

# EJB 3.0 Migration

*An Oracle White Paper*  
*October 2005*



Why Migrate? .....	4
What's Changed in EJB 3.0 .....	4
Session Bean Migration .....	5
Session Bean Changes .....	5
Migrating Application Code using Session Beans.....	7
Migrating Message-Driven Beans (MDBs).....	8
EJB 3.0 Persistence MIGRATION.....	9
The New Persistence API.....	9
Migrating EJB 2.x CMP Entity Beans .....	9
DTOs and EJB 3.0 Entities.....	10
Migrating DTOs to EJB 3.0 Entities.....	11
Migrating EJB 2.x Entity Beans .....	12
Migrating EJB 2.x Entity Homes.....	16
Migration of lifecycle methods.....	17
Migration of O-R mapping.....	17
Exception handling .....	18
Migrating the CMP Client Application Code.....	18
Migrating CMP Clients .....	18
Migrating POJO Applications.....	19
Conclusion.....	19

Oracle's EJB 3.0 functionality is available  
at [otn.oracle.com/ejb3](http://otn.oracle.com/ejb3)

The programming model for Enterprise JavaBeans (EJB) has been dramatically simplified in EJB 3.0 and is being hailed by Java developers as the new standard of server-side business logic programming. Meanwhile, the existence of thousands of J2EE applications written with earlier versions of the EJB API has raised concerns about both the interoperability and migrating the applications to use EJB 3.0. This paper introduces many of the relevant issues.

## Why Migrate?

The question participants in J2EE projects should discuss when planning an upgrade to an EJB 3.0-enabled J2EE container is: Why should we migrate our application? Major application server vendors, such as Oracle, that already provide EJB 3.0 features will continue to support EJB 2.x in the new EJB 3.0 container. This means that applications written with EJB 2.x will continue to run without any change whatsoever. However, some organizations will certainly want to migrate their applications to use the EJB 3.0 API.

The real benefits in migrating from an EJB 2.x include:

- Reduction in the complexity surrounding EJBs. Interfaces, home interfaces, and metadata are greatly simplified.
- Ability to easily test entities and the components that use them. The EJB 2.x persistence solutions have led most applications to avoid important unit testing.
- Reduction in problematic code that works around the limitations of EJB 2.x entity beans. These patterns include data-transfer objects (DTOs) and Service Locator implementations.

These benefits are significant to developer productivity, application quality, and maintenance costs. The challenge is to understand these benefits and the costs of migrating and determine when it makes the most sense in your application's development cycles to perform such a migration.

## WHAT'S CHANGED IN EJB 3.0

The main goals of EJB 3.0 are to simplify the programming model and to define a persistence API for the Java platform. Following are some general changes EJB is making that will simplify life for EJB developers.

For the most up to date information on EJB 3.0 refer to Sun's draft specification

- EJBs are now Plain Old Java Objects (POJOs).
- XML descriptors are no longer necessary; annotations may be used instead.
- Defaults are assumed whenever possible.
- Unnecessary artifacts and lifecycle methods are optional.
- The client view is simplified by dependency injection.

Standardization of the POJO persistence model within the context of J2EE finally provides users with the inheritance and polymorphism that have until now been available only outside the container or in proprietary products. Furthermore, the POJO entity beans can now be used and tested both inside and outside the EJB container.

## SESSION BEAN MIGRATION

The main changes in EJB 3.0 session beans are to simplify development by making beans POJOs and providing the use of annotations as well as XML descriptors and dependency injection instead of JNDI lookups. The changes required in this migration involve changes to the beans themselves and the application code that uses the beans.

### Session Bean Changes

The changes that need to be made to EJB 2.x session beans to migrate to EJB 3.0 include:

- The remote and local interfaces do not have to implement `javax.ejb.EJBObject` or `javax.ejb.EJBLocalObject`. The component interface can become a business interface, and `@Remote` annotations can be used to mark interfaces that will be accessible from remote clients.
- `RemoteExceptions` are no longer required to be thrown by the methods on the remote interface. A simple remote EJB 2.x session bean is defined as:

```
public interface HelloWorld extends EJBObject {  
    String sayHello(String name) throws RemoteException;  
}
```

When migrated to EJB 3.0 the remote interface is simply:

```
@Remote  
public interface HelloWorld {  
    String sayHello(String name);  
}
```

- Bean classes do not implement `javax.ejb.SessionBean` but, rather, their business interfaces. The result of this is that lifecycle methods that are not required by the bean do not need to be implemented. Instead those lifecycle callbacks that are needed are indicated by annotations.

