a DOM parser. The SAX parser typically provides faster performance but results in complicated code that is more difficult to maintain. For developers, there are often simple solutions to familiar problems, such as the use of string buffers instead of string operation, but the size and complexity of J2EE applications can make these problems difficult to locate without the help of a profiler or performance analysis tool.

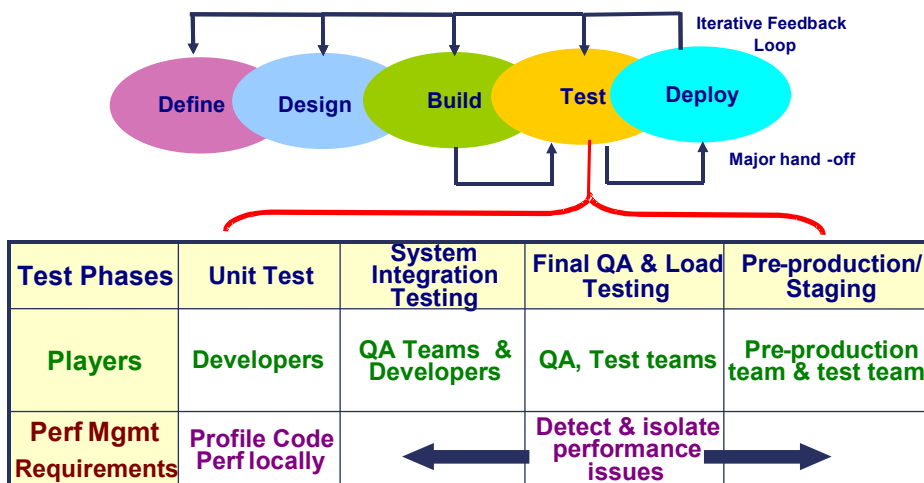
There are many such Java features that can create serious performance bottlenecks in J2EE applications without adequate testing and analysis. The need for tools and automation for measuring code performance is fundamental.

## **Testing performance prior to deployment**

For developers writing system code that involves using J2EE standardized services, such as JMS and JNDI, it is efficient and advantageous to detect J2EE system-level performance issues while performing local system integration testing on their development workstations. This step gives developers insight into how these services and other key J2EE components,

such as servlets and JSPs, are performing, and provides a means for correcting performance or reliability problems before test engineers test applications formally.

Quality assurance (QA) engineers or test teams perform full functional and load testing of a J2EE system. These teams also measure performance of the J2EE system and detect bottlenecks in the system. While the load test is running, it is absolutely essential to see and analyze the performance behavior of the J2EE system. At this time, the QA and the test teams



**Figure 3:** *Managing performance throughout the testing phases*

require simplified performance measurements for J2EE system components, such as JSP,<sup>™</sup> JDBC,<sup>®</sup> servlets, and JMS. These measurements need to explain how the components are performing overall in terms of the response times, CPU time, and the relative system-wide bandwidth consumed in real time. The performance data on the J2EE components should then be compared to initial goals set up during the design phase of implementing the J2EE system.

Stringent application performance requirements mean that collection and analysis of performance behavior data must be accomplished with low impact on the operation of the J2EE system. While testing, low overhead, real-time data collection must satisfy the needs for tactical performance monitoring while simultaneously providing comprehensive data needed for detailed root-cause analysis and strategic planning.

## **Performance and reliability requirements for J2EE systems**

J2EE is a distributed, multitier architecture with many interconnecting components.

Depending on the unique requirements of your J2EE application, you may want to focus on a specific J2EE component when investigating performance bottlenecks. For example, if an application is heavily dependent on database transactions, you will want to focus performance testing on your JDBC code and SQL statement execution.

Your analysis should start at the high level, where the application server is called. Examining the time and resource use associated with a high-level entry point, such as a URI, can lead to more in-depth discovery at the component or code level. Performance and reliability requirements for J2EE systems can be formulated by looking at performance metrics required for each J2EE component, as described below.

### **Performance considerations for Servlet and JSP™**

Sessions containing attributes that are not serializable or sessions that have grown too large can be detrimental to application reliability. The use of custom tags in JSPs can have an impact on application performance, depending on the code in the tag libraries. These tag libraries may come from a third party or another internal source, and may not have specific performance goals or targets. However, if you can measure how they impact the performance of the JSPs that use them, developers can work to resolve the problem. For this reason, it is useful to recognize when code from tag libraries is executing.

