

the Rational edge
e-zine for the rational community

developerWorks

Performance testing Java servlet-based Web applications: A guide for software test engineers

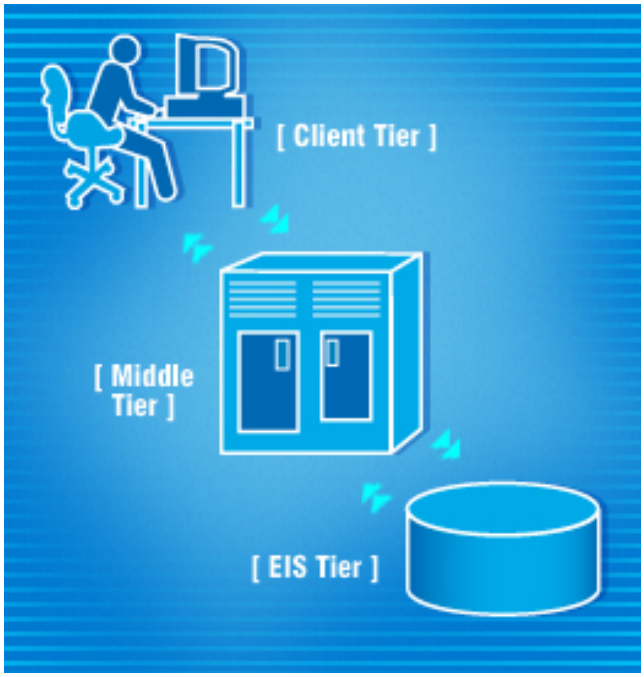
e-mail it!

Len DiMaggio
Software QE Engineer and Manager, IBM
30 0 2004

From The Rational Edge: As testing expert DiMaggio walks readers through the ins and outs of performance testing for applications built with Java servlets, he makes special note of ways that servlets' distinctive characteristics can affect overall application performance.

Contents:

Subscriptions:



This article is one in an ongoing series of "recipes" for testing various types of software. The goals of each article are to help testers understand issues that affect assessment of the target software and to provide specific, step-by-step instructions for thorough testing. In other words, my purpose is to describe the "whats, whys, and hows" of testing.

People use the phrase "speed kills" to warn drivers about the dangers of excess speed. When it comes to Web applications, however, it's a lack of speed that can be fatal.

A few years ago, when using the 'Net or Web was still a novelty, people were often willing to tolerate slow response times when viewing static text or playing with cute applets. Today, however, we all rely on rapid responses from Web applications for business, scientific, research, and, of course, entertainment purposes. And, we're not just viewing static text anymore. Now, we're accessing application servers and databases and moving

large quantities of data over the 'Net.

Now, another popular expression applies to the performance of Web applications: "People vote with their feet." Traditionally, this has meant that people will physically walk away from an unsatisfactory situation; in the Web applications world, it means that if you make clients wait, they'll take their business elsewhere. Accordingly, performance testing[1] should be a significant piece of your Web application testing. You won't be successful if you try to do it as an afterthought or last-minute fire drill. Your Web application's ability to respond quickly to clients—and handle ever-increasing numbers of them—will play a major role in the success of both your application and your business overall.

This article describes a step-by-step approach for testing a Java servlet-based Web application's performance. The first step is to understand performance-related issues that affect Java servlets.

Step 1: Understand what's special about servlet performance

Let's begin by examining some basic information about servlets. The Java Servlet Specification[2] defines a servlet as a:

> *…Java technology based Web component, managed by a container, that generates dynamic content. Like other Java-based components, servlets are platform independent Java classes that are compiled to platform neutral bytecode that can be loaded dynamically into and run by a Java enabled Web server. Containers, sometimes called "servlet engines," are Web server extensions that provide servlet functionality. Servlets interact with Web clients via a request/response paradigm implemented by the servlet container…*

There's quite a bit of information in this paragraph. Let's take a look at the key points from a performance testing perspective. Servlets have characteristics that are different from those of other types of programs (such as standalone Java applications), and these characteristics have implications for performance testing:

- **Servlets are managed by a container**. This is a major difference between servlets and standalone Java applications, because the performance of any servlet is directly affected by the performance of its container. To do performance testing on servlets, you have to understand servlets' container configuration and operation, including how (and how well) the container integrates with other software, such as databases.
- **Servlets are run by a Java-enabled Web server**. This is another major difference between servlets and standalone Java applications. For performance testing, you must take into account the performance and configuration of the Web server, and also deal with applications whose configurations are distributed across multiple servlets and multiple servers.
- **Servlets are in the Middle Tier of multi-tier J2EE applications**. The Java 2 platform, Enterprise Edition (J2EE) supports the creation of multi-tier, distributed applications written via Java extension APIs (e.g., the servlet API) and executed **within** a runtime infrastructure implemented via containers. Such applications are divided into "components" that run on different physical systems, based on the functions they perform. The J2EE application model is typically represented with three or four tiers, with servlets in the Web Tier portion of the Middle Tier:
    - **Client Tier**. These components are the Web clients (applets or other pages accessed via a browser) or Java applications running on client systems. The clients are typically "thin" in design; most data

processing tasks are performed by the servers.

- **Middle Tier**. This tier is sometimes divided into two other tiers:
  - **Web Tier**. These components are the servlet and Java Server Pages (JSP) containers[3] that run on the J2EE server systems.
  - **Application (or Business Logic) Tier**. These components are used to access (e.g., via Enterprise Java Beans [EJBs]) databases and other Enterprise Information Systems (EISs).
- **Back-End (Data or EIS) Tier**. The EISs in this tier may be database servers or other legacy systems.

Figure 1 shows how these tiers are configured in the J2EE platform.

