The new EJB 3.0 specification — why it's time to reevaluate Enterprise JavaBeans

by Lou Sacco

Lou Sacco
Senior Software Engineer

Lou Sacco has 12 years of industry experience in designing and developing business applications for a leading technology company in San Diego, CA. He specializes in various forms of persistence technologies, open source frameworks (Hibernate, Spring, Velocity, etc.), SOA, and enterprise application integration and aggregation. Lou is an IBM Certified Enterprise Developer – WebSphere 5.0, and he holds an MS degree in Computer & Information Science from the University of Michigan. You can visit Lou's blog at http://www.loutilities.com, or you can contact him at lou@loutilities.com.

Over the past 12 years, I've seen various forms of persistence technology come and go — embedded SQL in C/C++ code, ODBC, JDBC, EJB, JDO,[1] ORM. Whatever the form, a persistence technology at its essence is simply a way to help programmers do the same mundane things: store, retrieve, or delete some piece of data inside a database.

Before the Enterprise JavaBeans (EJB) specification, most Java developers were content to use the Java Database Connectivity (JDBC) API as an interface to their databases. JDBC isn't very object-oriented, however; it retrieves data as a tabular result set. It is also not very adept at supporting the architectural best practice of "separation of concerns"; JDBC persistence logic is often intermingled with business logic. With JDBC, you essentially have to write your own SQL queries and parse the result set into a Data Transfer Object (DTO) to enable the view component to display the results. Plus, you have to manage your own transactions and security within the realm of your business logic. All of these concerns find their way into your business domain logic, making for serious challenges when your code needs to be updated to support one concern or another.

When EJB first appeared in 1998, it promised transparent transaction support, declarative security, a distributed computing model (remotability), and the beginnings of Object Relational Mapping (ORM).[2] It was a bright, shiny object in the midst of the dot-com boom, and Java developers flocked to this new technology that seemed to address the shortcomings of straight JDBC. Vendors like SAP, IBM, Sun, BEA, and others started to develop Web application servers that had EJB containers to host EJBs.[3] The

[1] For more on JDO, see the article "Spend More Time Modeling Your Java Applications and Less Time Trying to Persist Your Data — Java Data Objects (JDO) Makes It Easy!" in the November/December 2004 issue of *SAP Professional Journal*.

[2] The early trendsetters in the ORM space include TopLink (acquired by Oracle through WebGain in 2002) and Enterprise Objects Framework (EOF, which is now part of WebObjects by Apple). More recent frameworks, like Hibernate, have brought ORM from obscurity to the mainstream.

[3] An EJB is any bean written to the EJB spec — a session, entity, or message-driven bean.

honeymoon didn't last long, however, as the shortcomings of EJB emerged, especially with respect to entity beans. Its deficiencies included poor performance, inadequate query language, and problems with implementing distributed components, to name a few. As a result, EJB's image became tarnished, and many developers and their managers deserted it for other technologies. Some headed for vendor-specific implementations of EJB. Others braved the open source world of meagerly supported ORM tools. Still others went back to basics with JDBC, developing their own in-house data access layers to provide for separation of concerns.

Now, however, there's a new EJB specification (Version 3.0, currently in final draft), and it has made some real headway in addressing the deficiencies of the previous specs. I believe that the things that were good about the original EJB spec have converged with today's successful vendor-specific implementations and open source frameworks under a common spec to become a de facto standard — one that vendors, the Java open source community, and developers can agree upon.

In this article, I highlight the features of the new EJB 3.0 specification and show how the shortcomings of the previous releases have been addressed to make for a more powerful ORM solution. Whether you have used EJBs in the past and abandoned them, are still using them, or are simply among the many architects and developers who want to be aware of emerging standards, this article provides the information and perspective you need to evaluate (or reevaluate) EJB technology for your environment. If you have been using this spec with the support provided by SAP Web Application Server (SAP Web AS), or if you have integrated open source tools like Hibernate into SAP Web AS, it's certainly worth learning how those technologies are converging with the EJB 3.0 spec so you can prepare for its introduction in a future release of SAP Web AS.

Pre-EJB 3.0: what went wrong?

In the introduction, you got an idea of the failings of previous releases of the EJB specification. Let's round out the picture with a closer look at some of those problems so that the improvements stand out when we examine the new spec.

From a developer's point of view, EJB has a steep learning curve. Although vendor tools, such as SAP NetWeaver Developer Studio,[4] do make EJB development a lot easier, to be proficient with EJB, you have to master many programming concepts — such as Core J2EE Patterns design (http://java.sun.com/blueprints/corej2eepatterns), transactions, and the EJB life cycle, to name a few — plus a host of acronyms (BMP/CMP, CMT, and CMR). Otherwise, you might find yourself with a poorly designed, poorly performing EJB application. Another hassle for developers is testing; because EJBs must be deployed in an EJB container, testing requires a lot of overhead. You need to make changes, compile EJBs, deploy, test, and repeat. Even testing tools that eliminate some of the tedium of manual testing, like JUnit and Cactus, don't help much because you have to deploy EJBs to the server to test them.

Using EJB prior to 3.0, especially for entity beans, you have to deal with complex workarounds to perform what would otherwise be simple tasks with JDBC or more recent ORM tools. For instance, when I first started programming with EJB, dealing with dates was cumbersome because of the disparity between how my database represented the dates and how EJB represented them — I had to write wrapper code to overcome the disparity. Sequences (database-generated indexes for primary keys) have been another challenge, as there is nothing in the pre-EJB 3.0 specification to address them; you usually have to come up with a workaround of some form or another to address the idiosyncrasies of sequences.

If all this weren't bad enough, EJB implementations have affected end users negatively. Performance has been slow, and time-to-market has been an issue due to EJB's steep learning curve for developers; for a long time, maintenance was cumbersome in

[4] For more on this tool, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal*.

production because monitoring tools to keep tabs on deployed EJBs were still in the infancy stages. Today, monitoring tools (like Wily Introscope or Tivoli) are much improved. For managers, EJB implementations have meant higher training costs, increased project cycles, and lost revenue on projects delivered late or scrapped as failures.

To address the shortcomings of EJB and tackle the biggest pain point — persistence — the open source community has responded, quite successfully, with frameworks like Hibernate, iBATIS, and ObjectRelationalBridge. The Hibernate and Spring frameworks especially have developed a loyal, ardent following. Open-source-based solutions have resulted in the introduction of hybrids like EJB+Hibernate+Spring.

