# IBM WebSphere Developer Technical Journal: The top Java EE best practices

Skill Level: Introductory

Keys Botzum
Senior Technical Staff Member
IBM

Kyle Brown
Distinguished Engineer
IBM

Ruth Willenborg
Senior Technical Staff Member
IBM

Albert Wong
I/T Architect
IBM

24 Jan 2007

This is an updated version of a similarly-named article published in the IBM® WebSphere® Developer Technical Journal in 2004. This revision takes into account changing technology trends and, more importantly, recommends certain practices that the authors assumed would be commonly followed, but, as they have learned, are not.

From the IBM WebSphere Developer Technical Journal.

## Introduction

Over nearly the entire last decade, much has been written about Java™ Platform, Enterprise Edition (Java EE) best practices. There are now dozens of books and hundreds (perhaps more) of articles that provide insight into how Java EE applications should be written. In fact, there are so many resources -- often with contradictory recommendations -- that merely navigating this maze has itself become an obstacle to adopting Java EE. Therefore, to provide some simple guidance for customers entering this world, we have compiled this best-of-the-best

list of what we feel are the most important and significant best practices for Java EE. Despite our earnest attempts, however, we weren't able to capture everything that needed to be said in a neat top-ten list. Thus, in order to avoid omitting critical best practices -- and also to honor the growth of Java EE -- our list instead is an essential "Top 19" best practices for Java EE.

## The best of the best practices

1. **Always use MVC.**
2. **Don't reinvent the wheel.**
3. **Apply automated unit tests and test harnesses at every layer.**
4. **Develop to the specifications, not the application server.**
5. **Plan for using Java EE security from Day One.**
6. **Build what you know.**
7. **Always use session facades whenever you use EJB components.**
8. **Use stateless session beans instead of stateful session beans.**
9. **Use container-managed transactions.**
10. **Prefer JSPs as your first choice of presentation technology.**
11. **When using HttpSessions, store only as much state as you need for the current business transaction and no more.**
12. **Take advantage of application server features that do not require your code to be modified.**
13. **Play nice within existing environments.**
14. **Embrace the qualities of service provided by the application server environment.**
15. **Embrace Java EE, don't fake it.**
16. **Plan for version updates.**
17. **At all points of interest in your code, log your program state using a standard logging framework.**
18. **Always clean up after yourself.**
19. **Follow rigorous procedures for development and testing.**

## 1. Always use MVC.

**Cleanly separate business logic (Java beans and EJB components) from controller logic (servlets/Struts actions) from presentation (JSP, XML/XSLT). Good layering can cover a multitude of sins.**

This practice is so central to the successful adoption of Java EE that there is no competition for the #1 slot. Model-View-Controller (MVC) is fundamental to the design of good Java EE applications. It is simply the division of labor of your programs into the following parts:

a. Those responsible for business logic (the Model -- often implemented using Enterprise JavaBeans™ or plain old Java objects).

b. Those responsible for presentation of the user interface (the View).

c. Those responsible for application navigation (the Controller -- usually implemented with Java servlets or associated classes like Struts controllers).

There are a number of excellent reviews of this topic with regard to Java EE; in particular, we direct interested readers to either [Fowler] or [Brown] (see Resources) for comprehensive, in-depth coverage.

There are a number of problems that can emerge from not following basic MVC architecture. The most problems occur from putting too much into the View portion of the architecture. Practices like using JSP tag libraries to perform database access, or performing application flow control within a JSP are relatively common in small-scale applications, but these can cause issues in later development as JSPs become progressively more difficult to maintain and debug.

Likewise, we often see migration of View layer constructs into business logic. For instance, a common problem is to push XML parsing technologies used in the construction of views into the business layer. The business layer should operate on business objects -- not on a particular data representation tied to the view.

However, just having the proper components does not make your application properly layered. It is quite common to find applications that have all three of servlets, JSPs, and EJB components, where the majority of the business logic is done in the servlet layer, or where application navigation is handled in the JSP. You must be rigorous about code review and refactoring to ensure that business logic is handled in the Model layer only, that application navigation is solely the province of the Controller layer, and that your Views are simply concerned with rendering model objects into appropriate HTML and Javascript™.

The value of this recommendation should be even clearer today than in the original version of this article. User interface technologies change rapidly and tying business logic to the user interface makes changes to "just the interface" deeply impact

existing systems. Just a few years ago, user interface developers for Web applications could choose from servlets and JSPs, struts, and perhaps XML/XSL transformation. Since then, Tiles and Faces have become popular, and now AJAX is gaining a strong following. It would be a shame to have to redevelop an application's core business logic every time the preferred user interface technology changes.

