**Hibernate Query Language – HQL**

References <https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html>

HQL or Hibernate Query Language is the object-oriented query language of Hibernate Framework. HQL is very similar to SQL except that we use Objects instead of table names, that makes it more close to object oriented programming.

**HQL and Case Sensitivity**: HQL is case-insensitive except for java class and variable names. So SeLeCT is the same as sELEct is the same as SELECT, but com.synergetics.model.Employee is not same as com.synergetics.model.EMPLOYEE.

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 **order 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.**
8. Let’s look at a simple example of using HQL in our program.

1. HQL Syntax

1.1. **Update** Operation

1.2. **Delete** Operation

1.3. **Insert** Operation

1.4. **Select** Operation

2. The **from Clause and Aliases**

3. The **select Clause and Projection**

4. Using **Named Parameters**

5. **Paging** Through the Result Set

6. Obtaining a **Unique** Result

7. **Sorting** Results with the **order by** Clause

8. **Associations**

9. **Aggregate** Methods

10. **Named Queries**

11. Using **Native SQL**

12. Enable **Logging** and **Commenting**

## 1. HQL Syntax

HQL syntax is defined as an [**ANTLR**](https://en.wikipedia.org/wiki/ANTLR) grammar. The grammar files are included in the grammar directory of the Hibernate core download. (ANTLR is a tool for building language parsers). Lets outline the syntax for the four fundamental CRUD operations here:

#### 1.1. HQL Update Statement

UPDATE alters the details of existing objects in the database. In-memory entities, managed or not, will not be updated to reflect changes resulting from issuing UPDATE statements. Here’s the syntax of the UPDATE statement:

|  |
| --- |
| hql update statement syntax |
| UPDATE [VERSIONED]     [FROM] path [[AS] alias] [, ...]     SET property = value [, ...]     [WHERE logicalExpression] |

* path – fully qualified name of the entity or entities
* alias – used to abbreviate references to specific entities or their properties, and must be used when property names in the query would otherwise be ambiguous.
* VERSIONED – means that the update will update time stamps, if any, that are part of the entity being updated.
* property – names of properties of entities listed in the FROM path.
* logicalExpression – a where clause.

An example of the update in action might look like this. In this example, we are updating employee data with **hql update query multiple columns**.

|  |
| --- |
| hql update statement example |
| **Query** query=session.createQuery("update **Employee** set age=:age where name=:name");//Employee is the name of the Object class not the name of the table  query.setInteger("age", 32);  query.setString("name", "Lokesh Gupta");  int modifications=query.executeUpdate(); |

#### 1.2. HQL Delete Statement

DELETE removes the details of existing objects from the database. In-memory entities will not be updated to reflect changes resulting from DELETE statements. This also means that Hibernate’s cascade rules will not be followed for deletions carried out using HQL. However, if you have specified cascading deletes at the database level (either directly or through Hibernate, using the @OnDelete annotation), the database will still remove the child rows.

Here’s the syntax of the DELETE statement:

|  |
| --- |
| hql delete statement syntax |
| DELETE     [FROM] path [[AS] alias]     [WHERE logicalExpression] |

In practice, deletes might look like this:

|  |
| --- |
| hql delete statement example |
| Query query=session.createQuery("delete from Account where accountstatus=:status");  query.setString("status", "purged");  int rowsDeleted=query.executeUpdate(); |

#### 1.3. HQL Insert Statement

An HQL INSERT **cannot be used to directly insert arbitrary entities**—it can only be used to insert entities constructed from information obtained from SELECT queries (unlike ordinary SQL, in which an INSERT command can be used to insert arbitrary data into a table, as well as insert values selected from other tables).

Here’s the syntax of the INSERT statement:

|  |
| --- |
| hql insert statement example |
| INSERT     INTO path ( property [, ...])     select |

The name of an entity is path. The property names are the names of properties of entities listed in the FROM path of the incorporated SELECT query. The select query is an HQL SELECT query (as described in the next section).

