Java persistence with JPA and Hibernate

The Java Persistence API (JPA) is a Java specification that bridges the gap between relational databases and object-oriented programming.

Object relations in JPA

Relational databases have existed as a means for storing program data since the 1970s. While developers today have many alternatives to the relational database, this type of database is scalable and well understood, and is still widely used in small- and large-scale software development.

Java objects in a relational database context are defined as entities. Entities are placed in tables where they occupy columns and rows. Programmers use foreign keys and join tables to define the relationships between entities--namely one-to-one, one-to-many, and many-to-many relationships. We can also use SQL (Structured Query Language) to retrieve and interact with data in individual tables and across multiple tables, using foreign key constraints. The relational model is flat, but developers can write queries to retrieve data and construct objects from that data.

Object-relations impedance mismatch

You may be familiar with the term object-relations impedance mismatch, which refers to the challenge of mapping data objects to a relational database. This mismatch occurs because object-oriented design is not limited to one-to-one, one-to-many, and many-to-many relationships. Instead, in object-oriented design, we think of objects, their attributes and behavior, and how objects relate. Two examples are encapsulation and inheritance:

If an object contains another object, we define this through encapsulation--a has-a relationship.
If an object is a specialization of another object, we define this through inheritance--an is-a relationship.

Association, aggregation, composition, abstraction, generalization, realization, and dependencies are all object-oriented programming concepts that can be challenging to map to a relational model.

ORM: Object-relational mapping

The mismatch between object-oriented design and relational database modeling has led to a class of tools developed specifically for object-relational mapping (ORM). ORM tools like Hibernate, EclipseLink, and iBatis translate relational database models, including entities and their relationships, into object-oriented models. Many of these tools existed before the JPA specification, but without a standard their features were vendor dependent.

First released as part of EJB 3.0 in 2006, the Java Persistence API (JPA) offers a standard way to annotate objects so that they can be mapped and stored in a relational database. The specification also defines a common construct for interacting with databases. Having an ORM standard for Java brings consistency to vendor implementations, while also allowing for flexibility and add-ons. As an example, while the original JPA specification is applicable to relational databases, some vendor implementations have extended JPA for use with NoSQL databases.

Getting started with JPA

The Java Persistence API is a specification, not an implementation: it defines a common abstraction that you can use in your code to interact with ORM products. Some of the important parts of the JPA specification:

O Define entities, fields, and primary keys in the database.
O Create relationships between entities in the database.
O Work with the EntityManager and its methods.

Defining entities

In order to define an entity, you must create a class that is annotated with the @Entity annotation. The @Entity annotation is a marker annotation, which is used to discover persistent entities.

```
@Entity
public class Employee {
   ...
```

```
}
```

By default, this entity will be mapped to the Employee table, as determined by the given class name. If you wanted to map this entity to another table (and, optionally, a specific schema) you could use the @Table annotation to do that.

Mapping fields to columns

With the entity mapped to a table, your next task is to define its fields. Fields are defined as member variables in the class, with the name of each field being mapped to a column name in the table. You can override this default mapping by using the @Column annotation.

```
@Entity
@Table(name="EMPS")
public class Employee {
   private String name;
   @Column(name="SSN_NUMBER")
   private String son;
   ...
}
```

The @Column annotation allows us to define additional properties of the field/column, including length, whether it is nullable, whether it must be unique, its precision and scale (if it's a decimal value), whether it is insertable and updatable, and so forth.

Specifying the primary key

One of the requirements for a relational database table is that it must contain a primary key, or a key that uniquely identifies a specific row in the database. In JPA, we use the @Id annotation to designate a field to be the table's primary key. The primary key is required to be a Java primitive type, a primitive wrapper, such as Integer or Long, a String, a Date, a BigInteger, or a BigDecimal.

```
@Entity
@Table(name="EMPS")
```

```
public class Employee {
   @Id
   private Integer id;
   ...
}
```

It is also possible to combine the @Id annotation with the @Column annotation to overwrite the primary key's column-name mapping.

Relationships between entities

Now that you know how to define an entity, let's look at how to create relationships between entities. JPA defines four annotations for defining entities:

@OneToOne
@OneToMany
@ManyToOne
@ManyToMany

The @OneToOne annotation is used to define a one-to-one relationship between two entities. For example, you may have a User entity that contains a user's name, email, and password, but you may want to maintain additional information about a user (such as age, gender, and favorite color) in a separate UserProfile entity. The @OneToOne annotation facilitates breaking down your data and entities this way.