## 2. Don't reinvent the wheel.

**Use common, proven frameworks like Apache Struts, JavaServer Faces, and Eclipse RCP. Use proven patterns.**

Back when we first started helping educate our clients in how to use the then-emerging Java EE standards, we discovered (as did many others) that developing a framework for user-interface development significantly improved developer productivity over building UI applications directly to the base servlet and JSP specifications. As a result, many companies developed their own UI frameworks that simplified the task of interface development.

As open-source frameworks like Apache Struts began to develop [Brown], we believed that the switchover to these new frameworks would be automatic and quick. We thought that the benefits of having an open-source community supporting the framework would be readily apparent to developers, and that they would gain universal acceptance very rapidly -- not only for new development, but in retro-fitted applications as well.

What has proven surprising is that this has turned out to not be the case. We still see many companies maintaining or even developing new user-interface frameworks that are functionally equivalent to Struts or JSF. There are many reasons why this could be true: organizational inertia, "not invented here" syndrome, lack of perceived benefit in changing working code, or possibly even a slight sense of hubris in thinking that you could do things "better" than the open-source developers did in a particular framework.

However, the time is long-past when any of these reasons is worth using as an excuse not to adopt a standard framework. Struts and JSF are not only well accepted in the Java community, but fully supported within the WebSphere runtimes and Rational® tool suites as well. Likewise, in the rich client arena, the Eclipse RCP (Rich Client Platform) has also gained wide acceptance for building standalone rich clients. While not a part of the Java EE standard, these frameworks are now a part of the Java EE community, and should be accepted as such.

Nearly as hubristic as not using an off-the-shelf UI framework is ignoring the lessons captured in [Alur] and [Fowler]. These two books detail the most common reusable patterns that occur in Enterprise Java applications. From simple patterns like session facade (discussed in a later recommendation) to more complex patterns like Fowler's persistence patterns (which have been implemented in many open-source persistence frameworks), these works capture the accumulated wisdom of the Java elders. With apologies to Santayana, those who do not learn from the past are condemned to repeat it -- if they're lucky enough to get the chance after their first

failures.

## 3. Apply automated unit tests and test harnesses at every layer.

**Don't just test your GUI. Layered testing makes debugging and maintenance vastly simpler.**

There has been quite a shake-up in the methodology world over the past several years as new, lightweight methods that call themselves Agile (such as SCRUM [Schwaber] and Extreme Programming [Beck1] in Resources) become more commonplace. One of the hallmarks of nearly all of these methods is that they advocate the use of automated testing tools to improve programmer productivity by helping developers spend less time regression testing, and to help them avoid bugs caused by inadequate regression testing. In fact, a practice called Test-First Development [Beck2] takes this practice even further by advocating that unit tests be written prior to the development of the actual code itself. However, before you can test your code, you need to isolate it into testable fragments. A "big ball of mud" is hard to test because it does not do a single, easily identifiable function. If each segment of your code does several things, it is hard to test each bit for correctness.

One of the advantages of the MVC architecture (and the Java EE implementation of MVC) is that the componentization of the elements make it possible (in fact, relatively easy) to test your application in pieces. Therefore, you can easily write tests to separately test persistence, session beans, and portions of the user interface outside of the rest of the code base. There are a number of frameworks and tools for Java EE testing that make this process easier. For instance, JUnit, which is an open-source tool developed by junit.org, and Cactus, which is an open source project of the Apache consortium, are both quite useful for testing Java EE components. [Hightower] discusses the use of these tools for Java EE in detail.

Despite all of the great information about deeply testing your application, we still see many projects where it is believed that if the GUI is tested (which may be a Web-based GUI or a standalone Java application), then the entire application has been comprehensively tested. GUI testing is rarely enough. There are several reasons for this.

1. With GUI testing, it's difficult to test every path through the system; the GUI is only one way of affecting the system. There may be background jobs, scripts, and various other access points that also need to be tested -- but these often don't have GUIs associated with them.

2. Testing at the GUI level is very coarse-grained. A GUI tests how the system behaves at the macro level of the system, meaning that if problems are found, entire subsystems must be considered, making finding any bugs identified extremely difficult.

3. GUI testing usually can't be done well until late in the development cycle

when the GUI is fully defined. This means that latent bugs won't be found systematically until very late.

4. Average developers probably don't have access to automatic GUI testing tools. Thus, when a developer makes a change, there is no easy way for that developer to retest the affected subsystem. This actually discourages good testing. If the developer has access to automated code level unit tests, the developer can easily run them to make sure the changes don't break existing function.

5. If automated builds are done, it is fairly easy to add an automated unit testing suite to the automated build process. By doing this, the system can be rebuilt regularly (often nightly) and regression tested with little human intervention.

In addition, we must emphasize that distributed, component-based development with EJBs and Web services makes testing your individual components absolutely necessary. When there is no GUI to test, you must then fall back on lower-level tests. It is best to start that way, and spare yourself the headache of having to retrofit your process to include those tests when the time comes to expose part of your application as a distributed component or Web service.

