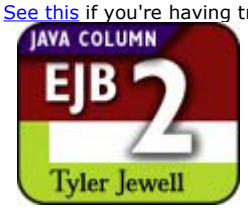
Published on **The O'Reilly Network** (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/onjava/2001/12/19/eejbs.html
See this if you're having trouble printing code examples




# Unlocking the True Power of Entity EJBs

12/19/2001

*Performance this*
*Performance that,*
*I think my EJB*
*Is a tiny bit flat!*

I've heard too much debate in the application developer community about how flawed the entity EJB model is, and that its performance isn't on par with a stateless session EJB (SLSB). I've sat, listened, and watched the debates. The conversations center around performance, the meaning of distributed component architecture, the relevance of JDO, the cumbersome nature of EJBs, and on and on. And through all of this, my concerns for the community grew, as no one was touting the real reasons entity EJBs are monstrously powerful. I told myself that they must see it -- that it should be clear -- but I still pondered, as the community seemed to be blinded to the *true* power of entity EJBs.

The time has come for entity EJBs to have a spokesperson that will champion their vision, potential, and *true* power to the world. Let's get started.

## Write Once, Deploy N-Times

Yes, EJBs have transactional, security, and interoperability support at the container level. It's been repeatedly demonstrated, however, that these capabilities are more relevant at the SLSB façade layer that accesses a data layer. Indeed, most architectures that employ EJBs have the SLSB façade handle these aspects. They configure their data access layer to use entity EJBs as a pass-through for these services (i.e., if a SLSB method is configured to start a new transaction, the entity EJB method invoked by the SLSB method will be configured to accept the already-started transaction).

Although important, these EJB features distract people from appreciating the subtle and powerful nature of entity EJBs: they can be tailored for every conceivable data scenario using a "write once and deploy n-times" model.

Every system has data that is used in different ways. Data can be read-only non-transactional, read-write transactional, read-write transactionally clustered cache, read for update over multiple requests, modified through batch updates, or used in bulk data retrieval. The number of ways that data is accessed and used is limitless. The complexity of these operations increases when relationships are factored in. If you disagree, think about how difficult it would be to implement a transactionally-aware clustered cache that performed eager relationship caching for entity EJBs with relationships three levels deep.

It's foolish to think that an entity EJB deployed in its basic configuration is suited to handle data for each of these scenarios with the best possible results. The out-of-the-box configuration for Entity EJB engines, such as WebLogic, are designed to handle read-write transactional data with the best possible performance. This is done because read-write data is the easiest way to represent life cycle data: data that is created, used for a period of time, and later destroyed. It allows EJB developers to quickly test the purview of functional operations of the data. This functional robustness is often interpreted as the suggested production deployment configuration!

**The real power of entity EJBs (especially robust CMP engines) is the ability for a developer to develop a single entity EJB and then deploy that component multiple times, each deployment tailored for a different data access scenario.**

Yes, there are a couple of studies that demonstrate entity EJBs with CMP have lackluster performance when compared with a

SLSB with JDBC. But there isn't a single study that leverages entity EJBs using the "develop-once, deploy n-times" model. A system using entity EJBs following this model will have greater overall performance, robustness, and reliability than a SLSB with multiple JDBC operations. Here is why:

- A SLSB with JDBC will never be as robust in its ability to handle different data in a single server or a cluster. Container Managed Persistence (CMP) engines already support lazy loading, aggressive loading, optimized writes, field groups for CMP fields, field groups that incorporate relationships, optimistic locking policies, and transparent cache management through activation and passivation. What's the level of effort that you would have to invest to write a SLSB with JDBC to correctly implement all of these scenarios? You could painstakingly accomplish this for a single server environment, but what about trying to coordinate actions for a clustered cache? SLSB with JDBC implementations have upper limits to their capability that entity EJB CMP containers have already surpassed.

- The configuration of entity EJBs allows developers to rapidly change the way a container behaves without any code modification. This provides a level of productivity not available with other technologies.

- Every test case used on entity EJBs for performance doesn't factor in the relative weights of how the data will be accessed in production. For example, Web-based systems hardly access data in a read-write transactional manner, and are primarily read-only systems. Typical test cases would use read-write transactional entity EJBs to gauge the performance for all scenarios. Yuck!

## The New Architecture

There are all kinds of approaches to the design and architecture of a system. Some architects design from the data up and others design from the presentation view down. Other approaches have developers focus on transaction and request granularity. If you want to make the most of entity EJBs, I propose a new methodology: architecture by usage pattern.