Vendors have also done a decent job of addressing some of the shortcomings of EJB. One drawback of a vendor implementation, of course, is that it can lead to applications that are tightly coupled to your Web application server provider, which is not very "write once, run anywhere" (the purported core value of Java and J2EE). Nonetheless, a vendor implementation allows EJB adopters to work around the specification's shortcomings. In SAP NetWeaver, SAP has added extensions to EJB 2.x, including several performance and throughput enhancements, use of the SAP enqueue server for pessimistic locks, EJB editors with an object-relational mapping tool in the Eclipse-based SAP NetWeaver Developer Studio, and a seamless integration of EJBs into the SAP persistence infrastructure Open SQL for Java.[5]

Why the need for a standard in persistence?

Given the vendor and open source success stories just mentioned, why is it so important to have a standard for how we persist information to a database? Why not continue to use the solutions we have today? The essential benefit of any standard is that it enables convergence between what vendors are doing and what the open source community is doing so that everyone can work to the same specification. Furthermore, a persistence standard unifies J2EE's strategic position in the marketplace against competing technologies, such as the Microsoft .NET framework.

For the developer, the specific benefits of a persistence standard are these:

- You only need to master one persistence methodology.
- You have a consistent way to perform persistent operations regardless of the vendor you choose.
- You have a larger base of developers you can collaborate with since IT shops usually like to rally around a standard (and there's less squabbling in the J2EE community over what persistence technology should look like).

From the manager's perspective, a persistence standard provides the following benefits:

- You can hire from a broader base of developers who have skills similar to yours, and you don't have to worry about ramping them up on an open source framework or, worse, on your own persistence framework.
- You can expect to run your application on various application servers and not be tied to a specific vendor implementation. You don't have to worry about the possibility of a contract being lost because you don't have experience with a different vendor's EJB implementation.
- You can expect your vendor's EJB 3.0 implementation to be consistent with the specification, and you can expect to get help resolving any issues you may have through your existing vendor support (whereas an open source solution would incur its own support cost).

To improve the EJB specification, members of the global community of Java developers and J2EE product vendors formed the Expert Group for Java Specification Request 220 (JSR 220): Enterprise JavaBeans 3.0. As the Java Community Process (JCP) standardization program converges with

[5] For more on this infrastructure, see the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" in the May/June 2004 issue of *SAP Professional Journal*.

vendor implementations, EJB is becoming a best-of-breed spec.

A (not so) new paradigm: transparent persistence

As long as we're going to have a persistence standard, wouldn't it be nice to have transparent persistence?

Transparent persistence is the idea that your business objects (the objects representative of your database) can be treated as first-class citizens in your application as they pass from a transient state (in-memory) to a persistent state (in-database) to a detached state (in the view layer after the connection with the database is closed). In other words, you can pass your business objects from layer to layer throughout your application, and all that really changes is their state. As an example, to persist a business object that was in a detached state while being operated on in the view layer, you call the persist() method to update its new state to the database. This method not only updates the database, it also transitions the business object's state to the "persistent" state until the session with the database is closed. The need to implement special interfaces, write basic CRUD (create, retrieve, update, delete) SQL statements, define transaction boundaries programmatically, and the like, just goes away. In recent years, transparent persistence has moved from the bleeding edge to become one of the most well-accepted persistence technologies, thanks to the grass-roots success of ORM frameworks like Hibernate.

If you think about it, there's really nothing transparent about either the JDBC or pre-3.0 EJB specifications. JDBC requires you to work with database row sets and SQL directly; so does EJB BMP (bean-managed persistence). Recall that BMP requires you to write your own persistence code (JDBC) within the EJB context, while container-managed persistence (CMP) is a mapping of EJB objects to database tables. EJB CMP is marginally closer to the transparent persistence idiom, but it requires you to pass data around by means of DTOs rather than the actual Java object entities.[6]

CMPs do not support the typically transparent persistence idioms, such as selectively loading information from the database (lazy loading/eager fetching), transitive persistence (cascading operations), or customizable mapping strategies. Moreover, CMP entity beans, which in general are more invasive, force you to implement certain interfaces, use various deployment descriptors, and have one class and four interfaces for each EJB.

How exactly does transparent persistence help? It begins with the notion of using POJOs (plain old Java objects) and their interrelationships for representing schemas from a database. Now you're dealing with objects, and that's something for which Java (an object-oriented language) and you (an object-oriented programmer) are well suited. Instead of JDBC row sets, you can call getter/setter methods of the object. A further benefit of transparent persistence relates to how the data is actually retrieved or updated from the database. You simply map the object's fields to the corresponding tables and columns in a database, typically using an XML configuration, XDoclet, or in the case of EJB 3.0, Java 5.0 annotations (I'll explain these shortly). Note that you're not implementing the EntityBean interface; you're just using POJOs (there's nothing invasive about that).

With the transparent persistence support in EJB 3.0, your life as a programmer gets better in these ways:

- You don't need to write any persistence code. Gone are the days of managing connections and writing high-performance SQL in your JDBC calls (something Java developers may not be adept at).
- You can deal directly with objects through your domain model and leave the details of persisting

[6] For more on how EJB CMP persists data, see the article "Persist Data for Your J2EE Applications with Less Effort Using Entity Beans with Container-Managed Persistence (EJB CMP)" in the July/August 2004 issue of *SAP Professional Journal*.

data to the transparent persistence framework, which makes your life a lot simpler.

- You no longer need to code mundane CRUD operations. You don't have to write this code for every object or table that exists (an especially nice benefit if you have been using stored procedures for these operations).
- You can freely make changes to the persistent Java objects. All changes are automatically tracked and written back to the database. With EJB 3.0, this synchronization can occur at any time during the transaction, not just when the transaction is committed, a much more efficient update process.
- Since you're dealing with POJOs, which can be serialized, you can easily pass graphs of these objects through your service layer directly to a Web application, or you can serialize your POJOs over a Web services call.

In short, transparent persistence results in a very clean domain model in which persistence code is not intermingled with your business logic, so your service layer code can deal with this code directly.

A deep dive into the EJB 3.0 specification

The upcoming sections address those aspects of the EJB 3.0 specification that I think you will find the most useful. We'll start with a look at some valuable aspects of EJB that are virtually unchanged from the previous spec. Then we'll get into what's new and different in EJB interfaces, entity beans, and the querying language, ending with a look at interceptors for crosscutting concerns. You'll get a chance to see transparent persistence in action in the discussion of entity beans.

