5 Hibernate Query Language

Objectives

At the end of this lesson, you will be able to:

- Introduction about the Hibernate Query Language, and how to leverage it to write database queries
- Prepare ourselves for cases where we need to write our own SQL by understanding how to accomplish its execution through Hibernate

Introduction

- So far we done the operations on single object (single row), here we will see modifications, updates on multiple rows of data (multiple objects) at a time.
- In hibernate we can perform the operations on a single row (or) multiple rows at a time, if we do operations on multiple rows at once, then we can call this as bulk operations.

Similar to SQL

- Object based. Instead of tables and columns, syntax includes objects and attributes

Understands inheritance

- Can issue a query using a superclass or interface

HQL

- HQL is the own query language of hibernate and it is used to perform bulk operations on hibernate programs
- An object oriented form of SQL is called HQL
- here we are going to replace table column names with POJO class variable names and table names with POJO class names in order to get HQL commands

Advantages Of HQL:

- HQL is database independent, means if we write any program using HQL commands then our program will be able to execute in all the databases with out doing any further changes to it
- HQL supports object oriented features like ***Inheritance***, ***polymorphism***, ***Associations***(Relation ships)
- HQL is initially given for selecting object from database and in hibernate 3.x we can do DML operations (insert, update…) too

Different Ways Of Construction HQL Select

- If we want to select a **Complete Object** from the database, we use POJO class reference in place of * while constructing the query
- In this case (select a complete object from the database) we can directly start our HQL command from, **from** key word

Example:

```
1  // In SQL
2  sql> select * from Product
3  Note: Product is the table name right....!!!
4
5  // In HQL
6  hql> select p from Product p
7       [ or ]
8       from Product p
9  Note: here p is the reference...!!
```

- If we want to load the **Partial Object** from the database that is only selective properties (selected columns) of an objects then we need to replace column names with POJO class variable names.

Example:

```
1  // In SQL
2  sql> select pid,pname from Product
3  Note: pid, pname are the columns Product is the table name right..!
4
5  // In HQL
6  hql> select p.productid,p.productName from Product p
7       [ or ]
8       from Product p ( we should not start from, from key word here because
9  we selecting the columns hope you are getting me )
10
11 Note: here p is the reference...!!
12          productid,productName are POJO variables
```

```
// In SQL
sql> select * from Product where pid=?
Note: Product is the table name right..!

// In HQL
hql> select p from Product p where p.productid=?
      [ or ]
      select p from Product p where p.productid=:prdId
      [ or ]
      from Product p where p.productid=?
      [ or ]
      from Product p where p.productid=:prdId

Note: Here p is the reference...!!
```

Procedure To Execute HQL Command:

- If we want to execute execute an HQL query on a database, we need to create a query object ” Query ” is an interface given in org.hibernate package
- In order to get query object, we need to call createQuery() method in the session Interface
- Query is an interface, QueryImpl is the implemented class we need to call list method for executing an HQL command on database, it returns java.util.List
- we need to use java.util.Iterator for iterating the List collection

Syntax:

```
1 Query qry = session.createQuery("--- HQL command ---");
2 List l = qry.list();
3 Iterator it = l.iterator();
4 while(it.hasNext())
5 {
6   Object o = it.next();
7   Product p = (Product)o;
8   ----- ------- ---------
9 }
```

Different Ways Of Executing HQL Commands

- We can execute our HQL command in 3 ways, like by selecting total object, partial object (more than one column), partial object (with single column).
  - **Case 1: [Selecting Complete Object]**
  - **Case 2: [Selecting Partial Object]**
  - **Case 3: [Selecting Partial Object]**

Case 1: [Selecting Complete Object]

- In this approach, we are going to select complete object from the database, so while iterating the collection, we need to typecast each object into our POJO class type only
- Internally hibernate converts each row selected from the table into an object of POJO class and hibernate stores all these POJO class objects into list so while iterating the collection, we typecast into POJO class type

Example:

```
1  Query qry = session.createQuery("-- our HQL command --");
2  List l =qry.list();
3  Iterator it = l.iterator();
4  while(it.hasNext())
5  {
6    Object o = it.next();
7    Project p = (Product)o;
8    ----- ------- ------
9  }
```

Case 2: [Selecting Partial Object]

- In this approach we are going to select partial object, (selected columns, i mean more than one column not single column)
- In this case hibernate internally stores the multiple column values of each row into an object array and stores these object arrays into List collection
- At the time of iterating the collection, we need to typecast the result into an object arrays

Example:

```
1 Query qry = session.createQuery("select p.pid,p.pname from Product p");
2 List l =qry.list();
3 Iterator it = l.iterator();
4 while(it.hasNext())
5 {
6   Object o[] = (Object o[])it.next();
7   System.out.println("-------");
8   ----- ------- ------
9 }
```