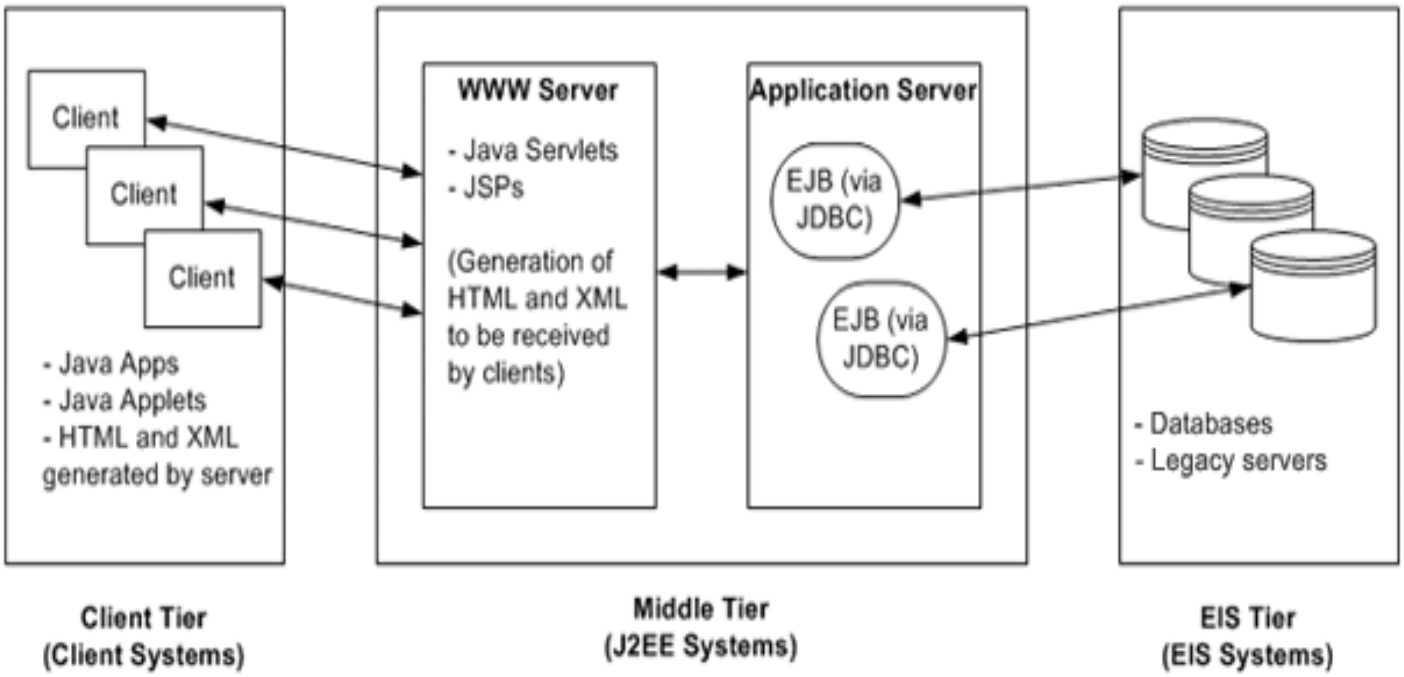


Figure 1: Simplified view of tiers in the J2EE platform

(Based on graphics in the Sun Java Spec (Java) Servlet Specification http://java.sun.com/products/servlet)

- **Servlets generate dynamic content**. One constant in the ever-changing world of Internet software is the need to supply more and more dynamic content. You can still display static text content on Web sites, but increasingly, most Web content is generated dynamically, based on user/customer-specific profiles, requests, and so on. This data is likely stored in databases, so performance testing should verify how efficiently a servlet under test can access its databases.
- **Servlets implement a request/response model**. Unlike other protocols, HTTP does not maintain a single live virtual circuit's "state" between clients and servers. Instead, it's a stateless protocol that establishes and tears down connections as needed and implements reliable communications via a request/response[4] mechanism. Because the protocol doesn't maintain a state, the servlet itself must maintain user session information. The mechanisms used to store this information (e.g., in a remote database) may affect the servlet's performance and have to be considered in planning performance tests.
- **Servlets are meant to run unattended for extended periods**. Unlike programs designed to run for brief periods and then close, servlets are designed to run unattended and stay running. For performance testing, this means that your tests should verify the servlet's longevity—in other words, its performance over time.

And these tests should not only verify the functional operation of the servlet, but also determine how it makes use of available system resources, such as system memory.

Before we consider specific performance tests for servlets, our next step is to examine how some basic software performance test set-up issues apply to servlets.

Step 2: Consider performance test factors before testing the servlet

In planning performance tests for any type of software, you should consider factors such as the testing environment and the optimum testing sequence. The following subsections describe how these factors affect servlets and include checklists highlighting issues of particular importance.

Vetting the test environment

An integral part of performance testing for any networked software is measuring the speed with which data is transmitted between clients and servers, between two or more servers, and so on. This is especially true for servlet testing, as J2EE supports the creation of multitier, distributed applications whose servlets may have to access multiple remote servers (e.g., database servers).

Your goal, in short, is to ensure that the operational characteristics of your test network provide a controlled environment that will not skew your test results or become a gating factor in test execution. In other words, you want the test network to function as the platform for running the tests; and you *don't* want to use the tests to debug your network.

The best test networks are physically separate from all other network traffic. You can construct such a network with dedicated routers, cables, DNS servers, PC domain controllers, and so on. You should also verify that the media you use can physically support the level of traffic you want to test for.

However, if you have limited networking (and financial) resources, you may not be able to execute all of your performance tests on a dedicated and physically separate network. The next-best alternative is to create a virtually separate test network by subnetting the physical network. First, verify that this virtual network is free of any traffic not generated by your tests; use the Unix "snoop" utility to listen in on traffic going "over the wire" before you start to run your tests.

Also, do not limit your vetting of the test environment to network hardware. Certain software-related factors can adversely affect performance tests. Here are some things to watch for:

- **Virus scanning**. The servlet under test, or any of its supporting software, may create files as it runs. If the scanning function for new files is enabled, then each of these newly created files will be scanned. If large numbers of files (e.g., temporary journal files to track individual transactions) are created, then such scanning can have a significant impact on performance. Should you therefore run performance tests with virus scanning disabled? It depends on the configurations your clients will use. Some servers (e.g., a servlet container server) will always run with virus scanning enabled, whereas others (e.g., Web servers to which content is deployed on a real-time basis) will run with it disabled.
- **Shared directories/file systems**. Remember how the space creatures in the movie "Alien" used human bodies as hosts? You should verify that your test servers aren't hosting shared directories or networked file

systems. If your servers are storing/updating large chunks of data, that may affect your performance tests.

