Published on **dev2dev** (http://dev2dev.bea.com/)
http://dev2dev.bea.com/pub/a/2006/01/ejb-3.html
See this if you're having trouble printing code examples

An Introduction to the Enterprise JavaBeans 3.0 (EJB 3) Specification

by Vimala Ranganathan and Anurag Pareek
03/29/2006

Abstract

The Enterprise JavaBeans (EJB) technology is a J2EE technology for the development and deployment of component-based business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multiuser secure.

In spite of the rich features, however, the complexity of the EJB architecture has hindered its wide adoption. Competing technologies are making inroads in the EJB space. For example, O/R mapping technologies such as Toplink and the open-source Hibernate framework have overtaken EJB as the preferred choice for developing persistence solutions. The introduction of the EJB 3.0 specification is a giant step forward and will go a long way toward luring developers back to EJBs. The goal of the specification is twofold:

- Make it easier for developers to develop EJBs.
- Standardize the persistence framework.

EJB 3.0 brings us closer to the dream of treating enterprise beans like regular JavaBeans. It decreases the number of programming artifacts for developers to provide, eliminates or minimizes callback methods required to be implemented, and reduces the complexity of the entity bean programming model and O/R mapping model. With EJB 3.0, J2EE now seems accessible to a much wider audience.

In this article, we first briefly discuss the limitations of EJB 2.1. Next, we describe how EJB 3.0 addresses these difficulties by describing the proposed significant changes one by one, including the impact on types of enterprise beans, the O/R mapping model, the entity-relationship model, and EJB QL (EJB Query Language). We conclude with code examples using EJB 3.0-based enterprise beans.

Limitations of EJB 2.1

Developing EJBs with EJB 2.1 hasn't been the easiest thing to do. The reasons are easy to find:

- To create a single EJB you need to create a multitude of XML deployment descriptors.
- A set of three source files must be created.
- Multiple callback methods must be implemented that usually are never used.
- You have to throw and catch several types of unnecessary exceptions.
- Yet another complaint is that the EJBs are completely untestable outside the context of the container since components like container-managed entity beans are abstract classes.
- Finally, EJB-QL in its current form is limited in functionality and difficult to use. These limitations force developers to use straight JDBC and SQL, or to use other persistence frameworks such as Toplink and Hibernate.

The sheer verbosity of the API has been one big annoyance, and EJB 3.0 makes a significant attempt to address most issues. This article covers the important aspects of this specification.

The End of the Road for Deployment Descriptors

The configuration of XML deployment descriptors was a major bottleneck in the path to simplifying development of EJBs. Therefore one of the primary goals of the EJB 3.0 specification was to shield the developer from having to work with XML files. This is accomplished by the use of metadata annotations that have been added to JDK 5.0 as part of the JSR 175 JCP specification. Annotations are a kind of attribute-oriented programming and are similar to XDoclet. However, unlike XDoclet, which requires pre-compilation, annotations are compiled into the classes by the Java compiler at compile-time. From the developer's point of view, annotations are modifiers like public/private and can be used in classes, fields, or methods:

```
import javax.ejb.*;
@Stateless
public class MyAccountBean implements MyAccount
{
  @Tx(TxType.REQUIRED)
  @MethodPermission({"customer"})
  public void deposit(double money) {...}
}
```

The annotations generally are self-explanatory. The `@Stateless` annotation indicates that the bean is stateless. The `@Tx` attribute specifies the transactional demarcation for the method, and the `@MethodPermission` attribute specifies the users who are allowed to access the method. So this means that there's no longer a need to write XML deployment descriptors to describe these properties. However, this does not eliminate the use of XML; it just makes it optional. The specification allows the use of XML deployment descriptors to override these annotations.

POJO Programming Model

The critical point to note is that the above stateless session bean example is complete in itself. Disregarding the annotations, this file is a JavaBean, also known as a Plain Old Java Object (POJO). Interfaces are optional for entity beans and required for session beans and message-driven beans. However, that does not mean that you have to define an interface for your session bean or message-driven bean. If you do not implement an interface, a bean interface will be generated for you. The type of generated interface, either local or remote, is dependent on the annotation you used in the bean class. All the public methods of the bean class will be included as part of the automatically generated business interface:

```
public interface ShoppingCart
{
  public void purchase(Product product, int quantity);
  public void emptyCart();
}
```

It is recommended that you generate the interface explicitly if you want to pick and choose the methods of the interface that should be exposed to the client, or want to give the interface a name different from the automatically generated name.