Here is the bean class for a simple EJB 2.x stateless session bean

```
public class HelloWorldBean implements SessionBean {
    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {}
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

The EJB 3.0 bean class will look like the following after the migration:

```
@Stateless
public class HelloWorldBean implements HelloWorld {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

- For stateful session beans the `ejbCreate` methods are replaced with business methods used to initialize the state that needs to be maintained by the container. For example, you can create a business method and annotate with `@PostConstruct` as follows:

```
@PostConstruct
public void initialize() {
    items = new ArrayList();
}
```

Removal methods, which indicate to the container when the Stateful bean is no longer needed by the application, are annotated with `@Remove`.

- Optionally, migrate session bean code to use annotations for transactions and security.

## Migrating Application Code using Session Beans

In addition to the beans themselves the code that uses session beans must be migrated. This includes application code within other J2EE and J2SE components. EJB 3.0 simplifies the use of resources and EJB references, by making use of the principle of dependency injection. Injection of resources or references can occur through either annotation of the injection target or specification of the target in the `ejb-jar.xml` descriptor.

EJB 2.x	EJB 3.0
Using another EJB: <ul style="list-style-type: none"> <li>• <code>ejb-ref</code> in XML descriptor</li> <li>• Do JNDI lookup to get home interface</li> <li>• Call <code>home.create</code> to obtain instance</li> </ul>	Using another EJB: <ul style="list-style-type: none"> <li>• <code>ejb-ref</code> no longer requires home interface</li> <li>• Change from the home interface to the migrated business interface</li> <li>• Remove <code>home.create</code> and directly invoke the methods on the EJB</li> <li>• Optionally annotate EJB business interface property/field to obtain instance</li> </ul>
Resource access <ul style="list-style-type: none"> <li>• Do JNDI lookup to obtain resource</li> </ul>	Resource access <ul style="list-style-type: none"> <li>• Optionally annotate resource property/field to obtain resource</li> </ul>

For example, if you are using a `CartEJB` in another EJB 2.1 bean you have to make a reference to the `CartEJB` in the deployment descriptor using `ejb-ref`.

```
<ejb-ref-name>MyCart</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>CartHome</home>
<remote>Cart</remote>
```

In lookup of the home interface for the `CartEJB`, using JNDI, and create an instance of the `CartEJB` is then possible.

```
Object homeObject = context.lookup("java:comp/env/MyCart");
CartHome home = (CartHome) PortableRemoteObject.narrow(homeObject,
    CartHome.class);
```

```

    Cart cart = (Cart) PortableRemoteObject.narrow(home.create(),
        Cart.class);

    cart.addItem("Item1");

```

After migration to the EJB 3.0 injection pattern, this code can become much simpler with the use of dependency injection.

```

    @EJB Cart cart;

    public void addItem() {
        cart.addItem("Item1");
    }

```

A session bean can also be used as a facade for CMP entity beans, giving rise to a migration strategy for moving CMP entity beans to the EJB 3.0 persistence API.

### MIGRATING MESSAGE-DRIVEN BEANS (MDBS)

In EJB 3.0, MDBs do not have to implement the `javax.ejb.MessageDriven` interface but can instead be annotated with `@MessageDriven`. Resources and EJB references can be injected into MDBs in the same way as session beans. Similarly, you can inject the context (`MessageDrivenContext` for MDB) into the message-driven bean. The following table summarizes the changes in MDB between EJB 2.x and EJB 3.0.

EJB 2.x	EJB 3.0
Implements  <code>javax.ejb.MessageDriven</code>	Annotate with:  <code>@MessageDriven</code>
Specify destination type, name, etc. in deployment descriptor	May be annotated with:  <code>@ActivationConfigProperty</code>
<code>MessageDrivenContext</code> is acquired using:  <code>setMessageDrivenContext()</code>	<code>MessageDrivenContext</code> is achieved using dependency injection e.g.  <code>@Resource javax.ejb.MessageDrivenContext mc;</code>
Resource usage such as Queue or Topic required resource-ref in deployment descriptor and JNDI lookup.	Can be done using dependency injection as follows:  <code>@Resource(name="jms/myQueue")</code> <code>private QueueConnectionFactory</code> <code>queueConnectionFactory;</code>