- **Default configurations: One size does not fit all**. When you're working with a system composed of multiple integrated subsystems, it can be very tempting to try using standard, default configuration settings for the software with which your servlet must integrate. Unfortunately, this can have an adverse impact on the servlet's performance. Earlier I pointed out that overly detailed logging level settings can negatively affect performance. Often, relying on the default logging setting for a servlet's or container's database will result in too much information being written to disk files; also, the default security level may require that every HTTP request be reauthenticated. Either of these situations can cause performance problems. Setting up a complex, integrated system can be a difficult task: You must wade through configuration data for each subsystem and determine which combinations of values will result in the best end-to-end performance for the system as a whole. However, this is the only way to ensure that the system is really performing according to requirements.

Checking for problems and making corrections in the test environment

Here's a brief description of aspects of the test environment that you'll want to check.

- **Routing**. Remember: A distinguishing characteristic of servlets is that they generate dynamic content. So the speed with which the servlet under test is able to access its database is important to its overall performance. Another distinguishing characteristic is that servlets are run by Java-enabled Web servers and can be configured into a distributed architecture. So inefficient routing can also affect servlet performance. Before you run any tests, you'll want to verify that the network configuration is not routing traffic through convoluted paths and extraneous "hops" that cause communication delays between clients and servers, or servlets and their supporting database servers. To do this, use the Unix "`traceroute`" or Windows "`tracert`" utility, and trace through the routes in both directions.
- **Logging**. Set logging at a high level, because servers, servlet containers, Web servers, and other system components can negatively impact servlet performance by either slowing down the servlet itself or tying up system resources. However, remember that servlets are managed by a container, and excessively high logging configuration settings for the container can adversely affect the servlet's performance. But don't turn off the container's logging. That might boost performance but also create an unsupportable configuration: If something goes wrong, you won't be able to find and debug the cause of the problem. Instead, set the logging at the level that your customers will likely use in their production environments.
- **Automatic software updates**. As we noted above, servlets are designed to run unattended for extended periods, so your tests will have to run for extended periods as well. However, do not run these tests unattended. In some instances, test servers may perform scheduled automatic software updates (e.g., for OS updates or antivirus) during the extended tests. Some of these will likely be very large and may slow down the servers by tying up system resources. If you see this happening, you might want to intervene manually and de-configure the updates until the test run is complete.
- **Configuring memory**. When you run a Java program, unless you specify how much memory the Java Virtual Machine (JVM) should make available for it, then the JVM will use its default value. This value will vary according to the version of the JVM: For JDK 1.2, the default was 16 MB; for JDK 1.4, the value is 64 MB. What value should you use for performance testing? Allocating additional memory will probably improve the servlet's performance, but you do not want to run tests in a larger configuration than your customers will use. It is safest not to depend on the default value—which may be different on different systems—and instead to explicitly set the memory limit in the servlet's Web server and container configuration files. You can do this with the "-mxNNNm" option. For example, the syntax

java –mx128m <program name>

will set the upper limit at 128MB.

If you want to find the default memory heap value for your servlet's test systems, the program[5] shown in Figure 2 can help.

```
1.  public class HeapTest {
2.    public static void main(String[] args) {
3.
4.      Runtime theRT = Runtime.getRuntime();
5.      java.util.Vector theVector = new java.util.Vector();
6.      while (true) {
7.        long size = theRT.freeMemory();
8.        System.out.println("Total memory used = "
9.          + (theRT.totalMemory() / 1024000)
10.         + ", free memory = " + (size / 1024000) );
11.       byte[] buffer = new byte[(int) size];
12.       theVector.addElement(buffer);
13.
14.     } // while
15.
16.   } // main
17.
18. } // class
```

Figure 2: Program to help discover the default heap value for a servlet's test systems

The program displays total and free memory and creates a byte array as large as the total amount of free memory. Each time this buffer is created, it gets added to a vector, so the space it uses cannot be reclaimed by the garbage collector. As a result, the program eventually encounters an out-of-memory exception and exits.

Figure 3 shows sample output from the program shown in Figure 2.

```
$ java HeapTest
Total memory used = 1, free memory = 1
Total memory used = 3, free memory = 1
Total memory used = 5, free memory = 1
Total memory used = 6, free memory = 1
Total memory used = 9, free memory = 3
Total memory used = 11, free memory = 1
Total memory used = 17, free memory = 5
Total memory used = 20, free memory = 3
Total memory used = 30, free memory = 10
Total memory used = 36, free memory = 6
Total memory used = 55, free memory = 18
Total memory used = 65, free memory = 9
Exception in thread "main" java.lang.OutOfMemoryError

$ java -mx16m HeapTest
Total memory used = 1, free memory = 1
Total memory used = 3, free memory = 1
Total memory used = 5, free memory = 1
Total memory used = 6, free memory = 1
Total memory used = 9, free memory = 3
Total memory used = 11, free memory = 1
Total memory used = 16, free memory = 4
Exception in thread "main" java.lang.OutOfMemoryError
```

Figure 3: Sample output from program in Figure 2

Why you must understand the "hows"

It's possible to execute some types of functional software testing by only mapping inputs to outputs, without any real understanding of the software's internal workings. It's tougher to tie inputs to outputs for performance testing, as you're not looking merely to verify that a function returns a specific value; you're also looking to quantify how fast the function can do that.

For example, let's say you observe that the servlet under test seems to process XML input much more slowly than it processes plain text. You can confirm this to some extent by running tests with various types of inputs and timing the results, but unless you know exactly how the servlet is processing its inputs, you won't be able to know why you're seeing that slower throughput. Maybe it's because both the client and the servlet are conversions between XML documents and data objects. Maybe it's because inefficient XML parsers are being used. Or maybe the servlet's logic is just plain convoluted when it comes to dealing with XML. All of these are valid hypotheses, but you won't be able to prove any of them unless you understand what's really going on "under the hood" of the servlet.

Looking at performance tests as complex equations

Executing software tests that exercise an entire system, including performance tests, can be a bit like solving a complex algebraic equation. Components of the equation are the servlet under test and its associated servers, databases, and so on. First, define the variables and constants that make up the test network and the tests, then establish a repeatable pattern of test results by using one set of values for the variables and constants, and finally,

change one value and measure differences in the test results. For example, after you've determined that the servlet is able to support a level of throughput, increase the logging level on its database server and measure to what extent that affects throughput.

