Entity Mapping and Persistence

By Ebenezer Arkoh-Addo

What are entities?

Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes. Entities represent persistent data stored in a relational database automatically using container-managed persistence. They are persistent because their data is stored persistently in some form of data storage system, such as a database: they do survive a server failure, failover, or a network failure. When an entity is re-instantiated, the state of the previous instance is automatically restored.

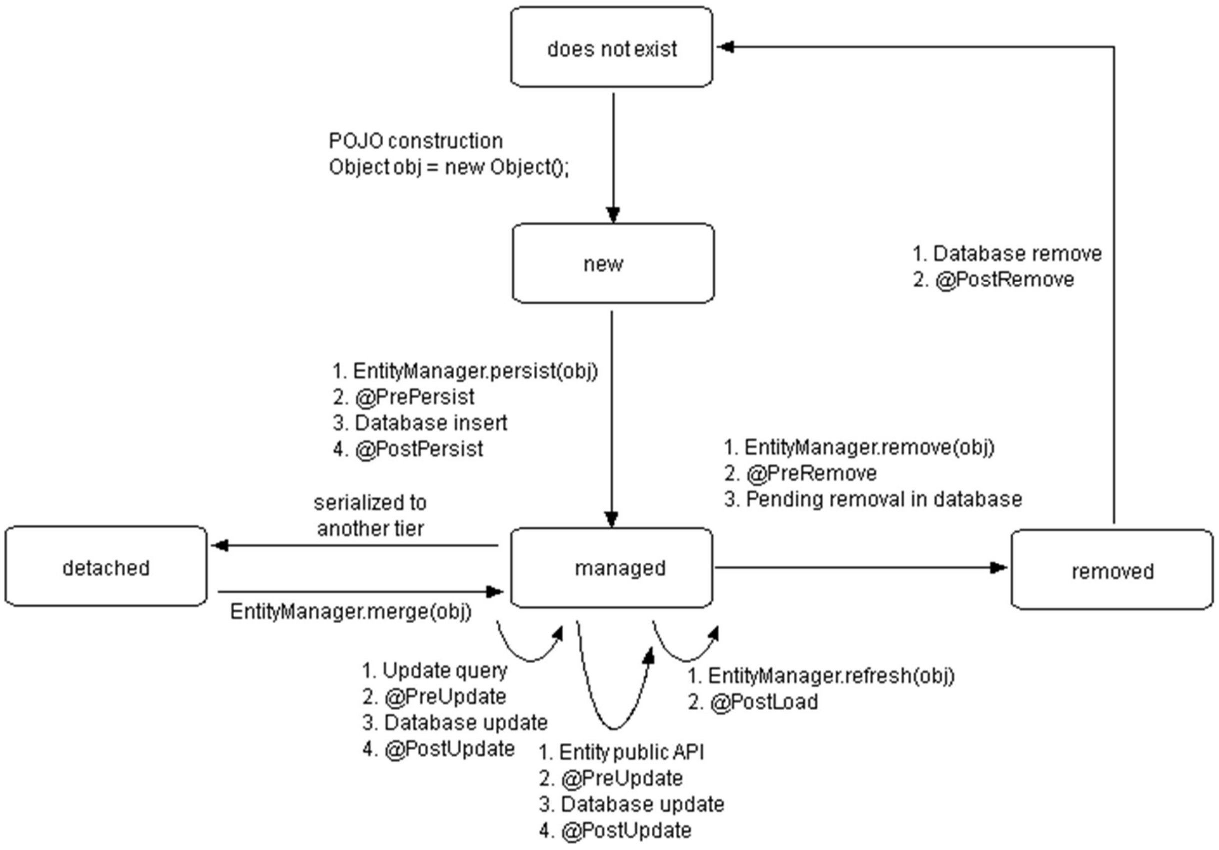
Entities are often used to facilitate business services that involve data and computations on that data. For example, you might implement an entity to retrieve and perform computation on items within a purchase order. Your entity can manage multiple, dependent, persistent objects in performing its tasks.

