CHAPTER 10

Advanced Queries Using Criteria

Hibernate provides three different ways to retrieve data. We have already discussed HQL and the use of native SQL queries; now we add criteria.

The Criteria Query API lets you 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 you supply example objects that contain the properties you 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 counts.

In this chapter, we explore the use of the Criteria API, employing the sample object model established in the previous chapter. (We are, however, going to move it to a new package, from `chapter09.model` to `chapter10.model`).

Using the Criteria API

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 `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();
```

When you run this example with our sample data, you will get all objects that are instances of the `Product` class. Note that this includes any instances of the `Software` class because they are derived from `Product`.

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

Using Restrictions with Criteria

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

Although you could create your own objects implementing the `Criterion` object, or extend an existing `Criterion` object, we recommend that you use Hibernate's built-in `Criterion` objects from your application's business logic. For instance, you could create your own factory class that returns instances of Hibernate's `Criterion` objects appropriately set up for your application's restrictions.

Use the factory methods on the `org.hibernate.criterion.Restrictions` class to obtain instances of the `Criterion` objects. To retrieve objects that have a property value that equals your restriction, use the `eq()` method on `Restrictions`, as follows:

```
public static SimpleExpression eq(String propertyName, Object value)
```

We would typically nest the `eq()` method in the `add()` method on the `Criteria` object. Here is an example of how this would look if we were searching for products with the name "Mouse":

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("description","Mouse"));
List<Product> results = crit.list()
```

Next, we search for products that do *not* have the name "Mouse." For this, we would use the `ne()` method on the `Restrictions` class to obtain a not-equal restriction:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ne("description","Mouse"));
List<Product> results = crit.list();
```

■ **Tip** You cannot use the not-equal restriction to retrieve records with a `NULL` value in the database for that property (in SQL, and therefore in Hibernate, `NULL` represents the absence of data, and so cannot be compared with data). If you need to retrieve objects with `NULL` properties, you will have to use the `isNull()` restriction, which we discuss further on in the chapter. You can combine the two with an `OR` logical expression, which we also discuss later in the chapter.

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. In either case, we have two different ways to call the method:

```
public static SimpleExpression like(String propertyName, Object value)
```

or

```
public static SimpleExpression like(String propertyName,
                                    String value,
                                    MatchMode matchMode)
```

The first `like()` or `ilike()` method takes a pattern for matching. Use the `%` character as a wildcard to match parts of the string, like so:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.like("name","Mou%"));
List<Product> results = crit.list();
```

The second `like()` or `ilike()` method 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

Here is an example that uses the `ilike()` method to search for case-insensitive matches at the end of the string:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ilike("description","ser", MatchMode.END));
List<Product> results = crit.list();
```

The `isNull()` and `isNotNull()` restrictions allow you to do a search for objects that have (or do not have) null property values. This is easy to demonstrate:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.isNull("name"));
List<Product> results = crit.list();
```

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();
```

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<Product> 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();
```

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.[1] 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<Product> results = crit.list();
```

The last type of restriction is the SQL restriction sqlRestriction(). This 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();
```

There are two other sqlRestriction() methods that permit you to pass JDBC parameters and values into the SQL statement. Use the standard JDBC parameter placeholder (?) in your SQL fragment.

[1]A disjunction represents data sets in which a data element fulfills one of a set of conditions, therefore an "or" construct. Given condition A (where value<4) and condition B (where value>9), and a data set consisting of 0, 5, and 10, a disjunction of A and B would give you 0 and 10.

A conjunction, conversely, represents an "and." A conjunction of A and B, with the same data set, would give you an empty set because no value is both less than 4 and greater than 9.

In practice, you'll probably never see these terms used outside of technical applications like this one. I can imagine the conversation now: "You could use a disjunction …" "No, thanks, I'd rather use an OR."

Paging Through the Result Set

One common application pattern that criteria can address is pagination through the result set of a database query. When we say pagination, we mean an interface in which the user sees part of the result set at a time, with navigation to go forward and backward through the results. A naïve pagination implementation might load the entire result set into memory for each navigation action, and would usually lead to atrocious performance.[2] Both of us have worked on improving performance for separate projects suffering from exactly this problem. The problem appeared late in testing because the sample data set that developers were working with was trivial, and they did not notice any performance problems until the first test data load.

If you are programming directly to the database, you will typically use proprietary database SQL or database cursors to support paging. Hibernate abstracts this away for you: behind the scenes, Hibernate uses the appropriate method for your database.

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. We have a very small data set in our sample application, so here is an admittedly trivial example:

```
Criteria crit = session.createCriteria(Product.class);
crit.setFirstResult(1);
crit.setMaxResults(2);
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. If you have only one result in your result set, Hibernate has a shortcut method for obtaining just that object.

Obtaining a Unique 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 (like `COUNT`, which we discuss later) or because your restrictions naturally lead to a unique result—when selecting upon a property under a unique constraint, for example. You may also limit the results of any result set to just the first result, using the `setMaxResults()` method discussed earlier. In any of these circumstances, if you want 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, we stress 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, which may not be what you would expect—Hibernate does not just pick the first result and return it.[3]

[2]When naïve pagination would *not* lead to atrocious performance, chances are the result set is small enough that you wouldn't need pagination in the first place.

[3]Thus, it's about a *unique* result and not just a single result. You have to enforce uniqueness if it's not already present.

Sorting the Query's Results

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.

Create an `Order` object with either of the two static factory methods on the `Order` class: `asc()` for ascending or `desc()` for descending. Both methods take the name of the property as their only argument. After you create an `Order`, use the `addOrder()` method on the `Criteria` object to add it to the query.

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.

Associations

To add a restriction on a class that is associated with your criteria's class, you will need to create another `Criteria` object. Pass the property name of the associated class to the `createCriteria()` method, and you will have another `Criteria` object. You can get the results from either `Criteria` object, although you should pick one style and be consistent for readability's sake. We find that getting the results from the top-level `Criteria` object (the one that takes a class as a parameter) makes it clear what type of object is expected in the results.

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<Supplier> 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<Product> results = crit.list();
```

