Application Development Framework
Application Express
Business Intelligence
Cloud Computing
Communications
Database Performance & Availability
Data Warehousing
.NET
Dynamic Scripting Languages
Embedded
Enterprise 2.0
Enterprise Architecture
Enterprise Management
Grid Computing
Identity & Security
Java
Linux
Service-Oriented Architecture
SQL & PL/SQL
Server and Storage Administration
Server and Storage Development
Systems Hardware and Architecture
Virtualization

The Java Persistence API - A Simpler Programming Model for Entity Persistence

By Rahul Biswas and Ed Ort, May 2006

The major theme of version 5 of the Java Platform, Enterprise Edition (Java EE, formerly referred to as J2EE) is ease of development. Changes throughout the platform make the development of enterprise Java technology applications much easier, with far less coding. Significantly, these simplifications have not changed the platform's power: The Java EE 5 platform maintains all the functional richness of the previous version, J2EE 1.4.

Enterprise developers should notice dramatic simplification in Enterprise JavaBeans (EJB) technology. Previous articles, such as Introduction to the Java EE 5 Platform and Ease of Development in Enterprise JavaBeans Technology have described the simplifications made in EJB 3.0 technology, an integral part of the Java EE 5 platform.

A major enhancement in EJB technology is the addition of the new Java Persistence API, which simplifies the entity persistence model and adds capabilities that were not in EJB 2.1 technology. The Java Persistence API deals with the way relational data is mapped to Java objects ("persistent entities"), the way that these objects are stored in a relational database so that they can be accessed at a later time, and the continued existence of an entity's state even after the application that uses it ends. In addition to simplifying the entity persistence model, the Java Persistence API standardizes object-relational mapping.

In short, EJB 3.0 is much easier to learn and use than was EJB 2.1, the technology's previous version, and should result in faster development of applications. With the inclusion of the Java Persistence API, EJB 3.0 technology also offers developers an entity programming model that is both easier to use and yet richer.

The Java Persistence API draws on ideas from leading persistence frameworks and APIs such as Hibernate, Oracle TopLink, and Java Data Objects (JDO), and well as on the earlier EJB container-managed persistence. The Expert Group for the Enterprise JavaBeans 3.0 Specification (JSR 220) has representation from experts in all of these areas as well as from other individuals of note in the persistence community.

This article supplements the earlier articles by focusing on entity-related code. Here you'll be able to examine EJB 2.1 entity beans in an application and compare them to EJB 3.0 entities in an equivalent application. More specifically, you'll be able to view side by side the source code for EJB 2.1 entity beans that use container-managed persistence and relationships and compare them to the source code for equivalently functioning EJB 3.0 entities written to the Java Persistence API. Note that in the Java Persistence API, what used to be called entity beans are now simply called entities. You'll see how much easier and streamlined the EJB 3.0 technology code is.

The article highlights some of the important features that simplify the EJB 3.0 version of the code. Although this article focuses on the Java Persistence API and its use in an EJB 3.0 container, the API can also be used outside the container -- for instance, in applications for the Java Platform, Standard Edition (Java SE, formerly referred to as J2SE). The API also provides support for pluggable, third-party persistence providers. For example, a persistence implementation from one vendor can be used with an EJB container from another vendor, provided that the container and the persistence implementation both conform to JSR 220.

This article assumes that you're familiar with the basic concepts of EJB technology that underlie EJB 2.1. If you're not, see the chapter "Enterprise Beans" in the J2EE 1.4 Tutorial. For more information about EJB 3.0 and Java Persistence concepts, see the chapter "Enterprise Beans" in the Java EE 5 Tutorial.

Simplicity at a Glance

The Java Persistence API simplifies the programming model for entity persistence and adds capabilities that were not in EJB 2.1. Here's a quick list of its simplifications and additions:

- Requires fewer classes and interfaces
- Virtually eliminates lengthy deployment descriptors through annotations
- Addresses most typical specifications through annotation defaults
- Provides cleaner, easier, standardized object-relational mapping
- Eliminates the need for lookup code
- Adds support for inheritance, polymorphism, and polymorphic queries.
- Adds support for named (static) and dynamic queries.
- Provides a Java Persistence query language -- an enhanced EJB QL
- Makes it easier to test entities outside of the EJB container
- Can be used outside of the container
- Can be used with pluggable, third-party persistence providers

Contents

First, let's look at the application in use.

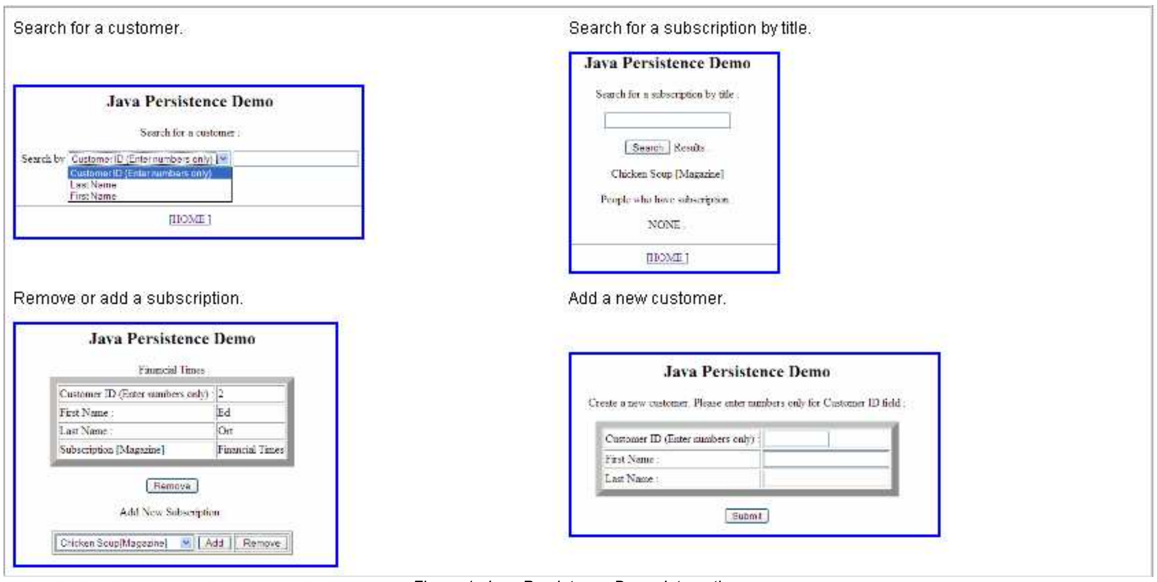
The Application

This article uses the EJB 2.1 technology version of the application, the CMP Customer Sample Application, from the J2EE 1.4 samples bundle. You can download the samples bundle from the J2EE 1.4 Downloads page. The samples bundle includes instructions for installing and running the EJB 2.1 version of the application.

The EJB 3.0 version of the same application is called the Java Persistence Demo and is available for download here. The download bundle includes instructions for installing and running the EJB 3.0 version of the application.

Both versions of the application do the same thing: They interact with a relational database to store and display information about customer subscriptions to periodicals. The database stores information such as a customer's name and address, as well as the type of periodical to which the customer is subscribed -- that is, magazine, journal, or newspaper. You can submit requests to the application to display subscription-related information. For example, you can get a list of all subscriptions for a particular customer.

