Guide to Hibernate Criteria Queries

Hibernate provides three different ways to retrieve data from a database. We have already discussed [HQL and native SQL queries](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/). Now we will discuss our third option i.e. **hibernate criteria queries**.

The criteria query API lets us build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL.

The Criteria API also includes **query by example (QBE)** functionality. This lets us supply example objects that contain the properties we would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count().

The Hibernate Criteria API had been deprecated back in Hibernate 5.x and these have been **removed in Hibernate 6.0**. Usually, all queries using the legacy API can be modeled with the **JPA Criteria API** that is still supported.

**1. A Simple Example**

The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains several overloaded createCriteria() methods.

Pass the persistent object’s class or its entity name to the createCriteria() method, and hibernate will create a Criteria object that returns instances of the persistence object’s class when your application executes a criteria query.

The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

Criteria crit = session.createCriteria(Product.**class**);

List<Product> results = crit.list();

Moving on from this simple criteria example, we will add constraints to our criteria queries so we can whittle down the result set.

**2. Using Restrictions**

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction.

We can have more than one restriction for a criteria query.

**2.1. Restrictions.eq()**

To retrieve objects that have a property value that “**equals**” your restriction, use the eq() method on Restrictions, as follows:

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.eq("description","Mouse"));

List<Product> results = crit.list()

The above query will search all products having a description as “Mouse”.

**2.2. Restrictions.ne()**

To retrieve objects that have a property value “not equal to” your restriction, use the ne() method on Restrictions, as follows:

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.ne("description","Mouse"));

List<Product> results = crit.list()

The above query will search all products having descriptions anything but not “Mouse”.

**2.3. Restrictions.like() and Restrictions.ilike()**

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the like() or the ilike() method. The ilike() method is case-insensitive.

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));

List<Product> results = crit.list();

The above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:

* *ANYWHERE*: Anyplace in the string
* *END*: The end of the string
* *EXACT*: An exact match
* *START*: The beginning of the string

**2.4. Restrictions.isNull() and Restrictions.isNotNull()**

The isNull() and isNotNull() restrictions allow you to do a search for objects that have (or do not have) null property values.

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.isNull("name"));

List<Product> results = crit.list();

**2.5. Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le()**

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison is le().

We can do a quick retrieval of all products with prices over $25 like this, relying on Java’s type promotions to handle the conversion to Double:

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.gt("price", 25.0));

List<Product> results = crit.list();

**2.6. Combining Two or More Restrictions**

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.lt("price",10.0));

crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));

List<Product> results = crit.list();

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the or() method on the Restrictions class, as follows:

Criteria crit = session.createCriteria(Product.**class**);

Criterion priceLessThan = Restrictions.lt("price", 10.0);

Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);

LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);

crit.add(orExp);

List results=crit.list();

The*orExp* logical expression that we have created here will be treated like any other criterion. We can therefore add another restriction to the criteria:

Criteria crit = session.createCriteria(Product.**class**);

Criterion price = Restrictions.gt("price",**new** Double(25.0));

Criterion name = Restrictions.like("name","Mou%");

LogicalExpression orExp = Restrictions.or(price,name);

crit.add(orExp);

crit.add(Restrictions.ilike("description","blocks%"));

List results = crit.list();

**2.7. Using Disjunction Objects with Criteria**

If we wanted to create an OR expression with more than two different criteria (for example, “price > 25.0 OR name like Mou% OR description not like blocks%”), we would use an org.hibernate.criterion.Disjunction object to represent a disjunction.

You can obtain this object from the disjunction() factory method on the Restrictions class. The disjunction is more convenient than building a tree of OR expressions in code.

To represent an AND expression with more than two criteria, you can use the conjunction() method, although you can easily just add those to the Criteria object. The conjunction can be more convenient than building a tree of AND expressions in code. Here is an example that uses the disjunction:

Criteria crit = session.createCriteria(Product.**class**);

Criterion priceLessThan = Restrictions.lt("price", 10.0);

Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);

Criterion browser = Restrictions.ilike("description", "browser", MatchMode.ANYWHERE);

Disjunction disjunction = Restrictions.disjunction();

disjunction.add(priceLessThan);

disjunction.add(mouse);

disjunction.add(browser);

crit.add(disjunction);

List results = crit.list();

**2.8. Restrictions.sqlRestriction()**

sqlRestriction() restriction allows you to directly specify SQL in the Criteria API. It’s useful if you need to use SQL clauses that Hibernate does not support through the Criteria API.

Your application’s code does not need to know the name of the table your class uses. Use {alias} to signify the class’s table, as follows:

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.sqlRestriction("{alias}.description like 'Mou%'"));

List<Product> results = crit.list();

**3. Pagination**

One common application pattern that criteria can address is pagination through the result set of a database query. There are two methods on the Criteria interface for paging, just as there are for Query: 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 retrieve a fixed number of objects with the setMaxResults() method. Using both of these together, we can construct a paging component in our web or Swing application.

Criteria crit = session.createCriteria(Product.**class**);

crit.setFirstResult(1);

crit.setMaxResults(20);

List<Product> results = crit.list();

As you can see, this makes paging through the result set easy. You can increase the first result you return (for example, from 1, to 21, to 41, etc.) to page through the result set.

**4. Fetch a Single Result**

Sometimes you know you are going to return only zero or one object from a given query. This could be because you are calculating an aggregate or because your restrictions naturally lead to a unique result.

If you want to obtain a single Object reference instead of a List, the uniqueResult() method on the Criteria object returns an object or null. If there is more than one result, the uniqueResult() method throws a HibernateException.