As this HQL statement can only use data provided by an HQL select, its application can be limited. An example of copying users to a purged table before actually purging them might look like this:

|  |
| --- |
| hql insert statement example |
| Query query=session.createQuery("insert into purged\_accounts(id, code, status) "+      "select id, code, status from account where status=:status");  query.setString("status", "purged");  int rowsCopied=query.executeUpdate(); |

#### 1.4. HQL Select Statement

An HQL SELECT is used to query the database for classes and their properties. Here’s the syntax of the SELECT statement:

|  |
| --- |
| hql select statement example |
| [SELECT [DISTINCT] property [, ...]]     FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]     WHERE logicalExpression     GROUP BY property [, ...]     HAVING logicalExpression     ORDER BY property [ASC | DESC] [, ...] |

The fully qualified name of an entity is path. The alias names may be used to abbreviate references to specific entities or their properties, and must be used when property names used in the query would otherwise be ambiguous.

The property names are the names of properties of entities listed in the ***FROM*** path.

If ***FETCH ALL PROPERTIES*** is used, then lazy loading semantics will be ignored, and all the immediate properties of the retrieved object(s) will be actively loaded (this does not apply recursively).

WHERE is used to create **hql select query with where clause**.

When the properties listed consist only of the names of aliases in the FROM clause, the SELECT clause can be omitted in HQL. If we are using the JPA with JPQL, one of the differences between HQL and JPQL is that the SELECT clause is required in JPQL.

## 2. HQL – from clause and aliases

The most important feature in HQL to note is the **alias**. Hibernate allows us to assign aliases to the classes in our query with the as clause. Use the aliases to refer back to the class inside the query.

Take for example:

|  |
| --- |
| hql alias example |
| from Product as p    //or    from Product as product |

The 'as' keyword is optional. We can also specify the alias directly after the class name as follows:

|  |
| --- |
| hql alias example |
| from Product product |

If we need to fully qualify a class name in HQL, just specify the package and class name. Hibernate will take care of most of this behind the scenes, so we really need this only if we have classes with duplicate names in our application. If we have to do this in Hibernate, use syntax such as the following:

|  |
| --- |
| from com.howtodoinjava.geo.usa.Product |

The from clause is very basic and useful for working directly with objects. However, if you want to work with the object’s properties without loading the full objects into memory, you must use the select clause as explained in next section.

## 3. HQL select clause and projection

The select clause provides more control over the result set than the from clause. If you want to obtain the properties of objects in the result set, use the select clause. For instance, we could run a projection query on the products in the database that only returned the names, instead of loading the full object into memory, as follows:

|  |
| --- |
| select product.name from Product product |

The result set for this query will contain a List of Java String objects. Additionally, we can retrieve the prices and the names for each product in the database, like so:

|  |
| --- |
| select product.name, product.price from Product product |

If you’re only interested in a few properties, this approach can allow you to reduce network traffic to the database server and save memory on the application’s machine.

## 4. HQL Named Parameters

Hibernate supports named parameters in its HQL queries. This makes writing queries that accept input from the user easy—and you do not have to defend against SQL injection attacks.

When using JDBC query parameters, any time you add, change, or delete parts of the SQL statement, you need to update your Java code that sets its parameters, because the parameters are indexed based on the order in which they appear in the statement. Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters around in the query.

The simplest example of named parameters uses regular SQL types for the parameters:

|  |
| --- |
| String hql = "from Product where price > :price";//named parameter, to provide value at runtime  Query query = session.createQuery(hql);//after creating the query  //we need to set the named parameter value to the query  query.setDouble("price",25.0);  //then list the query  List results = query.list(); |

## 5. HQL – Paging Through the ResultSet

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the Query interface for paging: setFirstResult() and setMaxResults(). The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the setMaxResults() method. Your HQL is unchanged—you need only to modify the Java code that executes the query.

|  |
| --- |
| Query query = session.createQuery("from Product");  query.setFirstResult(1);  query.setMaxResults(2);  List results = query.list();  displayProductsList(results); |

If you turn on SQL logging, you can see which SQL commands Hibernate uses for pagination. For the open-source HSQLDB database, Hibernate uses top and limit. Microsoft SQL Server does not support the limit command, so Hibernate uses only the top command. If your application is having performance problems with pagination, this can be very helpful for debugging.

If you only have one result in your HQL result set, Hibernate has a shortcut method for obtaining just that object as discussed next.

## 6. HQL – Get a Unique Result

HQL’s Query interface provides a uniqueResult() method for obtaining just one object from an HQL query. Although your query may yield only one object, you may also use the uniqueResult() method with other result sets if you limit the results to just the first result. You could use the setMaxResults() method discussed in the previous section.