This interface class is a Plain Old Java Interface (POJI). Both the interface and the bean class do not have to throw unnecessary exceptions such as `RemoteException`.

Callback Methods

In the EJB 2.1 specification, the developer had to implement a variety of callback methods in the bean class, such as `ejbActivate()`, `ejbPassivate()`, `ejbLoad()`, and `ejbStore()`, most of which were never used. With 3.0, there is no compulsion to implement any of these methods. In EJB 3.0, bean developers do not have to implement unnecessary callback methods and instead can designate any arbitrary method as a callback method to receive notifications for life cycle events. Any callback method has to be annotated with one of the pre-defined life cycle event callback annotations. Examples of life cycle event callback method annotations include `PostConstruct`, `PreDestroy`, `PostActivate`, or `PrePassivate`. Some of the event callback methods are common to all types of enterprise beans, while some are specific to bean types such as `PostPersist` for entity beans.

Callback methods can be defined either in the bean class itself or in a bean listener class. A bean listener class is denoted using the `CallbackListener` annotation on the bean class with which it is associated. The annotations used for callback methods are the same in both cases; only the method signatures are different. A callback method defined in a listener class must take an object as a parameter, which is not needed when the callback is in the bean itself. This object parameter can be used to pass the bean instance to the method in the listener class. Here's an example of putting a callback method in an entity bean:

```
@Entity
public class AccountBean{
  @PostPersist insertAccountDetails(AccountDetails accountDetails)
  public void createAccount(){}
}
```

Let's look at an example of creating a listener class and adding it to a bean class. The following code defines the callback listener AccountListener:

```
/* Adds callback listener to bean class */
@CallbackListener AccountListener
  public class AccountBean{
  public void createAccount(){}
}
```

The following code will add the callback listener AccountListener to the Account Bean:

```
/* Callback method defined inside a Listener class*/
public class AccountListener{
 @PostPersist insertAccountDetails(
                                 AccountDetails accountDetails){}
}
```

Since the `@PostPersist` is used to register a method to be called on an object that has just been inserted into the database, in this case, the method `insertAccountDetails()` will be invoked every time as soon as an account has been inserted using the `createAccount()` method in the `AccountBean`.

Configuration by Exception

The "Configuration by Exception" approach is the guiding methodology used in all aspects of EJB 3.0 to simplify the development effort. The intent is to simplify things for developers by forcing them to code things only where defaults are not adequate.

For instance, in many cases, defaults can be used instead of explicit metadata annotation elements. In these cases, a developer doesn't have to specify a metadata annotation to obtain the same result as if the annotation was fully specified. For example, by default, an entity bean (annotated by `@Entity`) has a default entity type of CMP, indicating that it has container-managed persistence. These defaults can make annotating enterprise beans very simple. The defaults always represent the most common specifications. For example, container-managed transaction demarcation (where the container, as opposed to the bean, manages the commitment or rollback of a unit of work to a database) is assumed for an enterprise bean if no annotation is specified. Similarly a default business interface is generated for session and message-driven beans which exposes all the public methods of the bean in the interface, since that is the most common use case.

Object-relational Mapping

The O/R mapping or persistence model has significantly changed from the abstract-persistence-schema-based approach, to one inspired by the various POJO-related approaches en vogue today. The O/R mapping is specified using annotations. The O/R mapping metadata expresses requirements and expectations of the application to map entities and relationship of the application domain to the database.

In EJB 2.1, developers used their own mechanisms to do certain database-specific operations like primary key generation. With EJB 3.0, support for several database-specific operations has been provided. The O/R mapping model has intrinsic support for native SQL. In this article, we do not provide details on the persistence framework, although we do provide an outline while discussing the changes in entity beans. For details check the EJB 3.0 API specification and download the EJB 3.0 persistence documentation.

Encapsulation of JNDI Lookups Using Annotations

EJB 3.0 addresses the encapsulation of environmental dependencies and JNDI access through the use of annotations, dependency injection mechanisms, and simple lookup mechanisms.

The enterprise bean's context comprises its container context and its resource and environment context. The bean may gain access to its resource references and other environment entries in its context in two ways:

1. Having the container supply it with those references such as using injections; for instance, `@EJB public AddressHome addressHome;` automatically looks up the EJB with the JNDI name "AddressHome."

2. Use the method `Object lookup(String name)` that is added to the `javax.ejb.EJBContext` interface. This method can be used to look up resources and other environment entries bound in the bean's JNDI environment naming context.