Figure 1 shows what some of the interactions look like. Click to enlarge each image.


Search for a customer.

Java Persistence Demo

Search for a customer :

Search by: Customer ID (Enter numbers only)

- Customer ID (Enter numbers only)
- Last Name
- First Name

[HOME]

Search for a subscription by title.

Java Persistence Demo

Search for a subscription by title :

Search Results

Chicken Soup [Magazine]

People who have subscription

NONE

[HOME]

Remove or add a subscription.

Java Persistence Demo

Financial Times

Customer ID (Enter numbers only) :	2
First Name :	Ed
Last Name :	Ort
Subscription [Magazine]	Financial Times

Remove

Add New Subscription

Chicken Soup[Magazine] Add Remove

Add a new customer.

Java Persistence Demo

Create a new customer. Please enter numbers only for Customer ID field :

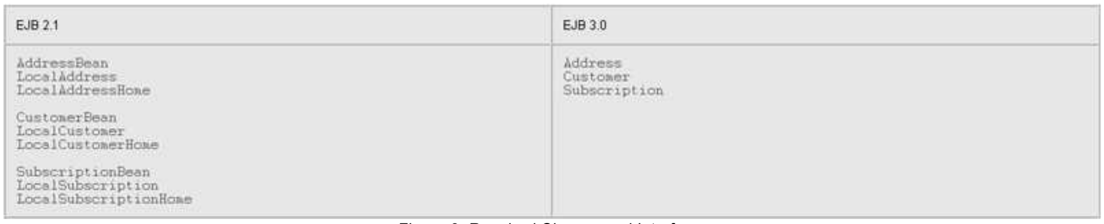
Customer ID (Enter numbers only) :	
First Name :	
Last Name :	

Submit

Figure 1: Java Persistence Demo Interactions

The Entities

Let's examine the entity-related code within the application. First, let's look at what classes and interfaces are required for the entities in the application. Look at Figure 2. Compare the list in the EJB 2.1 version with that of the EJB 3.0 version.



EJB 2.1	EJB 3.0
AddressBean LocalAddress LocalAddressHome CustomerBean LocalCustomer LocalCustomerHome SubscriptionBean LocalSubscription LocalSubscriptionHome	Address Customer Subscription

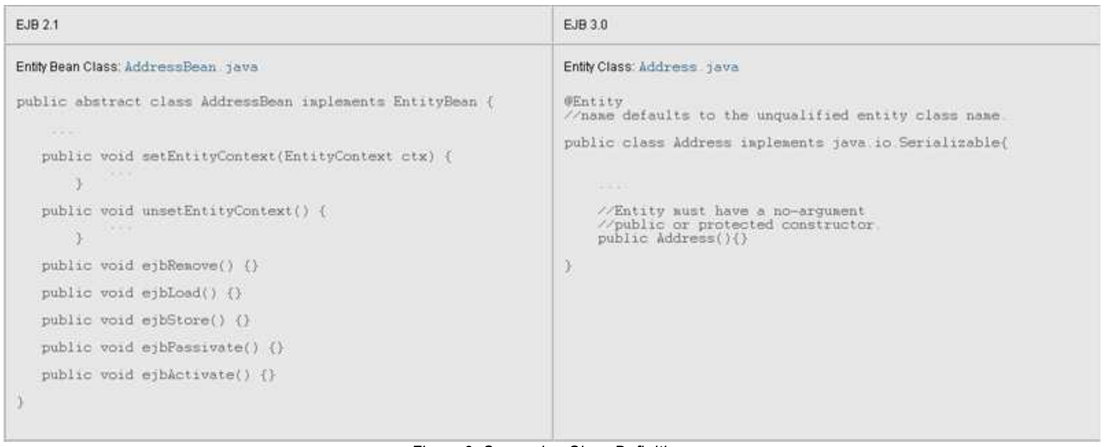
Figure 2: Required Classes and Interfaces
Click here for a larger image

You need to code fewer classes and interfaces

For an EJB 3.0 entity, you no longer need to code interfaces such as LocalAddressHome and LocalAddress -- or even a deployment descriptor. All you need to provide is an entity class. So in the application's EJB 3.0 version, what's required for an entity has been reduced from three classes -- for local interfaces and a business class -- to one entity class. This simplification is not limited to entities. EJB 3.0 technology has eliminated the requirement for a home interface in enterprise beans of any type. In addition, you no longer need to implement the EJBObject and EJBLocalObject interfaces. For example, a session bean now requires only a bean class and a business interface, which is a simple Java technology interface.

The Class Definition

Let's now look at the code for the entities. Let's start by comparing some of the key parts of the code for the EJB 2.1 AddressBean class and the EJB 3.0 Address class. First, we'll look at the class definition. You can view the entire entity bean class and entity class by clicking on their names in the Figure 3.



EJB 2.1

Entity Bean Class: AddressBean.java

```
public abstract class AddressBean implements EntityBean {

    ...

    public void setEntityContext(EntityContext ctx) {
        ...
      }

    public void unsetEntityContext() {
        ...
      }

    public void ejbRemove() {}

    public void ejbLoad() {}

    public void ejbStore() {}

    public void ejbPassivate() {}

    public void ejbActivate() {}
}
```

EJB 3.0

Entity Class: Address.java

```
@Entity
//name defaults to the unqualified entity class name.

public class Address implements java.io.Serializable{

    ...

    //Entity must have a no-argument
    //public or protected constructor.
    public Address(){}

}
```

Figure 3: Comparing Class Definitions
Click here for a larger image

An entity is a Plain Old Java Object (POJO), so no boilerplate is required

The EJB 2.1 version of the entity bean class implements the javax.ejb.EntityBean interface. In fact, an EJB 2.1 entity bean class must implement this interface to interact with the EJB 2.1 container. Because of that, the bean class must also implement all of the methods in the interface: ejbRemove, ejbActivate, ejbPassivate, ejbLoad, ejbStore, setEntityContext, and unsetEntityContext The class must implement these callback methods even if it doesn't use them, as is the case for most of these methods in the EJB 2.1 example in Figure 3.

By comparison, an EJB 3.0 entity class is a simple, nonabstract, concrete class -- a POJO, a class that you can instantiate like any other simple Java technology class, with a new operation. It does not implement javax.ejb.EntityBean or any other container-imposed interface. Because the entity class no longer implements javax.ejb.EntityBean, you are no longer required to implement any callback methods. However, you can still implement callback methods in the entity class if you need them to handle life-cycle events for the entity. Also notice the no-argument public constructor. In EJB 3.0 technology, an entity class must have a no-argument public or protected constructor.