The uniqueResult() method on the Query object returns a single object, or null if there are zero results. If there is more than one result, then the uniqueResult() method throws a NonUniqueResultException.

|  |
| --- |
| String hql = "from Product where price>25.0";  Query query = session.createQuery(hql);  query.setMaxResults(1);  Product product = (Product) query.uniqueResult(); |

## 7. HQL – Sorting Results with the ‘order by’ clause

To sort your HQL query’s results, you will need to use the **order by clause**. You can order the results by any property on the objects in the result set: either ascending (asc) or descending (desc). You can use ordering on more than one property in the query, if you need to. A typical HQL query for sorting results looks like this:

|  |
| --- |
| from Product p where p.price>25.0 order by p.price desc |

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas. For instance, you could sort by product price and the supplier’s name, as follows:

|  |
| --- |
| from Product p order by p.supplier.name asc, p.price asc  [/ql]    <a name="associations"></a>  <h2>8. HQL associations</h2>    Associations allow you to use <strong>more than one class in an HQL query</strong>, just as SQL allows you to use joins between tables in a relational database. Youadd an association to an HQL query with the join clause. Hibernate supports five different types of joins: <strong>inner join, cross join, left outer join, rightouter join, and full outer join</strong>.    If you use cross join, just specify both classes in the from clause (from Product p, Supplier s). For the other joins, use a join clause after the from clause. Specify the type of join, the object property to join on, and an alias for the other class.    You can use inner join to obtain the supplier for each product, and then retrieve the supplier name, product name, and product price, as so:      select s.name, p.name, p.price from Product p inner join p.supplier as s |

You can retrieve the objects using similar syntax:

|  |
| --- |
| from Product p inner join p.supplier as s |

## 9.HQL Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects. You may use the count(\*) syntax to count all the objects in the result set, or **count(product.name)** to count the number of objects in the result set with a name property. Here is an example using the count(\*) method to count all products:

|  |
| --- |
| select count(\*) from Product product |

The aggregate functions available through HQL include the following:

1. avg(property name): The average of a property’s value
2. count(property name or \*): The number of times a property occurs in the results
3. max(property name): The maximum value of the property values
4. min(property name): The minimum value of the property values
5. sum(property name): The sum total of the property values

## 10. HQL Named Queries

Named queries are created via class-level annotations on entities; normally, the queries apply to the entity in whose source file they occur, but there’s no absolute requirement for this to be true.

Named queries are created with the @NamedQueries annotation, which contains an array of @NamedQuery sets; each has a query and a name.

An example of named queries may look like this:

|  |
| --- |
| @NamedQueries({          @NamedQuery(name = "supplier.findAll", query = "from Supplier s"),          @NamedQuery(name = "supplier.findByName",                  query = "from Supplier s where s.name=:name"),  }) |

Executing above named query is even simpler.

|  |
| --- |
| Query query = session.getNamedQuery("supplier.findAll");  List<Supplier> suppliers = query.list(); |

Read More – [Hibernate named query tutorial](https://howtodoinjava.com/hibernate/hibernate-named-query-tutorial/)

## 11. HQL – Native SQL

Although you should probably use HQL whenever possible, Hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL. Another reason is that you may want to call stored procedures from your Hibernate application.

You can modify your SQL statements to make them work with Hibernate’s ORM layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with {objectname.\*}, or you can specify the aliases directly with {objectname.property}.

Hibernate uses the mappings to translate your object property names into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support. You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables, so that you select the right properties off the table.

Underlying Hibernate’s native SQL support is the org.hibernate.SQLQuery interface, which extends the org.hibernate.Query interface. Your application will create a native SQL query from the session with the createSQLQuery() method on the Session interface.

|  |
| --- |
| public SQLQuery createSQLQuery(String queryString) throws HibernateException |

After you pass a string containing the SQL query to the createSQLQuery() method, you should associate the SQL result with an existing Hibernate entity, a join, or a scalar result. The SQLQuery interface has addEntity(), addJoin(), and addScalar() methods.

