OBJECT™ TECHNOLOGIES

Hibernate Query Language and Criteria Queries

What Will Be Covered

OBJECT TECHNOLOGY

- Retrieving records as objects
- Updating record
- Deleting record
- What is HQL and advantages
- HQL examples
- Named queries
- Native SQL queries
- What are criteria queries
- Examples of Criteria queries

Retrieving records as objects

OBJECT

- Hibernate Session provide different methods to fetch data from database. Two of them are – get() and load(). There are also a lot of overloaded methods for these, that we can use in different circumstances.
- Both get() and load() seems similar because both of them fetch the data from database.
- get() returns the object by fetching it from database or from hibernate cache whereas load() just returns the reference of an object that might not actually exists, it loads the data from database or cache only when you access other properties of the object.
- When we use get() to retrieve data that doesn't exists, it returns null. With load(), we are able to print the id but as soon as we try to access other fields, it fires database query and throws org.hibernate.ObjectNotFoundException if there is no record found with the given identifier. It's hibernate specific Runtime Exception, so we don't need to catch it explicitly.

Retrieving records as objects

OBJECT

```
                //Prep Work
                SessionFactory sessionFactory =
HibernateUtil.getSessionFactory();
                Session session = sessionFactory.openSession();
                Transaction tx = session.beginTransaction();

                //Get Example
                Employee emp = (Employee)
session.get(Employee.class, new Long(2));
                System.out.println("Employee get called");
                System.out.println("Employee ID= "+emp.getId());
                System.out.println("Employee Get Details::
"+emp+"\n");

                //load Example
                Employee emp1 = (Employee)
session.load(Employee.class, new Long(1));
                System.out.println("Employee load called");
                System.out.println("Employee ID=
"+emp1.getId());
                System.out.println("Employee load Details::
"+emp1+"\n");
```

Updating record

OBJECT

- We can update an object in hibernate by calling the update() method, provided by the org.hibernate.Session.
- We can directly update the record in the database without load or getting the record from the database. If we know the record id, then we can directly create a new object and assign the primary key to it and save using update() method.

```
Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student student = new Student();
student.setId(111);
student.setName("chandra shekhar");
student.setRollNumber(8469);
session.update(student);
```

Updating record

OBJECT TECHNOLOGIES

- If we don't set any one of the property, then its default values (null) will be updated in the database.
- We can even update an object with loading the object from the database. We can load the object from the database by calling load() or get() methods.

```
Session session = factory.openSession();

Transaction tx = session.beginTransaction();
Student student = session.load(Student.class, 111);
student.setName("Johnson");
session.update(student);
tx.commit();
```

Deleting record

OBJECT

- The Session.delete(Object) method allows us to remove a transient instance of the entity with an identifier associated with existing persistent state.

```
Product product = new Product();
product.setId(37);
session.delete(product);
```

- This way is simple and straightforward, because we don't have to load a persistent instance from the datastore before deleting it. However, its drawback is that it doesn't remove the associated instances, even an appropriate cascade type is specified in the class mappings/annotations.
- A persistent instance can be loaded using the Session.load(Class, ID) method before deleting it.

Deleting record

```
Serializable id = new Long(17);
Object persistentInstance = session.load(Category.class, id);
if (persistentInstance != null) {
    session.delete(persistentInstance);
}
```

- The advantage of above way is it will delete even all associated entities based on cascade type mentioned. For example, the above code snippet deletes a category with id 17 with associated products.

What is HQL

OBJECT

- Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.
- All the clauses of SQL like select ,from, group by, order by, having are applicable as it is in HQL
- Keywords like SELECT , FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL.
- HQL (and JPQL) are loosely based on SQL and are easy to learn for anyone familiar with SQL.