Case 3: [Selecting Partial Object]

- In this case we are going to select partial object with single column from the database
- In this case hibernate internally creates an object of that value type and stores all these objects into the list collection
- At the time of iterating the collection, we need to typecast into that object type only

Example:

```
1  Query qry = session.createQuery("select p.pid from Product p");
2  List l =qry.list();
3  Iterator it = l.iterator();
4  while(it.hasNext())
5  {
6    Integer i = (Integer)it.next();
7    System.out.println(i.intValue());
8    ----- ------- ------
9  }
```

Note: it.next() return type is always **Object**

Hibernate Query Language, Using HQL Select Query

- Let us see the program on HQL select command, which is going to cover complete object, partial object (More than one column), partial object (Single column)
  - Product.java (POJO class)
  - Product.hbm.xml (Xml mapping file)
  - hibernate.cfg.xml (Xml configuration file)
  - ForOurLogic.java (java file to write our hibernate logic)

Ref : Description on HQL.docx

Hibernate Query Language, Passing Runtime Values

- Now we will see, how to pass the values at time time while using the HQL select query, actually same concept for 3 cases.
  - Product.java (POJO class)
  - Product.hbm.xml (Xml mapping file)
  - hibernate.cfg.xml (Xml configuration file)
  - ForOurLogic.java (java file to write our hibernate logic)

Ref : Description on HQL

HQL Update,Delete Queries

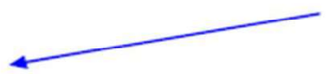
- So far we have been executed the programs on Hibernate Query Language (HQL) select only
- Now we will see the DML operations in HQL like insert, delete, update., you know some thing..? this delete, update query's are something similar to the select query only but insert query in Hibernate Query Language(HQL) is quite different
- Remember.., while we are working with DML operations in HQL we have to call executeUpdate(); to execute the query, which will returns one integer value after the execution

Ref Description on HQL.docx

- **Main class used for building and executing HQL**
- **Similar to a JDBC prepared statement**
  - Bind parameter values
    - setLong(), setString(), setDate() etc…
    - setParameter();
      - Generic way of binding variables
  - Submit Requests
    - list() ;
      - Execute, and return a collection of result objects
    - uniqueResult();
      - Execute and return a single result object
- **Created using the Hibernate Session**

Basic Object Queries

```
// return all CheckingAccounts
Query getAllCheckingAccounts =
  session.createQuery("from CheckingAccount");

List checkingAccounts = getAllCheckingAccounts.list();


// return all Account types
Query getAllAccounts =
  session.createQuery("from Account");

List accounts = getAllAccounts.list();


// return ALL object types
Query getAllAccounts =
  session.createQuery("from java.lang.Object");

List objects = getAllObjects.list();
```



Does not require a select clause, just the object class name

Binding Query Parameters

- **Position-based**
  - Just like JDBC
  - Set parameters in an ordered fashion, starting with zero
- **Name-based**
  - Use names as placeholders
  - Set parameters by name
- **Pros/Cons**
  - Position-based faster on executing variable substitution
  - Name-based doesn't require code changes if a new parameter gets added in the middle of the statement

Position-Based Parameters

```
// return all Accounts based on
// balance and creation date
String query = "from Account a where"
             + " a.balance > ?"
             + " and a.creationDate > ?";

// deprecated. for demo only
Date date = new Date(2008, 12, 01);

Query getAccounts = session.createQuery(query)
   .setLong(0, 1000)
   .setDate(1, date);

 List accounts = getAccounts.list();
```

Can alias objects, just like in SQL

Can set parameters in order, just like a JDBC PreparedStatement

Name-Based Parameters

```
// return all Accounts based on
// balance and creation date
String query = "from Account a where"
             + " a.balance > :someBalance"
             + " and a.creationDate > :someDate";

// deprecated. for demo only
Date date = new Date(2008, 12, 01);

// order doesn't matter
Query getAccounts = session.createQuery(query)
   .setDate("someDate", date)
   .setLong("someBalance", 1000);

 List accounts = getAccounts.list();
```

Setting Parameters Generically

```
// return all Accounts based on
// balance and creation date
String query = "from Account a where"
             + " a.balance > :someBalance"
             + " and a.creationDate > :someDate";

// deprecated. for demo only
Date date = new Date(2008, 12, 01);

// order doesn't matter.
// Temporal (time) values need to be specified
Query getAccounts = session.createQuery(query)
   .setParameter("someBalance", 1000)
   .setParameter("someDate", date, Hibernate.DATE);

List accounts = getAccounts.list();
```