Annotations minimize what needs to be specified

The @Entity metadata annotation marks the EJB 3.0 class as an entity. EJB 3.0 and the Java Persistence API rely heavily on metadata annotation, a feature that was introduced in J2SE 5.0. An annotation consists of the @ sign preceding an annotation type, sometimes followed by a parenthetical list of element-value pairs. The EJB 3.0 specification defines a variety of annotation types. Some examples are those that specify a bean's type, such as

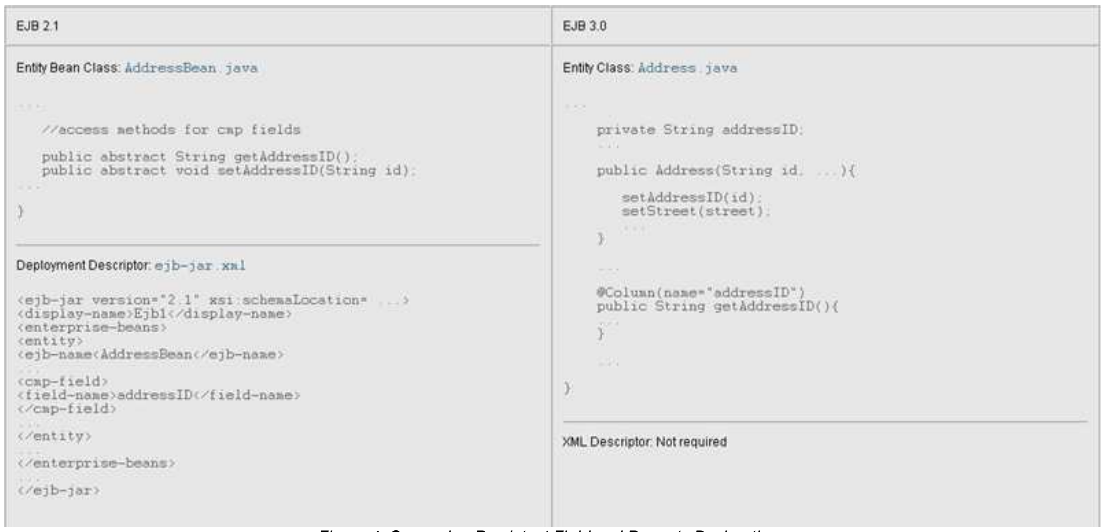
`@Stateless`; those that specify whether a bean is remotely or locally accessible, such as `@Remote` and `@Local`; transaction attributes, such as `@TransactionAttribute`; and security and method permissions, such as `@MethodPermissions`, `@Unchecked`, and `@SecurityRoles`. The Java Persistence API adds annotations, such as `@Entity`, that are specific to entities. The reliance on annotations in EJB 3.0 and the Java Persistence API demonstrates a significant design shift.

Defaults make things even easier

In many cases, the application can use defaults instead of explicit metadata annotation elements. In these cases, you don't have to completely specify a metadata annotation. You can obtain the same result as if you had fully specified the annotation. For example, the `@Entity` annotation has a `name` element that is used to specify the name to be used in queries that reference the entity. The `name` element defaults to the unqualified name of the entity class. So in the code for the `Address` entity, an annotation of `@Entity` is enough. There's no need to specify a name in the annotation. These defaults can make annotating entities very simple. In many cases, defaults are assumed when an annotation is not specified. In those cases, the defaults represent the most common specifications. For example, container-managed transaction demarcation -- in which 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. These defaults illustrate the coding-by-exception approach that guides EJB 3.0 technology. The process is simpler for the developer. You need to code only when the default is inadequate.

Persistent Fields and Properties

Let's now compare how persistence is declared for fields and properties.



EJB 2.1

Entity Bean Class: AddressBean.java

```
...
   //access methods for cmp fields

   public abstract String getAddressID();
   public abstract void setAddressID(String id);
...

}
```

Deployment Descriptor: ejb-jar.xml

```
<ejb-jar version="2.1" xsi:schemaLocation= ...>
<display-name>Ejb1</display-name>
<enterprise-beans>
<entity>
<ejb-name<AddressBean</ejb-name>
...
<cmp-field>
<field-name>addressID</field-name>
</cmp-field>
...
</entity>
...
</enterprise-beans>
...
</ejb-jar>
```

EJB 3.0

Entity Class: Address.java

```
...
    private String addressID;
    ...

    public Address(String id, ...){

       setAddressID(id);
       setStreet(street);
       ...
    }

    ...

    @Column(name="addressID")
    public String getAddressID(){
    ...
    }

    ...

}
```

XML Descriptor: Not required

Figure 4: Comparing Persistent Field and Property Declarations
Click here for a larger image

Persistence declarations are simpler

In EJB 2.1 technology, you specify which fields of the class need to be persisted in a database by defining public abstract getter and setter methods for those fields and by making specifications in a deployment descriptor -- an approach that many programmers find clumsy and far from intuitive. EJB 3.0 technology does not require these specifications. In essence, persistence is built into an entity. The persistent state of an entity is represented either by its persistent fields or persistent properties.

Recall that an entity is a POJO. Like a POJO, it can have nonabstract, private instance variables, such as the `AddressID` variable in the `Address` class. In the Java Persistence API, an entity can have field-based or property-based access. In field-based access, the persistence provider accesses the state of the entity directly through its instance variables. In property-based access, the persistence provider uses JavaBeans-style get/set accessor methods to access the entity's persistent properties. All fields and properties in the entity are persisted. You can, however, override a field or property's default persistence by marking it with the `@Transient`annotation or the Java keyword `transient`

No XML descriptor needed to declare persistent fields

EJB 2.1 technology, entity bean fields are identified as persistent fields in the bean's deployment descriptor, `ejb-jar.xml`, an often large and complex XML file. In addition to coding public accessor methods in the entity bean class, you must specify in the deployment descriptor a `cmp-field` element for each persistent field. In the Java Persistence API, you no longer need to provide a deployment descriptor, called an XML descriptor in the Java Persistence API, to specify an entity's persistent fields.

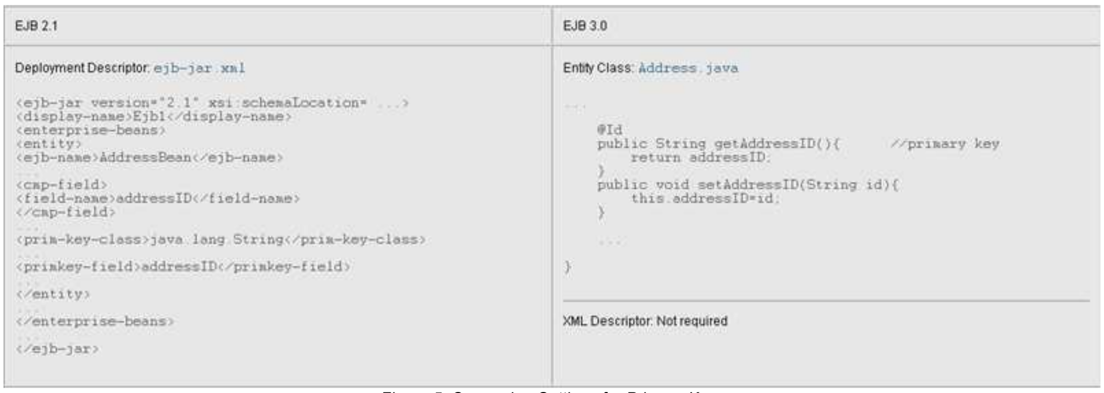
Default mappings are used where possible

In EJB 2.1 technology, you define the mapping between an entity bean's fields and their corresponding database columns in a vendor-specific deployment descriptor, such as `sun-ejb-jar.xml`. By contrast, the Java Persistence API does not require XML descriptors. Instead, you specify the mappings in the entity class by marking the appropriate persistent field or property accessor method with the `@Column` annotation. You can, however, take advantage of annotation defaults. If you don't specify an `@Column` annotation for a specific persistent field or property, a default mapping to a database column of the same name as the field or property name is assumed. Although you don't need to specify XML descriptors, you have the option of using them as an alternative to annotations or to supplement annotations. Using an XML descriptor might be useful in externalizing object-relational mapping information. Also, multiple XML descriptors can be useful in tailoring object-relational mapping information to different databases.