Avoid the temptation to change multiple test characteristics (i.e., equation variables) in a single run, which would make it much harder to identify cause and effect relationships between test inputs and results.

When to start performance testing

A common practice is to start performance testing only after functional, integration, and system testing are complete; that way, you know that the target software is "sufficiently sound and stable" to ensure valid performance test results.

However, the *problem* with this approach is that it delays performance testing until the latter part of the development lifecycle. Then, if the tests uncover performance-related problems, you'll have to resolve problems with potentially serious design implications at a time when the corrections you make might invalidate earlier test results. In addition, your changes might destabilize the code just when you want to freeze it, prior to beta testing or the final release.

A better approach is to begin performance testing as early as you can, just as soon as any of your application components can support the tests. This will enable you to establish some early benchmarks against which you can measure performance as the components are developed. It is an especially apt way to begin testing a servlet; the multitier J2EE architecture may allow you to execute performance tests for all components in a single tier. Remember, however, that these tier-specific tests will not satisfy the need for end-to-end testing once the entire application is testable.

When to stop performance testing

The short answer is: *maybe never*.

The conventional approach is to stop testing once you have executed all planned tests and can see a consistent, reliable pattern of performance. This approach gives you accurate performance information at that instant in time. However, you can quickly fall behind by just standing still.[6] The environment in which clients will run your servlet will always be changing, so it's a good idea to run ongoing performance tests.

Undoubtedly, you'll run performance tests whenever you create new versions of the servlet. However, your clients, will likely install new versions of software in their environment (e.g., patches or other updates to their operating system, databases, Java JREs, and servlet container software) on schedules that are not in sync with your own servlet release schedule. It makes sense to monitor these changes and periodically revisit—and reexecute—your servlet performance tests.

Another alternative is to set up a continual performance test and periodically examine the results. You can "overload" these tests by making use of real world conditions. Of course, regardless of how well you plan, construct, and vet your "clean" in-house test network, you'll never be able to reproduce all the conditions that your servlet will have to contend with in the real-world Internet environment: intermittent delays between servlets and their supporting database when misconfigured routers make dynamic routing changes; spikes in network

traffic when news sites are bombarded by users during international incidents; and so forth. However, you can configure a reasonable test by dispersing the various components in your application across multiple, physically separate servers, establishing and maintaining a constant but low level of traffic, and monitoring the results over time.

Step 3: Planning the tests

So much for how to start and end the testing; now it's time to talk about what tests to run. The following sections describe various types of servlet performance tests.

A common thread: Measure servlet longevity

Although each type of performance test is different, they share a common thread: They measure specific aspects of how well the servlet under test can perform if it runs unattended for an extended period. As we noted at the beginning of this article, running unattended is a factor that distinguishes servlets from other types of programs. Once a servlet is instantiated and loaded by its container, it remains available to service requests until it is either made unavailable or until the servlet container is stopped. The application supported by the servlet should always be "on"— that is, it should be capable of running unattended 24/7.

One measure of longevity is *whether performance is consistent over time*. Does throughput start off at one level but then degrade over time? If so, the servlet may be experiencing a memory leak, or the servlet (or other application components) may be consuming excessive system resources, which adversely affects performance. We'll discuss more about servlet longevity as we describe various performance tests below, and we'll return to the issue of measuring system resource consumption in a later section.

Measure the data movement rate

The primary measurement you'll likely look for from performance or throughput tests is the rate at which the servlet can accept, process, and serve data. I mentioned earlier how the overall performance of the servlet will be affected by the configuration and performance of its container and Web server(s). You'll want to run capacity tests, applying increasing levels of usage/traffic and measuring the servlet's throughput. But what else should you do?

See how variations in test data affect performance. I described earlier the importance of understanding how the servlet under test processes data, as opposed to treating it as a black box into which you can "pour" input, extract output, and measure the results. When you execute throughput tests, you build on this understanding with varying sets of concrete data. For example:

- **Data sizes**. Are bottlenecks based on the number of bits being moved? On the number of discrete transactions (e.g., requests made to the servlet) that are performed? Which takes longer to process—a single 100-MB transaction, or one hundred 1-MB transactions?
- **Data formats**. Do different types of input files/data take more or less time to process? Are the bottlenecks caused by processing/converting some types of data? Earlier I referred to a hypothetical situation involving the processing of XML data. Do other types of supported input (e.g., HTML, comma separated values) result in greater or lesser delays? What about data types that may be unexpected, such as XML tags embedded in HTML files?

Find out where the servlet is really spending its time. It used to be that shipping was a large part of the cost of producing and delivering a product to market. Today, however, personnel costs are the biggest expense. (That's why US and European manufactured goods are often assembled in cheaper labor markets, from domestically created components.) There's a parallel situation when it comes to testing servlets: Throughput bottlenecks may not relate to the actual movement of data over a network, but rather to preparing and processing that data. When you build and run performance tests you will want to identify these bottlenecks by checking the following.

- **Data collection**. Is user input converted into a different form before it's processed? For example, does the servlet's client send requests to the servlet that include data in the form of serialized objects that the servlet converts to XML? Or does the client send the servlet a large `STRING` object that contains XML tags that must be parsed into objects by the servlet?
- **Data aggregation and calculation**. When the servlet responds to a request from a client, does it "fire off" a large number of requests to database or application servers? Remember: A distinguishing characteristic of servlets is that they generate dynamic content, so they will be accessing databases or other servers. Does the servlet under test have to wait for responses from slow servers? And once it receives those responses, is its logic so convoluted that it wastes time performing unnecessary calculations? Does it take a long time to determine whether it has received invalid input?
- **Data reporting**. In a Java-based model-view-control application architecture, JSPs are often the most efficient way to display output for users. Does the servlet make use of JSPs for output, or does it painstakingly generate its own output? Does the servlet have to perform all the data conversions that the client performed, but in reverse?
- **Caching**. Is the servlet caching commonly requested information, or is it recalculating everything, every time? Earlier, I described a scenario in which a test designer didn't understand that his automated test was invalidated by caching, which the software under test was taking advantage of. In performance testing, you determine whether the servlet IS taking advantage of caching.