## **EJB 3.0 PERSISTENCE MIGRATION**

Persistence is one of the greatest challenges facing J2EE developers, compounded further by the lack of a standard persistence API for the Enterprise Java platform. J2EE applications typically use one of the following persistence choices:

- EJB 2.x Container-Managed Persistence (CMP) entity beans
- A POJO persistence framework such as Oracle TopLink™, JBoss Hibernate™, or another custom O-R mapping framework
- Data Access Objects with JDBC
- Java Data Objects

The new persistence API defined in EJB 3.0 will deliver a much-needed standard to reconcile all of these options. Each existing approach to persistence will provide its own unique migration challenges and opportunities. This paper will focus on the migration from EJB 2.x CMP as it is the most dramatic change and will effect customers who embraced the current solution defined in the 2.x specification.

### **The New Persistence API**

The core of this new persistence API is the `EntityManager`. The `EntityManager` is the primary interface used by the application to retrieve and modify persistent entities. In addition to this interface being available within J2EE components the new persistence API also provides for support outside of an integrated EJB 3.0 container.

This support is referred to as Application Managed and is primarily focused, but not limited to, testing of application code and EJB components outside of the container. This will allow for developers to construct simple test cases to validate the behavior of their entities and the code that interacts with them. This is one of the most significant benefits of the persistence API.

### **Migrating EJB 2.x CMP Entity Beans**

The process of moving from EJB 2.x persistence to EJB 3.0 can range from coexistence to a full migration and adoption of the new approach. Migration involves changing the model and its mappings, the queries, and the applications' code to use the persistence API.

The migration can be divided into three basic approaches:

1. Migrate the entity DTOs to EJB 3.0 entities.
2. Migrate the EJB 2.x entity beans to EJB 3.0 entities.
3. Create a new EJB 3.0 entity model.

The first two approaches are intended to minimize code changes in your application. The third involves creating a new model, possibly using forward generation tools to reduce coding and configuration work. Although this may be

the most efficient way to create the EJB 3.0 persistence model, it will most likely incur the most migration costs within the application code.

### **DTOs and EJB 3.0 Entities**

Before a discussion of which approach is best for a given application, the issue of DTOs and their role in EJB 3.0 persistence should be clarified. In EJB 2.x, DTOs were required for getting persistent data outside of the container for use in another tier of the application. In EJB 3.0, entities can be serialized and the need to use DTOs is removed. This does not mean that DTOs cannot be used if they provide necessary decoupling, but it does address the common practice of having DTOs that mirror entity beans and replicate business logic. These will be referred to as “entity DTOs,” and they are no longer required.

Another flavor of DTO is the “view DTO.” This is an object that does not mirror the CMP entities but instead provides a more coarse-grained view of data that may come from one or more entity beans. View DTOs are used to optimize interaction between tiers. EJB 3.0 does not remove the need for this style of DTO but instead provides facilities to simplify their use. Whereas in traditional applications, the underlying entities are retrieved and programmatically populated into view DTOs, EJB 3.0 query results may now be directly projected onto view DTOs without any additional binding code.

In EJB 3.0, an EJB QL query can return results by using nonpersistent classes. In this case, the query statement defines the attributes needed and the selection criteria relative to the persistence model, but instead of returning one of these types, a nonpersistent class is specified. The NEW operator is used for this specification and defines the class and constructor to be used when building these result objects.

```
SELECT
    NEW EmpView(e.id, e.firstName, e.lastName, d.name)
FROM Employee e JOIN e.department d
WHERE e.salary > 50000
```

Deciding to migrate your entity DTOs to become your EJB 3.0 entities is a good choice for minimizing changes on the client tier that use these serialized objects. It is also a good choice if your application logic in the EJB tier is minimal or also focused on the entity DTOs.

This approach requires that the entity DTO is a valid JavaBean and maintains a one-to-one correspondence with the data elements in the associated entity bean. It is also highly recommended that a DAO layer be in place, so that the persistent logic can be updated in isolation once the DTOs have been converted. If your application logic makes direct use of EJB 2.x entities, the techniques described in the section on migrating entity beans may be more suitable.

### Migrating DTOs to EJB 3.0 Entities

The first step in the migration process is to mark the entity DTO as a persistent object, by adding the `@Entity` annotation to the class definition. Without any further changes, the entity DTO is now a persistent object that maps to the database, using the EJB 3.0 defaults. The table name is assumed to be the class name, and each public JavaBean getter method is assumed to correspond to a column in the table with the same name as the property. You can customize these default mappings by using `@Table`, `@Column`, and other EJB 3.0 annotations.

