

6 Best Practices for Better WebSphere Performance

Turbocharge your WebSphere applications.

By Michael S. Pallos, Senior Solution Architect, Candle Corporation



When architecting WebSphere Application Server (WAS) applications, software engineers are faced with many complex issues and choices. Do you incorporate nested transactions, flat transactions, compensating transactions, or no transactions at all? Do you use JDBC calls from a servlet to access local application server data? Or, do you design data access using an Enterprise JavaBean (EJB) entity bean?

There is no one solution for all organizations. Currently, the industry doesn't have the knowledge required to build a single architecture that satisfies all business requirements, and perhaps never will. Why? Because corporations are different, choose to operate differently, and have a vast array of business requirements. And, as any good architect will inform you, business requirements drive technology.

There are many complex choices, all of which impact application performance. According to IBM, designing e-business applications with acceptable performance characteristics is one of the most challenging aspects of the development process. In this article, I'll share with you some hard-won lessons learned, best practices, and engineering ideas.

As I architect complex distributed solutions, some common implementable design themes/patterns always seem to surface. Although these six items may not all apply to your organization's business requirements, they seem to be of interest to most WebSphere application designers. These areas include:

1. A proven design pattern
2. Servlets and JavaServer Pages (JSP)
3. Sessions

4. Connecting to legacy systems and databases
5. Memory usage
6. Enterprise JavaBeans (EJB)

Design pattern

Developers often use the Model-View-Control (MVC) design pattern when building Java applications. The MVC pattern separates the process of serving a user request into three distinct pieces:

Model—Responsible for the data or business process (persistence/business logic)

View—Responsible for the user's display (CUI/GUI)

Controller—Responsible for the flow of the application (traffic cop)

When you design applications, you must separate the presentation from the business logic and make sure your application is J2EE-compliant. Often, when developing an application using JSPs, you're tempted to place business logic in the JSP. This trap can easily snowball, creating a non-scalable, maintenance nightmare for your organization in the future. Make sure your application design is J2EE-compliant and separates the presentation from the business logic. Following industry standards is always the safe, strategic choice. A vendor may offer a tactical, non-compliant quick-fix. However, in the long run, these widgets will erode the benefit of the initial time savings. Following the MVC design pattern will ensure a tactical implementation approach, yielding a long-term strategic payback (figure 1). For more information on the MVC pattern, see Jason Gibson and John Trollinger's articles in the January/February, March, April, May/June, and September 2002 issues.

VERSIONS

- IBM WebSphere Application Server 5/4/3.5

TECHNOLOGIES

- Enterprise JavaBeans
- JavaServer Pages
- Design Patterns

Michael S. Pallos is a senior solution architect for Candle Corporation, actively architecting WebSphere enterprise solutions for customers throughout the United States. He is an IBM certified e-business designer, e-business technologist, and WebSphere MQ specialist. Pallos has more than 17 years of experience in distributed systems, holding numerous copyrights for international applications he has developed. He is also a student at Walden University, working on his Ph.D. in applied management and decision sciences—information technology. He resides in Florida with his wife, Laura, and two children, Danielle and Mike. <http://www.candle.com>, michael_pallos@candle.com.

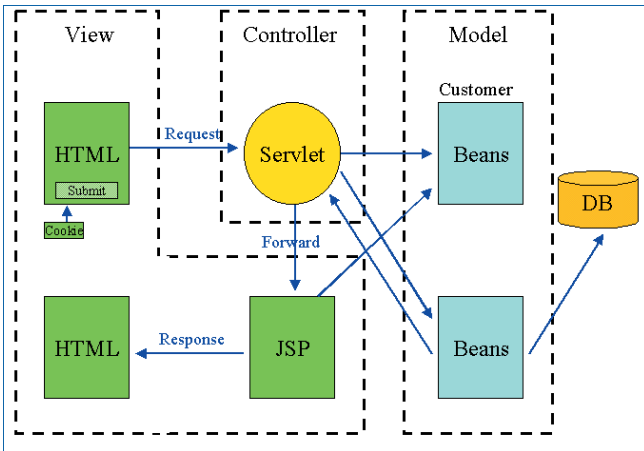


Figure 1: Model-View-Controller example—This simple guide offers separation of (a) presentation via JSPs, (b) interfacing between business logic and presentation via servlets, and (c) business logic and data access via EJBs or other frameworks.