The User class below has a single UserProfile instance. The UserProfile maps to a single User instance.

```
@Entity
public class User {
   @Id
   private Integer id;
   private String email;
   private String name;
```

```
    private String password;
    @OneToOne(mappedBy="user")
    private UserProfile profile;
    ...
}

@Entity
public class UserProfile {
    @Id
    private Integer id;
    private int age;
    private String gender;
    private String favoriteColor;
    @OneToOne
    private User user;
    ...
}
```

The JPA provider uses UserProfile's user field to map UserProfile to User. The mapping is specified in the mappedBy attribute in the @OneToOne annotation.

One-to-many and many-to-one relationships

The @OneToMany and @ManyToOne annotations facilitate both sides of the same relationship. Consider an example where an Order can have only one Customer, but a Customer may have many Order. The Order entity would define a @ManyToOne relationship with Customer and the Customer entity would define a @OneToMany relationship with Order.

```
@Entity
public class Order {
    @Id
    private Integer id;
    ..

    @ManyToOne
    @JoinColumn(name="EMP_ID")
```

```
    private Employee emp;
    ...
}

@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Integer id;
    private String name;
    @OneToMany(mappedBy = "emp")
    private List<Order> orders = new ArrayList<>();
    ...
}
```

In this case, the Employee class maintains a list of all of the orders by that customer and the Order class maintains a reference to its single employee. Additionally, the @JoinColumn specifies the name of the column in the Order table to store the ID of the Employee.

Many-to-many relationships

Finally, the @ManyToMany annotation facilitates a many-to-many relationship between entities. Here's a case where an Order entity has multiple Employee:

```
@Entity
public class Order {
    @Id
    private Integer id;
    ..

    @ManyToMany
    @JoinTable(name="ORDER_EMPLOYEES",
                joinColumns=@JoinColumn(name="ORDER_ID"),
                inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    private Set<Employee> employees = new HashSet<>();
    ...
}
```

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Integer id;
    ..
    @ManyToMany(mappedBy = "employees")
    private Set<Order> orders = new HashSet<>();
    ...
}
```

In this example, we create a new table, ORDER_EMPLOYEES, with two columns: ORDER_ID and EMP_ID. Using the joinColumns and inverseJoinColumns attributes tells your JPA framework how to map these classes in a many-to-many relationship. The @ManyToMany annotation in the Employee class references the field in the Order class that manages the relationship; namely the employees property.

Working with the EntityManager

EntityManager is the class that performs database interactions in JPA. It is initialized through a configuration file named persistence.xml. This file is found in the META-INF folder in your CLASSPATH, which is typically packaged in your JAR or WAR file. The persistence.xml file contains:

The named "persistence unit," which specifies the persistence framework you're using, such as Hibernate or EclipseLink.
A collection of properties specifying how to connect to your database, as well as any customizations in the persistence framework.
A list of entity classes in your project.

Let's look at an example.

Configuring the EntityManager

First, we create an EntityManager using the EntityManagerFactory retrieved from the Persistence class:

```
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("Orders");
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

In this case we've created an EntityManager that is connected to the "Orders" persistence unit, which we've configured in the persistence.xml file.

The EntityManager class defines how our software will interact with the database through JPA entities.

Here are some of the methods used by EntityManager:

O find retrieves an entity by its primary key.

O createQuery creates a Query instance that can be used to retrieve entities from the database.

O createNamedQuery loads a Query that has been defined in a @NamedQuery annotation inside one of the persistence entities. Named queries provide a clean mechanism for centralizing JPA queries in the definition of the persistence class on which the query will execute.

O getTransaction defines an EntityTransaction to use in your database interactions. Just like database transactions, you will typically begin the transaction, perform your operations, and then either commit or rollback your transaction. The getTransaction() method lets you access this behavior at the level of the EntityManager, rather than the database.

O merge() adds an entity to the persistence context, so that when the transaction is committed, the entity will be persisted to the database. When using merge(), objects are not managed.

O persist adds an entity to the persistence context, so that when the transaction is committed, the entity will be persisted to the database. When using persist(), objects are managed.

O refresh refreshes the state of the current entity from the database.

O flush synchronizes the state of the persistence context with the database.

merge() or persist()?
The difference between merge() and persist() is that merge() adds the entity to the persistence context, but your reference to the object is not "managed." If you make changes to your object after invoking merge(), those changes will not be sent to the database. The persist() method marks your object as "managed." Any changes you make to your object after invoking persist() will be sent to the database.