Dependency Injections

A bean declares a dependency upon a resource or other entry in its environment context through a dependency annotation. A dependency annotation specifies the type of object or resource on which the bean is dependent, its characteristics, and the name through which it is to be accessed. The following are examples of dependency annotations:

```
@EJB(name="mySessionBean", beanInterface=MySessionIF.class)
@Resource(name="myDB", type="javax.sql.DataSource.class")
```

Dependency annotations may be attached to the bean class or to its instance variables or methods. The amount of information that needs to be specified for a dependency annotation depends upon its usage context and how much information can be inferred from that context.

Injecting arbitrary resources with `@Resource`

The `@EJB` annotation only injects EJB stubs. A more generic dependency injection annotation is `@Resource`. Using the `@Resource` annotation, you can inject any service object from the JNDI using the object's JNDI name. Both global (java:/) and local (java:comp/env) JNDI trees are searched. The following examples inject a messaging connection factory and a messaging queue:

```
@Resource (name="ConnectionFactory") QueueConnectionFactory
factory;

@Resource (name="queue/A") Queue queue;
```

For "well-known" objects such as `TimerService` and `SessionContext`, the JNDI names are standard, and therefore the `@Resource` annotation can inject these objects without an explicit specification of the "name" attribute:

```
@Resource TimerService tms;

@Resource SessionContext ctx;
```

Similar to the `@EJB` annotation, the `@Resource` annotation can be applied to setter methods, and the `@Resources` annotation can be applied to arrays. Both the `@EJB` and `@Resource` annotations are specifically tailored to the resources they inject. They simplify the developer's work.

Code sample

In the example below, the variable `customerDB` will be assigned a DataSource object with JNDI name `myDB`. The "name" attribute needs to be specified because the name of the variable we have chosen, `customerDB`, is different from the JNDI name `myDB`. The "type" attribute does not need to be specified because it can be derived from the type of the variable (for example, `DataSource`):

```
@Stateless public class MySessionBean implements MySession {

//type is inferred from variable
@Resource(name="myDB") public DataSource customerDB;

public void myMethod1(String myString){
try  {
  Connection conn = customerDB.getConnection();
  ...
catch (Exception ex)
  }
}
```

Changes to the Four Types of Enterprise Beans

As we all know, there are four kinds of EJBs, and needless to say, EJB 3.0 made some changes to each type of EJB. In this section, we will look at the changes proposed for each type of EJB. One of the main advantages is that in EJB 3.0, all the managed service objects are POJOs (for example, session beans) or very lightweight components (such as message-driven beans). As you'll see, EJB 3.0 has made the development of EJBs much easier and simpler.

Stateless Session Beans

An EJB 3.0 session bean is a POJO managed by the EJB container.

The functionality of a session bean is defined by its service interface (a.k.a. business interface), which is a plain old Java interface. Using the interface class name, the session bean client retrieves a stub object of the bean from the server's JNDI. The stub object implements the bean's service interface. The client can then make calls to the bean interface methods against the stub object. The stub object simply passes the calls to the actual bean instance objects in the container, which have the implementations of those methods and do the actual work. The stub object is automatically generated by the EJB container, and it knows how to route the bean method calls to the container—you do not need to provide an implementation for the stub object. In a stateless session bean, the client-side stub object can route your method call to any bean instance that happens to be available in the container-managed object pool. Therefore, you should not have any field variables to store the bean state in the bean class.

Business interfaces

Business interfaces are required for stateless session beans. It is not always necessary to define one. When undefined, they will be automatically generated for you. The type of generated interface, either local or remote, is dependent on the annotation you used in the bean class and will be a local interface if there is no annotation. All the public methods of the bean class will be included as part of the automatically generated business interface.

Home interfaces

Stateless session beans do not need home interfaces. The client may acquire a reference to a stateless session bean by means of injection or annotation of variables.