Servlets and JSPs

Good servlet and JSP design is based on a few simple guiding principles. First, keep servlets thin. Ideally, a servlet should be approximately 50 lines of code (*good* code, that is). Someone I worked with in the UNIX/C days of the early 1990s believed all functions should fit on a screen. At first, I thought he was a bit idealistic and over simplistic. But as time progressed, I found that small, clean code is extremely maintainable, reduces ongoing maintenance costs, and usually increases application performance. The same is true with servlets.

Avoid unnecessary synchronize statements. A synchronize statement is Java's mechanism to ensure that only one thread at a time will access a piece of code and the instance variables of the locked object. Synchronizing large amounts of code effectively makes the application single-threaded (figure 2). In multi-threaded applications, synchronize statements are necessary. Use them sparingly and with caution to avoid deadlock situations.

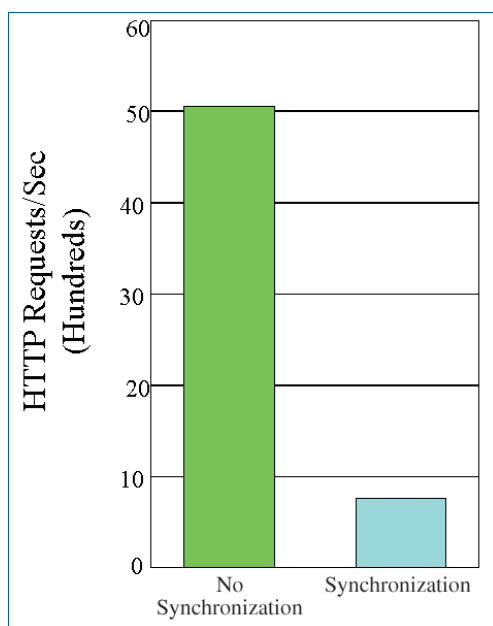


Figure 2: Servlet synchronization—Throughput decreases as large portions of code are synchronized (source: <http://www.developer.ibm.com>).

Avoid serializing servlets. Serialization is the expensive process of writing the servlet to disk, or flattening it out to transfer over the wire. As an alternative, consider assigning a unique servlet to each application task. Then, each servlet can return a few basic choices, and you can share data among servlets via a persistent store or session state. Also, avoid the SingleThreadModel, which ensures there is never more than one request thread accessing a single instance of your servlet.

Remember your first servlet? If your code was like mine, there were System.Out.println() statements throughout the code. It was easy, and I could trace what was going on. Prior to effective Java debuggers, print lines were great. Well, those days are gone. According to IBM, print line statements can reduce the number of HTTP requests per second from 72,000 per second to 10,000 per second. WebSphere Studio Application Developer (WSAD) has a state-of-the-art debugger. Don't make your code seven times slower to assist with debugging.

Another habit left over from the UNIX/C days is string concatenation. String concatenation creates temporary objects to assist with the process. This is expensive and is no longer needed. Java offers StringBuffer that are two and a half times as fast as string concatenation.

Sessions

As with servlets, it's good design practice to keep session objects small. Design your application so it only uses session data for the life of the session; this way, the data is destroyed after the session ends. Also, even though implicit invalidation is possible, be sure to explicitly invalidate all sessions. During implicit validation, the program is relying on WAS to free up the memory. This introduces a timing issue where WAS may serialize the object prior to session timeout termination. When you explicitly invalidate the object, you ensure the object is released immediately. For this reason, you should also keep all object graphs out of stateful sessions. Because object graphs are large, the process required for read/write time and disk i/o of the stateful session data is extended, creating slower applications. Often times, you may be tempted to use session EJBs as an alternative to entity EJBs. This tactical solution is short-sighted, non-scalable, and will cause problems as the system attempts to horizontally scale. For detailed information on how session and entity beans differ, go to <http://www.Advisor.com/doc/07570>.

