JPA And Entity Beans

## JPA

Persistence is the ability to have data contained in Java objects automatically stored into a relational database like Oracle, SQL Server, and DB2. Persistence in EJB 3 is managed by the JPA. It automatically persists the Java objects using a technique called object-relational mapping (ORM). ORM is essentially the process of mapping data held in Java objects to database tables using configuration. It relieves you of the task of writing low-level, boring, and complex JDBC code to persist objects into a database.

In EJB 3 terms, a persistence provider is essentially an ORM framework that supports the EJB 3 Java Persistence API (JPA). The JPA defines a standard for

-The creation of ORM configuration metadata for mapping entities to relational tables

-The EntityManager API standard API for performing CRUD (create, read, update, and delete)/persistence operations for entities

-The Java Persistence Query Language (JPQL), for searching and retrieving persisted application data

### Entity Bean

If you’re using JPA to build persistence logic of your applications, then you have to use entities. Entities are the Java objects that are persisted into the database. Just as session beans model processes, entities model lower-level application concepts that high-level business processes manipulate.

While session beans are the “verbs” of a system, entities are the “nouns.” Examples include an Employee entity, a User entity, an Item entity, and so on. Here’s another perfectly valid (and often simpler to-understand) way of looking at entities: they are the OO representations of the application data stored in the database. In this sense, entities survive container crashes and shutdown. You must be wondering how the persistence provider knows where the entity will be stored. The real magic lies in the ORM metadata; an entity contains the data that specifies how it is mapped to the database.

### The EntityManager

The JPA EntityManager interface manages entities in terms of actually providing persistence services. While entities tell a JPA provider how they map to the database, they do not persist themselves. The EntityManager interface reads the ORM metadata for an entity and performs persistence operations. The Entity-Manager knows how to add entities to the database, update stored entities, and delete and retrieve entities from the database. In addition, the JPA provides the ability to handle lifecycle management, performance tuning, caching, and transaction management.

## Containers

In the Java world, containers aren’t just limited to the realm of EJB 3. You’re probably familiar with a web container, which allows you to run web-based applications using Java technologies such as servlets, JSP, or JSF. A Java EE container is an application server solution that supports EJB 3, a web container, and other Java EE APIs and services. BEA WebLogic Server, Sun Microsystems’s GlassFish, IBM WebSphere, JBoss Application Server, and Oracle Application Server 10g are examples of Java EE containers. The relationship between the Java EE container, web container, EJB container, and JPA persistence provider is shown in figure 1.7.

If you install a Java EE compliant application server such as GlassFish, it will contain a preconfigured web container, EJB container, and a JPA provider. However, some vendors and open source projects may provide only a web container such as Tomcat or an EJB 3¨Ccompliant persistence provider such as Hibernate. These containers provide limited functionality compared to what you get with a complete Java EE 5 container.

**Figure 1.7** *Java EE container typically contains web and EJB containers and a persistence provider. The stateless session bean (Credit Check EJB) and stateful session bean (Cart EJB) are deployed and run in the EJB container. Entities (Customer and Catalog) are deployed and run within an EJB persistence provider and can be accessed by either web or EJB container components.A description...*

**Table 1.1** Major EJB 3 component services and why they are important to you. The persistence services are provided by the JPA provider.

A description...

A description...

**Table2.1**

*Major metadata annotations introduced in Java EE. Although primarily geared toward EJB, these annotations apply to Java EE components such as servlets and JSF managed beans as well as application clients. Annotations defined in the javax.annotation.\* package are defined by the Common Metadata Annotations API (JSR-250).*

A description...

**Interceptors:**

Interceptors are objects that are automatically triggered when an EJB method is invoked.

Interception takes place at various points (called point cuts) including at the beginning of a method, at the end of a method, and when an exception is triggered.

Interceptors can be applied to both session and message-driven beans.

Eg:

@Stateless

public class PlaceBidBean implements PlaceBid {

...

@Interceptors(ActionBazaarLogger.class)

public void addBid(Bid bid) {

...

