IBM WebSphere Developer Technical Journal: The top 10 (more or less) J2EE best practices

Kyle Brown
Keys Botzum
Ruth Willenborg

May 12, 2004

An updated version of this article is available: The top Java EE best practices.

Introduction

Over the last five years, a lot has been written about J2EE best practices. There now are probably 10 or more books, along with dozens of articles that provide insight into how J2EE applications should be written. In fact, there are so many resources, often with contradictory recommendations, navigating the maze has become an obstacle to adopting J2EE itself. To provide some simple guidance for customers entering this maze, we set out to compile the following "top 10" list of what we feel are the most important best practices for J2EE. Unfortunately, 10 was not enough to capture everything that needed to be said, especially when you consider Web services development as a part of J2EE. Thus, in honor of the growth of J2EE, we have decided to make our "top 10" list a "top 12" list instead.

And so without further ado -- the Top 10 (+ 2) Best Practices for J2EE ...

The best practices

1. Always use MVC.
2. Apply automated unit tests and test harnesses at every layer.
3. Develop to the specifications, not the application server.
5. Build what you know.
6. Always use Session Facades whenever you use EJB components.
7. Use stateless session beans instead of stateful session beans.
8. Use container-managed transactions.
9. Prefer JSPs as your first choice of presentation technology.
10. When using HttpSession, store only as much state as you need for the current business transaction and no more.
11. In WebSphere, turn on dynamic caching and use the WebSphere servlet caching mechanism.
12. Prefer CMP Entity beans as a first-pass solution for O/R mapping due to the programmer productivity benefits.

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 J2EE that there is no competition for the #1 slot. Model-View-Controller (MVC) is fundamental to the design of good J2EE 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 Java™Beans or plain old Java objects).

b. Those responsible for presentation of the user interface (the View -- usually implemented with JSP and tag libraries, but sometimes with XML and XSLT).

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 J2EE; in particular, we direct interested readers to either [Fowler] or [Brown] (see Related topics) 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.
2. 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 Related topics) 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 J2EE 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 Entity beans, Session beans, and JSPs outside of the rest of the code base. There are a number of frameworks and tools for J2EE 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 J2EE components. [Hightower] discusses the use of these tools for J2EE in detail.

Despite all of the great information about deeply testing your application, we still see many projects that believe that if they test the GUI (which may be a Web based GUI or a standalone Java application), then they have comprehensively tested the entire application. GUI testing is rarely enough. There are several reasons for this. First, with GUI testing, it is 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. Often, however, they do not have GUls. Secondly, testing at the GUI level is very coarse grained. It tests at the macro level of the system how the system behaves. This means that if problems are found, entire subsystems must be considered, making finding the bugs identified difficult. Third, GUI testing usually cannot be done well until late in the development cycle when the GUI is fully defined. This means that latent bugs will not be found systematically until very late. Fourth, average developers probably do not 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 do not break existing function. Finally, 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.

3. 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 J2EE allows you to do. We find developers dig themselves into a hole by trying something that they think will work "a little better" than what J2EE 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 J2EE 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 J2EE 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.

Other common problems include using persistence mechanisms that are not tied into the J2EE spec (making transaction management difficult), relying on inappropriate J2SE facilities like threading or singletons within your J2EE programs, and "rolling your own" solutions for program-to-program communication instead of staying within supported mechanisms like JCA, JMS, or Web services. Such design choices cause no end of difficulty when moving from one J2EE compliant server to another, or even when moving to new versions of the same server. Using elements outside of J2EE 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 may recommend using vendor-provided solutions where available (such as the Scheduler facility in WebSphere® Application Server Enterprise), or to use third-party tools where these are not available. 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 J2EE specification, or into a vendor’s product, is often a recipe for disaster. Support is critical -- if your vendor does not directly support a particular technology proposed in a JSR but not yet accepted into J2EE, you should probably not pursue it. 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.


**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 is a continual source of astonishment to us how few customers we work with originally plan to turn on WebSphere’s J2EE security. In our estimate around 50% of the customers we see initially plan to use this feature. For instance, we have worked with several major financial institutions (banks, brokerages, and so on) that did not plan on turning security on; luckily this problem was fixed in review prior to deployment.

Not leveraging J2EE security is a dangerous game. Assuming your application requires security (almost all do), you are betting that your developers can build a better security infrastructure than the one you bought from the J2EE 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 J2EE security include: fear of performance degradation, belief that other security products like Netegrity SiteMinder handle this, or ignorance of the features and capabilities of WebSphere Application Server security. Do not fall into these traps. In particular, while products like SiteMinder provide excellent security features, they alone cannot secure an entire J2EE application. They must work hand in hand with the J2EE application server to secure all aspects of the system.

Another common reason given for not using J2EE 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 not a reason to avoid J2EE security. Instead, leverage the J2EE authentication model and J2EE 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 J2EE authentication information (the user’s ID and roles).

5. Build what you know.

**Iterative development allows you to gradually master all the moving pieces of J2EE. Build small, vertical slices through your application rather than doing everything at once.**
Let's face it, J2EE is big. If a development team is just starting with J2EE, it is far too difficult to try to learn it all at once. There are simply too many concepts and APIs to master. The key to success in this environment is to take J2EE 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 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 EJB components or JDBC-based Data Access Objects (DAOs) 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 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 do development 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.

**6. Always use Session Facades whenever you use EJB components.**

Never expose Entity beans directly to any client type. Only use Local EJB interfaces for Entity types.

Using a session facade is one of the best-established best practices for the use of EJB components. 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 Related topics.)

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. Because the Session Facade and the entity EJBs it wraps should be local to each other, we recommend using local interfaces for the entity beans behind the Session Facade. However, the implementation of the Session Facade itself, typically a stateless session bean, should be designed for remote interfaces.