For better performance when creating session objects, don't create HttpSession objects by default on JSPs. Even though this is part of the J2EE spec, there is an impact on WAS performance. Use `<%@page session="false"%>` for best performance. This may sound contradictory: Do you follow the J2EE spec or not? Follow the spec, but if one of the spec's choices creates slow performance, follow another choice contained within the spec that offers better processing speed.

A final point to remember when accessing data: Consider writing your own Java Database Connectivity (JDBC) calls. Partitioning servlet state data and storing it using JDBC, instead of storing a single object using HTTP, improves performance. This minimizes the size of state data, and you can double the speed. Remember to clean up the session's table explicitly.

Connecting to databases and back-end systems

Leveraging back-end systems, including databases, can be extremely complex. The back-end application design and numerous variations offer a diverse array of options. For example, will integration be intrusive or

non-intrusive in the application? If the legacy application is a CICS program, is the application Distributed Program Link (DPL) enabled, or is the application using WebSphere MQ on MVS? Perhaps the solution will use CICS Bridge.

Whatever the back-end system, there are a few guiding principles for building your application. First, keep the back-end connections clean. Whatever middleware you've chosen, or are considering, make sure it can handle the required throughput and won't choke under a heavy load. Two ways to assist with the workload processing are to use connection pooling or multi-threaded traffic. WebSphere connection pooling returns results up to four times faster than non-connection pooling results (figure 3).

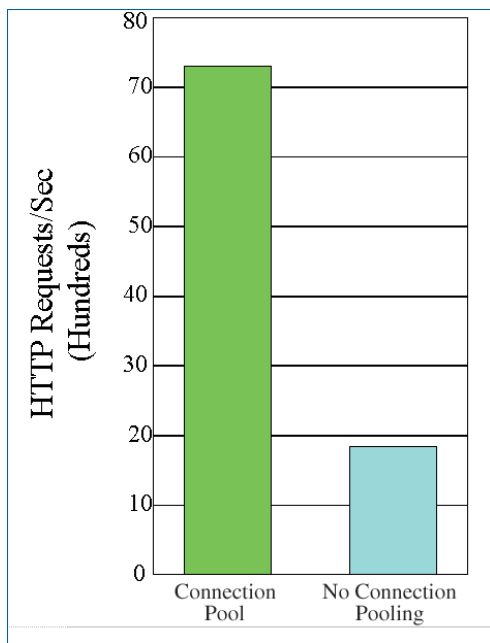


Figure 3: WebSphere connection pooling—When using connection pooling, keep one connection per thread, manage the pool size from a central point, always close all connections, and return idle connections to the pool promptly (source: <http://www.developer.ibm.com>).

Setting up the data source connection is expensive. Therefore, obtain the data source once and keep reusing it (figure 4). The best way to do this is to create the data source in the object initialization phase, such as the `Servlet.init()` method.

There are more JDBC drivers on the market than you might expect. So, make sure you've obtained the best driver for your environment. Your choice should include a Type 2 JDBC driver. Gather any performance data you can find on the driver and work around the specific weaknesses. If your application or corporation is large enough, you may be able to work with the vendor to keep the driver up-to-date.

Memory usage

There's just never enough memory. I haven't yet heard a customer or vendor say, "No, let's go with less memory." Still, you can't address all application memory with a SIM card. There are three aspects of memory to address when designing WebSphere applications: allocation, garbage collections, and health checks.