Measure system resource consumption

When you measure the level at which the servlet under test consumes system resources such as memory, you'd like to see that level remain constant when the traffic level remains constant, and rise or fall in relationship to changes in the traffic level, over time. You don't want to see that resource usage level either increase in response to consistent traffic or consistently rise over time, regardless of the traffic level.

Which system resources should you be concerned with? Let's carry forward the assumption that network capacity— the rate at which you can push bits "over the wire"—will not be in short supply. This is a fairly safe assumption, given the high physical capacity of most internal and Internet networks. What does this leave us to worry about? Memory quickly comes to mind (pardon my word play), but you should also think about disk capacity and I/O rates.

Memory: Make sure the "garbage collector" is working for your servlet. One of the JVM's most useful features is its "garbage collector," which analyzes memory to find objects that are no longer in use and deletes them to free up memory. In spite of the garbage collector's best efforts, however, you can still have memory leaks in Java programs.

Let's look at a little background information on garbage collection. The garbage collector works by looking for objects in memory that are no longer referenced. A running Java program contains threads, and each thread

executes methods. Each method, in turn, references objects via arguments or local variables. We refer to these references collectively as a "root" set of nodes. The garbage collector works through the objects in each node, the objects *referenced in* each node, the objects referenced *by* those nodes, and so on. All other objects on the "heap" (memory used by all JVM threads[7]) are tagged as unreachable and are therefore eligible for garbage collection. Memory leaks occur when objects are reachable but no longer in use.[8]

So far so good, but nothing here is unique to servlets. Any Java program can hit a memory leak, can't it? Yes, in practice, a Java program that is invoked, runs for a short time, and exits, may include a memory leak. However, the program does not run long enough for that leak to grow very large, so even if the garbage collector does not run before the program exits, that is no great problem. In contrast, recall one of the distinguishing characteristics of servlets: They are meant to run unattended for extended periods. Accordingly, even a small memory leak can become a *big* problem for a servlet if the leak is allowed to grow over time.

When you do your testing, you'll want to simulate real-world conditions in which the servlet under test will have to function 24/7. So you'll have to establish a level of traffic between test clients and the servlet. Several tools (such as IBM Rational Robot) on the market can help you simulate user sessions. However, the client is not your main concern. What matters most is how the servlet reacts to a regular, sustained level of incoming requests.

To examine this, you can monitor memory usage in several ways. The easiest way is simply to use utilities supplied by the operating system. On Microsoft Windows, the Task Manager enables you to view total system memory in use as well as the memory being used by any single process. On Unix systems, similar tools such as "`sar`" (on Solaris), or "`vmstat`" (on BSD-based Unix flavors) or "`top`" will provide you with some basic measurements of memory usage. What you should check for is increases in memory usage by the servlet's container to handle the client traffic—and instances in which the memory is never released by the container when the traffic stops. If you do locate a pattern or memory loss with these basic tools, then you can investigate further with more specialized tools such as IBM Rational Purify to identify the offending classes/methods.

Disk capacity and I/O rates. My in-laws are really wonderful folks but a bit quirky (who isn't?): They never throw anything away. Some applications are like that, too. Unfortunately, this "pack rat" mentality can have an adverse effect on performance if it is unchecked. I've already mentioned that logging at too high a level can be a drain on performance. What else might cause problems for a servlet with "pack rat" tendencies?

Temporary files can cause problems. When you're dealing with large amounts of data, it's often a good idea to break it into more manageable pieces for processing or transmission, and then to reassemble it into a single (and large) mass only when necessary. Your servlet might work this way—that is, working with data in pieces and writing intermediate data to temporary work files to avoid saving it in memory. Building an application in this way provides an extra measure of auditing in the event that one or more servers used by the application fails while processing a transaction. If the intermediate data is made persistent, then the transactions can be reconstructed.

There are some potential problems with this approach, however. First, disk I/O operations take a toll on system performance; they can affect performance of the servlet simply because, even with today's high-speed hard disks, accessing data on a disk is always slower than accessing data in memory. And sooner or later, those temporary files will have to be cleaned out. Having the servlet "clean up" after itself by deleting these files can further affect its performance.

Verify the efficiency of software integrations

Integration testing is becoming a more and more important class of testing, as companies increasingly make use of software components (commercial or open source) supplied by third parties. In planning performance tests for your servlet, you have to consider not only whether integrations function successfully (between the servlet and its container and/or Web server; between the container or Web server and other software; and between the servlet itself and supporting software such as databases), but also the efficiency with which they function.

Dealing with the databases. As I mentioned earlier, for a servlet to generate dynamic content, using flexible selection criteria to customize (or "personalize") data for a specific user, the servlet must extract data from a persistent data store, most likely a database. The servlet's throughput is naturally dependent on the efficiency with which the servlet works with the database. In planning and executing performance tests for the servlet, here are some problems to watch for.