Bean class

A stateless session bean must be annotated with the stateless annotation or denoted in the deployment descriptor as a stateless session bean. The bean class need not implement the `javax.ejb.SessionBean` interface. The `@Stateless` annotation indicates that this bean is a stateless session bean:

```
@Stateless
public class TraderBean implements Trader {
public void buy (String symbol, int quantity){
System.out.println("Buying "+quantity+ " of "+ symbol);
}
public void sell (String symbol, int quantity);{
System.out.println("Selling "+quantity+ " of "+ symbol);
}
}
```

The session bean client

Once the session bean is deployed into the EJB 3.0 container, a stub object is created, and it is registered in the server's JNDI registry. The client code obtains a stub of the bean using the class name of the interface in the JNDI. Below is an example on how to retrieve a stub instance of the `TraderBean` for this JSP page. You can make method calls against the stub object, and the call is transparently delegated to the bean instance in the EJB 3.0 container:

```
private Trader tr = null;
public void initialize () {
    try {
      InitialContext ctx = new InitialContext();
      tr = (Trader) ctx.lookup(
                  Trader.class.getName());
    }catch (Exception e) {
      e.printStackTrace ();
    }
}
// ... ...
public void service (Request req, Response rep) {
    // ... ...
    double res = tr.buy("SNPS",1000);
}
```

Callbacks for stateless session beans

The following life cycle event callbacks are supported for stateless session beans:

- `PostConstruct`
- `PreDestroy`

The `PostConstruct` callback occurs after any dependency injection has been performed by the container and before the first business method invocation on the bean. The `PostConstruct` method is invoked in an unspecified transaction context and security context.

The `PreDestroy` callback occurs at the time the bean instance is destroyed. The `PreDestroy` method executes in an unspecified transaction and security context.

Remote and local interfaces

A session bean can also implement multiple interfaces, each interface targeting a different type of client. By default, the interface is for a "local" client that runs in the same JVM as the EJB 3.0 container. Method call invocations over Java references are fast and efficient. Another type of session bean interface, the remote interface, is for remote clients. When a client looks up the session bean stub via the remote interface, the container returns a serialized stub object that implements the remote interface. The remote stub knows how to pass remote procedure calls (RPCs) to the server, even in a clustered environment. The remote interface is also a plain old Java interface.

Note that using the remote interface involves the serialization and deserialization of the stub, and all calls to the bean instance are made over the network. This approach is considerably less efficient than using the local interface. You should avoid looking up a remote interface from a local client.

In the session bean implementation, you can use the `@Local` and `@Remote` annotations to specify the local and remote interfaces for this bean. Here is an example bean that implements both a local and remote interface:

```
@Stateless
@Local ({Trader.class})
@Remote ({RemoteTrader.class})
public class TraderBean implements Trader, RemoteTrader {

  public void buy (String symbol, int quantity){
    System.out.println("Buying "+quantity+ " of "+ symbol);
  }

  public void sell (String symbol, int quantity);{
    System.out.println("Selling "+quantity+ " of "+ symbol);
  }
}
```

The `@Local` and `@Remote` annotations can also be used to tag session bean interfaces instead of the bean implementation class. For instance, the following code snippet specifies that the RemoteTrader is a remote interface. With that, you no longer need the `@Remote` tag on `TraderBean`.

Stateful Session Beans

The stateful session bean is a session bean that maintains its internal states. If the client invokes method calls against the same bean stub, the calls are always tunneled to the same bean instance in the container. So, all field variables in the bean instance retain their values as long as the client application retains the bean stub (or reference for a local client).

Business interface

The business interface of a stateful session bean on the EJB 3.0 API is also a plain Java interface. Business interfaces are required for stateful session beans. It is not always necessary to define one. When undefined they will be automatically generated for you. The type of generated interface, either local or remote, is dependent on the annotation you used in the bean class and will be a local interface if there is no annotation. All the public methods of the bean class will be included as part of the automatically generated business interface.

Home interface

Stateful session beans do not need home interfaces.

Bean class

A stateful session bean must be annotated with the stateful annotation or denoted in the deployment descriptor as a stateful session bean. The bean class does not need to implement the `javax.ejb.Session` Bean interface. A stateful session bean may implement the `SessionSynchronization` interface.

The implementation of the stateful `TraderBean` is straightforward. We annotated the implementation class as `@Stateful` and used Java objects (like Integer, String) to back up the bean properties defined in the session bean interface. The Java objects are initialized for each bean instance when it is created, at the beginning of a client session. Below is the complete code for the `TraderBean` class. It is important to note that the stateful session bean class must implement the serializable interface so that the container can serialize the bean instances and store them to preserve the state information when the instances are not in use.

```
@Stateful
public class TraderBean implements Trader, Serializable {

  public String symbol = "";
  public int quantity = 0;

  public void buy (String symbol, int quantity){
    System.out.println("Buying "+quantity+ " of "+ symbol);
  }
  public void sell (String symbol, int quantity);{
    System.out.println("Selling "+quantity+ " of "+ symbol);
  }
  public String getSymbol(){
    return symbol;
  }
  public int getQuantity(){
    return quantity;
  }
  // Other getter methods for the attributes ...
}
```