In summary, by using automated unit tests, defects are found sooner, defects are easier to find, testing can be made more systematic, and thus, overall quality is improved.

## 4. Develop to the specifications, not the application server.

**Know the specifications by heart and deviate from them only after careful consideration. Just because you can do something doesn't mean you should.**

It is very easy to cause yourself grief by trying to play around at the edges of what Java EE anables you to do. We find developers dig themselves into a hole by trying something that they think will work "a little better" than what Java EE allows, only to find that it causes serious problems in performance, or in migration (from vendor to vendor, or more commonly from version to version) later. In fact, this is such an issue with migrations, that [Beaton] calls this principle out as the primary best practice for migration efforts.

There are several places in which not taking the most straightforward approach can definitely cause problems. A common one today is where developers take over Java EE security through the use of JAAS modules rather than relying on built-in spec compliant application server mechanisms for authentication and authorization. Be very wary of going beyond the authentication mechanisms provided by the Java EE specification. This can be a major source of security holes and vendor compatibility problems. Likewise, rely on the authorization mechanisms provided by the servlet and EJB specs, and where you need to go beyond them, make sure you use the spec's APIs (such as getCallerPrincipal()) as the basis for your implementation. This way you will be able to leverage the vendor-provided strong security infrastructure

and, where business needs require, support more complex authorization rules. (For more on authorization, see [Ilechko].)

Other common problems include using persistence mechanisms that are not tied into the Java EE spec (making transaction management difficult), relying on inappropriate Java Standard Edition facilities (like threading or singletons) within your Java EE programs, and "rolling your own" solutions for program-to-program communication instead of staying within supported mechanisms like Java 2 Connectors, JMS, or Web services. Such design choices cause no end of difficulty when moving from one Java EE compliant server to another, or even when moving to new versions of the same server. Using elements outside of Java EE often causes subtle portability problems. The only time you should ever deviate from a spec is when there is a clear problem that cannot be addressed within the spec. For instance, scheduling the execution of timed business logic was a problem prior to the introduction of EJB 2.1. In cases like this, we might recommend using vendor-provided solutions where available (such as the Scheduler facility in WebSphere Application Server), or to use third-party tools where these are not available. Today, of course, the EJB specification now provides for time-based function so we encourage the use of standard interfaces. In this way, maintenance and migration to later spec versions becomes the problem of the vendor, and not your own problem.

Finally, be careful about adopting new technologies too early. Overzealously adopting a technology before it has been integrated into the rest of the Java EE specification, or into a vendor's product, is often a recipe for disaster. Support is critical -- if your vendor doesn't directly support a particular technology, you should carefully consider if you should use it. People (particularly developers) tend to focus too much on easing the development process and neglect to consider the long term consequences of depending on large amounts of code developed outside your organization which is not supported by a vendor. We've seen far too many project teams that are enamored with new technology (for example, the latest open source framework) and quickly become dependent upon it without considering the very real costs to the business. Frankly, the decision to use any technology beyond what you've bought from your vendors should be carefully reviewed by corporate architecture, business, and legal teams (or their equivalent in your environment), just as normal product purchasing decisions are evaluated. After all, with rare exceptions, most of us are in the business of solving business problems, not advancing technology for the sheer fun of it.

## 5. Plan for using Java EE security from Day One.

**Turn on WebSphere security. Lock down all your EJBs and URLs to at least all authenticated users. Don't even ask -- just do it.**

It's a continual source of astonishment to us how few customers we work with originally plan to turn on WebSphere Application Server's Java EE security. In our estimate, only around 50% of the customers we see initially plan to use this feature. We have even worked with several major financial institutions (banks, brokerages, and so on) that did not plan on turning security on; luckily this situation was usually

addressed in review prior to deployment.

Not leveraging Java EE security is a dangerous game. Assuming your application requires security (almost all do), you are betting that your developers can better build a security infrastructure than the one you bought from the Java EE vendor. That's not a good bet. Securing a distributed application is extraordinarily difficult. For example, you need to control access to EJBs using a network-safe encrypted token. In our experience, most home-grown security infrastructures are not secure -- with significant weaknesses that leave production systems terribly vulnerable. (Refer to chapter 18 of [Barcia] for more.)

Reasons cited for not using Java EE security include: fear of performance degradation, belief that other security products like IBM Tivoli® Access Manager and Netegrity SiteMinder handle this already, or ignorance of the features and capabilities of WebSphere Application Server security. Do not fall into these traps. In particular, while products like Tivoli Access Manager provide excellent security features, they alone cannot secure an entire Java EE application. They must work hand in hand with the Java EE application server to secure all aspects of the system.