Binding by Object

- **Name-based binding accepts an entire object for setting query parameters**
  - Placeholder names must match object attribute names
  - Hibernate uses reflection/java bean properties to map the attributes
- **Doesn't work with temporal data types**
  - Like Date

```
// return all Accounts based on
// balance and creation date
String query = "from EBill e where"
             + " e.balance > :balance"
             + " and e.ebillerId > :ebillerId";

EBill queryParams = new EBill();
queryParams.setBalance(1000);
queryParams.setEbillerId(1);

// this will use java bean properties/reflection
// to bind the variables
Query getEBills = session.createQuery(query)
    .setProperties(queryParams);

List accounts = getEBills.list();
```

Assume an object with attribute names that matched the placeholder names...

...pass that object in to set the parameter values

Pagination

- **Break up large result sets into smaller groups (pages)**
  - setFirstResults(int startRow);

    Set the starting record position

    Zero-based indexing
  - setMaxResults(int numberToGet);

    Set the number of records to retrieve
- **Keep track of current index in order to continue paging data through the data**

Pagination

```
// retrieve initial page, up to 50 records
Query getAccountsPage1 =
   session.createQuery("from Account")
      .setMaxResult(50);

...

// retrieve subsequent pages, passing
// in the first record to start with
Query getAccountsNextPage =
   session.createQuery("from Account")
      .setFirstResult(:startingIndex)
      .setMaxResult(50);
```

Setting Timeout

- **Set the time allowed for a specified query to execute**
  - setTimeout(int second);
  - Hibernate will throw an exception if limit is exceeded
- **Based on the JDBC timeout implementation**

Setting Timeout

```
try {

   // retrieve accounts, allow 30 seconds
   Query getAccounts =
      session.createQuery("from Account")
         .setTimeout(30);

   List accounts = getAccountsPage1.list();
}
catch (HibernateException) {
   ...
}
...
```

Setting Fetch Size

- **Optimization hint leveraged by the JDBC driver**
  - Not supported by all vendors, but if available, Hibernate will user this to optimize data retrieval
- **Used to indicate the number of records expected to be obtained in a read action**
  - If paging, should set to page size

Setting Fetch Size

```
// retrieve initial page, up to 50 records
Query getAccountsPage1 =
   session.createQuery("from Account")
      .setMaxResult(50)
      .setFetchSize(50);

...

// retrieve subsequent pages, passing
// in the first record to start with
Query getAccountsNextPage =
   session.createQuery("from Account")
      .setFirstResult(:startingIndex)
      .setMaxResult(50)
      .setFetchSize(50);
```

Adding Comments to Query

- **Developer provided comments included in the log along with the Hibernate SQL statement**
  - setComment(String comment);
  - Need to enable 'user_sql_comments' in the Hibernate configuration
- **Assists in distinguishing usergenerated queries vs. Hibernategenerated**
  - Also be used to explain query intention

Adding Comments to Query

```
// retrieve initial page, up to 50 records
Query getAccountsPage1 =
   session.createQuery("from Account")
      .setMaxResult(50)
      .setComment("Retrieving first page of
                   Account objects");

...

// retrieve subsequent pages, passing
// in the first record to start with
Query getAccountsNextPage =
   session.createQuery("from Account")
      .setFirstResult(:startingIndex)
      .setMaxResult(50)
      .setComment("Retrieving page: " + pageNum);
```

Externalizing Queries

- **Define queries in object mapping files**
- **Can be 'global' or included inside class definition**
  - If inside class definition, need to prefix with fully qualified class name when calling
- **Isolates the SQL statements**
  - Useful if you want to modify all queries

    Optimize queries

    Switch vendors

    May not require recompiling code

```
<hibernate-mapping>
  <class name="courses.hibernate.vo.Account"
         table="ACCOUNT">
    <id name="accountId" column="ACCOUNT_ID">
      <generator class="native" />
    </id>
    <property name="creationDate" column="CREATION_DATE"
              type="timestamp"    update="false" />
    <property name="accountType" column="ACCOUNT_TYPE"
              type="string"       update="false" />
    <property name="balance" column="BALANCE"
              type="double" />
  </class>
  <query name="getAllAccounts" fetch-size="50"
         comment="My account query" timeout="30">
     <![CDATA[from Account]]>
   </query>
</hibernate-mapping>
```