After each entity DTO is marked as a persistent object and the database mappings are set up correctly, the next step is to define the relationships between the new entities. EJB 3.0 includes a full set of object-relational annotations to identify entity relationships and control their behavior during persistent operations. The `<ejb-relation>` descriptor in the EJB 2.x `ejb-jar.xml` file can be directly mapped onto each pair of entities involved in the relationship. Consider the following relationship descriptor:

```
<ejb-relation>
    <ejb-relation-name>Dept-Emps</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>Dept-has-Emps
            </ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>DeptBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>employees</cmr-field-name>
            <cmr-field-type>java.util.Set</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
<ejb-relationship-role>
    <ejb-relationship-role-name>
        Emps-have-Dept
    </ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <relationship-role-source>
```

```

        <ejb-name>EmpBean</ejb-name>

    </relationship-role-source>

    <cmr-field>

        <cmr-field-name>dept</cmr-field-name>

    </cmr-field>

</ejb-relationship-role>

</ejb-relation>

```

This fragment from an Employee entity demonstrates how the Emps-have-Dept role would be mapped:

```

@Entity
public class Employee {

    private Department department;

    // ...

    @ManyToOne

    public Department getDepartment() {

        return department;

    }

    public void setDepartment(Department dept) {

        this. department = dept;

    }

}

```

Just as the table and field names are assumed by default and can be overridden, so also can the foreign key column names be defaulted. For these the relationship mapping annotations make use of `@JoinColumn` to provide column names that are different from the defaults.

It is important to note that at this point, there are no behavioral changes to the existing application. We have merely added metadata to the original entity DTOs that makes it possible to use them as persistent EJB 3.0 entities. Assuming there is a DAO layer in place that manages the entity DTOs, this layer can now be updated to use the `EntityManager` to retrieve, store, and update the newly annotated persistent objects.

### **Migrating EJB 2.x Entity Beans**

An alternative to migrating DTOs to be the entities is to leverage the existing EJB 2.x entity model. For the purposes of this discussion, we'll assume that only local interfaces to the entity beans are used, because these are the most prevalent in today's applications and because entities as remote objects are not supported in

EJB 3.0. For those applications using entity beans as remote objects, the migration will involve more-fundamental design changes.

The goal in this type of migration is to minimize the changes to client code, which is bound to the local and home interfaces of an entity:

1. Remove the abstract designation on the bean class, and provide concrete implementations of the property accessor (get/set) methods.
2. Annotate the bean class with the object-relational annotations that map the entity to the database, and describe its relationships.
3. Rename the bean class to be that of the local interface, and remove the local interface. This will minimize changes to application code that already uses the bean through this interface.
4. Convert finders and selects associated with the bean to `@NamedQuery` annotations on the bean class.

If your entity bean acts as a client to other beans, this code must be migrated as application client code, as discussed later.

Consider the following EJB 2.x Department bean:

```
public abstract class DepartmentBean implements EntityBean {  
    public abstract Long getId();  
    public abstract void setId(Long id);  
    public abstract String getName();  
    public abstract void setName(String name);  
    public abstract Collection getEmployees();  
    public abstract void setEmployees(Collection employees);  
  
    public void ejbCreate(Long id) throws CreateException {}  
    public void ejbPostCreate(Long id) throws CreateException {}  
    public void ejbStore() {}  
    public void ejbLoad() {}  
    public void ejbRemove() throws RemoveException {}  
    public void ejbActivate() {}  
    public void ejbPassivate() {}  
    public void setEntityContext(EntityContext ctx) {}  
    public void unsetEntityContext() {}  
}
```

It depends upon the following finder definitions:

```

<query>
    <description></description>
    <query-method>
        <method-name>findAll</method-name>
        <method-params/>
    </query-method>
    <result-type-mapping>Local</result-type-mapping>
    <ejb-ql>Select OBJECT(d) From Department d</ejb-ql>
</query>

<query>
    <description></description>
    <query-method>
        <method-name>findByDeptName</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <result-type-mapping>Local</result-type-mapping>
    <ejb-ql>SELECT DISTINCT OBJECT(d) FROM Department d WHERE
d.name = ?1</ejb-ql>
</query>