#### 11.1. Hibernate sql query example

Using native SQL with scalar results is the simplest way to get started with native SQL. Sample Java code looks like this:

|  |
| --- |
| String sql = "select avg(product.price) as avgPrice from Product product";  SQLQuery query = session.createSQLQuery(sql);  query.addScalar("avgPrice",Hibernate.DOUBLE);  List results = query.list(); |

A bit more complicated than the previous example is the **native SQL** that returns a result set of objects. In this case, we will need to map an entity to the SQL query.

|  |
| --- |
| String sql = "select {supplier.\*} from Supplier supplier";  SQLQuery query = session.createSQLQuery(sql);  query.addEntity("supplier", Supplier.class);  List results = query.list();    //Hibernate modifies the SQL and executes the following command against the database:    select Supplier.id as id0\_, Supplier.name as name2\_0\_ from Supplier supplier |

## 12. HQL – Enable Logs and Comments

Hibernate can output the underlying SQL behind your HQL queries into your application’s log file. This is especially useful if the HQL query does not give the results you expect, or if the query takes longer than you wanted. This is not a feature you will have to use frequently, but it is useful should you have to turn to your database administrators for help in tuning your Hibernate application.

#### 12.1. HQL Logs

The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the “**show\_sql**” property. Set this property to true in your **hibernate.cfg.xml**configuration file and Hibernate will output the SQL into the logs. When you look in your application’s output for the Hibernate SQL statements, they will be prefixed with “Hibernate:”.

If you turn your log4j logging up to debug for the Hibernate classes, you will see SQL statements in your log files, along with lots of information about how Hibernate parsed your HQL query and translated it into SQL.

#### 12.2. HQL Comments

Tracing your HQL statements through to the generated SQL can be difficult, so Hibernate provides a commenting facility on the Query object that lets you apply a comment to a specific query. The Query interface has a setComment() method that takes a String object as an argument, as follows:

|  |
| --- |
| public Query setComment(String comment) |

Hibernate will not add comments to your SQL statements without some additional configuration, even if you use the setComment() method. You will also need to set a Hibernate property, **hibernate.use\_sql\_comments**, to true in your Hibernate configuration.

If you set this property but do not set a comment on the query programatically, Hibernate will include the HQL used to generate the SQL call in the comment. I find this to be very useful for debugging HQL.

Use commenting to identify the SQL output in your application’s logs if SQL logging is enabled.

### HQL Example Database Setup

I am using MySQL database for my example, below script will create two tables Employee and Address. They have one-to-one mapping and I am inserting some demo data for my example.

CREATE TABLE `Employee` (

`emp\_id` int(11) unsigned NOT NULL AUTO\_INCREMENT,

`emp\_name` varchar(20) NOT NULL,

`emp\_salary` double(10,0) NOT NULL DEFAULT '0',

PRIMARY KEY (`emp\_id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `Address` (

`emp\_id` int(11) unsigned NOT NULL,

`address\_line1` varchar(50) NOT NULL DEFAULT '',

`zipcode` varchar(10) DEFAULT NULL,

`city` varchar(20) DEFAULT NULL,

PRIMARY KEY (`emp\_id`),

CONSTRAINT `emp\_fk\_1` FOREIGN KEY (`emp\_id`) REFERENCES `Employee` (`emp\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `Employee` (`emp\_id`, `emp\_name`, `emp\_salary`)

VALUES

(1, 'Smita', 100),

(2, 'David', 200),

(3, 'Lisa', 300),

(4, 'Jack', 400);

INSERT INTO `Address` (`emp\_id`, `address\_line1`, `zipcode`, `city`)

VALUES

(1, 'Albany Dr', '95129', 'San Jose'),

(2, 'Arques Ave', '95051', 'Santa Clara'),

(3, 'BTM 1st Stage', '560100', 'Mumbai'),

(4, 'City Centre', '100100', 'New Delhi');

commit;

Create a maven project in Eclipse or the IDE you are using, our final project will look like below image.

### Hibernate Maven Dependencies

Our final pom.xml contains dependencies for Hibernate and MySQL driver.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.synergetics.hibernate</groupId>

<artifactId>HQLExample</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>4.3.5.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.0.5</version>

</dependency>

</dependencies>

</project>

### Hibernate Configuration XML

Our hibernate configuration xml file contains database connection related properties and mapping classes. I will be using annotations for Hibernate mapping.

hibernate.cfg.xml code:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.password">root</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost/test</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="hibernate.current\_session\_context\_class">thread</property>

<property name="hibernate.show\_sql">true</property>

<mapping class="com.synergetics.hibernate.model.Employee"/>

<mapping class="com.synergetics.hibernate.model.Address"/>

</session-factory>

</hibernate-configuration>

### Hibernate SessionFactory Utility class

We have a utility class to configure hibernate SessionFactory.