External: Inside Class

```
<hibernate-mapping>
  <class name="courses.hibernate.vo.Account"
         table="ACCOUNT">
    <id name="accountId" column="ACCOUNT_ID">
      <generator class="native" />
    </id>
    <property name="creationDate" column="CREATION_DATE"
              type="timestamp"     update="false" />
    <property name="accountType" column="ACCOUNT_TYPE"
              type="string"        update="false" />
    <property name="balance" column="BALANCE"
              type="double" />
    <query name="getAccountByBalance"  fetch-size="50"
           comment="Get account by balance"
           timeout="30">
           <![CDATA[from Account where
                    balance=:balance]]>
    </query>
  </class>
</hibernate-mapping>
```

Calling Externalizing Queries

```
// globally named query
Query getAccounts =
   session.getNamedQuery("getAllAccounts")

List accounts = getAccounts.list();

...

// defined within class definition
Query getAccountByBalance =
   session.getNamedQuery(
     "courses.hibernate.vo.Account.getAccountByBalance")
   .setParameter("someBalance", 1000)

List accounts = getAccountByBalance.list();
```

Specifying Order

```
...

Query getAccounts =
   session.createQuery("from Account
  order by balance desc, creationDate
  asc")

List accounts = getAccounts.list();

...
```

Using SQL/Database Functions

```
Query getAccountOwners =
   session.createQuery(
   "select upper(lastName),
           lower(firstName),
           sysdate
           from AccountOwner");
```

HQL Aggregation Functions

- **Functions that operate against groups of resulting records**
- **Supported functions include:**
  - count();
  - min();
  - max();
  - sum();
  - avg();

Count Function

```
Query countQuery =
   session.createQuery(
      "select count(ao) from
       AccountOwner ao "

long cnt =
  (Long)countQuery.uniqueResult();
```

Min, Max, and Avg Functions

```
Query accountStatsQuery =
   session.createQuery(
      "select min(a.balance), max(a.balance),
              avg(a.balance) from Account a");

List listOfRowValues = accountStatsQuery.list();

for (Object[] singleRowValues : listOfRowValues) {
   // pull off the values
   double min = (Double)singleRowValues[0];
   double max = (Double)singleRowValues[1];
   double avg = (Double)singleRowValues[2];
}
```

Group By and Having

- **Group subsets of returned results**
  - ‘group by’ clause, just like SQL
- **Restrict groups returned**
  - ‘having’ clause, also like SQL

Group By Aggregation

```
Query avgTxAmountPerAccountQuery =
  session.createQuery(
    "select atx.account.accountId,
            avg(atx.amount)
     from
            AccountTransaction atx
     group by
            atx.account.accountId");

List listOfRowValues =
  avgTxAmountPerAccountQuery.list();
```

Having Aggregation Restrictions

```
Query avgTxAmountPerAccountQuery =
  session.createQuery(
    "select atx.account.accountId,
            avg(atx.amount)
     from
            AccountTransaction atx
     group by
            atx.account.accountId
     having
            count(atx) > 20");
List listOfRowValues =
  avgTxAmountPerAccountQuery.list();
```

Native SQL Queries

- **Write traditional SQL statements and execute them through the Hibernate engine**
  - Hibernate can handle the result set
- **Needed for very complicated queries or taking advantage of some database features, like hints**

Returning Scalar Values – All Columns

```
Query getEBills =
   session.createSQLQuery("SELECT * FROM EBILL");

List listOfRowValues =
     getEBills.list();

for (Object[] singleRowValues : listOfRowValues) {
   // returned in the order on the table
   long id = (long)singleRowValues[0];
   double balance = (balance)singleRowValues[1];
   ...
}
```

Return List of Objects

```
Query getEBills =
   session.createSQLQuery(
      "SELECT * FROM EBill")
   .addEntity(EBill.class);

List ebills =
     getEBills.list();
```

Hibernate VS JPA

JPA	Hibernate
Entity Classes	Persistent Classes
EntityManagerFactory	SessionFactory
EntityManager	Session
Persistence	Configuration
EntityTransaction	Transaction
Query	Query
Persistence Unit	Hibernate Config

Hibernate Configuration

```
<!-- in the Hiberante.cfg.xml file -->
<session-factory>
   <property name="hibernate.hbm2ddl.auto">
      create|create-drop
   </property>
   ...
</sessionFactory>
```

```
// programmatically
Configuration cfg =
   new Configuration().configure();
SchemaUpdate schemaUpdate =
   new SchemaUpdate(cfg);
schemaUpdate.execute();
```

Summary

In this lesson, we have learned

- **Learned how to use HQL to execute queries by binding dynamic parameters and settings**
  - Named and position based binding
  - Paging, fetch-size, timeout, comments
- **Saw how to externalize our queries for maintenance purposes**
- – In mapping files globally, or within class definitions
- **Aggregations:**
  - Grouping and Having
- **Native SQL**
  - Returning both scalar and object results