}

}

-------

public class ActionBazaarLogger {

@AroundInvoke

public Object logMethodEntry(

InvocationContext invocationContext)

throws Exception {

System.out.println("Entering method: ”

+ invocationContext.getMethod().getName());

return invocationContext.proceed();

}

}

Description of the above code:

The interceptor class, ActionBazaarLogger, is attached to the addBid method of the PlaceBid stateless session bean using the @javax.interceptor.Interceptors annotation. The ActionBazaarLogger object’s log-MethodEntry method is annotated with @javax.interceptor.AroundInvoke and will be invoked when the addBid method is called. The logMethodEntry method prints a log message to the system console, including the method name entered using the javax.interceptor.InvocationContext. Finally, the invocation context’s

proceed method is invoked to signal to the container that the addBid invocation can proceed normally.

## Entity Bean Example:

### Entity Bean:

package ejb3inaction.example.persistence;

import java.io.Serializable;

import java.sql.Date;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

import javax.persistence.GenerationType;

import javax.persistence.GeneratedValue;

@Entity

@Table(name="BIDS")

public class Bid implements Serializable {

private Long bidID;

private Long itemID;

private Long bidderID;

private Double bidAmount;

private Date bidDate;

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

@Column(name="BID\_ID")

public Long getBidID() {

return bidID;

}

public void setBidID(Long bidID) {

this.bidID = bidID;

}

@Column(name="ITEM\_ID")

public Long getItemID() {

return itemID;

}

public void setItemID(Long itemID) {

this.itemID = itemID;

}

@Column(name="BIDDER\_ID")

public Long getBidderID() {

return bidderID;

}

public void setBidderID(Long bidderID) {

this.bidderID = bidderID;

}

@Column(name="BID\_AMOUNT")

public Double getBidAmount() {

return bidAmount;

}

public void setBidAmount(Double bidAmount) {

this.bidAmount = bidAmount;

}

@Column(name="BID\_DATE")

public Date getBidDate() {

return bidDate;

}

public void setBidDate(Date bidDate) {

this.bidDate = bidDate;

}

}

### Description

1)The @Entity annotation signifies the fact that the Bid class is a JPA entity.

Note that Bid is a POJO that does not require a business interface, unlike session

and message-driven beans.

2)The @Table annotation tells JPA that the Bid entity is mapped to the BIDS table.

3)The @Column annotations indicatewhich Bid properties map to which BIDS table fields.

Note that entities need not use getter- and setter-based properties. Instead, the field mappings could have been placed directly onto member variables exposed through nonprivate access modifiers.

### Using Entity bean:

package ejb3inaction.example.buslogic;

...

import javax.persistence.PersistenceContext;

import javax.persistence.EntityManager;

...

@Stateless

public class PlaceBidBean implements PlaceBid {

@PersistenceContext(unitName="actionBazaar")

private EntityManager entityManager;

...

public Bid addBid(Bid bid) {

System.out.println("Adding bid, bidder ID=" + bid.getBidderID()+ ", item ID=" + bid.getItemID() + ", bid amount="+ bid.getBidAmount() + ".");

return save(bid);

}

private Bid save(Bid bid) {

entityManager.persist(bid);

return bid;

}

}

Important Points :

-Deployment descriptor entries override configuration values hard-coded into EJB components

- EJB should always implement interface.

Dependeny Injection:

EJB 3 DI using the @EJB annotation reduces all this mechanical JNDI lookup code to a single statement!

In a nontrivial application, this can easily translate to eliminating hundreds of lines of redundant, boring, error-prone code. You can think of EJB 3 DI as a high-level abstraction over JNDI lookups.

**Field- vs. property-based persistence**

An entity maintains its state by using either fields or properties (via setter and getter methods).The variables in the @Entity annotated class, can be persisted either using the getter/setter methods(property-based persistence) or by directly accessing the variables(field based persistence).@Id annotation indicates that how to persist the data.

If the @Id annotation is on a varibale then directly persist the variable.

If the @Id is on the getter method then getter/ steer methods will be used to persist the data.

Annotations used with a setter method are ignored by the persistence provider for property-based access.Persisted setters and getters cannot be declared final, as the persistence provider may need to override them.