The Session Bean Client

Here is a sample client:

```
Trader tr = null;
if (tr == null) {
  try {
    InitialContext ctx = new InitialContext();
    tr = (Trader) ctx.lookup(
                  Trader.class.getName());

  } catch (Exception e) {
    e.printStackTrace ();
  }
}

// Make use of the tr object
```

Callbacks for stateful session beans

Stateful session beans support callbacks for the following life cycle events: construction, destruction, activation, and passivation. The EJB 3.0 specification defines several annotations the bean can use to specify callback methods during the life cycle of the bean. The container automatically calls the annotated methods at different stages of the session bean life cycle. You can use the following annotations to tag any method in the bean class:

- `@PostConstruct`: The annotated method is called by the container immediately after a bean instance is instantiated. This annotation is applicable to both stateless and stateful session beans.

- `@PreDestroy`: The annotated method is called before the container destroys an unused or expired bean instance from its object pool. This annotation is applicable to both stateless and stateful session beans.
- `@PrePassivate`: If a stateful session bean instance is idle for too long, the container may passivate it and store its state to a cache. The method tagged by this annotation is called before the container passivates the bean instance. This annotation is applicable only to stateful session beans.
- `@PostActivate`: When the client uses the passivated stateful session bean again, a new instance is created and the bean state is restored. The method that tagged this annotation is called when the activated bean instance is ready. This annotation is only applicable to stateful session beans.
- `@Init`: This annotation designates initialization methods for a stateful session bean. It is different from the `@PostConstruct` annotation in that multiple methods can be tagged with `@Init` in a stateful session bean. However, each bean instance can have only one `@Init` method invoked. The EJB 3.0 container determines which `@Init` method to invoke depending on how the bean is created (see the EJB 3.0 specification for details). The `@PostConstruct` method is called after the `@Init` method.

Another life cycle method annotation for a stateful session bean is the `@Remove` tag. It is not a callback method since the application, not the container, calls the `@Remove` method on the bean stub to remove the bean instance in the container object pool.

Entity Beans

An EJB 3.0 entity is a lightweight persistent domain object. Entity beans are marked with the `@Entity` annotation, and all properties/fields in the entity bean class not marked with the `@Transient` annotation are considered persistent. Entity bean persistent fields are exposed through JavaBean-style properties or just as public/protected Java class fields.

Entity beans can use helper classes for representing entity bean state, but instances of these classes don't have a persistent identity. Instead, their existence is tied strongly to the owning entity bean instance; also these objects are not shareable across entities.

Home interface

Entity beans do not need home interfaces.

Business interface

Entity beans do not need business interfaces. They are optional.

Entity class

- The entity class must be annotated with the entity annotation or denoted in the XML descriptor as an entity.
- The entity class must have a no-arg constructor. The entity class may have other constructors as well.
- The no-arg constructor must be public or protected.

Persistent fields and properties

For single-valued persistent properties, the method signatures are:

- `T getProperty()`
- `void setProperty(T t)`

Code sample

As you can see from the following code sample, an entity bean is annotated with a `@Entity` tag. In the sample, we have some member variables and their corresponding getters and setters. Also the code sample shows how to annotate the CMR relationship.

A one-to-many relationship is shown using the `@OneToMany` tag. In this example, the `Customer` bean has a one-to-many relationship with the `Orders`code> bean (one customer can have multiple orders).

Similarly, `Customer`code> has a many-to-many relationship with the `Phones`code> bean. Some business methods will be defined in the business interface and implemented in the bean, for example, `addPhone()` which adds a phone record and associates it with the customer:

```
@Entity
public class Customer implements Serializable {
  private Long id;
  private String name;
  private Address address;
  private Collection orders = new HashSet();
  private Set phones = new HashSet();
```

```
// No-arg constructor
public Customer() {}
@Id
public Long getId() {
  return id;
}
public void setId(Long id) {
  this.id = id;
}
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;
}
@OneToMany
public Collection getOrders() {
  return orders;
}
public void setOrders(Collection orders) {
  this.orders = orders;
}
@ManyToMany
public Set getPhones() {
  return phones;
}
public void setPhones(Set phones) {
  this.phones = phones;
}
// Business method to add a phone number to the customer
public void addPhone(PhoneNumber phone) {
  this.getPhones().add(phone);
  // Set the phone's ref to this customer
  phone.setCustomer(this);
}
```

```
  }
}
```

Message-driven Beans

Let's now look at the final type of EJB: message-driven beans.