The Java Persistence API standardizes object-relational mapping, enabling fully portable applications.

Entity Identity

There are also simplifications in the way primary and composite keys for entities are specified.



EJB 2.1

Deployment Descriptor: ejb-jar.xml

```
<ejb-jar version="2.1" xsi:schemaLocation= ...>
<display-name>Ejb1</display-name>
<enterprise-beans>
<entity>
<ejb-name>AddressBean</ejb-name>
...
<cmp-field>
<field-name>addressID</field-name>
</cmp-field>
...
<prim-key-class>java.lang.String</prim-key-class>
...
<primkey-field>addressID</primkey-field>
...
</entity>
...
</enterprise-beans>
...
</ejb-jar>
```

EJB 3.0

Entity Class: Address.java

```
...
    @Id
    public String getAddressID(){      //primary key
        return addressID;
    }
    public void setAddressID(String id){
        this.addressID=id;
    }

    ...

}
```

XML Descriptor: Not required

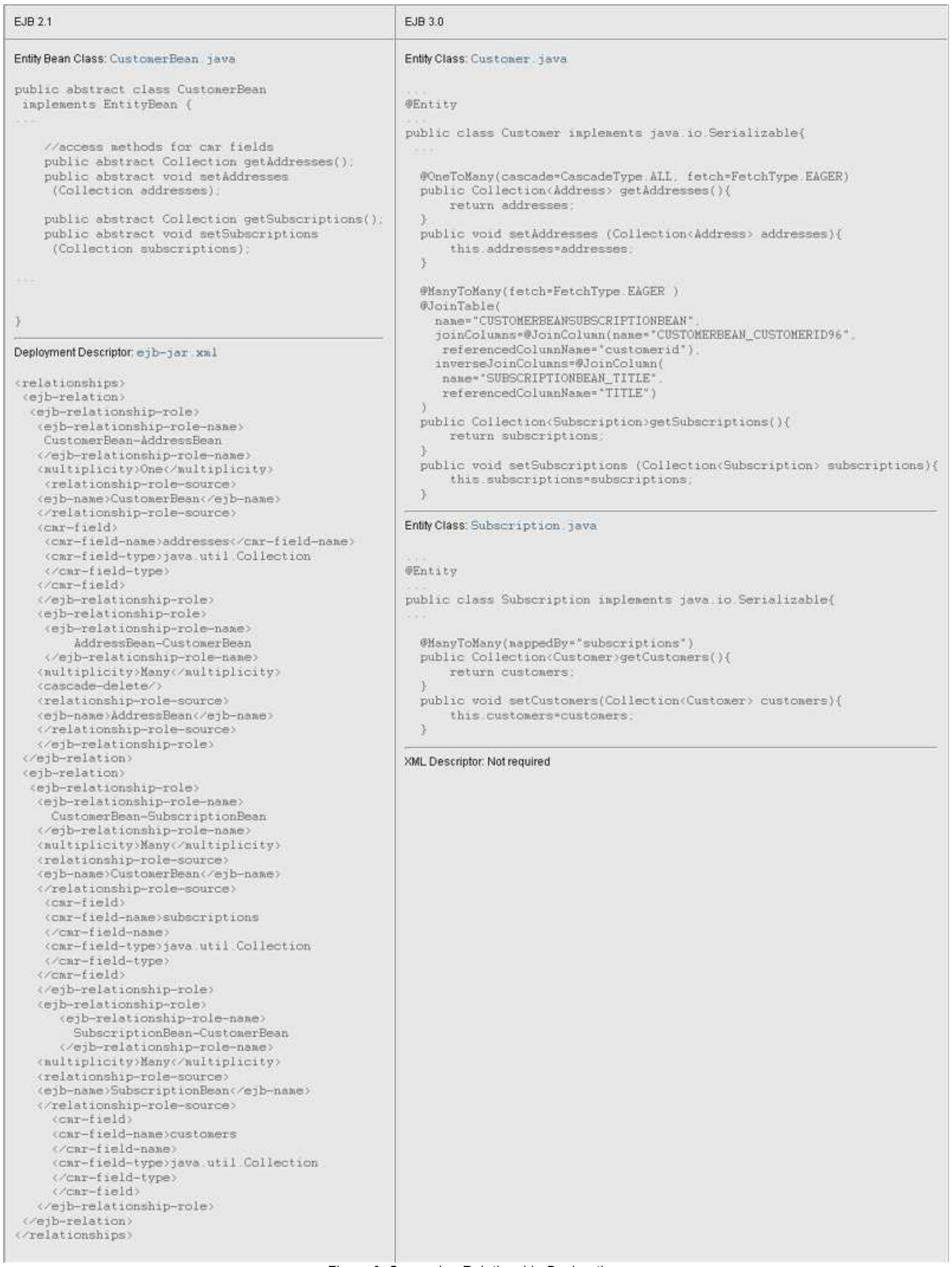
Figure 5: Comparing Settings for Primary Keys
Click here for a larger image

No XML descriptor needed to specify the primary key

In EJB 2.1 technology, the primary key for an entity bean -- that is, its unique identifier -- is specified not in the entity bean class but rather in its deployment descriptor. In the Java Persistence API, you don't need to provide an XML descriptor to specify an entity's primary key. Instead, you specify the primary key in the entity class by marking an appropriate persistent field or persistent property with the `@Id` annotation. You can specify composite keys in two different ways, using an `@IdClass` annotation or an `@EmbeddedId` annotation.

Relationships

So far, the focus has been on the coding simplifications that relate to basic persistence, that is, the persistence of entities and their fields or properties. Now, let's look at the coding simplifications that relate to entity relationships. To do that, let's focus on the `CustomerBean` class and compare it to the `CustomerBean` entity. The `CustomerBean` class has a one-to-many unidirectional relationship with `AddressBean` and a many-many bidirectional relationship with `SubscriptionBean`. Let's see how much easier it is to specify an equivalent set of relationships between the EJB 3.0 entities.



EJB 2.1

Entity Bean Class: CustomerBean.java

```
public abstract class CustomerBean
 implements EntityBean {
...

    //access methods for cmr fields
    public abstract Collection getAddresses();
    public abstract void setAddresses
     (Collection addresses);

    public abstract Collection getSubscriptions();
    public abstract void setSubscriptions
     (Collection subscriptions);


...


}
```

