Day4: JPQL

JPQL: Java Persistence Query language

JPQL is Java Persistence Query Language defined in JPA specification. It is used to create queries against entities to store in a relational database. JPQL is developed based on SQL syntax. But it won't affect the database directly.

JPQL can retrieve information or data using SELECT clause, can do bulk updates using UPDATE clause and DELETE clause. EntityManager.createQuery() API will support querying language.

Query Structure:

JPQL syntax is very similar to the syntax of SQL. Having SQL like syntax is an advantage because SQL is a simple structured query language and many developers are using it in applications. SQL works directly against relational database tables, records and fields, whereas JPQL works with Java classes and instances.

For example, a JPQL query can retrieve an entity object rather than field result set from database, as with SQL. The JPQL query structure as follows.

```
SELECT ... FROM ... [WHERE ...] [GROUP BY ... [HAVING ...]] [ORDER BY ...] The structure of JPQL DELETE and UPDATE queries is simpler as follows. DELETE FROM ... [WHERE ...] UPDATE ... SET ... [WHERE ...]
```

Scalar and Aggregate Functions

Scalar functions return resultant values based on input values. Aggregate functions return the resultant values by calculating the input values.

Follow the same example of employee management used in previous chapters. Here we will go through the service classes using the scalar and aggregate functions of JPQL.

Let us assume the employee table contains the following records.

Eid	Ename	Salary	Deg
1201	Gopal	40000	Technical Manager
1202	Manisha	40000	Proof Reader
1203	Masthanvali	40000	Technical Writer
1204	Satish	30000	Technical Writer
1205	Krishna	30000	Technical Writer
1206	Kiran	35000	Proof Reader

```
package com.masai; import java.util.List; import javax.persistence.EntityManager; import javax.persistence.EntityManagerFactory; import javax.persistence.Persistence; import javax.persistence.Query; public class Demo { public static void main( String[ ] args ) { EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "empUnit" ); EntityManager entitymanager = emfactory.createEntityManager(); //Scalar function Query query = entitymanager.createQuery("Select UPPER(e.ename) from Employee e"); List<String> list = query.getResultList(); for(String e:list) { System.out.println("Employee NAME :"+e); } //Aggregate function Query query1 = entitymanager.createQuery("Select MAX(e.salary) from Employee e"); Double result = (Double) query1.getSingleResult(); System.out.println("Max Employee Salary :" + result); } } Output: Employee NAME :GOPAL Employee NAME :MANISHA Employee NAME :MASTHANVALI Employee NAME :SATISH Employee NAME :KRISHNA Employee NAME :KIRAN ax Employee Salary :40000.0
```

Between, And, Like Keywords:

'Between', 'And', and 'Like' are the main keywords of JPQL. These keywords are used after Where clause in a query.

Example:

```
package com.masai; import java.util.List; import javax.persistence.EntityManager; import
javax.persistence.EntityManagerFactory; import javax.persistence.Persistence; import
javax.persistence.Query; import com.tutorialspoint.eclipselink.entity.Employee; public
class BetweenAndLikeFunctions { public static void main( String[ ] args ) {
EntityManagerFactory emfactory = Persistence.createEntityManagerFactory(
"Eclipselink_JPA" ); EntityManager entitymanager = emfactory.createEntityManager();
//Between Query query = entitymanager.createQuery( "Select e " + "from Employee e " +
"where e.salary " + "Between 30000 and 40000" ); List<Employee> list=
(List<Employee>)query.getResultList( ); for( Employee e:list ){
System.out.print("Employee ID :" + e.getEid( )); System.out.println("\t Employee salary
:" + e.getSalary( )); } //Like Query query1 = entitymanager.createQuery("Select e " +
"from Employee e " + "where e.ename LIKE 'M%'"); List<Employee> list1=
(List<Employee>)query1.getResultList( ); for( Employee e:list1 ) {
System.out.print("Employee ID :"+e.getEid( )); System.out.println("\t Employee name
:"+e.getEname( )); } } } Output: Employee ID :1201 Employee salary :40000.0 Employee ID
:1202 Employee salary :40000.0 Employee ID :1203 Employee salary :40000.0 Employee ID
:1204 Employee salary :30000.0 Employee ID :1205 Employee salary :30000.0 Employee ID
:1206 Employee salary :35000.0 Employee ID :1202 Employee name :Manisha Employee ID
:1203 Employee name :Masthanvali
```