The following short example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

Criteria crit = session.createCriteria(Product.**class**);

Criterion price = Restrictions.gt("price",**new** Double(25.0));

crit.setMaxResults(1);

Product product = (Product) crit.uniqueResult();

Again, please note that you need to make sure that your query returns only one or zero results if you use the uniqueResult() method. Otherwise, Hibernate will throw a NonUniqueResultException exception.

**5. Distinct Results**

If you would like to work with distinct results from a criteria query, Hibernate provides a result transformer for distinct entities, org.hibernate.transform.DistinctRootEntityResultTransformer, which ensures that no duplicates will be in your query’s result set.

Rather than using SELECT DISTINCT with SQL, the distinct result transformer compares each of your results using their default hashCode() methods, and only adds those results with unique hash codes to your result set. This may or may not be the result you would expect from an otherwise equivalent SQL DISTINCT query, so be careful with this.

Criteria crit = session.createCriteria(Product.**class**);

Criterion price = Restrictions.gt("price",**new** Double(25.0));

crit.setResultTransformer( DistinctRootEntityResultTransformer.INSTANCE )

List<Product> results = crit.list();

An additional performance note: the comparison is done in Hibernate’s Java code, not at the database, so non-unique results will still be transported across the network.

**6. Sorting**

Sorting the query’s results works much the same way with criteria as it would with HQL or SQL. The Criteria API provides the org.hibernate.criterion.Order class to sort your result set in either ascending or descending order, according to one of your object’s properties.

This example demonstrates how you would use the Order class:

Criteria crit = session.createCriteria(Product.**class**);

crit.add(Restrictions.gt("price",10.0));

crit.addOrder(Order.desc("price"));

List<Product> results = crit.list();

You may add more than one Order object to the Criteria object. Hibernate will pass them through to the underlying SQL query. Your results will be sorted by the first order, then any identical matches within the first sort will be sorted by the second order, and so on.

Beneath the covers, **Hibernate passes this on to an SQL ORDER BY clause after substituting the proper database column name for the property**.

**7. Associations or JOINS**

The association works when going from **either one-to-many or from many-to-one**. First, we will demonstrate how to use one-to-many associations to obtain suppliers who sell products with a price over $25. Notice that we create a new Criteria object for the *products*property, add restrictions to the products’ criteria we just created, and then obtain the results from the supplier *Criteria* object:

Criteria crit = session.createCriteria(Supplier.**class**);

Criteria prdCrit = crit.createCriteria("products");

prdCrit.add(Restrictions.gt("price",25.0));

List results = crit.list();

Going the other way, we obtain all the products from the supplier MegaInc using many-to-one associations:

Criteria crit = session.createCriteria(Product.**class**);

Criteria suppCrit = crit.createCriteria("supplier");

suppCrit.add(Restrictions.eq("name","Hardware Are We"));

List results = crit.list();

**8. Projections and Aggregates**

Instead of working with objects from the result set, you can treat the results from the result set as a set of rows and columns, also known as a projection of the data. This is similar to how you would use data from a SELECT query with JDBC.

To use projections, start by getting the org.hibernate.criterion.Projection object you need from the org.hibernate.criterion.Projections factory class.

The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances. After you get a Projection object, add it to your Criteria object with the setProjection() method.

When the Criteria object executes, the list contains object references that you can cast to the appropriate type.

**8.1. Single Aggregate (Getting Row Count)**

Criteria crit = session.createCriteria(Product.**class**);

crit.setProjection(Projections.rowCount());

List<Long> results = crit.list();

Other aggregate functions available through the Projections factory class include the following:

* avg(propertyName): Gives the average of a property’s value
* count(propertyName): Counts the number of times a property occurs
* countDistinct(propertyName): Counts the number of unique values the property contains
* max(propertyName): Calculates the maximum value of the property values
* min(propertyName): Calculates the minimum value of the property values
* sum(propertyName): Calculates the sum total of the property values

**8.2. Multiple Aggregates**

We can apply more than one projection to a given Criteria object. To add multiple projections, get a projection list from the projectionList() method on the Projections class.

The org.hibernate.criterion.ProjectionList object has an add() method that takes a Projection object. You can pass the projections list to the setProjection() method on the Criteria object because ProjectionList implements the Projection interface.

Criteria crit = session.createCriteria(Product.**class**);

ProjectionList projList = Projections.projectionList();

projList.add(Projections.max("price"));

projList.add(Projections.min("price"));

projList.add(Projections.avg("price"));

projList.add(Projections.countDistinct("description"));

crit.setProjection(projList);

List<object[]> results = crit.list();

**8.3. Getting Selected Columns**

Another use of projections is to retrieve individual properties, rather than entities. For instance, we can retrieve just the name and description from our product table, instead of loading the entire object representation into memory.

Criteria crit = session.createCriteria(Product.**class**);

ProjectionList projList = Projections.projectionList();

projList.add(Projections.property("name"));

projList.add(Projections.property("description"));

crit.setProjection(projList);

crit.addOrder(Order.asc("price"));

List<object[]> results = crit.list();

**9. Query By Example (QBE)**

In QBE, instead of programmatically building a Criteria object with Criterion objects and logical expressions, you can partially populate an instance of the object. You use this instance as a template and have Hibernate build the criteria for you based upon its values. This keeps your code clean and makes your project easier to test.

For instance, if we have a user database, we can construct an instance of a user object, set the property values for type and creation date, and then use the Criteria API to run a QBE query. Hibernate will return a result set containing all user objects that match the property values that were set.

Behind the scenes, Hibernate inspects the Example object and constructs an SQL fragment that corresponds to the properties on the Example object.

The following basic example searches for suppliers that match the name on the example Supplier object:

Criteria crit = session.createCriteria(Supplier.**class**);

Supplier supplier = **new** Supplier();

supplier.setName("MegaInc");

crit.add(Example.create(supplier));

List results = crit.list();