Deployment Descriptor: ejb-jar.xml

```
<relationships>
 <ejb-relation>
  <ejb-relationship-role>
   <ejb-relationship-role-name>
    CustomerBean-AddressBean
   </ejb-relationship-role-name>
   <multiplicity>One</multiplicity>
    <relationship-role-source>
   <ejb-name>CustomerBean</ejb-name>
   </relationship-role-source>
   <cmr-field>
    <cmr-field-name>addresses</cmr-field-name>
    <cmr-field-type>java.util.Collection
    </cmr-field-type>
   </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
    <ejb-relationship-role-name>
        AddressBean-CustomerBean
    </ejb-relationship-role-name>
   <multiplicity>Many</multiplicity>
   <cascade-delete/>
   <relationship-role-source>
   <ejb-name>AddressBean</ejb-name>
   </relationship-role-source>
   </ejb-relationship-role>
 </ejb-relation>
 <ejb-relation>
  <ejb-relationship-role>
   <ejb-relationship-role-name>
     CustomerBean-SubscriptionBean
   </ejb-relationship-role-name>
   <multiplicity>Many</multiplicity>
   <relationship-role-source>
   <ejb-name>CustomerBean</ejb-name>
   </relationship-role-source>
    <cmr-field>
    <cmr-field-name>subscriptions
    </cmr-field-name>
    <cmr-field-type>java.util.Collection
    </cmr-field-type>
   </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
      <ejb-relationship-role-name>
        SubscriptionBean-CustomerBean
      </ejb-relationship-role-name>
   <multiplicity>Many</multiplicity>
   <relationship-role-source>
   <ejb-name>SubscriptionBean</ejb-name>
   </relationship-role-source>
     <cmr-field>
     <cmr-field-name>customers
     </cmr-field-name>
     <cmr-field-type>java.util.Collection
     </cmr-field-type>
     </cmr-field>
   </ejb-relationship-role>
 </ejb-relation>
</relationships>
```

EJB 3.0

Entity Class: Customer.java

```
...
@Entity
...
public class Customer implements java.io.Serializable{
 ...

  @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
  public Collection<Address> getAddresses(){
      return addresses;
  }
  public void setAddresses (Collection<Address> addresses){
      this.addresses=addresses;
  }

  @ManyToMany(fetch=FetchType.EAGER )
  @JoinTable(
    name="CUSTOMERBEANSUBSCRIPTIONBEAN",
    joinColumns=@JoinColumn(name="CUSTOMERBEAN_CUSTOMERID96",
     referencedColumnName="customerid"),
    inverseJoinColumns=@JoinColumn(
     name="SUBSCRIPTIONBEAN_TITLE",
     referencedColumnName="TITLE")
  )
  public Collection<Subscription>getSubscriptions(){
      return subscriptions;
  }
  public void setSubscriptions (Collection<Subscription> subscriptions){
      this.subscriptions=subscriptions;
  }
```

Entity Class: Subscription.java

```
...
@Entity
...
public class Subscription implements java.io.Serializable{
...

  @ManyToMany(mappedBy="subscriptions")
  public Collection<Customer>getCustomers(){
      return customers;
  }
  public void setCustomers(Collection<Customer> customers){
      this.customers=customers;
  }
```

XML Descriptor: Not required

Figure 6: Comparing Relationship Declarations
Click here for a larger image

Relationship declarations are simpler

Specifying container-managed relationships in EJB 2.1 technology can be quite complex. If entity bean A has a relationship with entity bean B, you must specify abstract getter and setter methods in entity bean A for the related entity bean B. In addition, you must provide a rather lengthy entry for the relationship in the `ejb-jar.xml` deployment descriptor. In the Java Persistence API, you specify the relationship as you would for any POJO -- through a reference to the related entity object. In addition, you specify annotations that describe the semantics of the relationship and any database table-related metadata. You do not need an XML descriptor to specify entity relationships.

Note, however, that unlike in EJB 2.1 with container-managed persistence, the application that uses the Java Persistence API is responsible for managing relationships. For example, unlike EJB 2.1 technology, the Java Persistence API requires the backpointer reference in a bidirectional relationship to be set. Assume entity A has a bidirectional relationship to entity B. In EJB 2.1 technology, all you need to do is set the relationship from A to B -- the underlying persistence implementation is responsible for setting the backpointer reference from B to A. The Java Persistence API requires the references to be set on both sides of the relationship. This means that you have to explicitly call `b.setA(a)` and `a.setB(b)`.

Annotations specify multiplicity and related information

You declare the relationships in annotations that reference the related entities. Annotations on the persistent fields and properties specify the multiplicity of a relationship, such as one-to-many or many-to-many, and other information, such as cascade and eager fetching. (By comparison, in EJB 2.1 technology, this information is specified in vendor-specific settings.) For example, the `@OneToMany` annotation on the `getAddresses` accessor method in the `Customer` entity, specifies a one-to-many relationship with the `Address` entity.

The `Customer` entity is the owning side of the relationship. There is no associated annotation in the inverse side of the relationship. So this is a one-to-many unidirectional relationship with the `Address` entity. The value specified for the `cascade` element in the annotation specifies that life-cycle operations on the `Customer` entity must be cascaded to the `Address` entity, the target of the relationship. In other words, when a `Customer` instance is made persistent, any `Address` instances related to the `Customer` are also made persistent. It also means that if a `Customer` instance is deleted, the related `Address` instances are also deleted. The `FetchType` value specifies eager fetching -- this tells the container to prefetch the associated entities.

The owning side specifies the mapping

Notice that the many-to-many relationship between the `Customer` and `Subscription` entity classes is specified using a `@ManyToMany` annotation in both classes and a `@JoinTable` annotation in the `Customer` class. Either `Customer` or `Subscription` could have been specified as the owning class. In this example, `Customer` is the owning class, and so the `@JoinTable` annotation is specified in the that class. The `@ManyToMany` annotation in the `Subscription` class refers to the `Customer` class for mapping information through a `mappedBy` element. Because this example uses a join table for the relationship, the `@JoinTable` annotation specifies the foreign key columns of the join tables that map to the primary key columns of the related entities.

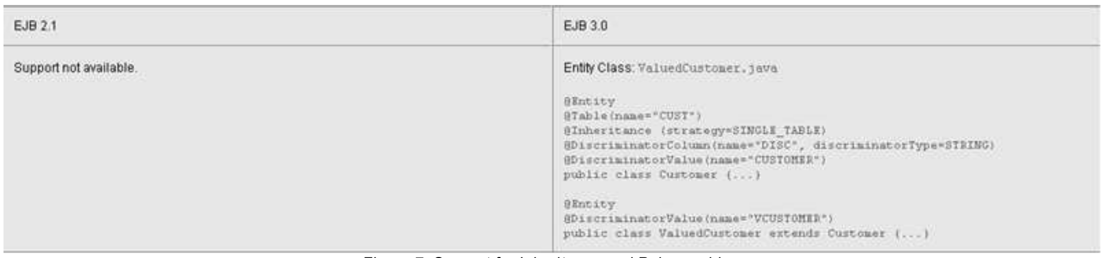
Default mappings are used for relationships where possible

You can take advantage of default mappings to simplify the coding for relationship mapping even further. If you look at the create.sql file under the `setup/sql` directory for the Java Persistence Demo, you will see a join table named `CUSTOMER_ADDRESS`. Notice that no annotations are needed in the `Customer` entity to specify the mapping of the `Customer` and `Address` entities to the columns in the `CUSTOMER_ADDRESS` table. That's because the table name and join column names are the defaults.

Inheritance and Polymorphism

An important capability of the Java Persistence API is its support for inheritance and polymorphism, something that was not available in EJB 2.1. You can map a hierarchy of entities, where one entity subclasses another, to a relational database structure, and submit queries against the base class. The queries are treated polymorphically against the entire hierarchy.