```

Following the steps above, we end up with the following:

```

@Entity
@Table (name="DEPT")
@NamedQueries ({
    @NamedQuery (name="DepartmentFindAll",
        queryString="SELECT OBJECT(d) FROM Department d"),
    @NamedQuery (name="DepartmentFindByName",
        queryString="SELECT OBJECT(d) FROM Department d
            WHERE d.name=?1")
})
public class Department {

```

```

private long id;

private String name;

private Collection<Employee> employees;

public Department() {}

@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; }
@OneToMany
public Collection getEmployees() { return employees; }
public void setEmployees(Collection employees) {
    this.employees = employees;
}
}

```

Of special interest is the use of EJB 2.x container-managed relationships (CMR). In EJB 3.0, relationship management is performed with standard Java coding practices. It is the responsibility of the developer to ensure that both sides of a relationship are updated correctly when relationships are formed. Generally speaking, one additional line of code is required for every relationship operation to ensure that the model is consistent. With respect to collections, we recommend the use of helper methods such as `add` and `remove` instead of direct manipulation of the collection in client code.

```

@Entity
public class Department {
    private Collection<Employee> employees;

    public void addEmployee(Employee emp) {
        employees.add(emp);
        emp.setDepartment(this);
    }
}

```

Besides these more common migration changes, there is additional EJB 2.x entity bean implementation that may require work. These include the use of the `EntityContext` within the bean itself and `ejbHome` methods. An EJB 3.0 entity will

not have an EntityContext. The entity can simply use “this” to resolve to itself and pass itself into other beans’ methods. Any ejbHome methods implemented can simply remain as is or be converted into static methods to make their use more obvious.

### **Migrating EJB 2.x Entity Homes**

The home interface is the primary mechanism that enables application code to locate and manage entity beans. Because EJB 3.0 has nothing equivalent to home interfaces, it also presents the greatest challenge in migration.

To minimize changes to client code, the easiest approach to migrating homes is to create a helper class that holds onto an EntityManager and implement the methods called on the home by the application code.

The following example demonstrates a helper class that functions as a replacement home:

```
public class DepartmentHome {
    private EntityManager em;

    public DeptHome(EntityManager em) {
        this.em = em;
    }

    public Department create(long id) {
        Department dept = new Department(id);
        em.persist(dept);
    }

    public void remove(Department dept) {
        em.remove(dept);
    }

    public Department findByPrimaryKey(long id) {
        return em.find(Department.class, id);
    }

    public Department findByName(String name) {
        return (Department)
            em.createNamedQuery("DepartmentFindByName")
                .setParameter(0, name)
                .getSingleResult();
    }

    public Collection findAll() {
```

```

        return em.createNamedQuery("DepartmentFindAll")
            .listResults();
    }
}

```

This helper class leverages the named queries we defined on the entity class to implement finders, although dynamic queries could also have been used. The remove method on this helper replaces the `EJBLocalObject.remove()` previously on the entity bean. It is important that all calls to the bean's remove method be migrated to call this one provided on the helper or to directly make the call with the `EntityManager`.

### Migration of lifecycle methods

EJB 2.x has a strict requirement to implement the `EntityBean` interface that defined the complete suite of lifecycle methods. Some of these lifecycle events no longer even apply.

The following table suggests strategies for migrating existing implementations of these methods.

EJB 2.x lifecycle method	What to do in EJB 3.0
<code>ejbCreate</code>	Implement logic in init methods/constructors
<code>ejbPostCreate</code>	Implement logic in constructor or create a business method and annotate as <code>@PostPersist</code>
<code>ejbRemove</code>	Create a business method and annotate with <code>@PreRemove</code>
<code>setEntityContext</code> <code>unSetEntityContext</code>	No longer applies
<code>ejbActivate</code>	Create a business method and annotate as <code>@PostLoad</code>
<code>ejbPassivate</code>	No longer applies (can be removed)
<code>ejbStore</code>	Create a business method and annotate with either <code>@PrePersist</code> or <code>@PreUpdate</code>

### Migration of O-R mapping

Before EJB 3.0, O-R mappings were always in the domain of the vendor and, as such, were typically stored in a vendor-specific deployment descriptor. Now that the O-R mappings have been integrated into the EJB standard, they can be defined either as annotations on the bean class or in a standard XML file. Most vendors should be providing ways to translate their mappings into the standard EJB mappings, either through an automated migration tool or from a mapping editor.

### Exception handling

Exceptions are unchecked, so the same `CreateException`, `RemoveException`, and `FinderException` catch phrases do not apply.