Requirements for Entity Classes

An entity class must follow these requirements.

- The class must be annotated with the jakarta.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance is passed by value as a detached object, such as through a session bean’s remote business interface, the class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class’s methods. Clients must access the entity’s state through accessor or business methods.

Entity Life Cycle



Annotation	Description
@PrePersist	This optional method is invoked for an entity before the corresponding Entity Manager persist operation is executed. This callback will be invoked on all entities to which these operations are cascaded. If this callback throws an Exception, it will cause the current transaction to be rolled back.
@PostPersist	This optional method is invoked for an entity after the corresponding Entity Manager persist operation is executed. This callback will be invoked on all entities to which these operations are cascaded. This method will be invoked after the database insert operation. This may be directly after the persist operation, a flush operation, or at the end of a transaction. If this callback throws an Exception, it will cause the current transaction to be rolled back.

Annotation	Description
@PreRemove	This optional method is invoked for an entity before the corresponding Entity Manager remove operation is executed. This callback will be invoked on all entities to which these operations are cascaded. If this callback throws an Exception, it will cause the current transaction to be rolled back.
@PostRemove	This optional method is invoked for an entity after the corresponding EntityManager remove operation is executed. This callback will be invoked on all entities to which these operations are cascaded. This method will be invoked after the database delete operation. This may be directly after the remove operation, a flush operation, or at the end of a transaction. If this callback throws an Exception, it will cause the current transaction to be rolled back.
@PreUpdate	This optional method is invoked before the database update operation on entity data. This may be at the time of the entity state update, a flush operation, or at the end of a transaction. OC4J calls this method only if it determines that an actual update is required (only if it is prepared to send SQL to the database). Contrast this with a post-update callback which is called regardless of whether or not an actual change was required.
@PostUpdate	This optional method is invoked after the database update operation on entity data. This may be at the time of the entity state update, a flush operation, or at the end of a transaction. OC4J calls this method even if it determines that no actual update is required (even if it determines that no SQL needs to be sent to the database). Use the pre-update callback if you want to be notified only when the object has actually been changed.
@PostLoad	This optional method is invoked after the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it and before a query result is returned or accessed or an association is traversed.

Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed through either the entity's instance variables or properties. The fields or properties must be of the following Java language types:

- Java primitive types
- java.lang.String
- Other serializable types, including:
- Wrappers of Java primitive types
- User-defined serializable types
- Enumerated types
- Other entities and/or collections of entities

Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity-class instance variables directly. All fields not annotated jakarta.persistence.Transient or not marked as Java transient will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

Persistent Properties

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names. For every persistent property property of type Type of the entity, there is a getter method getProperty and setter method setProperty. If the property is a Boolean, you may use isProperty instead of getProperty. For example, if a Customer entity uses persistent properties and has a private instance variable called firstName, the class defines a getFirstName and setFirstName method for retrieving and setting the state of the firstName instance variable.

The method signatures for single-valued persistent properties are as follows:

Type getProperty()

void setProperty(Type type)

The object/relational mapping annotations for persistent properties must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated @Transient or marked transient.

Validating Persistent Fields and Properties

Jakarta Bean Validation provides a framework for validating application data, integrated into Jakarta EE containers, allowing consistent validation across different layers of an enterprise application. Constraints can be applied to entity classes, embeddable classes, and mapped superclasses, with validation typically triggered after the PrePersist, PreUpdate, and PreRemove lifecycle events.

Constraints are applied through annotations on fields or properties, depending on the access strategy (field or property access). Built-in constraints, provided by the `jakarta.validation.constraints` package, can be used or combined to create custom constraints with associated validator classes.

An example is the `Contact` entity class, which includes various Bean Validation constraints:

- `@NotNull` on `firstName` and `lastName` ensures these fields are mandatory.

- `@Pattern` on `email` checks for a valid email format.