**@Transient**

If necessary, you can stop an entity property from being persisted by marking the getter with the @Transient annotation or marking the field with @Transient annotation.

**The @Id annotation**

Using the javax.persistence.Id annotations is the simplest way of telling the persistence

provider where the entity identity is stored. The @Id annotation marks a field or property as identity for an entity. This marks the field as the primary-key. EJB 3 supports primitives,

primitive wrappers, and Serializable types like java.lang.String, java.util.Date, and java.sql.Date as identities. In addition, when choosing numeric types you should avoid types such as float, Float, double, and so forth because of the indeterminate nature of type precision. Another type you should avoid choosing as identifier is TimeStamp.

Assume that we changed our minds and decided that a Category is uniquely identified by its name and creation date. There are two ways we can accomplish this: by using either the @IdClass or @EmbeddedId annotation.

**@Table**

@Table specifies the table containing the columns to which the entity is mapped.

In sample code shown below, the @Table annotation makes the USERS table’s columns available for ORM.The @Table annotation itself is optional. If it’s omitted, the entity is assumed to be mapped to a table in the default schema with the same name as the entity class. If

the name parameter is omitted, the table name is assumed to be the same as the

name of the entity.

@Entity

@Table(name="USERS")

@SecondaryTable(name="USER\_PICTURES",

pkJoinColumns=@PrimaryKeyJoinColumn(name="USER\_ID"))

public class User implements Serializable {

@Id

@Column(name="USER\_ID", nullable=false)

protected Long userId;

@Column(name="USER\_NAME", nullable=false)

protected String username;

@Column(name="FIRST\_NAME", nullable=false, length=1)

protected String firstName;

@Column(name="LAST\_NAME", nullable=false)

protected String lastName;

@Enumerated(EnumType.ORDINAL)

@Column(name="USER\_TYPE", nullable=false)

protected UserType userType;

@Column(name="PICTURE", table="USER\_PICTURES")

@Lob

@Basic(fetch=FetchType.LAZY)

protected byte[] picture;

@Column(name="CREATION\_DATE", nullable=false)

@Temporal(TemporalType.DATE)

protected Date creationDate;

@Embedded

protected Address address;

public User() {}

}

@Embeddable

public class Address implements Serializable {

@Column(name="STREET", nullable=false)

protected String street;

@Column(name="CITY", nullable=false)

protected String city;

@Column(name="STATE", nullable=false)

protected String state;

@Column(name="ZIP\_CODE", nullable=false)

protected String zipCode;

@Column(name="COUNTRY", nullable=false)

protected String country;

}

**@Enumerated**

This effectively means that any data type defined as UserType (like our persistent

field in the User object) can only have the four values listed in the enumeration.

Can be of 2 types:

1)EnumType.ORDINAL : This means that if the value of the field is set to UserType.SELLER, the value 0 will be stored into the database.

Eg :

@Enumerated(EnumType.ORDINAL)

...

protected UserType userType;

2)EnumType.STRING : In this case a UserType.ADMIN value would be saved into the database as "ADMIN". By default an enumerated field or property is saved as an ordinal. This

Eg:

@Enumerated(EnumType.STRING)

...

protected UserType userType;

**@GeneratedValue:**

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

@Column(name="USER\_ID")

protected Long userId;

This code assumes that an identity constraint exists on the USERS.USER\_ID column. Note that when using IDENTITY as the generator type, the value for the identity field may not be available before the entity data is saved in the database because typically it is generated when a record is committed. The two other strategies, SEQUENCE and TABLE, both require the use of an externally defined generator: a SequenceGenerator or TableGenerator must be created

and set for the GeneratedValue. You’ll see how this works by first taking a look at the sequence generation strategy.

**@JoinColumn**

If the underlying table for the referencing entity is the one containing the foreign

key to the table to which the referenced “child” entity is mapped, you can map the

relationship using the @JoinColumn annotation.

In our User-BillingInfo example shown below, the USERS table contains a foreign key named USER\_BILLING\_ID that refers to the BILLING\_INFO table’s BILLING\_ID primary key.