Another common reason given for not using Java EE security is that the role-based model does not provide sufficiently granular access control to meet complex business rules. Though this is often true, this is no reason to avoid Java EE security. Instead, leverage the Java EE authentication model and Java EE roles in conjunction with your specific extended rules. If a complex business rule is needed to make a security decision, write the code to do it, basing the decision upon the readily available and trustable Java EE authentication information (the user's ID and roles). (For more on authorization, see [Ilechko].)

## 6. Build what you know.

**Iterative development enbales you to gradually master all the moving pieces of Java EE. Build small, vertical slices through your application rather than doing everything at once.**

Let's face it, Java EE is big. If a development team is just starting with Java EE, it is far too difficult to try learning it all at once. There are simply too many concepts and APIs to master. The key to success in this environment is to take Java EE on in small, controlled steps.

This approach is best implemented through building small, vertical slices through your application. Once a team has built its confidence by building a simple domain model and back-end persistence mechanism (perhaps using JDBC), and has thoroughly tested that model, they can then move on to mastering front-end development with servlets and JSPs that use that domain model. If a development team finds a need for EJBs, they could likewise start with simple session facades atop container-managed persistence EJBs or JDBC-based DAOs (Data Access Objects) before moving on to more sophisticated constructs like message-driven beans and JMS.

This approach is nothing new, but relatively few teams actually build their skills in this way. Instead, most teams cave in to schedule pressures by trying to build everything at once -- they attack the View layer, the Model layer, and the Controller layer in MVC, simultaneously. Instead, consider adopting some of the new Agile development methods, such as Extreme Programming (XP), that foster this kind of incremental learning and development. There is a procedure often used in XP called ModelFirst [Wiki] that involves building the domain model first as a mechanism for organizing and implementing your user stories. Basically, you build the domain model as part of the first set of user stories you implement, and then build a UI on top of it as a result of implementing later user stories. This fits very well with letting a team learn technologies one at a time, as opposed to sending them to a dozen simultaneous classes (or letting them read a dozen books), which can be overwhelming.

Also, iterative development of each application layer fosters the application of appropriate patterns and best practices. If you begin with the lower layers of your application and apply patterns like Data Access Objects and session facades, you should not end up with domain logic in your JSPs and other View objects.

Finally, when you develop in thin vertical slices, it makes it easier to start early in performance testing your application. Delaying performance testing until the end of an application development cycle is a sure recipe for disaster, as [Joines] relates.

## 7. Always use session facades whenever you use EJB components.

**Use local EJBs when architecturally appropriate.**

Using a session facade is one of the best-established best practices for the use of EJBs. In fact, the general practice is widely advocated for any distributed technology, including CORBA, EJB, and DCOM. Basically, the lower the distribution "cross-section" of your application, the less time will be wasted in overhead caused by multiple, repeated network hops for small pieces of data. The way to accomplish this is to create very large-grained "facade" objects that wrap logical subsystems and that can accomplish useful business functions in a single method call. Not only will this reduce network overhead, but within EJBs, it also critically reduces the number of database calls by creating a single transaction context for the entire business function. (This is described in detail in [Brown]. [Alur] has the canonical representation of this pattern, but it is also described in [Fowler] (which generalizes it beyond just EJBs) and in [Marinescu]. See Resources.) The careful reader will realize that this is actually one of the core principles of Service Oriented Architecture (SOA).

EJB local interfaces, introduced as part of the EJB 2.0 specification, provide performance optimization for co-located EJBs. Local interfaces must be explicitly called by your application, requiring code changes and preventing the ability to later distribute the EJB without application changes. If you are certain the EJB call will always be local, take advantage of the optimization of local EJBs. However, the

implementation of the session facade itself, typically a stateless session bean, should be designed for remote interfaces. This way, the EJB itself can be used remotely by other clients without major breakage to existing business logic. Since EJBs can have both local and remote interfaces at the same time, this is quite feasible.

For performance optimization, a local interface can be added to the session facade. This takes advantage of the fact that most of the time, in Web applications at least, your EJB client and the EJB will be co-located within the same JVM. Alternatively, Java EE application server configuration optimizations, such as WebSphere "No Local Copies," can be used if the session facade is invoked locally but using the remote interface. However, you must be aware that these alternatives change the semantics of the interaction from pass-by-value to pass-by-reference. This can lead to subtle errors in your code. It is best to use local EJBs, since the behavior is controllable on a bean by bean basis, rather than affecting the entire application server.

If you use a remote interface (as opposed to a local interface) for your session facade, then you may also be able to expose that same session facade as a Web service in a Java EE 1.4 compliant way. (This is because JSR 109, the Web services deployment section of Java EE 1.4, requires you to use the remote interface of a stateless session bean as the interface between an EJB Web service and the EJB implementation.) Doing so is often desirable, since it can increase the number of client types for your business logic.