This process requires an understanding of the business domain, a data model, and some expectations as to the patterns of usage of the data.

1. Understand your business requirements and model your entity EJBs in such a way that you create in-memory domain objects that encapsulate the data in a format most accessible by the rest of the system.

2. Implement your entity EJBs to this model, ignoring behavior of the data.

3. Test the EJB using a standard read-write transactional locking scheme to make sure that you can create, read, update, and destroy instances.

4. Compile a list of usage patterns that the system will need to perform on the data and their relative weights to one another. For example, I interviewed a couple of companies that have built Web-based systems. This is what we came up with:

| Usage Pattern | Relative Weight |
|---|---|
| Read data for display | 85% |
| Read-write data, inserting records, and deleting records (all requiring transactional support) | 10% |
| Batch update data | 5% |

5. Create a separate deployment for each entity EJB for each usage pattern. For example, in WebLogic Server, changing certain flags implements different usage patterns. (For more information about WebLogic deployment and capabilities, please see BEA's site. WebLogic is merely used here as an example of how this methodology can work.)

| Usage Pattern | WebLogic Configuration |
|---|---|
| Read data for display | `read-only <concurrency-strategy>` with invalidations from a separate read-write deployment and `NotSupported` transaction demarcation. |

| | |
|---|---|
| Read-write data, inserting records, and deleting records (all requiring transactional support) | `Database <concurrency-strategy>` with `<finders-load-bean>` set to `true` (aggressive loading) and `Required` transaction demarcation. |
| Batch update data | `<finders-load-bean>` set to `false`, `Required` transaction demarcation, and only invoking `setXXX(...)` methods of EJBs. This will cause a `findXXX(...)` method to return a `Collection` of primary keys (PKs), but no data will be loaded into cache. A client can iterate through the `Collection`, modifying fields by calling individual `setXXX(...)` methods. When the transaction is committed, only the optimized fields of all of the PKs will be written to the database. |

6. Partition your presentation logic along the lines of the usage patterns that you have defined. Different use cases will access different deployments. So, for the "Read data for display" use case, this might be implemented by a single SLSB that directly accesses the read-only deployment. Since each usage pattern is a different deployment, they will each have a different JNDI name to which they are bound. Your presentation logic will determine which deployment to use by the lookup performed on the naming server.

## The Power and Drawback

Of course, the example displayed above is contrived, but demonstrates the possibilities. In fact, the number of usage patterns that you could derive for your system is quite extensive. For WebLogic Server's EJB container, by taking the permutation of available values for each data-access deployment descriptor tag, I counted nearly 40,000 different ways that an entity EJB could be configured to access a persistent store. Of course, most of these combinations would never be used, but this still quantifies the possibilities.

Of course, there is a penalty for using this methodology: memory consumption. Each deployment of an entity EJB will consume additional memory, based upon your cache settings. If not configured carefully, you could have the same PK in many different caches (one for each usage pattern). If you design a system that has each entity EJB deployed in 10 different formats, you are potentially replicating the same data nine times!

There is, however, a solution: determine the overarching cache size available for a single EJB, and then set the individual deployment limits to be a ratio based upon their expected usage. For example, if you have an entity EJB X that can be allocated 100,000 instances in its cache for all usage patterns, the "Read data for display" deployment would be set to 85,000, since its expected usage pattern was 85% of the activity in the system. The other usage patterns would have their cache sizes set appropriately. This is a simple, yet elegant, way to allocate the appropriate amount of memory to each operation, based upon its activity level in the system.



## Conclusion

EJB 2.0 is just getting warmed up. Vendors are excited about EJB 2.0 not because it is a new version of a popular specification, but because of the possibilities that a solid container implementation can fulfill. Even though EJB 2.1 is currently in development, it will be two to three years before vendors provide container implementations that fully cover every conceivable data-access scenario; the number of container value-add features that can be incorporated into the container is nearly limitless.

Future articles will focus on drilling down into the specifics of the container and provide some insight into how a container can provide optimizations that doing straight SQL would find challenging.

*Tyler Jewell , Director, Technical Evangelism, BEA Systems Tyler oversees BEA's technology evangelism efforts that are focused on driving early adoption of strategic BEA technologies into the ISV and developer community.*

*Read more **EJB 2** columns.*

Return to ONJava.com.