JPA with Hibernate

Configuring Hibernate

We include four dependencies:

O hibernate-core: Hibernate's core functionality.
O hibernate-entitymanager: Hibernate's support for an EntityManager.
O hibernate-jpa-2.1-api: The JPA API.
O h2: The embedded H2 database.

Configuring the EntityManager
Recall that JPA's EntityManager is driven by the persistence.xml file.

The persistence.xml file begins with a persistence node that can contain one or more persistence-units. A persistence-unit has a name, which we'll use later when we create the EntityManager, and it defines the attributes of that unit.

We could:

o Specify HibernatePersistenceProvider, so the application knows we're using Hibernate as our JPA provider.
o Define the entities
o Define the database configuration via JDBC.
o Configure Hibernate, including setting the Hibernate dialect, so that Hibernate knows how to communicate with the database.

The domain model
In this step you have to model the entities.

About JPA Query Language (JPQL)
JPQL is similar to SQL, but it operates on entities, their fields, and their relationships, rather than on database column names. JPA refers to these queries as running on top of an "abstract schema," which gets translated to a proper database schema. Entities are mapped to that database schema. Here's the format for a simple JPQL query:

SELECT returnedEntity FROM entityName var WHERE whereClause

For example:

SELECT e FROM Employee b WHERE b.name = :name

Parameters can be referenced by name or by position. When referencing a parameter by name, you specify the name your query expects, such as "name" prefaced by a ":". If, instead, you wanted to reference the name by position, you could replace ":name" with "?1". When executing the query, you would set the parameter with position "1."

CascadeType
CascadeType is an enumerated type that defines cascading operations to be applied in a given relationship.
In this case, CascadeType defines operations performed on the employee for example, that should be propagated to the order. CascadeTypes include the following:

o DETACH: When an entity is detached from the EntityManager, detach the entities on the other side of the operation, as well.

o MERGE: When an entity is merged into the EntityManager, merge the entities on the other side of the operation, as well.

o PERSIST: When an entity is persisted to the EntityManager, persist the entities on the other side of the operation, as well.

o REFRESH: When an entity is refreshed from the EntityManager, also refresh the entities on the other side of the operation.

o FLUSH: When an entity is flushed to the EntityManager, flush its corresponding entities.

o ALL: Includes all of the aforementioned operation types.

For example, when any operation is performed on an employee, its orders should be updated. This makes sense because an order cannot exist without its employee.

Repositories in JPA
We could create an EntityManager and do everything inside the sample application class, but using external repository classes will make the code cleaner. As defined by the Repository pattern, creating a repository isolates the persistence logic for each entity.

Many-to-many relationships in JPA
Many-to-many relationships define entities for which both side of the relationship can have multiple references to each other.

Unidirectional or bidirectional?

In JPA we use the @ManyToMany annotation to model many-to-many relationships. This type of relationship can be unidirectional or bidirectional:

O In a unidirectional relationship only one entity in the relationship points the other.
O In a bidirectional relationship both entities point to each other.

We use the mappedBy attribute of the @ManyToMany annotation to create this mapping.

Fetching strategies

The thing to notice in the @ManyToMany annotation is how we can configure the fetching strategy, which can be lazy or eager. If you we've set the fetch to EAGER, when we retrieve a Customer from the database for example, we'll also automatically retrieve all of its corresponding Orders.

If we chose to perform a LAZY fetch instead, we would only retrieve each Order for example as it was specifically accessed.

Join tables

JoinTable is a class that facilitates the many-to-many relationship.

The following properties are tipically applied to the @ManyToMany annotation:

O mappedBy references a field name on the class that manages the many-to-many relationship.

O cascade is configured to CascadeType.PERSIST, which means that when an Order is saved its corresponding Customer entities should also be saved.

O fetch tells the EntityManager how it should retrieve an order's customers eagerly: when it loads an Order, it should also load all corresponding Customer entities for example.

When configuring entities for persistence, it isn't enough to simply add a customer to an order; we also need to update the other side of the relationship. This means we need to add the order to the customer. When both sides of the relationship are configured properly, so that the order has a reference to the customer and the customer has a reference to the order, then the join table will also be properly populated.

JPA repositories

We could implement all of our persistence code directly in the sample application, but creating repository classes allows us to separate persistence code from application code.