## 8. Use stateless session beans instead of stateful session beans.

**This makes your system more amenable to failover. Use the HttpSession to store user-specific state.**

Stateful session beans are, in our opinion, an idea whose time has come ... and gone. If you think about it, a stateful session bean is exactly the same, architecturally, as a CORBA object -- a single object instance, tied to a single server, which is dependent upon that server for its life. If the server goes down, the object values are lost, and any clients of that bean are thus out of luck.

Java EE application servers providing for stateful session bean failover can workaround some issues, but stateful solutions are not as scalable as stateless ones. For example, in WebSphere Application Server, requests for stateless session beans are load-balanced across all of the members of a cluster where a stateless session bean has been deployed. In contrast, Java EE application servers cannot load-balance requests to stateful beans. This means load may be spread disproportionately across the servers in your cluster. In addition, the use of stateful session beans pushes state to your application server, which is undesirable. Stateful session beans increase system complexity and complicate failure scenarios. One of the key principles of robust distributed systems is the use of stateless behavior whenever possible.

Therefore, we recommend that a stateless session bean approach be chosen for most applications. Any user-specific state necessary for processing should either be passed in as an argument to the EJB methods (and stored outside the EJB through a mechanism like the HttpSession), or be retrieved as part of the EJB transaction from a persistent back-end store (for instance, through the use of Entity beans). Where appropriate, this information can be cached in memory, but beware of the potential challenges that surround keeping the cache consistent in a distributed environment. Caching works best for read-only data.

In general, you should make sure that you plan for scalability from day one. Examine all the assumptions in your design and see if they still hold if your application will run on more than one server. This rule applies not only in application code in the cases outlined above, but also to situations like MBeans and other administrative interfaces.

Avoiding statefulness is not merely an IBM/WebSphere recommendation based on supposed limitations of the IBM tool suite; it is a basic Java EE design principle. See [Jewell] for Tyler Jewell's acerbic opinions on stateful beans, which echo the statements made above.

## 9. Use container-managed transactions.

**Learn how two-phase commit transactions work in Java EE and rely on them rather than developing your own transaction management. The container will almost always be better at transaction optimization.**

Using container-managed transactions (CMTs) provides two key advantages that are nearly impossible to obtain without container support: composable units of work, and robust transactional behavior.

If your application code explicitly begins and ends transactions (perhaps using javax.jts.UserTransaction, or even native resource transactions), future requirements to compose modules, perhaps as part of a refactoring, often requires changing the transaction code. For example, if module A begins a database transaction, updates the database and then commits the transaction, and module B does the same, consider what happens when you try to use both from module C. Now, module C, which is performing what is a single logical action, is actually causing two independent transactions to occur. If module B were to fail during an operation, module A's work is still committed. This is not the desired behavior. If, instead, module A and module B both used CMTs, module C can also start a CMT (typically implicitly via the deployment descriptor) and the work in modules A and B will be implicitly part of the same unit of work without any need for complex rework.

If your application needs to access multiple resources as part of the same operation, you need two-phase commit transactions. For example, if a message is removed from a JMS queue and then a record is updated in a database based on that message, it is important that either both operations occur -- or that neither occurs. If the message was removed from the queue and then the system failed without updating the database, this system is inconsistent. Serious customer and business

implications result from inconsistent states.

We occasionally see client applications trying to implement their own solutions. Perhaps the application code will try to "undo" the queue operation if the database update fails. We don't recommend this. The implementation is much more complex than you initially think and there are many corner cases (imagine what happens if the application crashes in the middle of this). Instead, use two-phase commit transactions. If you use CMT and access to two-phase commit capable resources (like JMS and most databases) in a single CMT, WebSphere Application Server will take care of the dirty work. It will make sure that the transaction is entirely done or entirely not done, including failure cases such as a system crash, database crash, or whatever. The implementation maintains transactional state in transaction logs. We can't emphasize enough the need to rely on CMT transactions if the application accesses multiple resources. If the resources you are accessing cannot provide for two-phase commit, then of course you have no choice but to use a more complex approach -- but you should do everything as possible within your power to avoid this situation.

## 10. Prefer JSPs as your first choice of presentation technology.

**Use XML/XSLT only in cases where you have multiple presentation output types that must be supported by a single controller and back-end.**

There is a common argument that we often hear for why you should choose XML and XSLT as your presentation technology over JSP, and this is that JSP "allows you to mix model and view" too much, and that XML/XSLT is somehow free from this problem. Unfortunately, this is not quite true -- or at least not as black and white as it may seem. XSL and XPath are, in reality, programming languages. In fact, XSL is Turing-complete, even though it may not match most people's definition of a programming language in that it is rules-based and does not have all of the control facilities that programmers may be used to.