If you haven't had a chance to get to know the Java 5.0 annotation feature, please see the sidebar on the next page. EJB 3.0 makes heavy use of annotations so having a basic understanding of them will help you follow along.

Admirable aspects of pre-3.0 EJB that carry over

While the persistence aspect of EJB prior to release 3.0 falls woefully short, other aspects of the specification not dealing directly with persistence are quite worthwhile and carry over to 3.0 almost unchanged. Let's review some of them now.

One of the success stories of EJB has been container-managed transactions (CMT), a declarative programming approach to applying transaction boundaries to your code without having to write transaction code. CMT, which is typically accomplished through XDoclet or the use of a deployment descriptor (as in SAP Web AS), is very nice because it enables you to separate another concern (transaction management) from your business logic. You can easily update the deployment descriptor or XDoclet tags if the transaction boundaries need to be changed.

Another success story has been message-driven beans (MDBs), an integral component of service-oriented architectures (SOAs). Essentially, MDBs provide the means for asynchronous processing within a J2EE application, which is important in SOA as it frees up your application to continue working and returning results back to the client, instead of being blocked as in a synchronous call. If something in the asynchronous call fails, the application can make a callback to the client, say through a notification service.[7]

Finally, let's not forget the ubiquitous Façade pattern that SAP, IBM, and Sun pushed as a core J2EE pattern early on. This pattern encapsulates business logic as course-grained calls made to entity beans, Data Access Objects (DAOs), MDBs, and the like; it helps eliminate the chatter over the network of fine-grained calls made to the underlying components. By decoupling the calling client from the underlying components, the Façade pattern also makes it easy

[7] For more on using MDBs for asynchronous message delivery, see the article "Using Advanced Java Message Service (JMS) Features to Increase the Efficiency and Maintainability of Your Distributed Java Applications" in the September/October 2004 issue of *SAP Professional Journal*.

Annotation basics

Annotations allow you to add metadata to your Java classes. They are typically used to automate the generation of source code or configuration files (e.g., deployment descriptors) from within a bean class, thus eliminating the drudgery of managing a deployment descriptor somewhere else in your application. If you are familiar with XDoclet, the open source tool for generating XML descriptors and interfaces, then you can think of annotations as the @ tags you use in XDoclet, except that a compiler is required to check annotations in Java 5.0.

Annotations start with a definition of an annotation type followed by a declaration of the annotation in your source code. This declaration is preceded by the @ symbol. Annotation types must follow certain rules, such as no parameters, primitive return types, and no throw clauses. Let's look at an example from the EJB 3.0 specification, the TransactionAttribute annotation:

```
public enum TransactionAttributeType {
  MANDATORY,
  REQUIRED,
  REQUIRES_NEW,
  SUPPORTS,
  NOT_SUPPORTED,
  NEVER
}
@Target({METHOD, TYPE}) @Retention(RUNTIME)
public @interface TransactionAttribute {
  TransactionAttributeType value() default REQUIRED;
}
```

The first thing to notice is that an enumeration (also new in Java 5.0) defines the possible values. Second, note the meta-annotations @Target and @Retention. @Target says what type of element this annotation can be applied to, in this case METHOD or TYPE. @Retention says how long annotations are retained with the annotated type; with RUNTIME they are available in the Java Virtual Machine (JVM) after the class they annotate has been loaded.

The @interface annotation defines the annotation type TransactionAttribute, which has a value attribute that defaults to REQUIRED if it is not defined. To utilize this new way of defining transaction attributes in EJB 3.0, you declare the annotation by preceding any method definition you have in your EJB as follows:

```
@TransactionAttribute(value="MANDATORY")
public void setName(String) { ... }
```

In this example, you have the transaction attribute set to MANDATORY for this method call. This basic explanation of annotations should suffice for understanding how they are used in the new EJB spec. To learn more about annotations and code generation, see the "Resources" section at the end of this article.


```
// This interface is defined with @Remote allowing remote invocation
@Remote interface Calculator {
     float add (int a, int b);
     float subtract (int a, int b);
}
/*
* The bean class implements the Calculator business interface:
*/
@Stateless public class CalculatorBean implements Calculator {
     public float add (int a, int b) {
                    return a + b;
     }
     public float subtract (int a, int b) {
                    return a - b;
     }
}
```

Example 1 Defining a stateless session bean

to update those components without affecting the client. Transaction boundaries can be easily established at the level where the Façade is introduced because this level is usually a good point in the application to start a transaction or to join one already in progress.

In addition to the useful aspects of EJB mentioned here, you've probably found other features that are particularly useful to you — for example, EJB's declarative approach to security.

Where did all those interfaces go?

The first thing that you'll notice with the new specification is that there are no more home, local, and remote interfaces to implement. It used to be that you could potentially have as many as four interfaces and at least one class implementing a single EJB; now the number you need is between zero and two interfaces, depending on the type of EJB (session or entity). The reduction in interfaces, along with several other changes, makes EJB 3.0 easier to work with, which is a primary goal for this spec.

Recall the old 2.1 way of working with the home-local, home-remote, local, and remote interfaces: First, you had to acquire the EJB through a Java Naming and Directory Interface (JNDI) lookup to the EJB's home interface. Then you could get the local or remote interface to call any business methods on the bean class. The home interface never made much sense for session beans or MDBs, and it only had limited implications for entity beans, so the committee has eliminated it. Also, you no longer have to implement SessionBean in your bean class. Instead, for session beans and MDBs, you implement a business interface that you — not the specification — define. Let's look at a few examples that come from the EJB 3.0 spec and which I have adapted for demonstration purposes.

In **Example 1**, you see how to define a stateless session bean (SLSB) with a business interface for the methods that you want in the concrete implementation of the session bean. Note the use of @Remote,

```
@Stateless public class FooBean {
     private float total;
     ...
     @PostConstruct public void init() {...}
     @PreDestroy public void destroy() {...}
}
```

Example 2 Annotating your methods to introduce EJB callback methods

which enables remote access to the EJB. The notion of local and remote invocation is still pertinent; declarative annotations have replaced the interfaces. You define the bean class with the @Stateless annotation to provide the implementation based on the business interface. If you want to make the session bean stateful, simply use the @Stateful annotation.

