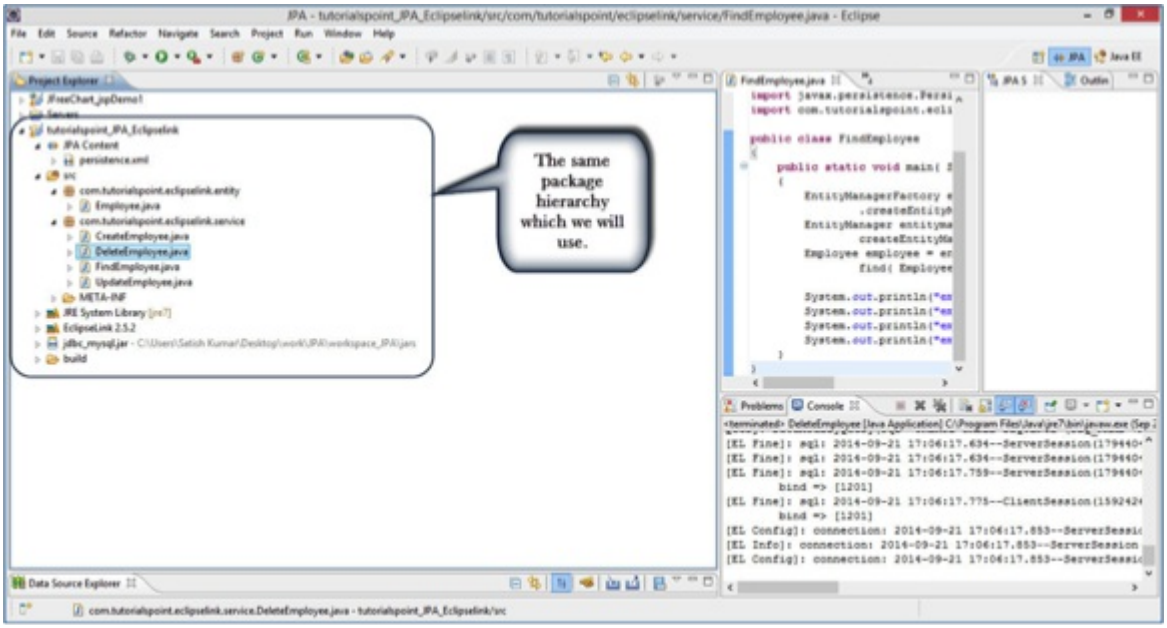
JPA - JPQL

 Copyright © tutorialspoint.com

This chapter tells you about JPQL and how it works with persistence units. In this chapter, examples follow the same package hierarchy, which we used in the previous chapter as follows:



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 for 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 returns resultant values based on input values. Aggregate functions returns the resultant values by calculating the input values.

Follow the same example employee management used in previous chapters. Here we will go

through the service classes using scalar and aggregate functions of JPQL.

Let us assume the jpadb.employee table contains 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

Create a class named **ScalarandAggregateFunctions.java** under **com.tutorialspoint.eclipselink.service** package as follows.

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class ScalarandAggregateFunctions {
   public static void main( String[ ] args ) {

      EntityManagerFactory emfactory = Persistence.createEntityManagerFactory(
"Eclipselink_JPA" );
      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);
   }
}
```

After compilation and execution of the above program you will get output in the console panel of Eclipse IDE as follows:

```
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.

Create a class named **BetweenAndLikeFunctions.java** under **com.tutorialspoint.eclipselink.service** package as follows:

```
package com.tutorialspoint.eclipselink.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 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( ));
      }
   }
}
```

After compilation and execution of the above program you will get output in the console panel of Eclipse IDE as follows:

```
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
```

Ordering

To Order the records in JPQL we use ORDER BY clause. The usage of this clause is same as the use in SQL, but it deals with entities. Follow the Order by example.