The code in the EJB 3.0 column below shows an example of the support for inheritance and polymorphism. This sample is not taken from the application code because the Java Persistence Demo does not use this feature. Here, `ValuedCustomer` is an entity that extends the `Customer` entity. The hierarchy is mapped to the `CUST` table:



EJB 2.1	EJB 3.0
Support not available.	Entity Class: ValuedCustomer.java

```
@Entity
@Table(name="CUST")
@Inheritance (strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="DISC", discriminatorType=STRING)
@DiscriminatorValue(name="CUSTOMER")
public class Customer {...}

@Entity
@DiscriminatorValue(name="VCUSTOMER")
public class ValuedCustomer extends Customer {...}
```

Figure 7: Support for Inheritance and Polymorphism
Click here for a larger image

Annotations specify inheritance

The `@Inheritance` annotation identifies a mapping strategy for an entity class hierarchy. In this example, the strategy specified by the value of the strategy element is `SINGLE_TABLE`. This means that the base class `Customer` and its subclass `ValuedCustomer` are mapped to a single table. By the way, you can also specify a strategy that joins the base class and its subclasses or that maps them to separate tables. A single table strategy requires a discriminator column in the table to distinguish between rows in the table for the base class and rows for the subclasses. The `@DiscriminatorColumn` annotation identifies the discriminator column, in this case, `DISC`. The `discriminatorType` element specifies that the discriminator column contains strings. The `@DiscriminatorValue` annotation is used to specify the value of the discriminator column for the associated entity. Here, `ValuedCustomer` instances are distinguished from other `Customer` instances by having a value of `VCUSTOMER` in the discriminator column.

Defaults can be used for various inheritance specifications

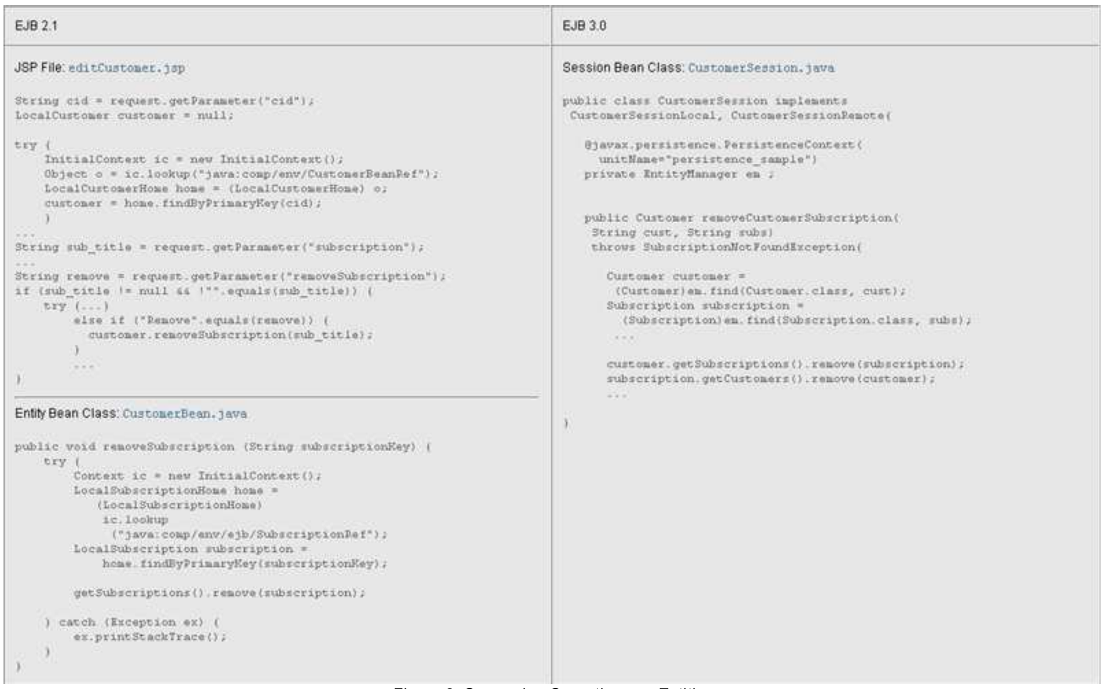
As is the case in many other parts of EJB 3.0 technology and the Java Persistence API, you can rely on defaults to further simplify coding. For example, `SINGLE_TABLE` is the default inheritance strategy, so the code sample does not need to specify it. In fact, you don't need to specify the `@Inheritance` annotation if you're using single-table inheritance. The default for the `discriminatorType` element in the `@DiscriminatorColumn` annotation is `STRING`, so the specification in the code sample is not necessary there either. Also, for string discriminator columns, you don't need to specify a discriminator value. The default is the entity name.

Entities can inherit from other entities and from non-entities

The EJB 3.0 example illustrates inheritance from an entity. However inheritance from POJOs that are not entities -- for behavior and mapping attributes -- is also allowed. The base entity or nonentity can either be abstract or concrete.

Operations on Entities

Another pronounced simplification in the Java Persistence API is the way entity operations are performed. For example, look at the way a subscription is removed in the `editCustomer.jsp` file and the EJB 2.1 `CustomerBean` compared to the way this is done in the business interface for the EJB 3.0 session bean, `CustomerSession`.



EJB 2.1

JSP File: editCustomer.jsp

```
String cid = request.getParameter("cid");
LocalCustomer customer = null;

try {
    InitialContext ic = new InitialContext();
    Object o = ic.lookup("java:comp/env/CustomerBeanRef");
    LocalCustomerHome home = (LocalCustomerHome) o;
    customer = home.findByPrimaryKey(cid);
    }
...
String sub_title = request.getParameter("subscription");
...
String remove = request.getParameter("removeSubscription");
if (sub_title != null && !"".equals(sub_title)) {
    try {...}
        else if ("Remove".equals(remove)) {
          customer.removeSubscription(sub_title);
        }
        ...
}
```

Entity Bean Class: CustomerBean.java

```
public void removeSubscription (String subscriptionKey) {
    try {
        Context ic = new InitialContext();
        LocalSubscriptionHome home =
           (LocalSubscriptionHome)
            ic.lookup
             ("java:comp/env/ejb/SubscriptionRef");
        LocalSubscription subscription =
            home.findByPrimaryKey(subscriptionKey);

        getSubscriptions().remove(subscription);

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

EJB 3.0

Session Bean Class: CustomerSession.java

```
public class CustomerSession implements
 CustomerSessionLocal, CustomerSessionRemote{

   @javax.persistence.PersistenceContext(
     unitName="persistence_sample")
   private EntityManager em ;


   public Customer removeCustomerSubscription(
    String cust, String subs)
    throws SubscriptionNotFoundException{

      Customer customer =
       (Customer)em.find(Customer.class, cust);
      Subscription subscription =
        (Subscription)em.find(Subscription.class, subs);
       ...

      customer.getSubscriptions().remove(subscription);
      subscription.getCustomers().remove(customer);
      ...