The issue is that given this flexibility, developers will take advantage of it. While everyone agrees that JSP makes it easy for developers to do "model-like" behaviors in the view, the fact is that it's possible to do some of the same kinds of things in XSL. While it's very difficult (if not impossible) to do things like calling databases from XSL, we've seen some incredibly complex XSLT stylesheets that perform difficult transformations that still amount to model code.

However, the most basic reason why you should choose JSP as your first option for presentation technology is simply because it's the best supported and best-understood Java EE view technology available. Given the introduction of custom tag libraries, the JSTL, and the JSP 2.0 features, it's become increasingly easy to build JSPs that do not require any Java code, and that cleanly separate model and view. There is significant support (including debugging support) for JSP built into development environments, like IBM Rational Application Developer, and many developers find developing with JSP easier than developing with XSL -- mainly due to how JSP is procedurally based, as opposed to rules-based. While Rational Application Developer supports XSL development, the graphical layout tools and

other features supporting JSP (especially when in the context of frameworks like JSF) make it much easier for developers to work in a WYSIWYG way -- something that isn't easily done with XSL.

This is not to say that you should *never* use XSL, however. There are certain cases where the ability of XSL to take a single representation of a fixed set of data and render it in one of several different ways based on different stylesheets (see [Fowler]) is the best solution for rendering your views. However, this kind of requirement is most often the exception rather than the rule. If you are only ever producing one HTML rendering for each page, then in most cases, XSL is overkill, and it will cause more problems for your developers than it will solve.

## 11. When using HttpSessions, store only as much state as you need for the current business transaction and no more.

**Enable session persistence.**

HttpSessions are great for storing information about application state. The API is easy to use and understand. Unfortunately, developers often lose sight of the intent of the HttpSession -- to maintain temporary user state. It's not an arbitrary data cache. We've seen far too many systems that put enormous amounts of data -- megabytes -- in each user's session. Well, if there are 1000 logged-in users, each with a 1 MB HTTP session, that's one gigabyte or more of memory in use just for sessions. Keep those HTTP sessions small. If you don't, your application's performance will suffer. A good rule of thumb is something under 2K-4K. This isn't a hard rule. 8K is still okay, but obviously slower than 2K. Just keep your eye on it and prevent the HttpSession from becoming a dumping ground for data that "might" be used.

One common problem is in using HttpSessions to cache information that can be easily recreated, if necessary. Since sessions are persisted, this is a very expensive decision forcing unnecessary serialization and writing of the data. Instead, use an in memory hash table to cache the data and just keep a key to the data in the session. This enables the data to be recreated should the user fail over to another application server. (See [Brown2] for more.)

Speaking of session persistence, don't forget to enable it. If you don't enable session persistence, should a server be stopped for any reason (a server failure or ordinary maintenance), any user that is currently on that application server will lose their session. That makes for a very unpleasant experience. They have to log in again and redo whatever they were working on. If instead, session persistence is enabled, WebSphere will automatically move the user (and their session) to another application server, transparently. They won't even know it happened. This works so well, that we've actually seen production systems that crash regularly (due to nasty bugs in native code -- not IBM code!) still provide adequate service.

## 12. Take advantage of application server features that do not

require your code to be modified.

**With features such as WebSphere Application Server caching and the Prepared Statement cache, the performance gains are substantial and the overhead is minimal.**

Best practice #4 above states a clear case as to why you should be very prudent in applying application-server-specific features that modify your code. It makes portability difficult and may make version migration challenging as well. However, there are a suite of application-server specific features, particularly in WebSphere Application Server, that you can and should take full advantage of precisely because they do not modify your code. Your code should be written to the specification, but if you know about these features and how to properly use them you can take advantage of significant performance gains.

For one example of this, in WebSphere Application Server, you should turn on dynamic caching and use servlet caching. The performance gains are substantial and the overhead minimal, while the programming model is unaffected. The merits of caching to improve performance are well understood. Unfortunately, the current Java EE specification does not include a mechanism for servlet/JSP caching. However, WebSphere Application Server provides support for page and fragment caching through its dynamic cache function without requiring any application changes. The cache policy is specified declaratively and configuration is through XML deployment descriptors. Therefore, your application is unaffected, remaining Java EE specification compliant and portable, while benefiting from the performance optimizations provided from WebSphere's servlet and JSP caching.

The performance gains from dynamic caching of servlets and JSPs can be substantial, depending on the application characteristics. Cox and Martin [Cox] showcase performance benefits up to a multiplier of 10 from applying dynamic caching to an existing RDF (Resource Description Format) site summary (RSS) servlet. Please recognize that this experiment involved a simple servlet, and this order of magnitude improvement may not be reflective of a more complex application mix.

For additional performance gains, the WebSphere Application Server servlet/JSP results cache is integrated with the WebSphere plug-in ESI Fragment processor, the IBM HTTP Server Fast Response Cache Accelerator (FRCA) and Edge Server caching capabilities. For heavy read-based workloads, significant additional benefits are gained through leveraging these capabilities. (See performance gains described in [Willenborg] and [Bakalova] in Resources.)