Create a class Ordering.java under **com.tutorialspoint.eclipselink.service** package as follows:

```
package com.tutorialspoint.eclipselink.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 Ordering {

   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 " +
"ORDER BY e.ename ASC" );

      List<Employee> list = (List<Employee>)query.getResultList( );

      for( Employee e:list ) {
         System.out.print("Employee ID :" + e.getEid( ));
         System.out.println("\t Employee Name :" + e.getEname( ));
      }
   }
}
```

After compilation and execution of the above program you will get output in the console panel of Eclipse IDE as follows:

```
Employee ID :1201  Employee Name :Gopal
Employee ID :1206  Employee Name :Kiran
Employee ID :1205  Employee Name :Krishna
Employee ID :1202  Employee Name :Manisha
Employee ID :1203  Employee Name :Masthanvali
Employee ID :1204  Employee Name :Satish
```

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.tutorialspoint.eclipselink.entity** package as follows:

```
package com.tutorialspoint.eclipselink.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(query = "Select e from Employee e where e.eid = :id", name = "find
employee by 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 [e;
   }
}
```

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

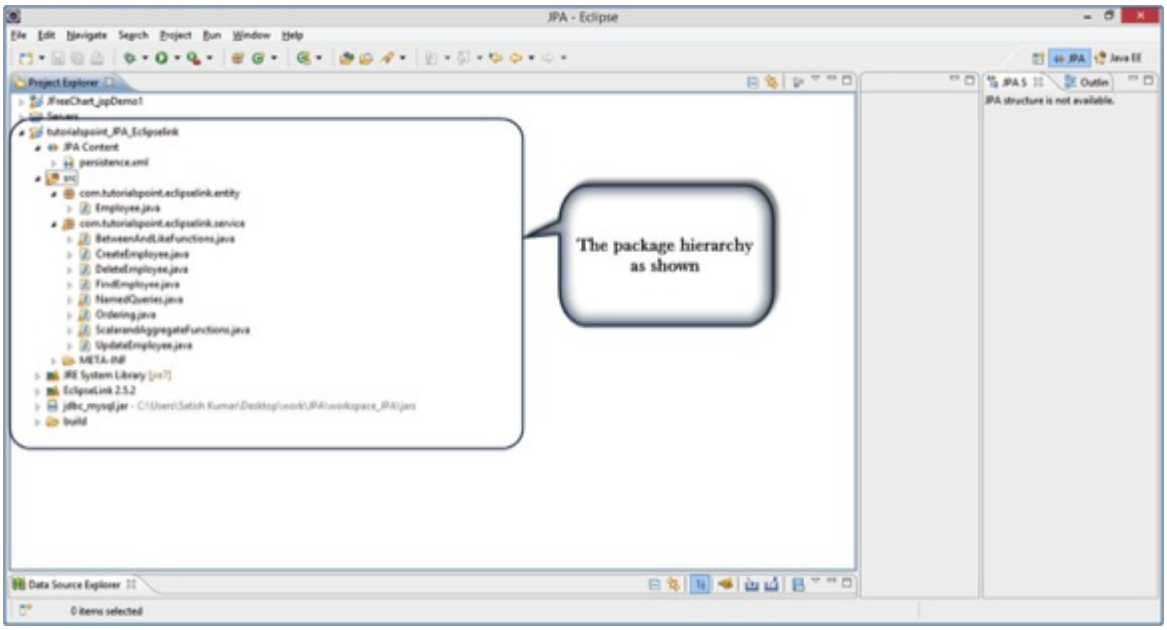
```
package com.tutorialspoint.eclipselink.service;

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( ));
      }
   }
}
```

After compilation and execution of the above program you will get output in the console panel of Eclipse IDE as follows:

```
Employee ID :1204  Employee Name :Satish
```

After adding all the above classes the package hierarchy is shown as follows:



Eager and Lazy Loading

The main concept of JPA is to make a duplicate copy of the database in cache memory. While transacting with the database, first it will effect on duplicate data and only when it is committed using entity manager, the changes are effected into the database.

There are two ways of fetching records from the database - eager fetch and lazy fetch.

Eager fetch

Fetching the whole record while finding the record using Primary Key.

Lazy fetch

It checks for the availability of notifies it with primary key if it exists. Then later if you call any of the getter method of that entity then it fetches the whole.

But lazy fetch is possible when you try to fetch the record for the first time. That way, a copy of the whole record is already stored in cache memory. Performance wise, lazy fetch is preferable.

Loading [MathJax]/jax/output/HTML-CSS/jax.js