package com.synergetics.hibernate.util;

import org.hibernate.SessionFactory;

import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

import org.hibernate.cfg.Configuration;

import org.hibernate.service.ServiceRegistry;

public class HibernateUtil {

private static SessionFactory sessionFactory;

private static SessionFactory buildSessionFactory() {

try {

// Create the SessionFactory from hibernate.cfg.xml

Configuration configuration = new Configuration();

configuration.configure("hibernate.cfg.xml");

System.out.println("Hibernate Configuration loaded");

ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();

System.out.println("Hibernate serviceRegistry created");

SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);

return sessionFactory;

}

catch (Throwable ex) {

System.err.println("Initial SessionFactory creation failed." + ex);

ex.printStackTrace();

throw new ExceptionInInitializerError(ex);

}

}

public static SessionFactory getSessionFactory() {

if(sessionFactory == null) sessionFactory = buildSessionFactory();

return sessionFactory;

}

}

### Model Classes with Annotation based mapping

Our model classes with JPA annotations looks like below.

package com.synergetics.hibernate.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.Table;

import org.hibernate.annotations.Cascade;

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "emp\_id")

private long id;

@Column(name = "emp\_name")

private String name;

@Column(name = "emp\_salary")

private double salary;

@OneToOne(mappedBy = "employee")

@Cascade(value = org.hibernate.annotations.CascadeType.ALL)

private Address address;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

}

package com.synergetics.hibernate.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.PrimaryKeyJoinColumn;

import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

import org.hibernate.annotations.Parameter;

@Entity

@Table(name = "ADDRESS")

public class Address {

@Id

@Column(name = "emp\_id", unique = true, nullable = false)

@GeneratedValue(generator = "gen")

@GenericGenerator(name = "gen", strategy = "foreign",

parameters = { @Parameter(name = "property", value = "employee") })

private long id;

@Column(name = "address\_line1")

private String addressLine1;

@Column(name = "zipcode")

private String zipcode;

@Column(name = "city")

private String city;

@OneToOne

@PrimaryKeyJoinColumn

private Employee employee;

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getAddressLine1() {

return addressLine1;

}

public void setAddressLine1(String addressLine1) {

this.addressLine1 = addressLine1;

}

public String getZipcode() {

return zipcode;

}

public void setZipcode(String zipcode) {

this.zipcode = zipcode;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public Employee getEmployee() {

return employee;

}

public void setEmployee(Employee employee) {

this.employee = employee;

}

}

### HQL Example Test Class

Let’s see how to use HQL in java programs.

package com.synergetics.hibernate.main;

import java.util.Arrays;

import java.util.List;

import org.hibernate.Query;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import com.synergetics.hibernate.model.Employee;

import com.synergetics.hibernate.util.HibernateUtil;

public class HQLExamples {

@SuppressWarnings("unchecked")

public static void main(String[] args) {

//Prep work

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Session session = sessionFactory.getCurrentSession();

//HQL example - Get All Employees

Transaction tx = session.beginTransaction();

Query query = session.createQuery("from Employee");

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

for(Employee emp : empList){

System.out.println("List of Employees::"+emp.getId()+","+emp.getAddress().getCity());

}

//HQL example - Get Employee with id

query = session.createQuery("from Employee where id= :id");

query.setLong("id", 3);

Employee emp = (Employee) query.uniqueResult();

System.out.println("Employee Name="+emp.getName()+", City="+emp.getAddress().getCity());

//HQL pagination example

query = session.createQuery("from Employee");

query.setFirstResult(0); //starts with 0

query.setFetchSize(2);

empList = query.list();

for(Employee emp4 : empList){

System.out.println("Paginated Employees::"+emp4.getId()+","+emp4.getAddress().getCity());

}

//HQL Update Employee

query = session.createQuery("update Employee set name= :name where id= :id");

query.setParameter("name", "Smita Kumar");

query.setLong("id", 1);

int result = query.executeUpdate();

System.out.println("Employee Update Status="+result);

//HQL Delete Employee, we need to take care of foreign key constraints too

query = session.createQuery("delete from Address where id= :id");

query.setLong("id", 4);

result = query.executeUpdate();

System.out.println("Address Delete Status="+result);

query = session.createQuery("delete from Employee where id= :id");

query.setLong("id", 4);

result = query.executeUpdate();

System.out.println("Employee Delete Status="+result);

//HQL Aggregate function examples

query = session.createQuery("select sum(salary) from Employee");

double sumSalary = (Double) query.uniqueResult();