When no @Remote or @Local annotation is declared, then the default will be @Local, which provides for only "local" invocation at runtime.

If you are familiar with the pre-3.0 EJB specification, you're probably wondering what happened to all the EJB callback methods, such as ejbCreate(), ejbRemove(), and so forth. They're still there. Now, however, you only implement them if you need them, which makes sense: you rarely need to implement them, so why have the empty stubs in your code? The spec also provides only the callbacks that make sense for a particular kind of EJB bean. To that end, in SLSBs, you only have the following two callback methods (you can use the existing callback methods, or you can use the annotations prefixed to an existing method of your own):

- **ejbCreate()** (or @PostConstruct) — occurs after the bean has been engaged but before the first method is called.
- **ejbRemove()** (or @PreDestroy) — occurs before the bean is destroyed.

Likewise, for stateful session beans that require serialization to the persistent store in order to retain their state during activation and passivation, there are these two callback methods:

- **ejbActivate()** (or @PostActivate)
- **ejbPassivate()** (or @PrePassivate)

An alternative to these four callback methods is to use an annotation, as shown in **Example 2**, where the code in the destroy() method is the same as if you had implemented an ejbRemove() method in this bean class.

The @Remove annotation is unique to stateful session beans; @Remove tells the container that it can remove a stateful session bean from the container after the method annotated with @Remove has been called.

What happened to JNDI lookups to the home interface?

Since there are no more home interfaces for acquiring an EJB, exactly how does a client acquire an EJB in the new specification? Recall that the previous spec requires you to use variations of a JNDI lookup depending on whether the bean is a local or remote bean, as demonstrated by the code in **Example 3**. But remember, this code is only the half of it; you also


```
try {
     Context context = new InitialContext();
     FooBeanHome fooHome =
          (FooBeanHome)PortableRemoteObject.narrow(context.lookup
     ("java:comp/env/ejb/FooBeanEjb"), FooBeanHome.class);
          FooBean fooBean = fooHome.create();
          ...
}
catch(RemoteException re) {
   System.err.println("Remote Exception: " + re.getMessage());
}
catch(NamingException ne) {
   System.err.println("EJB not found: " + ne.getMessage());
}
catch(CreateException ce) {
   System.err.println("Error creating EJB: " + ce.getMessage());
}
```

Example 3 A remote JNDI lookup pre-EJB 3.0

```
@Resource SessionContext ctx;
private FooBean foo = (FooBean)ctx.lookup("ejb/FooBeanEJB");
```

Example 4 JNDI lookup performed with the @Resource annotation

must have the EJB defined in your deployment descriptor as follows:

```
<ejb-ref>
  <ejb-ref-name>ejb/FooBeanEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>FooBeanHome</home>
  <remote>FooBean</remote>
</ejb-ref>
```

Well, in EJB 3.0 things just got a whole lot easier with the dependency injection (DI) approach made popular by lightweight containers, like PicoContainer (http://picocontainer.codehaus.org) and Spring (http://www.springframework.org). With DI, a class that has dependencies is not responsible for obtaining instances to the classes it's dependent on. Instead, the lightweight container injects those dependencies exactly where and when they are needed. The PicoContainer and Spring frameworks accomplish DI through constructor or setter dependency injection (see http://www.martinfowler.com/articles/injection.html for more details). As you might have already guessed, EJB 3.0 accomplishes DI through the use of annotations. Let's take a look.

In **Example 4**, you can see how the JNDI call has been streamlined using the @Resource annotation. Notice that we didn't have to do anything special, as

```
@EJB (name="ejb/FooBeanEJB")
private FooEJB fooEjb;
```

Example 5 Dependency injection using the @EJB annotation

```
// This is one way to add a dependency using an explicit name
@Resource(name="CustomerDB")
public void setDataSource(DataSource myDB) {
     this.ds = myDB;
}

// In this example, name is inferred from the property name after "set"
@Resource
public void setCustomerDB(DataSource myDB) {
     this.customerDB = myDB;
}
```

Example 6 Two ways to add dependencies using setter injection

we did in Example 3, to account for remote invocation of the EJB resource. The annotation transparently handles invocation, whether local or remote. There's an even easier way, however. As **Example 5** shows, you can use the @EJB annotation directly on the EJB variable to further streamline the code.

Finally, you may wish to take the approach of using setter injection to add dependencies to your EJB on the fly. This approach is necessary when establishing resources such as database references. **Example 6** provides a couple of examples of using setter injection to add dependencies.

Entity beans resurrected

At the start of this article, I made the case for taking another look at entity beans, despite past problems and contempt for them. I strongly believe that the designers of the EJB specification finally got it right in EJB 3.0 and that the new spec dramatically changes EJB entity beans for the better.

Transparent persistence in action

The EJB 3.0 specification provides a POJO-based implementation for transparent persistence. This approach offers the developer the advantage of being able to work transparently with database data elements as regular Java objects throughout the many layers of a J2EE application. Let's take a look at an example of transparent persistence (**Example 7**).

In this example, you're looking at an implementation of an EJB entity bean, though you'll notice that this code doesn't implement the EntityBean interface. Instead, it uses the annotation @Entity to identify the class as an EJB 3.0 entity bean, and it implements the java.io.Serializable interface to allow the Entity object to be serialized across remote invocations or over a Web service (i.e., no separate DTOs are required!).

Note too that the Customer entity bean is not an abstract class, and it has a public constructor — yes, that's right, it's a pure POJO that can be instantiated.

Most of the rest of the code in Example 7 is similar to old-style EJB implementations, with the exception that it uses annotations and implements relationships (I'll cover how this release specifies implementing relationships a little later). Below are descriptions of the annotations in Example 7 taken from the top down in the code:

- **@Entity** — defines the class as an EJB entity bean.

```
@Entity
@Table(name="EDI_CUSTOMER")
public class Customer implements java.io.Serializable {
       private Long id;
       private String name;
       private Address address;
       private Collection<Order> orders = new HashSet();
       private Set<PhoneNumber> phones = new HashSet();

       // No-arg constructor
       public Customer() {}

       @Id(generate=SEQUENCE, generator="CUST_SEQ")
       @Column(name="CUSTOMER_ID")
       public Long getId() {
                    return id;
       }
       public void setId(Long id) {
                    this.id = id;
       }
       @Column(name="CUST_NAME", nullable=false, length=100)
       public String getName() {
                    return name;
       }
       public void setName(String name) {
                    this.name = name;
       }
       public Address getAddress() {
                    return address;
       }
       public void setAddress(Address address) {
                    this.address = address;
```

Example 7 Implementation of an EJB entity bean using transparent persistence *Continues on next page*

Example 7 *continued*

```
        }
        @OneToMany
        public Collection<Order> getOrders() {
                    return orders;
        }
        public void setOrders(Collection<Order> orders) {
                    this.orders = orders;
        }
        @OneToMany
        public Set<PhoneNumber> getPhones() {
                    return phones;
        }
        public void setPhones(Set<PhoneNumber> phones) {
                    this.phones = phones;
        }
}
```

- **@Table** — is optional only if the class name matches the table name; otherwise, use this annotation to identify the table name to which to map and, if necessary, the schema or catalog.