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, J2EE application server configuration optimizations, such as WebSphere "No Local Copies," can be used if the Session Facade is invoked locally. However, you must be aware 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. To take advantage of these options, you should plan for this possibility from the start.

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 J2EE 1.4 compliant way. (This is because JSR 109, the Web services deployment section of J2EE 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.

7. 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.

J2EE 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, J2EE 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. It increases system complexity and complicates failure scenarios. One of the key principles of robust distributed systems is 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 J2EE design principle. See [Jewell] for Tyler Jewell's acerbic opinions on stateful beans, which echo the statements made above.

8. Use container-managed transactions.

Learn how 2-phase commit transactions work in J2EE 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 2-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 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 do not 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 2-phase commit transactions. If you use CMT and access to 2-phase commit capable resources (like JMS and most databases) in a single CMT, WebSphere 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 cannot emphasize enough the need to rely on CMT transactions if the application accesses multiple resources.

9. 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. 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, in fact, it is possible to do some of the same kinds of things in XSL. While it is very difficult, if not impossible, to do things like calling databases from XSL, we have seen some incredibly complex XSLT stylesheets that perform difficult transformations that still amount to model code.

However, the most basic reason why people should choose JSP as your first option for presentation technology is simply because it is the best supported and best-understood J2EE view technology available. Given the introduction of custom tag libraries, the JSTL, and the new JSP 2.0 features, it is becoming 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 WebSphere Studio, and many developers find developing with JSP easier than developing with XSL -- mostly due to how JSP is procedurally based, as opposed to rules-based. While WebSphere Studio 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 is not easily done with XSL.

The final reason to carefully consider using JSP is one of speed. Performance tests done at IBM comparing the relative speed of XSL and JSP show that in most cases a JSP will be several times faster at producing the same HTML output as an equivalent XSL transform, even when compiled XSL is used. While this is often not an issue, in performance-critical situations it can create problems.
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.

10. When using HttpSession, 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 is not an arbitrary data cache. We have seen far too many systems that put enormous amounts of data -- megabytes -- for each user's session. Well, if there are 1000 logged-in users, each with a 1 MB HTTP session, that is 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 using HttpSession to cache information that is easily recreated, should it be 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 allows 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 do not 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 have seen production systems that crash regularly due to nasty bugs in native code (not IBM code!) yet provide adequate service.

11. In WebSphere, turn on dynamic caching and use the WebSphere servlet caching mechanism.

The performance gains are substantial; the overhead minimal. The programming model is unaffected.

The merits of caching to improve performance are well understood. Unfortunately, the current J2EE specification does not include a mechanism for servlet/JSP caching. However, WebSphere
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 J2EE 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 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 Related topics.)

12. Prefer CMP Entity beans as a first-pass solution for O/R mapping due to the programmer productivity benefits.

Optimize performance through the WebSphere framework (readahead, caching options, isolation levels, and so on). If necessary, selectively apply patterns like Fast-Lane reader [Marinescu] to achieve performance goals.

Object/Relational (O/R) mapping is fundamental to building enterprise-scale applications in Java. Nearly every J2EE application needs some type of O/R mapping. J2EE vendors provide an O/R mapping mechanism that is portable across vendors, efficient, and well-supported by standards and tools. This is the CMP (Container-Managed Persistence) portion of the EJB specification.

Early CMP implementations had a (perhaps) well-deserved reputation for underperforming and not supporting many SQL constructs. However, as the EJB 2.0 and 2.1 specifications were developed and adopted by vendors, and tools like IBM WebSphere Studio Application Developer emerged, these concerns became no longer as valid as they once were.

CMP EJB components are now widely used in many high performing applications. WebSphere includes optimizations to enhance the performance of EJB components including life-time in cache and read-ahead capabilities. Both of these optimizations are deployment options, and do not require application modifications or impact portability.

Life-time in cache caches CMP state data and provides time-based invalidation. The performance gains from life-time in cache approach Option A caching performance, while still providing capabilities for your application to scale in a cluster. Read-ahead capabilities are used in conjunction with container managed relationships. This feature minimizes database interactions by optionally retrieving associated data in the same query as the parent data. This provides performance benefits if the associated data is typically going to be accessed with a subsequent
query. [Gunther] provides detailed descriptions and details the performance improvements possible from these features.

In addition, to fully optimize your EJB components, pay close attention when specifying the isolation level. Use the lowest isolation level possible while still maintaining integrity of your data. Lower isolation levels provide optimal performance and reduce the risk of database deadlocks.

This is by far the most contentious best practice of the lot. Volumes have been written in praise of CMP EJBs, and condemning them as well. However, the basic problem here is that database development is hard. You need to have a fundamental knowledge of how queries and database locks work before beginning to use any persistence solution. If you choose to use CMP EJBs, be sure you are well-educated in their use through books such as [Brown] and [Barcia]. There are subtle interactions in locking and contention that are difficult to understand, but which can be mastered given enough time and effort.

**Conclusion**

In this brief summary we have taken you through the core patterns and best practices that can make J2EE 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.
Related topics

- The top Java EE best practices is a complete revision of this article.
- [Beaton] Wayne Beaton, Migrating to IBM WebSphere Application Server, Part 1: Designing Software for change, IBM DeveloperWorks
- [Brown 2] Kyle Brown, Keys Botzum, Improving HttpSession Performance with Smart Serialization, IBM DeveloperWorks
- [Fowler] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002
- [Joines] Stacy Joines, Ken Hygh and Ruth Willenborg, Performance Analysis for Java Websites, Addison-Wesley, 2002
- [Marinescu] Floyd Marinescu, EJB Patterns, John Wiley & Sons, 2002

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)