@Entity

@Table(name="USERS")

public class User {

@Id

@Column(name="USER\_ID")

protected String userId;

...

@OneToOne

@JoinColumn(name="USER\_BILLING\_ID",

referencedColumnName="BILLING\_ID", updatable=false)

protected BillingInfo billingInfo;

}

@Entity

@Table(name="BILLING\_INFO")

public class BillingInfo {

@Id

@Column(name="BILLING\_ID")

protected Long billingId;

...

}

**Entity-Manager**

**A description...**

The EntityManager acts as a bridge between the OO and relational worlds. It interprets the O/R mapping specified for an entity and saves the entity in the database.

Besides providing these explicit SQL-like CRUD operations, the EntityManager also quietly tries to keep entities synched with the database automatically as long as they are within the EntityManager’s reach.

When we talk about managing an entity’s state, what we mean is that the Entity-Manager makes sure that the entity’s data is synchronized with the database. The EntityManager ensures this by doing two things. First, as soon as we ask an Entity-Manager to start managing an entity, it synchronizes the entity’s state with the database. Second, until the entity is no longer managed, the EntityManager ensures that changes to the entity’s data (caused by entity method invocations, for example) are reflected in the database. The EntityManager accomplishes this feat by holding an object reference to the managed entity and periodically checking for data freshness. If the EntityManager finds that any of the entity’s data has changed, it automatically synchronizes the changes with the database. The EntityManager stops managing the entity when the entity is either deleted or moves out of persistence provider’s reach. An entity can become attached to the EntityManager’s context when you pass the entity to the persist, merge, or refresh method. Also an entity becomes attached when you retrieve using the find method or a query within a transaction.

A managed entity becomes detached when it is out of scope, removed, serialized, or cloned.

A description...

**Example of using Container managed Entity-Manager**

@Stateless

public class ItemManagerBean implements ItemManager {

@PersistenceContext(unitName="actionBazaar")

private EntityManager entityManager;

public ItemManagerBean() {}

public Item addItem(String title, String description,

byte[] picture, double initialPrice, long sellerId) {

Item item = new Item();

item.setTitle(title);

item.setDescription(description);

item.setPicture(picture);

item.setInitialPrice(initialPrice);

Seller seller = entityManager.find(Seller.class, sellerId);

item.setSeller(seller);

entityManager.persist(item);

return item;

}

public Item updateItem(Item item) {

entityManager.merge(item);

return item;

}

public Item undoItemChanges(Item item) {

entityManager.refresh(entityManager.merge(item));

return item;

}

public void deleteItem(Item item) {

entityManager.remove(entityManager.merge(item));

}

}

**Application-managed EntityManager**

An application-managed EntityManager wants nothing to do with a Java EE container.

This means that you must write code to control every aspect of the Entity-

Manager’s lifecycle. By and large, application-managed EntityManagers are most

appropriate for environments where a container is not available, such as Java SE

or a lightweight web container like Tomcat.

However, a justification to use application-managed EntityManagers inside a

Java EE container is to maintain fine-grained control over the EntityManager lifecycle

or transaction management.

@Stateless

public class ItemManagerBean implements ItemManager {

@PersistenceUnit

private EntityManagerFactory entityManagerFactory;

private EntityManager entityManager;

public ItemManagerBean() {}

@PostConstruct

public void initialize() {

entityManager = entityManagerFactory.createEntityManager();

}

...

public Item updateItem(Item item) {

entityManager.joinTransaction();

entityManager.merge(item);

return item;

}

...

@PreDestroy

public void cleanup() {

if (entityManager.isOpen()) {

entityManager.close();

}

}

...