- **@Id** — is used to specify the primary key fields of the entity bean. The parameters (generate=SEQUENCE, generator="CUST_SEQ") indicate that the "CUST_SEQ" sequence will be used to generate the primary keys in the database.

- **@Column** — identifies the table column to which a persistent field of the entity bean is mapped. As illustrated by the name field, @Column can also take on several other parameters, such as nullable and length. These attributes (nullable and length) identify database constraints that are verified before the database ever gets a chance to complain about any violations — a nice efficiency.

- **@OneToMany** — see the next section for a description.

What happened to container-managed relationships (CMR)?

Look again at the bottom half of Example 7. Do you see the @OneToMany annotation related both to orders and phone numbers? As you probably guessed, this annotation refers to relationships with other objects, namely Order and PhoneNumber. Since the Customer class has a one-to-many relationship with the Order and the PhoneNumber class, the Customer class represents this relationship as having a set of each type. Therefore, the Order and the PhoneNumber classes will each have a customer field of type Customer, which would be represented as the foreign key in the database. In EJB 2.1, the container manages these relationships with what is known as CMR (container-managed relationships). In EJB 3.0, you have more flexibility and can manage them yourself using annotations and the Java 5.0 generic type feature (which allows you to identify what types of objects can exist in the collection).

CMR is actually pretty nice within SAP NetWeaver Developer Studio. There are several


```
@Entity
public class Employee {
         private Department department;
         @ManyToOne
         public Department getDepartment() {
                 return department;
         }
         public void setDepartment(Department department) {
                 this.department = department;
         }
         ...
}

@Entity
public class Department {
         private Collection<Employee> employees = new HashSet();
         @OneToMany(mappedBy="department")
         public Collection<Employee> getEmployees() {
                 return employees;
         }

         public void setEmployees(Collection<Employee> employees) {
                 this.employees = employees;
         }
         ...
}
```

Example 8 One-to-many relationship

editors for dealing with CMP entity beans, including editors for managing relationships and OR mapping; however, the tool uses the EJB 2.x specification, so you also have to do things in the ejbPostCreate() method to initialize CMR relationships, which can be cumbersome to manage and easy to forget.

With annotations, everything is in one central spot, and in my opinion, it's the best spot — the business object representing the database table. You have the same controls as you did with CMR, and you can see clear as day how each collection maps back to another object representative of the foreign key relationship in the database.

Let's examine how EJB 3.0 manages relationships.

Example 8 is a typical example of a one-to-many relationship between employees and departments (departments having many employees and an employee having only one department). In the Employee class, we declare the department as Department type. On the getter method, we introduce the @ManyToOne annotation, which makes the relationship bidirectional, thus allowing navigation to department through employee. In the Department class, note the Employee type collection of employees and the @OneToMany annotation with the attribute mappedBy="department". The mappedBy attribute

```
@Entity
public class Employee {
         private Cubicle assignedCubicle;
         @OneToOne
         public Cubicle getAssignedCubicle() {
                 return assignedCubicle;
         }
         public void setAssignedCubicle(Cubicle cubicle) {
         this.assignedCubicle = cubicle;
         }
         ...
}
@Entity
public class Cubicle {
         private Employee residentEmployee;
         @OneToOne(mappedBy="assignedCubicle")
         public Employee getResidentEmployee() {
                 return residentEmployee;
         }
         public void setResidentEmployee(Employee employee) {
                 this.residentEmployee = employee;
         }
         ...
}
```

Example 9 One-to-one relationship

designates the field in the entity that is the owner of the relationship (i.e., Employee's department field). Also, because the types are generic, the collection type implies that Employee is the object that contains the department field.

The one-to-many relationship is probably the most common one that you'll use. One-to-one and many-to-many relationships follow essentially the same rules. In **Example 9**, you see how to handle the one-to-one relationship for an entity bean. Here, you're mapping an object to an object rather than a collection of objects. **Example 10** shows just the opposite for the many-to-many relationship, where you map a collection of objects (projects) to another collection (employees). In both cases, the same semantics are at work.

So far, we have looked at bidirectional relationships. Recall that in EJB 2.1 you can restrict navigability to only one owning side. Well, you can do the same thing in EJB 3.0 — simply eliminate the annotations on the inverse side (the side without the foreign key) of the relationship. The code in **Example 11** looks very similar to that of Example 8, with the exception that the AnnualReview class does not contain the @ManyToOne annotation to allow navigability back to Employee — a nice feature in this case for data security.

Leveraging entity beans

Before moving on to the more advanced features of entity beans in EJB 3.0, let's look at how SLSBs integrate with entity beans, especially considering that there are no explicit home, remote, or local interfaces.

EJB 3.0 introduces the concept of an


```
@Entity
public class Project {
         private Collection<Employee> employees = new HashSet();;
         @ManyToMany
         public Collection<Employee> getEmployees() {
                 return employees;
         }
         public void setEmployees(Collection<Employee> employees) {
                 this.employees = employees;
         }
         ...
}
@Entity
public class Employee {
         private Collection<Project> projects = new HashSet();;
         @ManyToMany(mappedBy="employees")
         public Collection<Project> getProjects() {
                 return projects;
         }
         public void setProjects(Collection<Project> projects) {
                 this.projects = projects;
         }
         ...
}
```

Example 10 Many-to-many relationship

```
@Entity
public class Employee {
         private Collection<AnnualReview> annualReviews = new HashSet();;
         @OneToMany
         public Collection<AnnualReview> getAnnualReviews() {
                return annualReviews;
         }
         public void setAnnualReviews(Collection<AnnualReview> annualReviews)
         {
                 this.annualReviews = annualReviews;
         }
         ...
}
@Entity
public class AnnualReview {
...
}
```

Example 11 Unidirectional one-to-many relationship

```
@Stateless public class OrderEntry {
        @PersistenceContext EntityManager em;
        public void enterOrder(int custID, Order newOrder) {
                Customer cust = (Customer)em.find("Customer", custID);
                cust.getOrders().add(newOrder);
                newOrder.setCustomer(cust);
                em.persist(newOrder);
        }
}
```

Example 12 Using an EntityManager object to call entity beans

EntityManager (em) object using DI. The @PersistenceContext annotation allows us to inject the EntityManager object into our SLSB to perform CRUD operations using the entity bean POJOs you have created. Let's look at **Example 12**.