For another example of the principle (which we often observe customers not use simply because they don't know that it exists), take advantage of the WebSphere Prepared Statement Cache when writing JDBC code. By default, whenever you use a JDBC PreparedStatement in WebSphere Application Server, it will compile the statement once and then place it in a cache that will be reused not just later in the same method where the PreparedStatement is created, but across all points in your program where the same SQL code is used in the same or another

PreparedStatement. Saving this re-compilation step can result in a significantly lower number of calls to the JDBC driver and improve the performance of your application. You don't have to do anything special to take advantage of this; just write your JDBC code to use PreparedStatements. By writing your code to use a PreparedStatement instead of a regular JDBC Statement class (which uses purely dynamic SQL) you can take advantage of this performance enhancement while not losing any portability.

## 13. Play nice within existing environments.

**Deliver a Java EE EAR and configurable installation scripts, not a black box binary installer.**

In most realistic scenarios, large WebSphere Application Server users run multiple applications in the same shared cell. This means that if you provide an application to be installed, it must install reasonably into an existing infrastructure. This means two things: First, you must limit the number of assumptions you make about the environment, and ultimately because you can't possibly anticipate every variant, your installation process must be visible. By visible, we mean that providing a binary executable that is the installer is not acceptable. Administrators performing the installation need to understand what the install process is doing to their cell. To facilitate this, you should deliver an EAR file (or a set of EAR files), along with documentation and installation scripts. The scripts should be readable so that the installer can understand what they do and can validate that there is nothing dangerous being done by the scripts. For situations where the scripts are inappropriate, users may need to install your EARs using some other process that they already use - meaning you must document what your installer is doing!

## 14. Embrace the qualities of service provided by the application server environment.

**Design applications to be clusterable using WebSphere Application Server Network Deployment.**

We've already mentioned the importance of leveraging WebSphere Application Server security and transactional support. One more important area that we see ignored far too often is clustering. Applications need to be designed and delivered to run in a clustered environment. Most realistic environments require clustering for scalabity and reliabilty. Applications that don't cluster lead quickly to disaster.

Closely related to clustering is supporting WebSphere Application Server Network Deployment. If you are building an application that you will sell to others, make sure your application runs on WebSphere Application Server Network Deployment and not just the single server versions.

## 15. Embrace Java EE, don't fake it.

**Commit to building real Java EE applications that truly leverage Java EE function.**

One of the most disturbing things we've seen more than once is an application that claims to "run in WebSphere" but isn't really a WebSphere application. We've seen several examples where there is a thin piece of code (perhaps a servlet) in WebSphere Application Server and all of the remaining application logic is actually in a separate process; for example, a daemon process written in Java, C, C++ or whatever -- but not using Java EE -- does the real work. That's not a real WebSphere Application Server application. Virtually all of the qualities of service that WebSphere Application Server provides aren't available to such applications. This can be quite a rude awakening for folks that think this is a WebSphere Application Server application.

## 16. Plan for version updates.

**Change is inevitable. Plan for new releases and fix updates so that your customers can stay current.**

WebSphere Application Server continues to evolve, and so it should not surprise you that IBM regularly produces fixes for WebSphere Application Server, and that IBM periodically releases new major versions. You need to plan for this. There are two kinds of development organizations that this impacts: in-house developers and third party application vendors. The basic issues are the same, but each is impacted differently.

First, consider fixes. IBM regularly releases recommended updates that fix known bugs in our products. While it is likely impossible to always be running at the latest levels, it is prudent to not fall too far behind. How "far behind" is it okay to be? There is no right answer to this, but you should plan on supporting fix levels within a few months of their release. Yes, this means upgrades in production a few times a year. In-house developers can feel free to skip certain fix levels and support one fix level at a time to reduce testing costs. Application vendors aren't so lucky. If this is you, then you need to support multiple fix levels at the same time so that your customers can run your software in conjunction with other software. If you support only one fix level, it may quite literally be impossible to find fix levels compatible across multiple products. Really, the best approach for vendors is to go with the model of supporting "upwardly compatible fixes." This is the approach IBM uses with regard to support products of other vendors with which we integrate (such as Oracle®, Solaris™, and so on). Refer to our support policy for more information.

Second, consider major version upgrades. Periodically, IBM releases new major releases of our products with major functional upgrades. We continue to support older major releases, but not forever. This means you must plan for forced moves from one major release to another. This is simply unavoidable and must be considered in your cost model. If you are a vendor, this means you have to upgrade your product to support new versions of WebSphere Application Server from time to time, or your customers will be stranded on unsupported IBM products -- which is something we've seen happen more than once! If you are purchasing a product from

a vendor, we encourage you to ensure through due diligence that your vendor is committed to supporting new versions of IBM products. Being stranded on unsupported software is a very dangerous situation.