System.out.println("Sum of all Salaries= "+sumSalary);

//HQL join examples

query = session.createQuery("select e.name, a.city from Employee e "

+ "INNER JOIN e.address a");

List<Object[]> list = query.list();

for(Object[] arr : list){

System.out.println(Arrays.toString(arr));

}

//HQL group by and like example

query = session.createQuery("select e.name, sum(e.salary), count(e)"

+ " from Employee e where e.name like '%i%' group by e.name");

List<Object[]> groupList = query.list();

for(Object[] arr : groupList){

System.out.println(Arrays.toString(arr));

}

//HQL order by example

query = session.createQuery("from Employee e order by e.id desc");

empList = query.list();

for(Employee emp3 : empList){

System.out.println("ID Desc Order Employee::"+emp3.getId()+","+emp3.getAddress().getCity());

}

//rolling back to save the test data

tx.rollback();

//closing hibernate resources

sessionFactory.close();

}

}

Notice that I am using HQL for Select, Update and Delete operations. It also shows how to use HQL Join and HQL Aggregate functions.

When I run above hql example program, we get following output.

May 22, 2017 1:55:37 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>

INFO: HCANN000001: Hibernate Commons Annotations {4.0.4.Final}

May 22, 2017 1:55:37 PM org.hibernate.Version logVersion

INFO: HHH000412: Hibernate Core {4.3.5.Final}

May 22, 2017 1:55:37 PM org.hibernate.cfg.Environment <clinit>

INFO: HHH000206: hibernate.properties not found

May 22, 2017 1:55:37 PM org.hibernate.cfg.Environment buildBytecodeProvider

INFO: HHH000021: Bytecode provider name : javassist

May 22, 2017 1:55:37 PM org.hibernate.cfg.Configuration configure

INFO: HHH000043: Configuring from resource: hibernate.cfg.xml

May 22, 2017 1:55:37 PM org.hibernate.cfg.Configuration getConfigurationInputStream

INFO: HHH000040: Configuration resource: hibernate.cfg.xml

May 22, 2017 1:55:37 PM org.hibernate.cfg.Configuration doConfigure

INFO: HHH000041: Configured SessionFactory: null

Hibernate Configuration loaded

Hibernate serviceRegistry created

May 22, 2017 1:55:37 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure

WARN: HHH000402: Using Hibernate built-in connection pool (not for production use!)

May 22, 2017 1:55:37 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator

INFO: HHH000401: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost/TestDB]

May 22, 2017 1:55:37 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator

INFO: HHH000046: Connection properties: {user=Smita, password=\*\*\*\*}

May 22, 2017 1:55:37 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator

INFO: HHH000006: Autocommit mode: false

May 22, 2017 1:55:37 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure

INFO: HHH000115: Hibernate connection pool size: 20 (min=1)

May 22, 2017 1:55:37 PM org.hibernate.dialect.Dialect <init>

INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect

May 22, 2017 1:55:37 PM org.hibernate.engine.jdbc.internal.LobCreatorBuilder useContextualLobCreation

INFO: HHH000423: Disabling contextual LOB creation as JDBC driver reported JDBC version [3] less than 4

May 22, 2017 1:55:38 PM org.hibernate.engine.transaction.internal.TransactionFactoryInitiator initiateService

INFO: HHH000399: Using default transaction strategy (direct JDBC transactions)

May 22, 2017 1:55:38 PM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory <init>

INFO: HHH000397: Using ASTQueryTranslatorFactory

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_, employee0\_.emp\_name as emp\_name2\_1\_, employee0\_.emp\_salary as emp\_sala3\_1\_ from EMPLOYEE employee0\_

Hibernate: select address0\_.emp\_id as emp\_id1\_0\_0\_, address0\_.address\_line1 as address\_2\_0\_0\_, address0\_.city as city3\_0\_0\_, address0\_.zipcode as zipcode4\_0\_0\_, employee1\_.emp\_id as emp\_id1\_1\_1\_, employee1\_.emp\_name as emp\_name2\_1\_1\_, employee1\_.emp\_salary as emp\_sala3\_1\_1\_ from ADDRESS address0\_ left outer join EMPLOYEE employee1\_ on address0\_.emp\_id=employee1\_.emp\_id where address0\_.emp\_id=?