Here you see an SLSB using DI to introduce EntityManager. The client program calls enterOrder(), passing custID and a newOrder object. First, we use EntityManager to find and return a Customer object based on the custID, and then we bind the Order object with the Customer object by establishing a relationship between them. Finally, we call the persist() method to persist the objects to the database. This technique may seem a little unorthodox at first. However, you must fulfill both sides of the relationship because these objects are defined as having a bidirectional relationship in the entity beans. So first add the Order object to the Customer's order collection (the many side); then, add the Customer object in the Order (the one side).

As good stewards of J2EE design, we have just abstracted access to the entity bean using the Core J2EE Patterns Session Façade pattern. How will our client access the façade, and how is it that we can pass an entity bean into the enterOrder method here?

Here are a couple examples of how you can provide access to the session bean from the Web tier. The first is through DI as follows:

```
@EJB OrderEntry orderEntry;
```

The other way is through a JNDI lookup as follows:

```
@Resource SessionContext ctx;OrderEntry
orderEntry
= (OrderEntry)ctx.lookup("ejb/OrderEntry");
```

Both of these methods provide the same result and, in the spirit of the new specification, they greatly simplify the process of obtaining the session bean from the EJB container.

Having entity beans as POJOs is a huge advantage. You can now use the <jsp:usebean> tag to hold the entity beans in an HTTPSession object. When you are ready to store them in the database, you can pass these entity beans directly to the session bean method. In other words, these entity beans are serving two purposes (see the sidebar on the next page for a consideration to keep in mind when using entity beans for dual purposes). You no longer need to map a DTO back to the entity bean since the concepts of storing data from objects and transferring data as objects are both merged into a single entity bean business object.

Transitive persistence and lazy loading

By manually defining relationships between two or more objects, you have the ability to process these

No DTOs and optimistic locking

Using entity bean business objects for dual duty as DTOs is a good thing, but it brings an interesting problem to the table; namely, how does the object maintain data integrity in the face of concurrent usage? The scenario is as follows: User A reads from the database and then User B reads from the database. User A makes a change to the record, which persists that record to the database; slightly later, User B also makes a change that persists the record to the database. Since both users were working from the same original data, User A's changes would be lost (i.e., last in wins).

Enter optimistic locking. In this scenario, User A reads a row from the database, and the version column (usually a sequential index) has a value of 1. User B reads the same row before User A commits any changes, and the version column is also 1. User A commits his changes to the database, and the version column is automatically incremented to 2. When User B attempts to save her changes a short time later, the save fails because the version column value for the row she read (1) no longer matches what is in the database. In this case, an exception would be thrown; she could either reread the data and attempt to save again, or she could pass the exception to the view layer.

How is optimistic locking supported in EJB 3.0? It's simple. There's another annotation called @Version that helps solve problems with concurrent usage. All you need to do is define the version column in your entity bean as a numeric value and add the @Version annotation to the accessor method as follows:

```
@Version
@Column(name="EMP_VERSION", nullable=false)
public int getVersion() { return version; }
protected void setVersion(int version) {this.version = version;}
```

Optimistic locking is a great approach for solving concurrency issues, especially if concurrency is the exception rather than the norm in your application. Optimistic locking affords you better performance because it doesn't require you to use expensive row-level locks. Always look to use optimistic locking in favor of pessimistic locking if your application doesn't require a lot of concurrency at the row level.

objects (in the form of a graph) using a cascading mechanism provided by the new specification. This mechanism is commonly known as "transitive persistence" or "persistence by reachability" (i.e., if the root object being persisted can reach a sub-node object, it will persist the sub-node object as well). Transitive persistence enables you to perform the following cascading options:

- **PERSIST** — cascades the ability to persist an entity into the database.
- **MERGE** — cascades the ability to merge a detached entity onto persistence entities.
- **REMOVE** — cascades the ability to delete entities from the database.
- **REFRESH** — cascades the ability to refresh the objects in memory from the database. It overwrites any changes made in memory up to that point.
- **ALL** — allows the ability to do all of the above with one keyword.

```
@Entity
public class Project {
        private Collection<Employee> employees;
        @ManyToMany(cascade=PERSIST)
        public Collection<Employee> getEmployees() {
                return employees;
        }
        public void setEmployees(Collection<Employee> employees) {
                this.employees = employees;
        }
        ...
}
@Entity
public class Employee {
        private Collection<Project> projects;
        @ManyToMany(cascade=PERSIST, mappedBy="employees")
        public Collection<Project> getProjects() {
                return projects;
        }
        public void setProjects(Collection<Project> projects) {
                this.projects = projects;
        }
        ...
}
```

Example 13 Cascade PERSIST in a many-to-many relationship

In **Example 13**, you see how to implement transitive persistence using the PERSIST cascade attribute in the @ManyToMany annotation. Essentially, you are allowing Project objects or Employee objects to be persisted from either side of the relationship when either are the root node — but that's it! Be careful when using this feature: if you were to choose ALL or included REMOVE, you could inadvertently delete employees or projects from the database.

Lazy loading allows you to retrieve information on an as-needed basis. By default, most relationships are lazy loaded, which means if you want only the Employee object, you get just the employee and not all of his or her projects. When you need the projects, you bring them into scope by calling the getProjects() method. If you want projects right away, you can have Project objects fetched eagerly (as opposed to lazily) and cached. Example 13 would lazily load either projects or employees, depending on your entry point, because that is the default behavior for this relationship. Now let's look at modifying the relationships to support eager fetching.

In **Example 14** you see how to modify the Project's many-to-many relationship with Employee such that Project always uses the fetch attribute of the @ManyToMany annotation set to EAGER. This approach may be appropriate for retrieval through Project; I would not recommend doing the same thing in the Employee class for performance reasons (and so I have not included that in the example).