- `@Pattern` on `homePhone` and `mobilePhone` ensures valid 10-digit phone numbers.

- `@Past` on `birthday` ensures the date is in the past.

These constraints enforce data integrity and ensure that invalid data is caught before persistence.

Multiplicity in Entity Relationships

Multiplicities are of the following types.

One-to-one: Each entity instance is related to a single instance of another entity. For example, to model a physical warehouse in which each storage bin contains a single widget, Storage Bin and Widget would have a one-to-one relationship. One-to-one relationships use the jakarta.persistence.OneToOne annotation on the corresponding persistent property or field.

One-to-many: An entity instance can be related to multiple instances of the other entities. A sales order, for example, can have multiple line items. In the order application, CustomerOrder would have a one-to-many relationship with LineItem. One-to-many relationships use the jakarta.persistence.OneToMany annotation on the corresponding persistent property or field.

Many-to-one: Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship. In the example just mentioned, the relationship to CustomerOrder from the perspective of LineItem is many-to-one. Many-to-one relationships use the jakarta.persistence.ManyToOne annotation on the corresponding persistent property or field.

Many-to-many: The entity instances can be related to multiple instances of each other. For example, each college course has many students, and every student may take several courses. Therefore, in an enrollment application, Course and Student would have a many-to-many relationship. Many-to-many relationships use the jakarta.persistence.ManyToMany annotation on the corresponding persistent property or field.

Direction in Entity Relationships

The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

Bidirectional Relationships

In a bidirectional relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object. If an entity has a related field, the entity is said to "know" about its related object. For example, if CustomerOrder knows what LineItem instances it has and if LineItem knows what CustomerOrder it belongs to, they have a bidirectional relationship.

Bidirectional relationships must follow these rules.

- The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.

For many-to-many bidirectional relationships, either side may be the owning side.

Unidirectional Relationships

In a unidirectional relationship, only one entity has a relationship field or property that refers to the other. For example, LineItem would have a relationship field that identifies Product, but Product would not have a relationship field or property for LineItem. In other words, LineItem knows about Product, but Product doesn't know which LineItem instances refer to it.

Cascade Operations and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

The jakarta.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations. Cascade Operations for Entities lists the cascade operations for entities.

Cascade Operations for Entities

ALL

All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}

DETACH

If the parent entity is detached from the persistence context, the related entity will also be detached.

MERGE

If the parent entity is merged into the persistence context, the related entity will also be merged.

PERSIST

If the parent entity is persisted into the persistence context, the related entity will also be persisted.

REFRESH

If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.

REMOVE

If the parent entity is removed from the current persistence context, the related entity will also be removed.

Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

The roster example application demonstrates entity inheritance, as described in Entity Inheritance in the roster Application.

Abstract Entities

An abstract class may be declared an entity by decorating the class with @Entity. Abstract entities are like concrete entities but cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity:

@Entity

```
public abstract class Employee {
  @Id
  protected Integer employeeId;
}
@Entity
public class FullTimeEmployee extends Employee {
  protected Integer salary;
}
@Entity
public class PartTimeEmployee extends Employee {
  protected Float hourlyWage;
}
```

Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the @Entity annotation and is not mapped as an entity by the Jakarta Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the annotation jakarta. persistence.MappedSuperclass:

```
@MappedSuperclass
public class Employee {
  @Id
  protected Integer employeeId;
}
@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}
@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

Mapped superclasses cannot be queried and cannot be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations. Mapped superclasses can't be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the preceding code sample, the underlying tables would be FULLTIMEEMPLOYEE and PARTTIMEEMPLOYEE, but there is no EMPLOYEE table.

Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is nonpersistent, and any state inherited from the non-entity superclass by an entity class is nonpersistent. Non-entity superclasses may not be used in EntityManager or Query operations. Any mapping or relationship annotations in non-entity superclasses are ignored

Managing Entities

Entities are managed by the entity manager, which is represented by jakarta.persistence.EntityManager instances. Each EntityManager instance is associated with a persistence context: a set of managed entity instances that exist in a particular data store. A persistence context defines the scope under which particular entity instances are created, persisted, and removed. The EntityManager interface defines the methods that are used to interact with the persistence context.

The EntityManager Interface

The EntityManager API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

Managing an Entity Instance's Lifecycle

You manage entity instances by invoking operations on the entity by means of an EntityManager instance. Entity instances are in one of four states: new, managed, detached, or removed.

New entity instances have no persistent identity and are not yet associated with a persistence context.

Managed entity instances have a persistent identity and are associated with a persistence context.

Detached entity instances have a persistent identity and are not currently associated with a persistence context.

Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

Persisting Entity Instances

New entity instances become managed and persistent either by invoking the persist method or by a cascading persist operation invoked from related entities that have the cascade=PERSIST or cascade=ALL elements set in the relationship annotation. This means that the entity's data is stored to the database when the transaction associated with the persist operation is completed. If the entity is already managed, the persist operation is ignored, although the persist operation will cascade to related entities that have the cascade element set to PERSIST or ALL in the relationship annotation. If persist is called on a removed entity instance, the entity becomes managed. If the entity is detached, either persist will throw an IllegalArgumentException, or the transaction commit will fail. The following method performs a persist operation:

```
@PersistenceContext

EntityManager em;

public LineItem createLineItem(CustomerOrder order, Product product,

    int quantity) {

  LineItem li = new LineItem(order, product, quantity);
```

```
   order.getLineItems().add(li);
   em.persist(li);
   return li;
}
```

The persist operation is propagated to all entities related to the calling entity that have the cascade element set to ALL or PERSIST in the relationship annotation:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
   return lineItems;
}
```

Removing Entity Instances

Managed entity instances are removed by invoking the remove method or by a cascading remove operation invoked from related entities that have the cascade=REMOVE or cascade=ALL elements set in the relationship annotation. If the remove method is invoked on a new entity, the remove operation is ignored, although remove will cascade to related entities that have the cascade element set to REMOVE or ALL in the relationship annotation. If remove is invoked on a detached entity, either remove will throw an IllegalArgumentException, or the transaction commit will fail. If invoked on an already removed entity, remove will be ignored. The entity's data will be removed from the data store when the transaction is completed or as a result of the flush operation.

In the following example, all LineItem entities associated with the order are also removed, as CustomerOrder.getLineItems has cascade=ALL set in the relationship annotation:

```
public void removeOrder(Integer orderId) {
   try {
      CustomerOrder order = em.find(CustomerOrder.class, orderId);
      em.remove(order);
   }...
}
```

Persistence Units

A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the persistence.xml configuration file. The following is an example persistence.xml file:

```
<persistence>

  <persistence-unit name="OrderManagement">

    <description>This unit manages orders and customers.

      It does not rely on any vendor-specific features and can

      therefore be deployed to any persistence provider.

    </description>

    <jta-data-source>jdbc/MyOrderDB</jta-data-source>

    <jar-file>MyOrderApp.jar</jar-file>

    <class>com.widgets.CustomerOrder</class>

    <class>com.widgets.Customer</class>

  </persistence-unit>

</persistence>
```

This file defines a persistence unit named OrderManagement, which uses a Jakarta Transactions aware data source, jdbc/MyOrderDB. The jar-file and class elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasse

References

- Jakarta EE. (n.d.). *Introduction to persistence*. Jakarta EE. https://jakarta.ee/learn/docs/jakartaee-tutorial/current/persist/persistence-intro/persistence-intro.html
- Oracle. (n.d.). *Entities and persistence* (Oracle Database Documentation). Retrieved September 1, 2024, from https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm
- Girhe, N. (2020, September 25). *Java Persistence API (JPA) and object-relational mapping (ORM)*. Medium. https://medium.com/@niranjangirheindia/java-persistence-api-jpa-and-object-relational-mapping-orm-582c5905195b