- **Permitting global queries**. What type of SQL query do you run if you're not exactly sure what you're looking for? Something like "select * from tableName," right? The database responds by sending you the entire contents of the table, and you can manually search through a huge amount of data for the piece of information you need. What happens if your servlet runs a query like this? Its performance will be degraded, as it has to process potentially very large amounts of data. What sorts of tests should you run to detect this problem? Actually, you may need to test the servlet's client more than the servlet itself; you'll want to verify that it is not easy for the servlet to request huge queries from the database. But what if you can't prevent this from happening?
- **Receiving too much data, even with a narrowly focused query**. What if the servlet makes queries that, even though narrowly focused, still return too much data? Some time ago, I worked on a project with a servlet that controlled user creation of data objects and "tasks" that acted upon those objects. The data objects, task definitions, and task results, including logging information, were all made persistent in a database. After running tests for a while, we found a performance problem: When more than a few dozen tasks had been executed, the client seemed to hang when it displayed a summary table for all tasks. As it turned out, the problem was simply that when the user selected the program option to display the task information, the servlet responded by performing multiple queries to the database. Some of these queries—for example, queries for task execution logs—resulted in large amounts of data from the database. This was a serious scalability problem, as these tasks could be scheduled to run automatically (similar to a Unix system "cron job"). The solution was to modify the servlet to query the database only for subsets of the task-related data—just enough to fill one screen at a time. When the user traversed the task data by moving from screen to screen, new queries would be run. Modifying the servlet to take many small "bites" instead of a few big ones improved performance dramatically.
- **Failing to reuse or close connections**. Another potential database-related bottleneck that can adversely affect servlet performance is the time it takes for the servlet to actually log into the database. There may be no way to alter the actual speed with which the database will accept new connections, but the servlet can be modified to create and use those connections more efficiently. The servlet can pool and reuse the connections so that it isn't constantly opening new connections every time it has to access the database. Also, the servlet can be written to ensure that connections are properly closed when they are no longer needed. Connections left open can tie up system and database resources and can be a drain on system performance (and therefore servlet performance). Figure 4 shows an example of a coding error (in pseudocode) that can result in connections being left open unnecessarily.

```
1.        try {
2.           <open database connection>
3.        }
4.        catch (exception) {
5.           <log error, and exit>
6.        }
7.        .
8.        .
9.        try {
10.          <execute SQL queries, process data>
11.       }
12.       catch (exception) {
13.          <log error, and exit>
14.       }
15.       .
16.       .
17.       try {
18.          <close database connection>
19.       }
20.       catch (exception) {
21.          <log error, and exit>
22.       }
```

Figure 4: Faulty code that leaves connections open unnecessarily

If the SQL query on line 10 fails and the code exits, the database connection that was opened on line 2 will remain open until it times out or is closed by some database housekeeping routine.

Step 4: Executing the Tests

Now that we've discussed the "whats" and "whys" of performance testing servlet-based applications, it's time to discuss the "hows." An approach that I've found successful is to divide the execution of the tests into four stages:

- Stage 1: Verify the subsystems and perform low-impact aerobics.
- Stage 2: Vary the equation #1: Load testing.
- Stage 3: Vary the equation #2: Performance testing.
- Stage 4: Conduct ongoing "health checks."

Stage 1: Verify the subsystems and perform low-impact aerobics

How to begin? There's a great temptation to approach performance testing in the same way that some people approach rental cars: "Let's hit the gas and see how fast it can go!" This approach may sound like fun, but it will probably be unproductive. You may have a hard time reproducing the exact input conditions that result in failures and an even harder time determining the cause and effect relationships that affect end-to-end operation of the software under test. Instead of that approach, I recommend the following measures.

- **Reexamine functional tests**. Start by reexamining functional tests you run to verify the operation of the application's individual subsystems. This approach involves looking at those subsystems in isolation to the greatest extent possible, and tracing through how the subsystems execute discrete transactions.

Functional tests are concerned with verifying that function X returns value Y. When you look at these tests in the context of performance tests, your concern is *how* (i.e., how efficiently) function X returns value Y. For example, if the function in question retrieves data from a database, and you run a test to return (three) records, the functional test will succeed. However, if retrieving the three records requires three separate queries to the database, and each query returns excess data, (recall the dangers of global database queries we discussed earlier), then the test will either fail or at least be marked as an area for further investigation.[9] To identify potential program inefficiencies, you will want to trace individual transactions, initially through each of the application's subsystems, including the servlet's methods, and ultimately through the entire set of subsystems. This can be difficult detective work, especially if you don't have access to the servlet's or application's source code or designers, or if the design is extremely complicated. Certain tools can help, such as IBM Rational Quantify, a tool that lets you profile the performance of your entire application, including subsystems for which you don't have source code. Although it's not platform independent, IBM Rational Quantify[10] offers versions that run on both Windows and Unix systems, respectively.

- **To stub or not to stub?** As you reexamine subsystem functional tests, you'll have to decide whether to replace some subsystems with stub routines. This allows you to verify those subsystems' operation and performance in isolation, but it involves constructing test versions of the actual product's subsystems. For example, the Web application that you're testing may consist of a Java client and a Java servlet that connect to a database via EJBs. If you can stub out a test client that runs on the same server as the servlet, and also stub out the database access by accessing static test data stored in files, then you can concentrate on verifying how (and how well) the servlet processes data. So, should you do this? If you're sure that the test subsystems that you construct can run cleanly, without introducing spurious bugs, then give it a try. If you're successful, you'll effectively resolve at least some of the variables in the performance testing equation.
- **Perform low impact aerobics/verify longevity**. What next? After you've traced though discrete transactions and verified the operation of the application's subsystems in isolation, then it's time to attack the application end-to-end and look at its longevity. As you recall, servlets are intended to run unattended for extended periods, so your goal is to verify that the application and its servlet(s) can sustain a low level of traffic over an extended time. You will assess how the application consumes system resources and otherwise operates during this time. Why should you start with a low level of traffic? To reduce the number of variables. If you start with a "big bang" of traffic, the servlet may fail, but it may be hard to determine the exact cause (e.g., it could be a memory leak,[11] problems with multiple thread coordination, database record locking, etc.) and therefore hard to debug. With a "single thread" of traffic, you'll be better able to connect causes and effects. What types of problems, other than memory leaks, disk usage, and database connections should you look for? It all depends on the design of the software under test, but you may find some silly things, such as counters that either never get reset or that fail when the transaction count gets too high.

Stage 2: Vary the Equation #1: Load Testing

What's next? Once you've confirmed that the application can walk, it's time to make it run.[12] What you'll do now is keep some elements (the test configuration) in the testing equation constant while you change other variables (the test data and number of simulated users).

Here's the sequence of actions.

- **First, define a test configuration and establish a baseline**. The testing you've performed so far will have involved relatively low levels of traffic and users for extended time periods. To some extent, this testing will have verified that the application and servlets can run independently for a long time, but it will not provide a real basis/baseline against which to measure the application's performance. Your first action should be to establish that baseline.

  The inputs for doing this are the hardware and software test configuration, the test data, the traffic or usage level, and the time period for which you want to run the test. How do you decide on the "shape" of the data and the number of users for this baseline? You may be fortunate enough to be working on a project for which the operational load parameters, including the hardware configuration, are exactly defined and specific configurations are "recommended." If this is the case, then the decisions are made for you. However, it's more likely that the load test parameters will be less precisely defined; you'll have to deal with "grey" areas and complete the definition yourself. Don't try this on your own. Instead, make use of other resources; marketing personnel, for example, will understand the configurations the application should support, and pre-sales and post-sales support personnel will understand the configurations that clients actually use.