Advanced relationship features: multi-table mapping and inheritance

A nice feature of EJB 3.0 is the ability to do multi-table mappings to one object using the @SecondaryTable annotation. This annotation


```
@Entity
public class Project {
        private Collection<Employee> employees;
        @ManyToMany(cascade=PERSIST, fetch=EAGER)
        public Collection<Employee> getEmployees() {
                return employees;
        }
        public void setEmployees(Collection<Employee> employees) {
                this.employees = employees;
        }
        ...
}
```

Example 14 Eager fetching of employees for a Project

```
@Entity
@Table(name="EMPL")
@SecondaryTable(name="EMP_SALARY", pkJoin=@PrimaryKeyJoinColumn(name="EMP_ID",
referencedColumnName="ID"))
public class Employee implements Serializable {
        private Long id;
        private Long salary;

        public Employee() {}

        @Id(generate=TABLE)
        public Integer getId() { return id; }
        protected void setId(Integer id) { this.id = id; }

        ...

        @Column(name="EMP_SAL", secondaryTable="EMP_SALARY")
        public Long getSalary() { return salary; }
        public void setSalary(Long salary) {
                this.salary = salary;
        }
}
```

Example 15 Multi-table mappings

eliminates the necessity of having to create an object for every single table that exists. In essence, you can combine tables into one object whenever that makes sense, as demonstrated in **Example 15**.

In Example 15, you see the @SecondaryTable annotation being defined above the class; this annotation tells the class to join the primary key ID column in the EMPL table to the EMP_ID column in the EMP_SALARY table. In the getSalary() method, you see the annotation parameter secondaryTable, which says to use the EMP_SALARY column from the

defined table to obtain the salary amount. This annotation allows you to encapsulate the EMP_SALARY table (without having to create a new object) within the Employee object; it effectively provides a single business object represented from two tables. To that end, this concept helps keep object proliferation down to a minimum for the trivial fields that can be easily merged with other objects.

Multi-table mapping and inheritance are not part of the EJB 2.x spec. Additionally, there are severe conceptual issues with adding inheritance to EJB 2.x CMP entity beans. Though SAP's implementation of EJB 2.x does not support multi-table mappings or inheritance, both features are available in SAP Web AS through the SAP JDO implementation. JDO supports several mapping strategies for inheritance, including the single-table-per-class-hierarchy mapping strategy. In EJB 3.0, multi-table mapping and inheritance are now part of the spec. There are three types of inheritance model in release 3.0: Single_Table (one table per class hierarchy), Table_Per_Class (distinct table per class), and Joined (one joined table per subclass). More details can be found in the EJB 3.0 spec.

EJB-QL — a real query language for real people and projects

EJB-QL has come a long way in the EJB 3.0 specification. In this release, you have the option to perform bulk update and delete operations, JOIN operations (inner, outer, left), GROUP BY, HAVING, subqueries, arithmetic functions, and more. Also, this release introduces named parameters for use in static and dynamic queries. Basically, anything that you can dream up in SQL can now be done in EJB-QL — it's a real query language for real people and projects! The spec devotes nearly 40 pages to the new querying features. I'll just touch on some of the basics and my own personal favorites.

Parametric queries

You're probably already familiar with parametric queries from pre-3.0 EJB-QL; they allow you to develop dynamic queries by using parameters to define the inputs. In **Example 16**, you can see a parametric query that takes the employee's name as a parameter. Similar to the way the EntityManager object called CRUD operations in the section "Leveraging entity beans," the em.createQuery() method can do data retrieval. The ":" before the employee's name (empName) signifies that a parameter will be used for this value; the setParameter() method allows you to set the parameter with the name variable. Note that if you want to use positional parameters instead, you use a "?" before an integer (starting with 1).

Finally, you can limit the result set to whatever number you choose (it's 50 in Example 16) and getResultList() will return the results as a List collection.

Named queries

Formerly, static queries were defined through the EJB deployment descriptor. EJB 3.0 continues to allow you to create static queries by using the @NamedQuery annotation. If you are familiar with Hibernate, you may recall that you can put named queries directly in the HBM XML file. Whether that will happen in EJB 3.0 through a deployment descriptor (similar to 2.1) is yet to be seen.

Example 17 shows how to define and use a named query.

Polymorphic queries

All queries are polymorphic by default in EJB 3.0. For instance, if you take a typical inheritance example where Manager and ExemptEmployee are subclasses of Employee, the following query would result in the return of both types of employees with salaries greater than $80,000:

```
select e.salary from Employee e where
e.salary > 80000;
```



```
public List findByName(String name) {
        return em.createQuery("SELECT e FROM
        Employee e WHERE e.name LIKE :empName")
        .setParameter("empName", name).setMaxResults(50).getResultList();
        }
```

Example 16 Parametric query

```
@NamedQuery(
     name="findAllEmployeesWithName",
                  queryString="SELECT e FROM Employee e WHERE e.name LIKE
           :empName"
)

List employees = em.createNamedQuery("findAllCustomersWithName")
.setParameter("custName", "Smith").getResultList();
```

Example 17 Named query

```
Query q = em.createNativeQuery(
       "SELECT o.id, o.quantity, o.item, i.id, i.name, i.description "+
       "FROM Order o, Item i " +
       "WHERE (o.quantity > 25) AND (o.item = i.id)",
       "OrderItemResults");

@SqlResultSetMapping(name="OrderItemResults",
       entities={
                    @EntityResult(entityClass=com.acme.Order.class),
                    @EntityResult(entityClass=com.acme.Item.class)
       }
)
```

Example 18 Creating a native query with SQL

SQL queries

Depending on your target database, you might need to use native SQL queries, oftentimes for performance optimizations. Understandably, these queries would not be portable. However, the new specification allows you to write raw SQL queries if you need some function that your database vendor's implementation of SQL provides; you can map the result set back to your entity beans using the @SqlResultSetMapping annotation.

In **Example 18**, you see how to use the createNativeQuery() method of EntityManager to create an SQL statement. The first parameter of this method is the raw SQL statement, and the second

parameter is a string to define where the result set goes. In the @SqlResultSetMapping annotation, the name parameter utilizes that string and maps it to entities, in this case, the Order and Item classes. Therefore, the Order's ID and quantity, and the Item's name and description, are populated based on the query.

