Hibernate Query Examples –

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. It is a superset of the JPQL, the Java Persistence Query Language; a JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries. HQL is a language with its own syntax and grammar. It is written as strings, like “ from Product p “. your HQL queries are translated by Hibernate into conventional SQL queries; Hibernate also provides an API that allows you to directly issue SQL queries as well.

Table Of Contents

Basic HQL Syntax

Update Operation

Delete Operation

Insert Operation

Select Operation

The from Clause and Aliases

The select Clause and Projection

Using Named Parameters

Paging Through the Result Set

Obtaining a Unique Result

Sorting Results with the order by Clause

Associations

Aggregate Methods

Named Queries

Using Native SQL

Enable Logging and Commenting

Let’s discuss each and every item in more detail starting from basic stuff to more complex concepts.

Basic HQL Syntax

HQL’s syntax is defined as an 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 HQL operations here:

Update Operation

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:

Where fully qualified name of the entity or entities is “ path “. The “ alias ” names may be 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.

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

“ logicalExpression ” is just a where clause.

An example of the update in action might look like this:

Delete Operation

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:

In practice, deletes might look like this:

Insert Operation

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:

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:

Select Operation

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

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).

The from Clause and Aliases

The most important feature in HQL to note is the alias. Hibernate allows you to assign aliases to the classes in your query with the as clause. Use the aliases to refer back to the class inside the query. Take for example:

The as keyword is optional—you can also specify the alias directly after the class name, as follows:

If you 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 you really need this only if you have classes with duplicate names in your application. If you have to do this in Hibernate, use syntax such as the following:

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.

The 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:

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:

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.

Using 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:

Paging Through the Result Set

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.

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.

Obtaining 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 .

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:

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:

Associations

Associations allow you to use more than one class in an HQL query, just as SQL allows you to use joins between tables in a relational database. You add an association to an HQL query with the join clause. Hibernate supports five different types of joins: inner join, cross join, left outer join, right outer join, and full outer join. 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:

You can retrieve the objects using similar syntax:

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:

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

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:

Executing above named query is even simpler.

Read More: Hibernate named query tutorial

Using 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.

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.

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

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.

Enable Logging and Commenting

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.

1) Logging: 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.

2) Commenting: 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:

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 programmatically, 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.

That’s all for now related to HQL. Keep visiting for more on hibernate.

Happy Learning !!