}
```

Figure 8: Comparing Operations on Entities
Click here for a larger image

Entity operations are performed directly on the entity

In EJB 2.1 technology, a client must use the Java Naming and Directory Interface (JNDI) to lookup the bean. In doing this, JNDI acquires an object that represents the bean's home interface. In response, the EJB container creates an instance of the bean and initializes its state. The client can then call methods on the JNDI-acquired object to perform operations on the bean. In the EJB 2.1 technology code sample above, JNDI is used to acquire an object that represents the `LocalSubscriptionHome` interface. In the `removeSubscription()` business method of the `CustomerBean`, the `findByPrimaryKey()` method finds the pertinent `SubscriptionBean` instance and removes the reference to it. This relationship update is synchronized to the database when the transaction in which this `CustomerBean`'s business method takes part commits. Many developers view this process as needlessly indirect and complex.

By comparison, the Java Persistence API requires no JNDI-based lookup. An `EntityManager` instance is used to find the pertinent `Customer` and `Subscription` entities. The reference to the `Subscription` instance is then removed from the `Customer`'s subscription list. Also, the reference to the `Customer` instance is removed from the `Subscription`'s customer list -- just as with POJOs.

An entity manager manages the entities

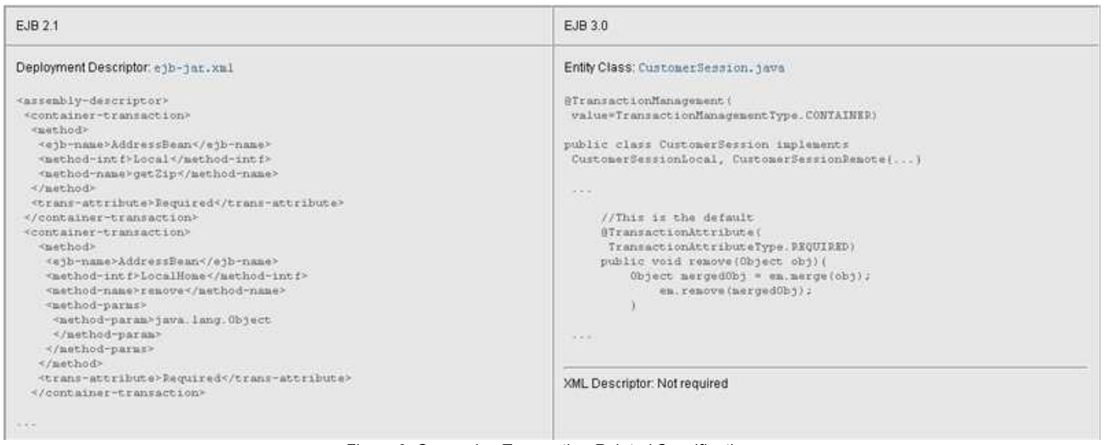
An `EntityManager` instance is used to manage the state and life cycle of entities within a persistence context. The entity manager is responsible for creating and removing persistent entity instances and finding entities by the entity's primary key. It also allows queries to be run on entities.

Dependencies can be injected

As illustrated in the EJB 3.0 technology code example in Figure 8, JNDI is no longer required to get references to resources and other objects in an enterprise bean's context. Instead, you can use resource and environment reference annotations in the bean class. These annotations inject a resource on which the enterprise bean has a dependency. In the EJB 3.0 technology code sample, the `@PersistenceContext` annotation injects an `EntityManager` with a transactional persistence context, on which the session bean has a dependency. The container then takes care of obtaining the reference to the needed resource and providing it to the bean. Dependency injection can dramatically simplify what you have to code to obtain resource and environmental references.

Transactions

Transaction-related specifications are also simplified.



EJB 2.1

Deployment Descriptor: ejb-jar.xml

```
<assembly-descriptor>
 <container-transaction>
  <method>
   <ejb-name>AddressBean</ejb-name>
   <method-intf>Local</method-intf>
   <method-name>getZip</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
 </container-transaction>
 <container-transaction>
   <method>
    <ejb-name>AddressBean</ejb-name>
    <method-intf>LocalHome</method-intf>
    <method-name>remove</method-name>
    <method-parms>
     <method-param>java.lang.Object
     </method-param>
    </method-parms>
   </method>
   <trans-attribute>Required</trans-attribute>
  </container-transaction>

...
```

EJB 3.0

Entity Class: CustomerSession.java

```
@TransactionManagement(
 value=TransactionManagementType.CONTAINER)

public class CustomerSession implements
 CustomerSessionLocal, CustomerSessionRemote{...}

 ...

     //This is the default
     @TransactionAttribute(
      TransactionAttributeType.REQUIRED)
     public void remove(Object obj){
         Object mergedObj = em.merge(obj);
             em.remove(mergedObj);
         }

 ...
```

XML Descriptor: Not required

Figure 9: Comparing Transaction-Related Specifications
Click here for a larger image

No XML descriptor needed to specify transaction attributes

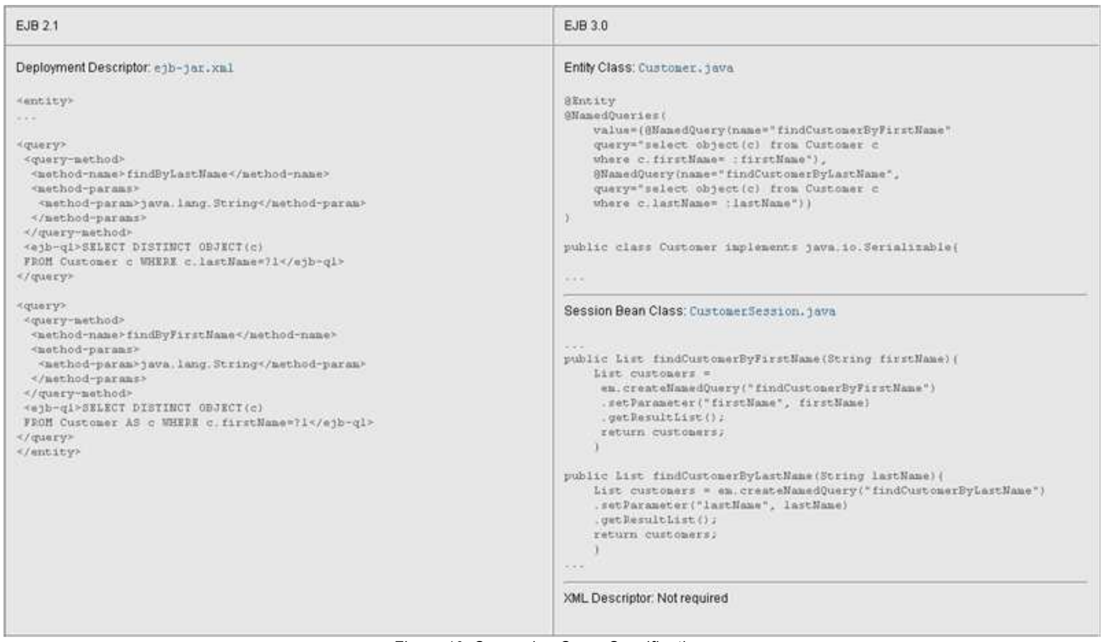
In EJB 2.1 technology, you specify the transaction attributes for container-managed transactions in an often lengthy and complex deployment descriptor. In EJB 3.0 technology, an XML descriptor is not needed to specify transaction attributes. Instead, you can use the `@TransactionManagement` annotation to specify container-managed transactions, as well as to specify bean-managed transactions, and the `@TransactionAttribute` annotation to specify transaction attributes. In the EJB 3.0 technology code sample, the `@TransactionManagement` annotation specifies container-managed transactions for the session bean. Because container-managed transactions are the default type of transaction demarcation, the annotation is not necessary here. The `@TransactionAttribute` annotation on the `remove()` method specifies a transaction attribute of `REQUIRED`, which is the default transaction type. So this annotation is not necessary either. You would need the annotation, however, to specify another transaction type, such as "Mandatory" or "Supports".

Container-managed entity managers are Java Transaction API entity managers

In the Java Persistence API, transactions involving `EntityManager` operations can be controlled in two ways, either through JTA or by the application through the `EntityTransaction` API. An entity manager whose transactions are controlled through JTA is called a JTA entity manager. An entity manager whose transactions are controlled by the `EntityTransaction` API is called a resource-local entity manager. A container-managed entity manager must be a JTA entity manager. JTA entity manager transactions are started and ended outside of the entity manager, and the entity manager methods share the transaction context of the session bean methods that invoke them.

Queries

Support for queries has been significantly enhanced in the Java Persistence API. Some of these enhancements are shown in Figure 10.



EJB 2.1	EJB 3.0
Deployment Descriptor: `ejb-jar.xml`	Entity Class: `Customer.java`

```
<entity>
...