Advantages of HQL

OBJECT

There are many advantages of HQL. They are as follows:

- database independent
- supports polymorphic queries
- easy to learn for Java Programmer

HQL examples

OBJECT

- It is an object oriented representation of Hibernate Query. The object of Query can be obtained by calling the createQuery() method Session interface.
- Example of HQL to get all the records

```
Query q = session.createQuery("from Emp",Emp.class);
List<Emp>  emps = q.getResultList();
for(Emp e : emps)
{
    System.out.println(e);
}  //List - element - Emp object
```

- Example of HQL query for retrieving few attributes

```
Query q = session.createQuery("select empid,ename from Emp where salary > 20000",Object[].class)
List<Object []> emps = q.getResultList();
//List - element - array(record) of object(column)
for( Object [] e : emps )
    System.out.println(e[0]+" : "+e[1]);
```

Fetching result of query

OBJECT

- The **getResultList** and **getResultStream** methods to get a query result containing multiple records,
- The **getSingleResult** method to get a query result that has precisely 1 record,
- Different versions of the setParameter method to set the value of a bind parameter used in your query,
- The **setFirstResult** and **setMaxResult** methods to define pagination,
- The **setHint** method to provide a query hint and
- Various methods to configure the cache handling.

Named query

OBJECT

- If there are a lot of queries, then they will cause a code mess because all the queries will be scattered throughout the project. That's why Hibernate provides Named Query that we can define at a central location and use them anywhere in the code.
- Named queries can be created for both HQL and Native SQL
- Hibernate Named Query can be defined in Hibernate mapping files or through the use of JPA annotations @NamedQuery and @NamedNativeQuery.
- Session interface provides methods for creating query object using methods createNamedSelectionQuery and createNamedMutationQuery

Named query

OBJECT

```
@Entity
@Table(name="emp")
@NamedQuery(name = "getByDept",query="select empid, salary from Emp where dept = :dept")
public class Emp
{
    @Id
    int empid;
```

```
Query<Emp> getByName= session.createNamedQuery("getByDept", Emp.class);
getByName.setParameter("p", "Porjects");
List<Emp> emps = getByName.getResultList();
for(Emp e : emps)
    System.out.println(e);
```

Native SQL queries

OBJECT

- It is even possible to express queries in the native SQL dialect of your database. This is useful if you want to utilize database-specific features.
- It also provides a clean migration path from a direct SQL/JDBC based application to Hibernate/Jakarta Persistence.
- Hibernate also allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and retrieve operations.

```
Query q = session.createNativeQuery("select * from emp",Emp.class);
List<Emp> list = q.getResultList();
for(Emp e : list)
{
    System.out.println(e);
}
```

What are criteria querries

OBJECT

- Criteria queries offer a type-safe alternative to HQL, JPQL and native SQL queries.
- Criteria queries are a programmatic, type-safe way to express a query. They are type-safe in terms of using interfaces and classes to represent various structural parts of a query such as the query itself, the select clause, or an order-by, etc.
- Criteria queries are essentially an object graph, where each part of the graph represents an increasing (as we navigate down this graph) more atomic part of the query.

Examples of Criteria queries

OBJECT

- Selecting the root entity

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery(Person.class);
Root<Person> root = criteria.from(Person.class);
criteria.select(root);
criteria.where(builder.equal(root.get(Person_.name), "John Doe"));

List<Person> persons = entityManager.createQuery(criteria).getResultList();
```

- Selecting an attribute

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<String> criteria = builder.createQuery(String.class);
Root<Person> root = criteria.from(Person.class);
criteria.select(root.get(Person_.nickName));
criteria.where(builder.equal(root.get(Person_.name), "John Doe"));

List<String> nickNames = entityManager.createQuery(criteria).getResultList();
```

Examples of Criteria queries

OBJECT

- Selecting multiple attributes using multiselect

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Object[]> criteria = builder.createQuery(Object[].class);
Root<Person> root = criteria.from(Person.class);

Path<Long> idPath = root.get(Person_.id);
Path<String> nickNamePath = root.get(Person_.nickName);

criteria.multiselect(idPath, nickNamePath);
criteria.where(builder.equal(root.get(Person_.name), "John Doe"));