To optimize memory allocation, avoid dynamic allocation. Static creates a class member, not an instance member, letting a single copy of the variable belong to all objects of that class type. This sharing

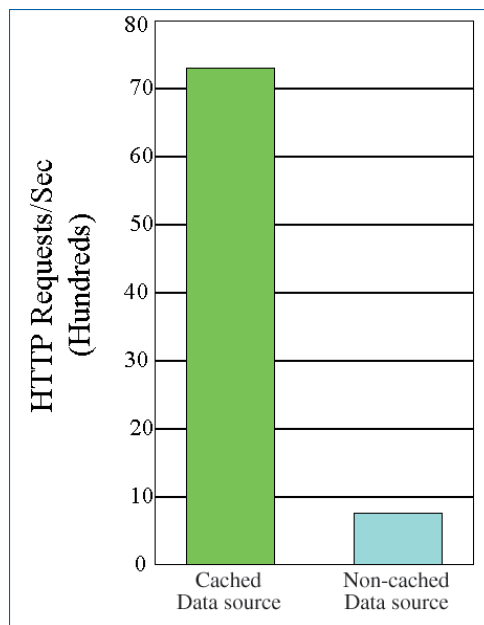


Figure 4: Reuse data sources—Reusing data sources minimizes the number of JNDI lookups, which are expensive (source: <http://www.developer.ibm.com>).

increases performance. When a variable is final, Java knows the value of the variable is constant and is then able to perform optimizations. Using a static final increases performance. Also, watch for hidden object creation. I touched on this in the servlet and JSP section. Use streams instead of string concatenation, and avoid using `PrintWriter`.

To optimize garbage collection, pool and reuse objects as much as possible. If applicable, use EJB intermediate caching to speed up processing. Also, keep the object size at the right level for your application. This is true for servlets, JSPs, and EJBs. Remember the 50-lines-of-code-per-servlet guideline.

You can keep memory healthy by watching for memory leaks. One way to identify a leak is to monitor the box running the application. If memory is slowly creeping up prior to a crash, you most likely have a memory leak. Ensure containers aren't holding objects no longer in use. Optimizing network traffic will also help keep memory healthy. Keep the page size, traffic to back-end systems, databases, and message sizes as small as possible.

Enterprise JavaBeans

Enterprise JavaBeans rock! Okay, that's just my personal perspective. But I experienced the Distributed Computing Environment (DCE) craze, and remember thinking any framework with 620 functions couldn't be all that bad. However, DCE was too complex. Then I was on the CORBA bandwagon. Interface Definition Language (IDL) was great. Sure, it was another language on top of the language I was developing in, but for distributed object development, it was ideal. Just run the IDL through the preprocessor and out pops two object files: the client and server object code, or stub and skeleton. I didn't think distributed object development life could get any better.

Then, EJBs came along. You just create the EJB in the WSAD environment and press a button. Now the environment creates the stub and skeleton for you, all without having to write in a third IDL language. The name may have changed from stub and skeleton to stub and tie; still, this is much easier to use. The Java Naming Directory Interface (JNDI) is automatically updated, saving you time and aggravation. Life is good.

EJBs are a great idea, but they aren't a one-size-fits-all solution. Some applications don't even need EJBs. You shouldn't use EJBs for the sake of using EJBs. Your application may not need them at all. The first step to EJB development is to ensure your application is going to require distributed beans for implementation.

To access an EJB, you need another bean. Clients don't access EJBs directly. Therefore, the choices made on various access methods impact WAS performance. Here are some options for EJB development. First, access entity beans (entity beans are EJBs that deal with persistence) from session beans (figure 5). This will triple the performance gained over accessing entity beans from the client code. The session beans are the entity bean "gate-keeper." This reduces the number of remote method calls. Access via session beans also provides a single transaction context for entity beans. Access from one session bean creates only one transaction, whereas access from client code causes each method call to become a transaction.

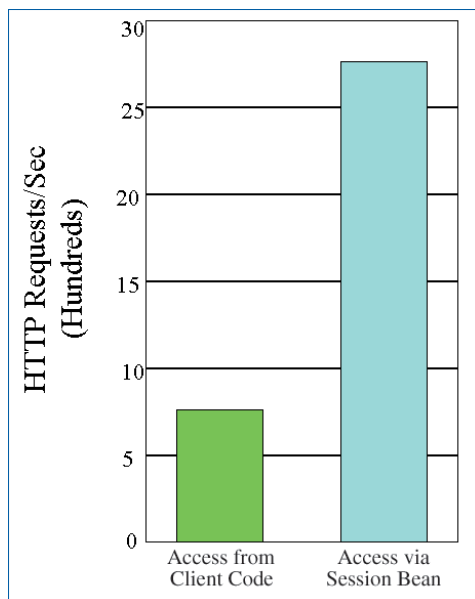


Figure 5: Entity bean access—Accessing entity beans from session beans and not directly from client code increases performance (source: <http://www.developer.ibm.com>).