  Be forewarned, though, that defining performance test configurations is sometimes an inexact science. Temptation can be strong to make decisions based on "gut" feel or a need to control configuration costs. Instead, you should gather information from multiple sources, make a decision based on the sum of that information, and then obtain "buy in" from all concerned parties. If this sounds difficult, well, it can be. But remember that a Java-servlet based application often encompasses multiple software components in addition to the servlet (the servlet container, databases, etc.) as well as multiple, potentially geographically separated, hardware servers.

- **Vary the data and number of users, and compare results to your baseline**. Once you've established a baseline, it's time to alter the test conditions by varying the test data and the number and characteristics of your simulated users.
  - **Varying the test data**. Part of the configuration you use to establish the test baseline consists of data that you pass to the target application. The "mix" you select for this data should be the best approximation possible of commonly used customer data. Most likely, your target clients will not use simply a single, static set of data; instead, their data will cover a whole ranges of values, sizes, formats, and so on.

    Varying the test data will help you look for throughput problems if the application under test (and its servlets) cannot handle these ranges of data. As we discussed earlier, you may find that the gating factor in the application's throughput is not the number of bits moved through the system, but the number of transactions. For example, a single transaction that moves 100 MB of data may run much faster than ten transactions moving 10 MB each. Likewise, the speed of data handling may depend on format. For example, plain text might be handled much faster that XML because the application is using inefficient parsers.

  - **Varying the number/characteristics of test users**. When you ran the "low impact aerobics" tests, your goal was to verify how well the application and its servlets were able to handle a single user's actions. Now, you want to start increasing the number of users. But wait! You can't simply assume that all users will perform the same tasks. The Web application under test probably supports

multiple user actions. You'll have to create classes of test users, distinguishing each class by the actions it performs.

In order to support these classes of tests, you'll need corresponding classes of data. One approach to defining these classes of tests and test data is software test design methodology that involves dividing inputs and outputs into groups or "classes"; tests involving individual members of each group yield results within the same group. This methodology is known as "equivalence class partitioning."[13] For example, one class of users may perform complex queries, while another writes new records/data, and still another deletes existing records, and so on. You'll have to perform tests with each class of users to establish baselines and then perform tests with multiple classes of users.

- **Should you vary both data and users in the same test?** Again, you should resist the temptation to make multiple, simultateous changes to variables in the performance test equation. It may seem like a good way to combine multiple actions into a single test, but you may then not be able to map the results to a single data or user change. If this happens, you'll have to take a step back, reset the variables, and then change only one at a time.

Stage 3: Vary the equation #2: Performance testing

In executing load tests, you work with a single "recommended" configuration. Since it's likely that the application under test supports multiple configurations, it's important to execute tests at those configurations' boundaries.

- **Testing at the boundaries — minimal and maximal hardware**. An obvious approach is to run tests on "minimal" and "maximal" configurations; the differentiating hardware characteristics are memory and CPU speed, and the differentiating software characteristrics are the versions of supporting utilities (e.g., databases).You'll want to verify that the smallest (and cheapest!) configuration the application officially supports is actually usable. You'll also want to quantify the application's performance on an optimal configuration. Some customers are willing to pay for high-end hardware if their return on investment is superior performance. What other configurations should you verify?
- **Testing at the boundaries — distributed subsystems**. Remember: Java servlets are managed by a container and run via a Web server. Also, the application may comprise multiple servlets and Web servers, as well as other supporting software subsystems such as databases. All of this means that you have to consider the extent to which the application's configuration is distributed across discrete servers in your performance testing. In this distributed context, a minimal configuration may mean using a smaller number of servers, and a maximal configuration may mean that the application's subsystems are distributed across a large number of discrete, and potentially geographically separated, servers.
    - **Multiple subsystems sharing servers**. If the application under test is aimed at target customers with a wide range of usage/capacity needs, then it will likely also support a wide range of configurations. At the low end, the application may support a configuration that allows many or all of its subsystems to be run on a single physical server. This trades off higher performance and capacity against lower cost and simplicity of installation, configuration, and maintenance. When you execute performance tests on this type of all-in-one configuration, you have to pay special attention to how the subsystems will share and compete for potentially scarce system resources, such as memory.

At one client site, we were running tests on a servlet that accessed a database, when we noticed that the size of the database process continued to grow in memory. This seemed strange, as the database was a well-known, off-the-shelf program. What we thought was a memory leak actually turned out to be a knowledge leak. We had overlooked the fact that the database could be configured to use either all the memory available or memory only up to a configured amount.

- **One server per subsystem**. When you execute tests on this type of "high-end" configuration, your focus shifts from assessing how well how the application's subsystems compete for resources on any one server to assessing how well those subsytems can handle interserver communications. As we discussed earlier, before you begin testing with this type of configuration, it's important to confirm that your test network isn't the gating factor in limiting the application's performance.

  What types of tests should you execute after you verify that the network is a suitable base? Well, as is often the case in testing software, your first step is to tear apart what you just created. You may want to run tests that simulate server outages (simply power down a server or two), so that you can observe how well the application's remaining subsystems react. Such outages will probably be more rare than instances in which a server is briefly unavailable because of network "glitches," or when a server is so heavily loaded that it becomes an application bottleneck. In such conditions, you'll want to "bounce" (i.e., restart) key subsystems (e.g., the servlet container's Web server) or tie up system resources by running multiple copies of memory-intensive programs to starve other key subsystems. By performing such tests, you'll assess how well the the application as a whole implements "retry" actions, as well as its ability to cope with real-world network conditions.

Stage 4: Ongoing "health checks"

Now you're all done, right? Well, maybe not.

To ensure that the application can continue executing in an evolving environment with constant updates to operating systems, databases, Java JREs, and servlet container software, you might have to run periodic tests.

This is the final stage of the testing. As we noted earlier, these tests will verify that changes in software components on which the application relies don't result in a general application failure. Remember, it's easy to fall behind by just standing still!