List<Object[]> idAndNickNames = entityManager.createQuery(criteria).getResultList();
```

HQL and Criteria queries

As now we are aware about performing operations on single instance, in this chapter, we are going to learn manipulation on group records. This can be done by writing queries similar to SQL in the form of HQL.

What is HQL

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.

Features of HQL

1. SQL similarity: HQL?s syntax is very similar to standard SQL. If you are familiar with SQL then writing HQL would be pretty easy: from SELECT, FROM, ORDERBY to arithmetic expressions and aggregate functions, etc.
2. Fully object-oriented: HQL doesn?t use real names of table and columns. It uses class and property names instead. HQL can understand inheritance, polymorphism and association.
3. Case-insensitive for keywords: Like SQL, keywords in HQL are case-insensitive. That means SELECT, select or Select are the same.
4. Case-sensitive for Java classes and properties: HQL considers case-sensitive names for Java classes and their properties, meaning Person and person are two different objects.

Some of the commonly supported clauses in HQL are:

1. HQL From: HQL From is same as select clause in SQL, from Employee is same as select * from Employee. We can also create alias such as from Employee emp or from Employee as emp.
2. HQL Join : HQL supports inner join, left outer join, right outer join and full join. For example, select e.name, a.city from Employee e INNER JOIN e.address a. In this query, Employee class should have a variable named address. We will look into it in the example code.
3. Aggregate Functions: HQL supports commonly used aggregate functions such as count(*), count(distinct x), min(), max(), avg() and sum().
4. Expressions: HQL supports arithmetic expressions (+, -, *, /), binary comparison operators (=, >=, <=, <>, !=, like), logical operations (and, or, not) etc.
5. HQL also supports ordre by and group by clauses.
6. HQL also supports sub-queries just like SQL queries.
7. HQL supports DDL, DML and executing store procedures too.

Some of the commonly supported clauses in HQL are:

1. HQL From: HQL From is same as select clause in SQL, from Employee is same as select * from Employee. We can also create alias such as from Employee emp or from Employee as emp.
2. HQL Join : HQL supports inner join, left outer join, right outer join and full join. For example, select e.name, a.city from Employee e INNER JOIN e.address a. In this query, Employee class should have a variable named address. We will look into it in the example code.
3. Aggregate Functions: HQL supports commonly used aggregate functions such as count(*), count(distinct x), min(), max(), avg() and sum().
4. Expressions: HQL supports arithmetic expressions (+, -, *, /), binary comparison operators (=, >=, <=, <>, !=, like), logical operations (and, or, not) etc.
5. HQL also supports ordre by and group by clauses.
6. HQL also supports sub-queries just like SQL queries.
7. HQL supports DDL, DML and executing store procedures too.

HQL examples

Basically HQL is a simple string which will look like SQL query where the name of attributes and classes are referred rather than columns and tables.

This query can be useful for retrieving collection of enetities directly without any explicit execution.

Simple example for retrieving all employees :

```
Query q = session.createQuery("from Emp",Emp.class);
 List<Emp>  emps = q.getResultList();
 for(Emp e : emps)
 {
    System.out.println(e);
 } //List - element - Emp object
```

Example of HQL for getting only few attributes :

```
Query q = session.createQuery("select empid,ename from Emp where salary > 20000",Object[].class);
 List<Object []> emps = q.getResultList();
// List - element - array(record) of object(column)
```

Example of HQL for getting only few attributes :

```
Query q = session.createQuery("select empid,ename from Emp where salary > 20000",Object[].class);
 List<Object []> emps = q.getResultList();
// List - element - array(record) of object(column)
 for( Object [] e : emps )
    System.out.println(e[0]+" : "+e[1]);
```

Two additional interfaces are introduced by Hibernate 6 as SelectionQuery and MutationQuery to separate the queries which retrieve the data and the queries which modifies the data in the database.

Named queries

If there are a lot of queries, then they will cause a code mess because all the queries will be scattered throughout the project. That's why Hibernate provides Named Query that we can define at a central location and use them anywhere in the code.

Named queries can be created for both HQL and Native SQL.

Hibernate Named Query can be defined in Hibernate mapping files or through the use of JPA annotations @NamedQuery and @NamedNativeQuery.

Session interface provides methods for creating query object using methods createNamedSelectionQuery and createNamedMutationQuery for queries retrieving data and mutating data respectively in the latest version of hibernate (in hibernate 6)

```
Query q = session.createNamedQuery("getByDept",Object[].class);
q.setParameter("dept","Admin");
List<Object[]> list = q.getResultList();
for(Object[] e : list)
{
   System.out.println(e[0]+" : "+e[1]);
}
```

The query having the name "getByDept" is written in bean class using annotation @NamedQuery