Secondly, any time you have to locate an EJB, WAS must go to the JNDI to resolve the EJB home, which is expensive. Therefore, after an EJB home is resolved, a good design principle is to cache the EJB homes and reduce the JNDI calls (figure 6).

Finally, another way to double processing time is to mark all getter methods as read-only in the deployment descriptor. Otherwise, you'll create a transaction every time you call a method. After all, if it's a getter method, you won't alter the data, anyway. You should also reduce the Tx (transaction) isolation level if possible. There are four Tx levels:

TRANSACTION_READ_UNCOMMITTED: The transaction can read uncommitted data.

TRANSACTION_READ_COMMITTED: The transaction can't read uncommitted data from other transactions.

TRANSACTION_REPEATABLE_READ: The transaction is guaranteed to read back the same data on each successive read.

TRANSACTION_SERIALIZABLE: All transactions are serialized, or fully isolated from one another.

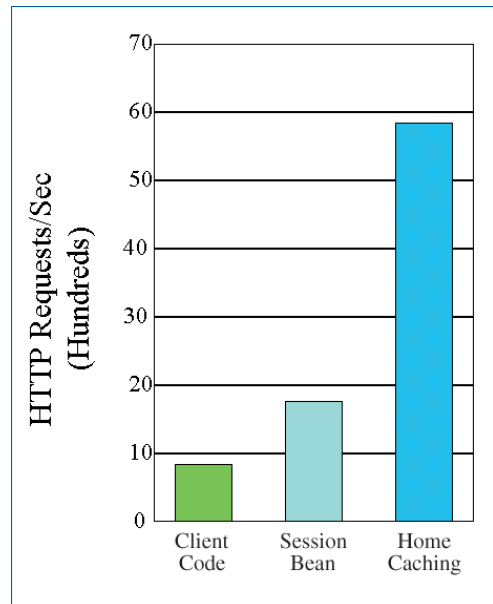


Figure 6: Reuse EJB homes—Simple applications can cache the home. For complex applications, consider creating an EJB home locator and caching class (source: <http://www.developer.ibm.com>).

If you don't need to use TX_SERIALIZABLE, then don't. It's tempting when you're new to EJBs to set this Tx isolation level to ensure data won't be misread and that phantom records or non-repeatable reads don't occur. Still, this is the most restrictive and isolated Tx level and your application will pay for it in performance. Whenever possible, use the TX_READ_UNCOMMITTED, which has the lowest overhead. Learning the different Tx meanings and properly using them in your EJB application will provide great processing rewards. Don't fall into the just-lock-it-down trap, otherwise your applications may not meet service-level requirements and be rejected due to unacceptable response times. EJBs provide a lot of power and flexibility, and you must use them correctly.

Meet business requirements

Distributed application architecture can be challenging. By focusing on the complexities of WebSphere application development, you can occasionally lose sight of the bottom-line business requirements of a specific application. It's the results that the application offers to the organization that are important. As such, performance and response times are critical factors to the real-world success of application deployment. Focusing on application performance during the design phase—and throughout the development and deployment lifecycle—ensures that new applications will have the flexibility, performance, and scalability to support new corporate requirements. **ADVISOR**

RESOURCES

- Brown, K., Craig, G., Hester, G., Niswonger, J., Pitt, D., Stinehour, R., *Enterprise Java Programming with IBM WebSphere*. Addison Wesley, 2001.
- Hall, M., *Core Servlets and JavaServer Pages*. Sun Microsystems. Prentice Hall PTR, 2000.
- *IBM Framework for e-Business Technology, Solution, and Design Overview* [Redbook]. IBM, 2001.
- Monson-Haefel, R., *Enterprise JavaBeans*. O'Reilly, 2000.
- <http://www.developer.ibm.com>