Business interface

The business interface of a message-driven bean (MDB) is the message-listener interface that is determined by the messaging type in use for the bean. The interface is `javax.jms.MessageListener`. The message-driven bean must implement the appropriate message listener interface for the messaging type that the message-driven bean supports or must designate its message-listener interface using the `@MessageDriven` annotation or the deployment descriptor.

Bean class

In EJB 3.0, the MDB bean class is annotated with the `@MessageDriven` annotation, which specifies the message queue this MDB monitors (such as queue/mdb).

The bean class needs to implement the MessageListener interface, which defines only one method, `onMessage()`. When a message arrives in the queue monitored by this MDB, the container calls the bean class's `onMessage()` method and passes the incoming message in as the call parameter.

In our example, the `TraderBean.onMessage()` method retrieves the message body, parses out the parameters, performs the trade, and saves the result to a static data manager class. The "sent" timestamp on the service request message serves as the unique ID for the calculation record (it works well for low-volume Web sites). A `check.jsp` JSP page picks up and displays the calculation record based on the message ID:

```
@MessageDriven(activateConfig =
{
  @ActivationConfigProperty(propertyName="destinationType",
    propertyValue="javax.jms.Queue"),
  @ActivationConfigProperty(propertyName="destination",
    propertyValue="queue/mdb")
})
public class TraderBean implements MessageListener {

  public void onMessage (Message msg) {
    try {
      TextMessage tmsg = (TextMessage) msg;
      Timestamp sent =
          new Timestamp(tmsg.getLongProperty("sent"));
      StringTokenizer st =
          new StringTokenizer(tmsg.getText(), ",");

           buy ("SNPS",1000);

      RecordManager.addRecord (sent, "BUY SUCCESSFUL");

    } catch (Exception e) {
      e.printStackTrace ();
    }
  }
  // ... ...
}
```

Sending a message

To use the message-driven bean, the client (such as the JSP page, trader.jsp, in this case) uses the standard JMS API to obtain the target message queue to the MDB by way of the queue name (queue/mdb), and then it sends the message to the queue:

```
try {
  InitialContext ctx = new InitialContext();
  queue = (Queue) ctx.lookup("queue/mdb");
  QueueConnectionFactory factory =
    (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
  cnn = factory.createQueueConnection();
  sess = cnn.createQueueSession(
           false,QueueSession.AUTO_ACKNOWLEDGE);
} catch (Exception e) {
    e.printStackTrace ();
}
TextMessage msg = sess.createTextMessage("SNPS",1000);
sender = sess.createSender(queue);
```

```
sender.send(msg);
```

Callbacks for message-driven beans

The following life cycle event callbacks are supported for message-driven beans:

- `PostConstruct`
- `PreDestroy`

What Happens to the Old Entity Model?

The old entity model is still going to remain a part of EJB, and it always will be a part of EJB, for compatibility reasons. The Expert Group is currently looking at a number of the new features in EJB 3.0, which would potentially be useful for people using the old programming model, and thinking about making those available for people using the older programming model. EJB 3.0 plans to extend the EJB-QL for the EJB 2.1-style CMP entity beans. So if you want to stick to the old programming model for a while, you will be able to do that and still use some of the new functionality.

Conclusion

EJB 3.0 goes a long way toward making the EJB programming experience a pleasant one by simplifying development, facilitating test-driven development, and focusing more on plain Java objects (POJOs) rather than on complex APIs. One of the important aspects that we have not covered in detail in this article is the new persistence framework defined in the specification. For details, check the EJB 3.0 API specification, and download the EJB 3.0 persistence documentation.

BEA Systems is working actively on its EJB3 implementation strategy in BEA WebLogic Server. Details on the implementation and the timeline will be provided on this Web site when they are finalized.

Additional Reading

- The JSR 220 - Enterprise JavaBeans 3.0 specification (JCP)
- EJB 3.0 API specification download (JCP)
- JSR 175 - Metadata for the Java language (JCP)
- SDO vs. EJB3 Persistence (Michael Rowley's dev2dev blog, August 2005)
- Tech Talk: EJB 3 (dev2dev live!)
- Visit the dev2dev EJB Technology Center

Vimala Ranganathan is a QA engineer on the workshop team. She has over eight years of experience in Java SE/Java EE technologies.

Anurag Pareek is an Escalation Engineer working for BEA Systems. He has extensive experience in implementing, tuning and troubleshooting mission-critical, high-availability applications.

Return to dev2dev.