Hibernate: select address0\_.emp\_id as emp\_id1\_0\_0\_, address0\_.address\_line1 as address\_2\_0\_0\_, address0\_.city as city3\_0\_0\_, address0\_.zipcode as zipcode4\_0\_0\_, employee1\_.emp\_id as emp\_id1\_1\_1\_, employee1\_.emp\_name as emp\_name2\_1\_1\_, employee1\_.emp\_salary as emp\_sala3\_1\_1\_ from ADDRESS address0\_ left outer join EMPLOYEE employee1\_ on address0\_.emp\_id=employee1\_.emp\_id where address0\_.emp\_id=?

Hibernate: select address0\_.emp\_id as emp\_id1\_0\_0\_, address0\_.address\_line1 as address\_2\_0\_0\_, address0\_.city as city3\_0\_0\_, address0\_.zipcode as zipcode4\_0\_0\_, employee1\_.emp\_id as emp\_id1\_1\_1\_, employee1\_.emp\_name as emp\_name2\_1\_1\_, employee1\_.emp\_salary as emp\_sala3\_1\_1\_ from ADDRESS address0\_ left outer join EMPLOYEE employee1\_ on address0\_.emp\_id=employee1\_.emp\_id where address0\_.emp\_id=?

Hibernate: select address0\_.emp\_id as emp\_id1\_0\_0\_, address0\_.address\_line1 as address\_2\_0\_0\_, address0\_.city as city3\_0\_0\_, address0\_.zipcode as zipcode4\_0\_0\_, employee1\_.emp\_id as emp\_id1\_1\_1\_, employee1\_.emp\_name as emp\_name2\_1\_1\_, employee1\_.emp\_salary as emp\_sala3\_1\_1\_ from ADDRESS address0\_ left outer join EMPLOYEE employee1\_ on address0\_.emp\_id=employee1\_.emp\_id where address0\_.emp\_id=?

List of Employees::1,San Jose

List of Employees::2,Santa Clara

List of Employees::3,Mumbai

List of Employees::4,New Delhi

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_, employee0\_.emp\_name as emp\_name2\_1\_, employee0\_.emp\_salary as emp\_sala3\_1\_ from EMPLOYEE employee0\_ where employee0\_.emp\_id=?

Employee Name=Lisa, City=Mumbai

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_, employee0\_.emp\_name as emp\_name2\_1\_, employee0\_.emp\_salary as emp\_sala3\_1\_ from EMPLOYEE employee0\_

Paginated Employees::1,San Jose

Paginated Employees::2,Santa Clara

Paginated Employees::3,Mumbai

Paginated Employees::4,New Delhi

Hibernate: update EMPLOYEE set emp\_name=? where emp\_id=?

Employee Update Status=1

Hibernate: delete from ADDRESS where emp\_id=?

Address Delete Status=1

Hibernate: delete from EMPLOYEE where emp\_id=?

Employee Delete Status=1

Hibernate: select sum(employee0\_.emp\_salary) as col\_0\_0\_ from EMPLOYEE employee0\_

Sum of all Salaries= 600.0

Hibernate: select employee0\_.emp\_name as col\_0\_0\_, address1\_.city as col\_1\_0\_ from EMPLOYEE employee0\_ inner join ADDRESS address1\_ on employee0\_.emp\_id=address1\_.emp\_id

[Smita Kumar, San Jose]

[David, Santa Clara]

[Lisa, Mumbai]

Hibernate: select employee0\_.emp\_name as col\_0\_0\_, sum(employee0\_.emp\_salary) as col\_1\_0\_, count(employee0\_.emp\_id) as col\_2\_0\_ from EMPLOYEE employee0\_ where employee0\_.emp\_name like '%i%' group by employee0\_.emp\_name

[David, 200.0, 1]

[Lisa, 300.0, 1]

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_, employee0\_.emp\_name as emp\_name2\_1\_, employee0\_.emp\_salary as emp\_sala3\_1\_ from EMPLOYEE employee0\_ order by employee0\_.emp\_id desc

ID Desc Order Employee::3,Mumbai

ID Desc Order Employee::2,Santa Clara

ID Desc Order Employee::1,San Jose

May 22, 2017 1:55:38 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop

INFO: HHH000030: Cleaning up connection pool [jdbc:mysql://localhost/TestDB]

Notice that once delete operation is performed, further operations are not showing that record data (sum of salary is 600). However I am rolling back the transaction, so the data in table will remain unchanged. Change the code to commit the transaction and it will be reflected in the database tables.