Closing thoughts

Java servlets represent a significant milestone in the evolution of Internet-based software. Their use is widespread today and will continue to grow. Odds are that if you're working with Internet software, then you're working with servlets. And, if you're testing servlets, the odds are very good that you'll be verifying their ability to support high-performance and high-capacity Web applications.

When you plan and execute performance tests for a Java servlet-based application, you should take into account more than just the "generic" factors that impact performance testing for any type of software (e.g., spurious traffic). Also consider servlet-specific factors that will affect the application. Although these factors enable the application to perform specific tasks, such as generating dynamic output, they may also constrain the servlet's performance (e.g., the container's performance will affect the servlet's performance).

References

Online resources

- Sun Java Servlet Web site: http://java.sun.com/products/servlet
- Sun Java J2EE Web site: http://java.sun.com/j2ee
- Sun Java JVM Specification: http://java.sun.com/docs/books/vmspec
- Tomcat Servlet Container: http://jakarta.apache.org/tomcat, http://java.sun.com/products/jsp/tomcat
- JUnit Test Framework Web site: http://www.junit.org
- “Does Java Technology Have Memory Leaks?” 1999 Java One Presentation, Ed Lycklama: http://industry.java.sun.com/javaone/99/pdfs/e618.pdf
- “An Introduction to Java Servlets," Hans Bergsten: http://www.Webdevelopersjournal.com/articles/intro_to_servlets.html
- “Learning Servlets”: http://www.javaskyline.com/learnservlets.html
- "FindTutorials.com”: http://tutorials.findtutorials.com/

Books

- Subrahmanyam Allamaraju, et al., *Java Server Programming, J2EE Edition*. Wrox Press Ltd., 2002.
- Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd edition. Addison-Wesley, 2000.
- Glenford J. Myers, *The Art of Software Testing*. Wiley, 1979.
- Larne Pekowsky, *Java Servlet Pages*. Addison-Wesley, 2000.
- George Baklarz, and Bill Wong, *DB2 Universal Database v8 Database Administration Certification Guide*. IBM Press, 2003.

Journal articles

- Michael S. Dougherty, “Optimizing for Top Performance.” *DB2 Magazine*, Oct. 2002. http://www.db2mag.com/db_area/archives/2002/q4/Webdev.shtml
- Len DiMaggio, “Testing at the Boundaries Between Integrated Software Systems - Part I: Where to Begin.” *The Rational Edge*, April 2003.
- Len DiMaggio, “Testing at the Boundaries Between Integrated Software Systems - Part II: A Roadmap for Testing.” *The Rational Edge*, May 2003.

Notes

[1] The blanket term “performance testing” is used in this article to describe a wide variety of software tests. IBM Rational Unified Process,® or RUP,® differentiates among types of performance testing such as “load” testing (in which the test configuration remains constant while operational conditions such as the test usage level varies), “performance” testing (in which the configuration varies but the operational conditions remain constant) and “stress” testing (which places the software under test under extreme conditions). The majority of tests described in this article fall into the “load” and “performance” types. Details on the specific tests and the sequence in which they should be run are in the **Executing the tests section** of the article. Details on RUP are available at http://www.rational.com/products/rup/index.jsp and in *The Rational Unified Process: An Introduction*, second

edition, by Philippe Kruchten (Addison-Wesley, 2000).

[2] Java(TM) Servlet Specification (http://java.sun.com/products/servlet)

[3] Several servlet containers are in widespread use these days. The easiest (and cheapest) way to begin is with Tomcat, a free, open-source implementation of Java Servlet and JSP technologies built by the "Jakarta" project at the Apache Software Foundation. Start at http://jakarta.apache.org/tomcat. You'll find user documentation, sample servlets, and links to other Jakarta projects. Tomcat is also bundled into the J2EE reference implementation from Sun and can be accessed at: http://java.sun.com/products/jsp/tomcat.

[4] The HttpServlet class implements methods (which can be overridden) for each of the request types supported by the HTTP protocol: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE. The RFC (#2068) for the protocol is available at many Web sites, for example: http://www.rfc-editor.org/rfc/rfc2068.txt

[5] This program, and other useful JVM memory usage related topics, can be found at: http://tutorials.findtutorials.com/read/id/216 (This is a really useful Web site—you'll find many great tutorials here.)

[6] I'd like to take credit for this line, but I can't. It's my favorite line from the play "Inherit the Wind" by Jerome Lawrence and Robert E. Lee (later made in to a great movie staring Spencer Tracy).

[7] As defined in the JVM Specification. You can actually print a free copy (only once!) from: http://java.sun.com/docs/books/vmspec

[8] In a 1999 Java One presentation, Ed Lycklama coined the phrase "loiterer" to describe these objects. His presentation (http://industry.java.sun.com/javaone/99/pdfs/e618.pdf) is a great place to start if you want to learn more about Java memory leaks. Also, if you do a Web search for "Java memory leak" you'll find many references to subsequent journal articles by Lycklama as well.

[9] An example of a useful tool for tracking open database connections is a DB2 "Event Monitor." These monitors are created using the SQL Data Definition Language. For details (and lots of other good DB2 information) see : *DB2 Universal Database v8 Database Administration Certification Guide* by George Baklarz and Bill Wong (IBM Press, 2003, p. 689-702).

[10] http://www.rational.com/products/quantify_nt/index.jsp

[11] If you're looking for memory leaks, then a tool you should consider is IBM/Rational Purify (http://www.rational.com/products/pqc/index.jsp). This multiplatform runtime analysis tool can be a big help in tracking down the locations of memory leaks.

[12] I really have to give credit where credit is due on this statement. When my wife and I bought our house, the inspector remarked to us that we shouldn't worry about the age of the house (built in 1918), as "it never ran; it only walked." In other words, the house was always well cared for by its owners (who luckily avoided the 1970s rage for pastel-colored bathtubs)!

[13] Glenford J. Myers, *The Art of Software Testing* (Wiley, 1979).

About the author

Len DiMaggio is a software quality engineer and manager at IBM Rational. Prior to joining Rational, he led software testing teams at BBN, GTE, and Genuity. Len has published articles on software testing in *STQE*, *Software Development*, *Dr. Dobbs Journal*, and the *QAI Journal*.