}

**Application-managed EntityManagers outside the Java EE container**

EntityManagerFactory entityManagerFactory =

Persistence.createEntityManagerFactory("actionBazaar");

EntityManager entityManager =

entityManagerFactory.createEntityManager();

try

{

EntityTransaction entityTransaction =

entityManager.getTransaction();

entityTransaction.begin();

entityManager.persist(item);

entityTransaction.commit();

}

finally

{

entityManager.close();

entityManagerFactory.close();

}

With JPA, you can use the following methods to retrieve entities and related data:

■ EntityManager.find with the entity’s primary key : entityManager.find(Seller.class, sellerId);

■ Queries written in JPQL

■ SQL queries native to the underlying database

Note : you will likely choose to work with JPQL rather than SQL, since a SQL query returns database records and a JPA query returns entities.

**Types of queries**

The query API supports two types of queries: named and dynamic. Named and dynamic queries have different purposes. Named queries are intended to be stored and reused. Dynamic (or ad hoc) queries are different from named queries in that they are created on the fly.

Example of dynamic query to retrieve all the categories in the system:

@PersistenceContext em;

...

public List findAllCategories() {

Query query = em.createQuery("SELECT c FROM Category c"); ...

return query.getResultList(); ............

}

Example of NamedQuery :

Say you want to create a named query on the Category entity to retrieve all categories by

passing a category name. To achieve this, use the @javax.persistence.Named-Query annotation:

@Entity

@NamedQuery(

name = "findAllCategories",

query = "SELECT c FROM Category c WHERE c.categoryName

LIKE :categoryName ")

public class Category implements Serializable {

…...

Query query = em.createNamedQuery("findAllCategories");

..

}

For a complex application, you’ll probably have multiple named queries. In that case, you can use the @javax.persistence.NamedQueries annotation to specify multiple named queries like this:

@Entity

@NamedQueries({

@NamedQuery(

name = "findCategoryByName",

query = "SELECT c FROM Category c WHERE c.categoryName

LIKE :categoryName order by c.categoryId"

),

@NamedQuery(

name = "findCategoryByUser",

query = "SELECT c FROM Category c JOIN c.user u

WHERE u.userId = ?1"

)})

@Table(name = "CATEGORIES")

public class Category implements Serializable {

}

note : named queries are globally scoped. You can create a named query instance from any component that has access to the persistence unit to which the entity belongs.

Let’s recap where we are now. We’ve created an instance of the EntityManager,

and we’ve created an instance of the query. The next step is the actual execution

of the query. The Query interface provides the methods we need.

query = em.createNamedQuery("findCategoryByName");

query.setParameter("categoryName", categoryName);

query.setMaxResults(10);

query.setFirstResult(3);

List categories = query.getResultList();

In this above example, we create a query instance from a named query that was defined

earlier. Here we want to retrieve a List of Category entities by name and hence we

set the parameter. We limit the maximum number of items returned to 10, and

we position the first entity to be returned at the third item. Finally we retrieve the

result

**Setting parameters for a query**

The number of entities retrieved in a query can be limited by specifying a WHERE

clause. If we want to retrieve all instances of the Item entity with a specific price,

the JPQL would look like this:

SELECT i FROM Item i WHERE i.initialPrice = ?1

In this statement, we’ve used a parameter (?1) for the WHERE clause. There are

two ways we can specify this parameter: by number or by name. When we have

an integer in the parameter, we call it a positional (or numbered) parameter.

Before we execute a query, we have to set the parameter for the query:

query.setParameter(1, 100.00);

In some cases you’ll want to specify multiple parameters for a query. Say you want

to retrieve all items with an initialPrice that falls within a particular range:

SELECT i FROM Item i WHERE i.initialPrice > ?1 AND i.initialPrice < ?2

The following code should do the trick:

query.setParameter(1, 100.00);

query.setParameter(2, 200.00);

When you specify a specific name for a parameter, it’s called a named parameter.

Eg :

SELECT i FROM Item i WHERE i.initialPrice = :price

query.setParameter("price", 100.00);

As you can see, the only difference between the positional parameter and the

named parameter is the notation of the parameter itself. A positional parameter

starts with a ? followed by the parameter’s position. A named parameter starts

with : and is followed by the name of the parameter.

To retrieve a single entity :

Category cat = (Category)query.getSingleResult();

To retrieve a collection of entities :

Category cat = (Category)query.getSingleResult();

**JPQL**

JPQL operates on classes and objects (entities) in the Java space. SQL operates on tables, columns, and rows in the database space. While JPQL and SQL look similar to us humans, they operate in two very different worlds.