Projection and sub-queries

Projection and sub-queries are features that you won't find in the previous EJB specification.

- **Projection queries** — allow you to project the results of a query directly to an entity, without having to iterate through a collection to produce the objects. The query in the following example returns new OverworkedEmployee objects for those employees with more than three projects:

```
SELECT NEW
com.loutilities.OverworkedEmployee
(e.id, e.firstname, e.lastname)
FROM Employee e JOIN c.projects p WHERE
p.count > 3
```

- **Sub-queries** — allow you to embed a query within another query. The following query yields those employees who have spouses working at the same company:

```
SELECT DISTINCT emp FROM Employee emp
        WHERE EXISTS (
                      SELECT spouseEmp FROM
Employee spouseEmp WHERE spouseEmp =
emp.spouse)
```

Bulk update and delete operations

Probably one of my favorite features in the 3.0 release is the new ability to perform bulk update and delete operations without having to make a special call to JDBC or a stored procedure.

For example, you can set the status of all employees with a released date that is less than the current date to "inactive" (bulk update) like this:

```
UPDATE employee e SET c.status =
'inactive'
WHERE e.date_released < CURRENT_DATE
```

To remove all employees that have a status set to "inactive" (bulk delete), do this:

```
DELETE FROM EMPLOYEE e WHERE e.status =
'inactive'
```

Interceptors to address your crosscutting concerns

The interceptors feature is one of the nicest introduced in the EJB 3.0 specification. A key element of AOP (Aspect-Oriented Programming) interceptors are classes (or "aspects" if you are using something like AspectJ) that allow you to manage crosscutting concerns across several objects in a single programming construct. A crosscutting concern could be something like exception handling, profile logging, security, or auditing. Instead of duplicating the code to handle these concerns in every class where you need it, you can use an interceptor class to provide the necessary mechanisms.

EJB 3.0 introduces the @Interceptors annotation to define the interceptor(s) that a call uses. In **Example 19**, you can see how an interceptor has been introduced into the AccountManagementBean class by using the @Interceptors annotation. The Metrics interceptor provides the ability to gather performance metrics on how long it takes to execute each method in the AccountManagementBean class. Note that it's not necessary to add anything to this class other than the interceptor at the top.

In the Metrics interceptor class, the @AroundInvoke annotation defines the desired functionality around each invocation of this interceptor. In this example, you see that we first capture the current time, allow the original method called from


```
@Stateless
@Interceptors({
com.acme.Metrics.class
})
       public class AccountManagementBean implements AccountManagement {
       public void createAccount(int accountNumber, AccountDetails
                       details) { ... }
       public void deleteAccount(int accountNumber) { ... }
       public void activateAccount(int accountNumber) { ... }
       public void deactivateAccount(int accountNumber) { ... }
       ...
}

public class Metrics {
       @AroundInvoke
       public Object profile(InvocationContext inv) throws Exception {
                     long time = System.currentTimeMillis();
                     try {
                             return inv.proceed();
                     } finally {
                             long endTime = time - System.currentTimeMillis();
                             System.out.println(inv.getMethod() + " took " +
                             endTime + "milliseconds.");
                             }
       }
}
```

Example 19 Using an interceptor to profile the AccountManagementBean class

AccountManagementBean to continue using the proceed() method; then, in the finally code block, the code reports the timing metrics to the console.

Even with an example this simple, you can see the power of interceptors. You can put common cross-cutting code into one area and then, in the classes that need it, introduce the interceptor with the annotation. All the while, there are no other modifications required in the calling classes. Pretty powerful stuff!

Where do we go from here?

At the time of this writing, EJB 3.0 has been released for public review. In August 2005, members of the Executive Committee for SE/EE[8] (including SAP) voted unanimously in favor of the new specification. EJB 3.0 is part of Java EE 5.0, which is scheduled for Q1 2006. Oracle is the co-specification lead, and it looks like the Oracle TopLink product will be used as the reference implementation, much to the dismay of the Hibernate community.[9]

[8] The SE/EE oversees Java technologies for desktops and servers. See the "Resources" section on the next page for a link to the list of committee members.

[9] See the discussion thread "Oracle becomes sponsor and co-specification lead of EJB3" at TheServerSide.com: http://www.theserverside.com/news/thread.tss?thread_id=34877.

JBoss (using Hibernate) and Oracle both have implementations of EJB 3.0 that conform to the current release of the specification. Another newcomer, Versant, also has an EJB 3.0 implementation that is available as a plug-in to the Eclipse IDE. SAP, too, has already started with its EJB 3.0 implementation. Though there isn't yet an official statement on when the SAP EJB 3.0 implementation will be available, there are rumors that preview versions will be made available to early adopters in mid-2006.

The current lack of an SAP implementation should not deter you from experimenting with the new spec, however. I know many of you have explored Hibernate with SAP Web AS. The creators of the most recent release of Hibernate (R3) have made great efforts to match the Hibernate API with the EJB 3.0 API. Certainly, if you want to explore outside the realm of SAP Web AS, you can take a look at some of the early vendor implementations mentioned above.

The nice thing about the spec is that an implementation that works on one platform, say Oracle, should work on SAP Web AS as well. So instead of vendor-specific EJB implementations, we get back to our "write once, run anywhere" Java roots. Even so, I'm looking forward to seeing what SAP NetWeaver will bring to the table in light of the new spec and how applications will leverage EJB 3.0. I hope, after reading this article, you are too. It's an exciting time to be a J2EE developer.

Resources

Aspect-Oriented Refactoring Series — Part 1: Overview and Process
http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1

Hibernate EntityManager for EJB3 (Hibernate's implementation of EntityManager)
http://www.hibernate.org/299.html

How to use Java 5's built-in annotations
http://www-128.ibm.com/developerworks/java/library/j-annotate1/

Java Community Process (JCP) Home — JSR-220: Enterprise JavaBeans 3.0
http://www.jcp.org/en/jsr/detail?id=220

JBoss TrailBlazer (An EJB tutorial of JBoss' implementation of the specification)
http://trailblazer.demo.jboss.com/EJB3Trail/

The Executive Committee info for J2SE/J2EE
http://www.jcp.org/participation/committee/index.jsp#SEEE

Oracle's EJB 3.0 Preview (Oracle's implementation of the specification)
http://www.oracle.com/technology/tech/java/ejb30.html

Versant's Open Source JSR220-ORM
http://www.versant.com/opensource/orm/