### Migrating the CMP Client Application Code

The application code that makes use of EJB 2.x entity beans is tightly coupled to the `Home` and `Local` or `Remote` interfaces of the entity beans. The migration of this code is least likely to be automated, but

- Home interface lookup and query execution must be modified to either directly use the `EntityManager` or a `Home`-helper class that is not managed by the container.
- Code using the entity beans interfaces must be changed to use the new EJB 3.0 entity. The entity class can be the interface, the entity `DTO`, or some new entity bean class. The selection and intrusiveness of these changes will depend on the model migration approach chosen.
- Entity `DTO` detachment and attachment code can be removed, in favor of detachment/attachment of the entities.
- View `DTO` creation code can be replaced with EJB QL queries that dynamically create the non-persistent view objects.

### Migrating CMP Clients

In EJB 2.x, entity beans were accessed by local and/or remote clients. A well-documented best practice was to keep the entity beans local and wrap them in a session facade. In EJB 3.0, entities, being regular Java objects, are always local and can never be remote (RMI) objects. They are obtained and queried for through the `EntityManager` API. Client code (a session bean in the case of a session facade) must change its use of entity home or component-specific methods to use the `EntityManager` to access the entities on which it operates.

Local and remote entity references used to be required declarations in the deployment descriptor. In the client code, the home would be looked up in JNDI, and home operations, such as `create` and `find` could then be performed. An example of a client lookup is as follows:

```
public void createNewCustomer(String name, String city)
    throws CreateException, NamingException, RemoteException {
    InitialContext ctx = new InitialContext();
    CustomerHome home = (CustomerHome) ctx.lookup(
        "java:comp/env/ejb/CustomerHome");
    CustomerLocal customer = null;
    customer = home.create(name, city);
}
```

```
}
```

The migrated version of this code would look like this:

```
@PersistenceContext private EntityManager em;

public void createNewCustomer(String name, String city) {

    Customer cust = new Customer();

    cust.setName(name);

    cust.setCity(city);

    em.persist(cust);

}
```

It is clear from the preceding discussion that migrating EJB 2.x entity beans to EJB 3.0 is the most complex task, will have an impact on clients, and needs careful planning.

### **Migrating POJO Applications**

Some frameworks have been persisting Java objects to relational databases for a long time, and vast numbers of applications are written by use of O-R frameworks such as Oracle TopLink. Applications that are currently using a POJO persistence framework are well positioned to migrate to the EJB persistence API, because the domain objects are already Java objects. Furthermore the persistence frameworks' transaction mechanisms and session-level APIs are similar to those of the new EJB persistence API. The result is that very little code rework is required. If you are currently evaluating a persistence framework for your J2EE applications probably using a POJO persistence framework like Oracle TopLink is the best choice because it can you easily get you to EJB 3.0 Persistence API.

Two steps remain for getting to the new persistence API:

- Change the session APIs used in the persistence framework to EntityManager APIs.
- Change proprietary O-R XML to O-R mapping annotations or O-R XML defined by the EJB 3.0 persistence API.

The technical details of migration of POJO persistence to EJB 3.0 will be covered in another article in future.

### **CONCLUSION**

Migration of J2EE applications to EJB 3.0 is certainly not rocket science but, like any other migration effort, takes education, resources, and planning. The first step is to become aware of where the platform is moving and understand where it plans to end up. Once you understand the technology, you should undertake planning to decide the best strategy for getting there.

The prudent approach to migration is to migrate a subsection of the application before applying the practices to the whole. This strategy will ferret out many of the difficult migration issues and allow for experimentation to discover which approach might be best suited to the application. Features such as interoperability between EJB 2.x and EJB 3.0 components are critical when you're pursuing this kind of strategy. You can start trying out EJB 3.0 with early implementation of EJB 3.0 such as Oracle Application Server 10g and get ready for migrating to EJB 3.0.



EJB 3.0 Migration

June 2005

Authors:

Debu Panda (debabrata.panda@oracle.com)

Doug Clarke (douglas.clarke@oracle.com)

Merrick Schincariol (merrick.schincariol@oracle.com)

Oracle Corporation  
World Headquarters  
500 Oracle Parkway  
Redwood Shores, CA 94065  
U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

[www.oracle.com](http://www.oracle.com)

Copyright © 2003, Oracle. All rights reserved.

This document is provided for information purposes only

and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.