### **Performance considerations for JDBC®**

Database access is an essential element of most enterprise applications, and performance bottlenecks associated with JDBC code or SQL statements can be very costly. Using proper caching techniques, minimizing duplicate accesses, preventing unintended EJB write-backs, and investigating the slowest SQL calls, are all useful methods for improving application performance.

Database operations that take a lot of time and are called infrequently are probably suspect SQL statements for the database team to investigate. Problems can be caused if key columns are not indexed properly. You may find that other, database-specific optimization can be used to improve performance of SQL statements. Database operations with high counts and high average times are likely candidates for improved caching. If a given SQL statement is called from many different locations, there may be common code-points, where better caching of database results can improve performance.

If, upon investigating single or small sets of hits, an SQL statement is found to be frequently used, it may be that database caching is not the best solution, and that higher-order caching is needed. For example, either eliminating computation of a `count (*)` query inside of a `for` loop, or reworking the logical flow of an application to store and pass around temporarily useful pieces of information can benefit performance. Performance can also degrade if update statements are improperly happening in methods on EJBs designed for read-only access.

In general, resources that are left open correspond to database resources that are not freed. Unless these resources are part of recycled pools (such as `JDBC Pools`), these correspond to bugs that can ultimately affect scalability. `JDBC` resources should be closed, but relying on the garbage collector to close open resources is not an acceptable practice in a high-scalability environment. Developers who optimize garbage collection algorithms need to optimize garbage collection for reclaiming memory, not to aggressively close open `JDBC` resources.

Some `JDBC` implementations suffer from circular referencing, and resources that are not closed never go away. Either way, it is to your advantage to close resources promptly, rather than relying on the system to do so on your behalf.

## **Performance considerations for EJB™**

Excessive activation and passivation activity indicates EJBs are not being reused correctly, or that there is a need to better configure caching parameters. Excessive load and store activities may be a sign that performance can be improved with EJBs that use container-managed persistence (CMP) instead of bean-managed persistence (BMP), or that you need to optimize the container caching parameters in your persistence layer to improve the performance of your

CMP beans. For BMP entity beans, the EJB data adds useful context to the JDBC calls used to load, store, and find your entity beans. While many beans may be container managed (therefore reducing the contextual usefulness of the load and store entries), the frequency of activate and passivate calls shows how often beans are entering and leaving the cache. EJB performance is heavily dependent on JDBC performance.

### **Performance considerations for JNDI**

JNDI is an essential element of J2EE applications. Although JNDI throughput varies from server to server, in general, even one or two unintended lookups per request can degrade response time. The great thing about many JNDI lookups, however, is that the results are cacheable. Whether this is a `DataSource` or an `EJBHome` interface, the result of a lookup should generally be cached and reused. Sometimes, developers cache these lookups in the wrong place (for example, as an instance variable rather than a class variable). Sometimes they fail to consider caching the return at all. The use of caching for JNDI lookups is critical to the overall performance of J2EE applications.

### **Performance considerations for JMS**

In the design phase, application architects make a choice between a JMS implementation that sends a high frequency of relatively short messages and an implementation that sends fewer messages of greater size and complexity. If the measured overhead associated with creating and sending messages is high, performance may be improved by consolidating messages, and sending fewer of them. Architects must also choose between synchronous messaging, which is often slower but simpler, and asynchronous messaging. If `QueueRequestor` and `TopicRequestor` methods are taking too much time, switching to asynchronous messaging may be justified by the consequent application performance gains.

## **Quality assurance**

Quality assurance (QA) teams offer greater value by detecting performance issues early. The role of quality assurance organizations has been expanded to emphasize pre-deployment

capacity, scalability, and stress testing. QA is now a critical component of performance management. QA engineers must design and run extensive testing to verify that applications have the ability to respond to high loads and to provide scalability to meet increasing system usage needs. Rapid application evolution means performance analysis during load testing must be performed quickly and efficiently.

QA teams are under constant pressure to deliver a high-quality J2EE application and system implementation in terms of functionality as well as performance. They stand to benefit considerably from a smarter tool for J2EE performance with capabilities such as ease-of-use, QA views, and performance error reporting. True value of such a tool comes from advanced capabilities such as automatic capture of diagnostics data for performance issues, and predictive error reporting. Such performance testing when combined with load testing increases the value of the QA team delivering high-performance J2EE systems to the enterprise.