The JPQL Query Parser or Processor Engine of a persistence provider, as shown in figure below, translates the JPQL query into native SQL for the database being used by the persistence provider.

A description...

Defining and using SELECT

Suppose we get jump-started with a simple JPQL query:

SELECT c

FROM Category c

WHERE c.categoryName LIKE :categoryName

ORDER BY c.categoryId

This JPQL query has (or can have) the following:

■ A SELECT clause that specifies the object type or entity or values being

retrieved

■ A FROM clause that specifies an entity declaration that is used by other

clauses

■ An optional WHERE clause to filter the results returned by the query

■ An optional ORDER BY clause to order the results retrieved by the query

■ An optional GROUP BY clause to perform aggregation

■ An optional HAVING clause to perform filtering in conjunction with

aggregation

Using UPDATE

Only one entity type can be specified with an UPDATE statement, and we should

provide a WHERE clause to limit the number of entities affected by the statement.

Here is the syntax for the UPDATE statement:

UPDATE entityName indentifierVariable

SET single\_value\_path\_expression1 = value1, ...

single\_value\_path\_expressionN = valueN

WHERE where\_clause

You can use any persistence field and single value association field in the SET

clause of the UPDATE statement. Assume that we want to provide Gold status and a

commissionRate of 10 percent to all Sellers whose lastName starts with Packrat.

Start with the following JPQL statement:

UPDATE Seller s

SET s.status = 'G', s.commissionRate = 10

WHERE s.lastName like 'PackRat%'

*Using DELETE*

Like UPDATE, DELETE in JPQL resembles its SQL cousin. You can specify only one

entity type with a DELETE statement, and again you should specify a WHERE clause

to limit the number of entities affected by the statement. Here is the syntax for

the DELETE statement:

DELETE entityName indentifierVariable

WHERE where\_clause

For example, if we want to remove all instances of Seller with Silver status we’d

use this:

DELETE Seller s

WHERE s.status = 'Silver'

From Clause

FROM Category c

Category is the domain that we want to query, and here we have specified c as an

identifier of type Category. Category is the entity name.

In the previous example, we are assuming the Category entity class that we discussed

in earlier chapters does not define a name. If we assume that the Category

class defines an entity name using the name element as follows:

@Entity(name = "CategoryEntity")

public class Category

then we must change the FROM clause of the query as follows:

FROM CategoryEntity c

Identifier variables

In our JPQL example, we defined an identifier variable named c, and we used

that variable in other clauses, such as SELECT and WHERE. A simple identifier variable

is defined using the following general syntax:

FROM entityName [AS] identificationVariable

The square brackets ([]) indicate that the AS operator is optional. The identifier

variable (which is not case sensitive) must be a valid Java identifier, and it must

not be a JPQL reserved identifier.

What is a path expression?

In our JPQL example we used expressions such as c.categoryName and c.categoryId.

Such expressions are known as path expressions. A path expression is an

identifier variable followed by the navigation operator (.), and a persistence or

association field. We normally use a path expression to narrow the domain for

a query by using it in a WHERE clause, or order the retrieved result by using an

ORDER BY clause.

if we have a many-to-many relationship between

Category and Item, we can utilize a query to find all Category entities that have

associated Items as follows:

SELECT distinct c

FROM Category c

WHERE c.items is NOT EMPTY

Here c.items is a collection type. Such expressions are known as collection-value

expressions.

To retrieve the Category instances that have a categoryId greater than 500, we’d have

to rewrite the query like this:

SELECT c

FROM Category c

WHERE c.categoryId > 500

Almost all types of Java literals such as boolean, float, enum, String, int, and so

forth are supported in the WHERE clause. You cannot use numeric types such as

octal and hexadecimals, nor can you use array types such as byte[] or char[] in the WHERE clause. Remember that JPQL statements are translated into SQL; SQL is actually imposing the restriction that BLOB and CLOB types cannot be used in a WHERE clause.