Named Queries:

A @NamedQuery annotation is defined as a query with a predefined unchangeable query string. Instead of dynamic queries, usage of named queries may improve code organization by separating the JPQL query strings from POJO. It also passes the query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

First of all, add @NamedQuery annotation to the Employee entity class named **Employee.java** under **com.masai.entity** package as follows:

```
package com.masai.entity; import javax.persistence.Entity; import
javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import
javax.persistence.Id; import javax.persistence.NamedQuery; import
javax.persistence.Table; @Entity @Table @NamedQuery(name = "find employee by id", query
= "Select e from Employee e where e.eid = :id") public class Employee { @Id
@GeneratedValue(strategy = GenerationType.AUTO) private int eid; private String ename;
private double salary; private String deg; public Employee(int eid, String ename, double
salary, String deg) { super( ); this.eid = eid; this.ename = ename; this.salary =
salary; this.deg = deg; } public Employee( ) { super(); } public int getEid( ) { return
eid; } public void setEid(int eid) { this.eid = eid; } public String getEname( ) {
return ename; } public void setEname(String ename) { this.ename = ename; } public double
getSalary( ) { return salary; } public void setSalary(double salary) { this.salary =
salary; } public String getDeg( ) { return deg; } public void setDeg(String deg) {
this.deg = deg; } @Override public String toString() { return "Employee [eid=" + eid +
", ename=" + ename + ", salary=" + salary + ", deg=" + deg + "]"; } }
```

Create a class named **NamedQueries.java** under **com.masai.service** package as follows:

```
package com.masai.service; import java.util.List; import
javax.persistence.EntityManager; import javax.persistence.EntityManagerFactory; import
javax.persistence.Persistence; import javax.persistence.Query; import
com.tutorialspoint.eclipselink.entity.Employee; public class NamedQueries { public
static void main( String[ ] args ) { EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" ); EntityManager entitymanager
= emfactory.createEntityManager(); Query query = entitymanager.createNamedQuery("find
employee by id"); query.setParameter("id", 1204); List<Employee> list =
query.getResultList( ); for( Employee e:list ){ System.out.print("Employee ID :" +
e.getEid( )); System.out.println("\t Employee Name :" + e.getEname( )); } } }
```

Native Query:

This can be especially true if you are migrating an older JDBC application which has already a stable and tuned list of queries.

In these situations it is convenient to use the **native SQL** for creating Entity Queries.

The simplest way to run a native SQL Query is to use the **createNativeQuery**() method of the EntityManager interface, passing in the query string and the entity type that will be returned.

Example:

```
public List<Customer> findAllCustomersNative() { Query query =
em.createNativeQuery("SELECT * from customer",Customer.class); List<Customer>
customerList = query.getResultList(); return customerList; }
```

Named Native Query:

Native SQL can also be used for named queries by defining a **@NamedNativeQuery** annotation.

example:

```
package com.masai.entity; import javax.persistence.Entity; import
javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import
javax.persistence.Id; import javax.persistence.NamedNativeQueries; import
javax.persistence.NamedNativeQuery; @Entity @NamedNativeQueries({ @NamedNativeQuery(
name = "Person.findAllPersons", query = "SELECT * " + "FROM Person ", resultClass =
Person.class ), @NamedNativeQuery( name = "Person.findPersonByName", query = "SELECT * "
+ "FROM Person p " + "WHERE p.name = ?", resultClass = Person.class) }) public class
Person { @Id @GeneratedValue(strategy = GenerationType.AUTO) Long id; String name;
String surname; // Getter and Setters omitted for brevity }
```

Executing the NamedNativeQuery:

```
public List<Person> findAll() { //Get the EntityManager object Query q =
```