## 17. At all points of interest in your code, log your program state using a standard logging framework.

**This includes exception handlers. Use a logging framework like JDK 1.4 logging or Log4J.**

Logging is sometimes the most tedious, undervalued part of programming, but it is the difference between long hours of debugging and going home at a reasonable time. As a general rule of thumb, at every transition point, log it. When you're passing parameters from one method to another method, or between classes, log it. When doing some transformation on an object, log it. When in doubt, log it.

Once you've made the decision to log, choose an appropriate framework. There are lots of good choices out there but we are partial to the JDK 1.4 trace APIs, as they are fully integrated into the WebSphere Application Server trace subsystem and are standards-based.

## 18. Always clean up after yourself.

**If you obtain an object from a pool, always make sure you return it back to the pool.**

One of the most common errors we see with Java EE applications, whether running in development, test, or production, are memory leaks. Nine times out of ten, it's because a developer forgot to close a connection (JDBC most of the time) or return an object back into the pool. Make sure that any objects that need to be explicitly closed or returned to the pool are so done. Don't be one of the culprits responsible for the offending code.

## 19. Follow rigorous procedures for development and testing.

**This includes adopting and following a software development methodology.**

Large scale system development is hard and it should be taken seriously. Yet, too many times we find teams that are lax in their policies, or that half-heartedly follow development methods that may not apply for the type of development that they are doing, or that they don't understand well. Perhaps the worst extreme of this is trying on the "Development method of the month" where a team will swing from RUP to XP to some other agile method within the lifecycle of a single project.

In short, almost any method will work for most teams provided that they are well-understood by the team members, followed rigorously, and adjusted carefully to

deal with the specific natures of the technology and team that is using that method. For teams that have not adopted a method, or have not fully embraced the method that they have chosen, we would refer them to classic works like [Jacobson], [Beck1], or [Cockburn]. Another useful source of information is the recently announced OpenUP plug-in for the Eclipse Process Framework [Eclipse]. And so that we don't repeat too much of what has been said on this topic already, we refer the reader to [Hambrick] and [Beaton2]. (See Resources.)

## Conclusion

In this brief summary we have taken you through the core patterns and best practices that can make Java EE development a manageable endeavor. While we have not shown all of the details necessary to put these patterns into practice, we have hopefully given you enough pointers and direction to help you determine where to go next.

## Acknowledgements

Thanks to all of those who first documented these patterns and best practices (and whom we reference below), and also to John Martinek, Paul Ilechko, Bill Hines, Dave Artus and Roland Barcia for their help in reviewing this article.

# About the authors

Keys Botzum
**Keys Botzum** is a Senior Technical Staff Member with IBM Software Services for WebSphere. Mr. Botzum has over 10 years of experience in large scale distributed system design and additionally specializes in security. Mr. Botzum has worked with a variety of distributed technologies, including Sun RPC, DCE, CORBA, AFS, and DFS. Recently, he has been focusing on J2EE and related technologies. He holds a Masters degree in Computer Science from Stanford University and a B.S. in Applied Mathematics/Computer Science from Carnegie Mellon University. Mr. Botzum has published numerous papers on WebSphere and WebSphere security. Additional articles and presentations by Keys Botzum can be found at http://www.keysbotzum.com, as well as on IBM developerWorks WebSphere. He is also an author (with Bill Hines) of IBM WebSphere: Deployment and Advanced Configuration.

---

Kyle Brown
**Kyle Brown** is a Distinguished Engineer with IBM Software Services for WebSphere. Kyle provides consulting services, education, and mentoring on object-oriented topics

and J2EE technologies to Fortune 500 clients. He is a co-author of Java Programming with IBM WebSphere, the WebSphere AEs 4.0 Workbook for Enterprise Java Beans, 3rd Edition, and The Design Patterns Smalltalk Companion. He is also a frequent conference speaker on the topics of Enterprise Java, OO design, and design patterns.

---

Ruth Willenborg

**Ruth Willenborg** is a Senior Technical Staff member in IBM's WebSphere Technology Institute working on virtualization. Prior to this assignment, Ruth was the manager of the WebSphere Performance team responsible for WebSphere Application Server performance analysis, performance benchmarking and performance tool development. Ruth has over 20 years of experience in software development at IBM. She is co-author of *Performance Analysis for Java Web Sites* (Addison-Wesley, 2002).

---

Albert Wong

**Albert Wong** is an IT Architect with IBM Retail On Demand Emerging Business Opportunities (EBO), the internal venture capitalist organization missioned to increase IBM business and technical solutions within the Retail industry. Due to the dynamic nature of the EBO, his skills span the whole spectrum of I/T business and technical development from technical pre-sales, solution design and implementation, offering development to ecosystem enablement.