```
@Entity
@Table(name="emp")
@NamedQuery(name = "getByDept",query="select empid, salary from Emp where dept = :dept")
public class Emp
```

The query having the name "getByDept" is written in bean class using annotation @NamedQuery

```
@Entity
@Table(name="emp")
@NamedQuery(name = "getByDept",query="select empid, salary from Emp where dept = :dept")
public class Emp
{
   //
}
```

Criteria queries

Criteria queries offer a type-safe alternative to HQL, JPQL and native SQL queries.

Criteria queries are a programmatic, type-safe way to express a query. They are type-safe in terms of using interfaces and classes to represent various structural parts of a query such as the query itself, the select clause, or an order-by, etc.

Criteria queries are essentially an object graph, where each part of the graph represents an increasing (as we navigate down this graph) more atomic part of the query.

The first step in performing a criteria query is building this graph. The jakarta.persistence.criteria.CriteriaBuilder interface is the first object that needs to be created for criteria queries. Its role is that of a factory for all the individual pieces of the criteria. This instance is created by calling the getCriteriaBuilder() method.

The next step is to obtain a jakarta.persistence.criteria.CriteriaQuery by using createQuery() method of CriteraBuilder.

The type of the criteria query (aka the <T>) indicates the expected types in the query result. This might be an entity, an Integer, or any other object.

The most common example is to select all the entity instances.

```
//create - select * from emp / from Emp
//1.create criteriabuilder instance
CriteriaBuilder cb = session.getCriteriaBuilder();
//2.create criteriaquery instance
CriteriaQuery<Emp> cquery = cb.createQuery(Emp.class);
//3.specify the root(table from which the records)
Root<Emp> root = cquery.from(Emp.class);
//4 specify the properties to be selected
cquery.select(root);
//5. prepare the query from the criteria
Query q = session.createQuery(cquery);
```

```
//create - select * from emp / from Emp
//1.create criteriabuilder instance
CriteriaBuilder cb = session.getCriteriaBuilder();
//2.create criteriaquery instance
CriteriaQuery<Emp> cquery = cb.createQuery(Emp.class);
//3.specify the root(table from which the records)
Root<Emp> root = cquery.from(Emp.class);
//4 specify the properties to be selected
cquery.select(root);
//5. prepare the query from the criteria
Query q = session.createQuery(cquery);
//6. get the result(list) from the query
List<Emp> emps = q.getResultList();
for(Emp e : emps)
   System.out.println(e.getEmpid()+" : "+e.getEname());
```

The simplest form of selecting an expression is selecting a particular attribute from an entity. But this expression might also represent an aggregation, a mathematical operation, etc. Following example shows selecting only few attributes from the entitiy.

```
//select empid,ename from EMP
//1.create criteriabuilder instance
CriteriaBuilder cb = session.getCriteriaBuilder();
//2.create criteriaquery instance
CriteriaQuery<Object[]> cquery = cb.createQuery(Object[].class);
//3.specify the root(table from which the records)
Root<Emp> root = cquery.from(Emp.class);
//4 specify the properties to be selected
cquery.multiselect(root.get("empid"),root.get("ename"));
//5. prepare the query from the criteria
Query q = session.createQuery(cquery);
//6. get the result(list) from the query
List<Object[]> emps = q.getResultList();
for(Object[] e : emps)
    System.out.println(e[0]+" : "+e[1]);
```

It is possible to select multiple attributes by using Tuple Criteria Queries

It is possible to select multiple attributes by using Tuple Criteria Queries

Assignments

1. Perform the update and delete operation on persistent entites. Update the name of the employee and delete an emp with a particular id.
2. Write HQL for retrieving all emp records as collection of Emp objects
3. Accept salary range from user and display only empid of those emps falling in that range using HQL.
4. Create named HQL for getting information about all the employees belonging to a perticular dept. Dept should be used as a query parameter
5. Write a native query for finding count of employees having salary greater then 10000
6. Write HQL for updating the salary of a given employee. Emp id and new salary should be used as query parameter.
7. Create criteria query for following
   a. Select all products
   b. Select information about a product using productid
   c. Select only prduct name and price from products