Conditional expressions and operators

example of a conditional expression:

c.categoryName = 'Dumped Cars'

Operator Type : Operator

*Navigational* : .

*Unary sign* : +, -

*Arithmetic* : \*, /,+, -

*Relational* : =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER

[OF]

*Logical* : NOT, AND, OR

Using a range with BETWEEN

You can use the BETWEEN operator in an arithmetic expression to compare a variable

with a range of values. You can also use the BETWEEN operator in arithmetic,

string, or DATETIME expressions to compare a path expression to a lower and

upper limit using the following syntax:

path\_expression [NOT] BETWEEN lowerRange and upperRange

Suppose you want to filter the results so that categoryId falls within a specified

range. You can use a WHERE clause and named parameters for the range this way:

WHERE c.categoryId BETWEEN :lowRange AND :highRange

NOTE The lower and upper range used in a BETWEEN operator must be the

same data type.

Using the IN operator

The IN operator allows you to create a conditional expression based on whether a

path expression exists in a list of values. Here is the syntax for the IN operator:

path\_expression [NOT] IN (List\_of\_values)

The list of values can be a static list of comma-separated values, or a dynamic list

retrieved by a subquery. Suppose you want to retrieve the results for userId that

exist in a static list of userIds. This WHERE clause will do the trick:

WHERE u.userId IN ('viper', 'drdba', 'dumpster')

If you want to retrieve the information from users that do not exist in the same

static list, then you can use this WHERE clause:

WHERE u.userId NOT IN ('viper', 'drdba', 'dumpster')

A subquery is a query within a query. A subquery may return a single or multiple

values. You’ll learn more about subqueries in section 10.3.8. Let’s review an

example of a subquery with an IN operator:

WHERE c.user IN (SELECT u

FROM User u

WHERE u.userType = 'A')

Using the LIKE operator

The LIKE operator allows you to determine whether a single-value path expression

matches a string pattern. The syntax for the LIKE operator is

string\_value\_path\_expression [NOT] LIKE pattern\_value\_

Here pattern\_value is a string literal or an input parameter. The pattern\_value

may contain an underscore (\_) or a percent sign (%). The underscore stands for a

single character. Consider the following clause:

WHERE c.itemName LIKE '\_ike'

The percent sign (%) represents any numbers of characters. Whenever you

want to search for all Category entities with a name that starts with Recycle, use

this WHERE clause:

WHERE c.categoryName LIKE 'Recycle%'

Dealing with null values and empty collections

You can use the IS NULL or IS NOT NULL operator to check whether a single-value path expression contains null or not null values.

Eg : WHERE c.parentCategory IS NOT NULL

JPQL provides the IS [NOT] EMPTY comparison operator to check whether a collection type path expression is empty.

Eg : WHERE c.items IS EMPTY

**PACKAGING**

A description...

**Dissecting the EAR file**

META-INF/application.xml

actionBazaar-ejb.jar

actionBazaar.war

actionBazaar-client.jar

lib/actionBazaar-commons.jar

application.xml is the deployment descriptor that describes the standard Java

EE modules packaged in each EAR file.

Applocation.xml content :

<application>

<module>

<ejb>actionBazaar-ejb.jar</ejb>

</module>

<module>

<web>

<web-uri>actionBazaar.war</web-uri>

<context-root>ab</context-root>

</web>

</module>

<module>

<java>actionBazaar-client.jar</java>

</module>

</application>

Exploring class loading

There is a misconception among many developers that all classes are loaded into memory when the JVM starts up; this is not true. Classes are loaded dynamically as and when they are needed at runtime. This process of locating the byte code for a given class name and converting that code into a Java class instance is known as class loading. Your application may have hundreds of EJBs and other resources; loading all these classes into the JVM consumes a lot of memory. Most application servers use a sophisticated mechanism to load classes as and when needed. Therefore, your EJB class will be loaded into memory only when a client accesses it. However, it is implementation specific. Application servers support the bean pooling mechanism, so EJB classes would be loaded into memory while some instances would be instantiated and put into the pool during deployment time.