## **Performance analysis tool selection**

Adopting and executing a thorough performance management strategy lets you methodically analyze and optimize the performance of your applications instead of reacting to performance problems when they have reached a level of disaster. A good performance management tool helps you predict problems in advance and helps you take remedial action before problems have a chance to affect application behavior for customers or business associates.

Selecting the best solution for J2EE performance analysis is not trivial—not just any tool will do. Helpful tools provide analysis of the entire J2EE application at a high-level, as well as at the individual component level. A performance tool for J2EE needs to offer complete application performance management from the desktop, across the network, through the application server, to the database and the storage device. This tool should be powerful enough to grow with the company's business needs, provide a basis to achieve true end-to-end application performance management throughout the entire design and development cycle of the J2EE system, and meet the evaluation criteria. Consider the following guidelines as the key criteria for selecting the best tool for your needs.

### **Non-intrusiveness operation**

Ideal analysis tools provide accurate, usable data without altering the source code or application performance. Such tools need to deliver the most time-critical operations with significantly lighter weight on the system. When configured correctly, the performance overhead associated with using such a tool is minimal.

### **Ease of use for QA and test teams**

The best tools are easy to learn and use. A performance analysis tool needs to facilitate ease of use for test teams and require minimal training and ready access by authorized users. Important information is shared among multiple users simultaneously or easily referenced for later examination.

### **Effectiveness**

Powerful tools let users make iterative improvements to their code. Such tools pinpoint existing application weaknesses and provide reliable, standard methodologies for improving application performance.

### **Smarter tool with predictive analysis**

A tool that can assess API use and determine whether existing code will cause faults or exceptions in a production environment can save time and trouble. A tool that can lead you to the root cause of the potential problem is even more beneficial.

### **Closing the loop with developers**

To ensure that developers effectively address problems found by QA or test engineers, it is important for the tool to generate and export test data that provides a comprehensive picture of the performance bottleneck, from the suspect application event down to the root cause in the source code. Providing the data that both points developers to the code that needs attention and points testers to the application events that indicated the problem allows developers and

testers to work together to perform iterative (and comparative) testing and development to quickly and effectively resolve problems.

## **Conclusion**

Performance management solutions for J2EE applications must meet many demanding requirements. IT staff require a full range of performance management functions addressing the specific challenges posed by the J2EE-based mid-tier. Rapid application development and deployment means that any solution must include extensive automation capabilities, and the product must be easy to install and configure, with minimal IT involvement. Additionally, instrumentation technology is required to function completely with third-party components as well as custom developed software.

Performance management solutions must support both monitoring and analysis. Solutions must present current application performance state in such a way that problems can be rapidly detected. Response time and throughput are essential performance metrics, as are indicators of concurrent user-induced application loading.

To facilitate rapid problem diagnosis, performance management solutions must provide detailed correlation, drill down, and analysis functions. Correlation provides the power of associating two dissimilar measurements to yield insight that is not otherwise available. For example, CPU resource consumption measured from the underlying operating system should be correlated with specific application activity.

Application activity must be correlated with user requests. Each application component active in fulfilling a user request should be correlated to other components to provide a complete view of application behavior. Application behavior on the application server should be correlated with requests to the database, and the performance of these requests should be re-correlated to application performance, and ultimately to the performance of user request processing. Each correlation must be made quantitatively so that QA and test teams are quickly guided to problem diagnosis by following the path of “greatest contributor first.” All correlation and analysis should be performed automatically.

Correlation of JVM activity with application activity provides additional critical insight into the causes of performance shortfalls. JVM sampling combined with application instrumentation is required to provide this information. Sampling can provide identification of CPU hot spots that can then be correlated with application components and user requests. Memory usage and garbage collection should be measured and correlated with application performance. CPU hot spots observed using sampling should be correlated with all levels of application components and user activity.

Performance management solutions must provide a historical perspective of application performance, including the ability for detailed analysis of any particular time in the past. A user interface which naturally guides the user through the complex data and its analysis is not just convenient, but essential to handle the complexities of J2EE application performance management.

**Made in Borland®** Copyright © 2003 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • [www.borland.com](http://www.borland.com) • Offices in: Australia, Brazil, Canada, China, Czech Republic, Finland, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, Mexico, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. 20108