<query>
 <query-method>
  <method-name>findByLastName</method-name>
  <method-params>
   <method-param>java.lang.String</method-param>
  </method-params>
 </query-method>
 <ejb-ql>SELECT DISTINCT OBJECT(c)
 FROM Customer c WHERE c.lastName=?1</ejb-ql>
</query>

<query>
 <query-method>
  <method-name>findByFirstName</method-name>
  <method-params>
   <method-param>java.lang.String</method-param>
  </method-params>
 </query-method>
 <ejb-ql>SELECT DISTINCT OBJECT(c)
 FROM Customer AS c WHERE c.firstName=?1</ejb-ql>
</query>
</entity>
```

```
@Entity
@NamedQueries(
    value={@NamedQuery(name="findCustomerByFirstName"
    query="select object(c) from Customer c
    where c.firstName= :firstName"),
    @NamedQuery(name="findCustomerByLastName",
    query="select object(c) from Customer c
    where c.lastName= :lastName")}
)

public class Customer implements java.io.Serializable{

...
```

Session Bean Class: `CustomerSession.java`

```
...
public List findCustomerByFirstName(String firstName){
    List customers =
     em.createNamedQuery("findCustomerByFirstName")
      .setParameter("firstName", firstName)
      .getResultList();
     return customers;
    }

public List findCustomerByLastName(String lastName){
    List customers = em.createNamedQuery("findCustomerByLastName")
    .setParameter("lastName", lastName)
    .getResultList();
    return customers;
    }
...
```

XML Descriptor: Not required

Figure 10: Comparing Query Specifications
Click here for a larger image

No XML descriptor needed to specify queries

In EJB 2.1 technology, you define a query for an entity bean using Enterprise JavaBeans query language (EJB QL). You specify the query in the deployment descriptor for the bean, and associate it there with a finder or select method for the bean. In the Java Persistence API, you can define a named -- or static -- query in the bean class itself. You also have the option of creating dynamic queries. To create a named query, you first define the named query using the `@NamedQuery` annotation. Then you create the previously-defined query using the `createNamedQuery` method of the `EntityManager`. In the EJB 3.0 technology example, two named queries are defined in the `Customer` entity class: `findCustomerByFirstName` and `findCustomerByLastName`. The named queries are created in the session bean, `CustomerSession`, which provides the client code for the entity. To create a dynamic query, you use the `createQuery` method of the `EntityManager`. The Java Persistence API provides a Java Persistence query language that extends EJB QL. You can use Java Persistence query language or native SQL in named or dynamic queries.

Support for dynamic queries and named queries is added

The Java Persistence API provides a Query API to create dynamic queries and named queries. For example, the `CustomerSession` class uses the Query API on the entity manager to create the named queries `findCustomerByFirstName` and `findCustomerByLastName`. The Query method `setParameter` binds an argument to a named parameter. (By the way, support for named parameters is also a new feature in the Java Persistence API: both named queries and dynamic queries can use named parameters as well as positional parameters, although a single query cannot mix both types of parameters.) For example, the `setParameter` method in `findCustomerByFirstName` binds the `firstname` argument to the named parameter `:firstName` in the named query definition. The `getResultList`method returns the query results.

All queries that use the Java Persistence API are polymorphic

This means that when a class is queried, all subclasses that meet the query criteria are also returned.

The Java Persistence Query Language is an enhanced query language

EJB QL has been a very popular facet of EJB technology. However, despite its popularity, EJB QL has lacked some of the features of a full structured query language, such as bulk update and delete operations, outer join operations, projection, and subqueries. The Java Persistence query language adds those features. It also adds support for outer join-based prefetching. The full range of the Java Persistence query language can be used in static or dynamic queries. However the Java Persistence Demo does not use the Java Persistence query language enhancements.

Testing Entities Outside of the Container

Although this can't be demonstrated in a side-by side code comparison, testing entities outside an EJB container is now much easier. Previously, the entity bean component model -- with its requirements for home and component interfaces, abstract entity bean classes, and virtual persistent fields -- made it difficult to test entity beans outside of the container. The Java Persistence API removes the requirement for these interfaces. The only thing required for an entity bean is a concrete entity bean class -- a POJO -- that has persistent fields or persistent properties. In addition, an entity's life cycle is controlled through the entity manager, not through a home interface whose life-cycle methods are implemented by the EJB container. Because all of this makes entities less dependent on intervention by the EJB container, the entities can be more easily tested outside of the container.

Summary

The aim of the new Java Persistence API is to simplify the development of persistent entities. It meets this objective through a simple POJO-based persistence model, which reduces the number of required classes and interfaces. You model your data using POJOs, and then annotate them to tell the container about an entity's characteristics and the resources it needs. You also use annotations for object-relational mappings and entity relationships, as well as deploy-time instructions. This annotation-based approach removes the need for often long and complex, XML-based descriptors. In many cases, the annotations' defaults are enough. You code specific attributes of the annotations only when the defaults are inadequate.

Beyond these simplifications, the Java Persistence API adds capabilities that were not in EJB 2.1 technology, giving you additional power and flexibility in developing and using persistent entities. You can take advantage of query language enhancements and new features such as inheritance and polymorphism to perform more powerful and encompassing queries. You can exercise more control over queries, and perform query optimizations specific to your needs. In short, using the Java Persistence API is much simpler and more intuitive than it's predecessors, yet it offers a more robust API for creating, managing, and using persistent entities.

This article only highlights the simplifications and enhancements that the Java Persistence API offers. You'll find more extensive information about the API, as well as other simplifications made in EJB 3.0 technology, by examining the Enterprise JavaBeans 3.0 Specification. Now is a good time to try out EJB 3.0 technology and the Java Persistence API. See the Try It Out! box at the beginning of this article for some good places to start.

For More Information

EJB 3.0 Specification (JSR-220)

Enterprise JavaBeans Technology

Introduction to the Java EE 5 Platform

Ease of Development in Enterprise JavaBeans Technology

The Java EE 5 Tutorial

Annotations

Java EE 5 SDK Preview

Preview: NetBeans IDE 5.5 with NetBeans Enterprise Pack 5.5

Project GlassFish

About the Authors

Rahul Biswas is a member of the Java Performance Engineering group at Sun. He is currently involved in the development of a generic performance benchmark for Java Persistence and the performance improvement of the persistence implementation in GlassFish.

Ed Ort is a is a staff member of java.sun.com. He has written extensively about relational database technology, programming languages, and web services.

```
                    </td>
                  </tr>
                   <tr>
```