Although we can use either `Criteria` object to obtain the results, it makes a difference which criteria we use for ordering the results. In the following example, we are ordering the supplier results by the supplier names:

```
Criteria crit = session.createCriteria(Supplier.class);
Criteria prdCrit = crit.createCriteria("products");
prdCrit.add(Restrictions.gt("price",25.0));
crit.addOrder(Order.desc("name"));
List<Supplier> results = prdCrit.list();
```

If we wanted to sort the suppliers by the descending price of their products, we would use the following line of code. This code would have to replace the previous `addOrder()` call on the supplier `Criteria` object.

```
prdCrit.addOrder(Order.desc("price"));
```

Although the products are not in the result set, SQL still allows you to order by those results. If you get mixed up about which `Criteria` object you are using and pass the wrong property name for the sort-by order, Hibernate will throw an exception.

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. 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.

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; also, Hibernate supports properties, aggregate functions, and the `GROUP BY` clause.

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.

The row-counting functionality provides a simple example of applying projections. The code looks similar to the restrictions examples we were working with earlier in the chapter:

```
Criteria crit = session.createCriteria(Product.class);
crit.setProjection(Projections.rowCount());
List<Long> results = crit.list();
```

The results list will contain one object, a `Long` that contains the results of executing the `COUNT` SQL statement. Other aggregate functions available through the `Projections` factory class include the following:

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

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. The following example demonstrates some of the aggregate functions, along with the projection list:

```
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();
```

When you execute multiple aggregate projections, you get a `List` with an `Object` array as the first element. The `Object` array contains all of your values, in order. For this projection, you'll have three `Double` references, and then a `Long` reference for the count.

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. Use the `property()` method on the `Projections` class to create a `Projection` for a property. When you execute this form of query, the `list()` method returns a `List` of `Object` arrays. Each `Object` array contains the projected properties for that row. The following example returns just the contents of the name and description columns from the `Product` data. Remember, Hibernate is polymorphic, so this also returns the name and description from the `Software` objects that inherit from `Product`.[4]

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.property("name"));
projList.add(Projections.property("description"));
crit.setProjection(projList);
List<Object[]> results = crit.list();
```

Use this query style when you want to cut down on network traffic between your application servers and your database servers. For instance, if your table has a large number of columns, this can slim down your results. In other cases, you may have a large set of joins that would return a very wide result set, but you are only interested in a few columns. Lastly, if your clients have limited memory, this can save you trouble with large data sets. But make sure you don't have to retrieve additional columns for the entire result set later, or your optimizations may actually decrease performance.

You can group your results (using SQL's `GROUP BY` clause) with the `groupProperty` projection. The following example groups the products by name and price:

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.groupProperty("name"));
projList.add(Projections.groupProperty("price"));
crit.setProjection(projList);
crit.addOrder(Order.asc("price"));
List<Object> results = crit.list();
```

[4]This kind of projection is supported by JPA, via JPQL, as well.

As you can see, projections open up aggregates to the Criteria API, which means that developers do not have to drop into HQL for aggregates. Projections offer a way to work with data that is closer to the JDBC result set style, which may be appropriate for some parts of your application.

Query By Example (QBE)

In this section, because of the confusing terminology, we will refer to excerpts from our demonstration code as "samples" rather than "examples," reserving "example" for its peculiar technical meaning in the context of 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. The `org.hibernate.criterion.Example` class contains the QBE functionality. Note that the `Example` class implements the `Criterion` interface, so you can use it like any other restriction on a criteria query.

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.

To use QBE, we first need to construct an `Example` object. Then we need to create an instance of the `Example` object, using the static `create()` method on the `Example` class. The `create()` method takes the `Example` object as its argument. You add the `Example` object to a `Criteria` object just like any other `Criterion` 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<Supplier> results = crit.list();
```

When Hibernate translates our `Example` object into an SQL query, all the properties on our `Example` objects get examined. We can tell Hibernate which properties to ignore; the default is to ignore null-valued properties. To search our products or software in the sample database with QBE, we need either to specify a price or to tell Hibernate to ignore properties with a value of zero, because we used a double primitive for storage instead of a `Double` object. The double primitive initializes to zero, while a `Double` would have been null; and so, left to its own devices, the QBE logic will assume that we are specifically searching for prices of zero, whereas we want it to ignore this default value.

We can make the Hibernate `Example` object exclude zero-valued properties with the `excludeZeroes()` method. We can exclude properties by name with the `excludeProperty()` method, or exclude nothing (compare for null values and zeroes exactly as they appear in the `Example` object) with the `excludeNone()` method. This sample applies the `excludeZeroes()` method to ignore the default zero prices:

```
Criteria crit = session.createCriteria(Product.class);
Product exampleProduct = new Product();
exampleProduct.setName("Mouse");
Example example = Example.create(exampleProduct);
example.excludeZeroes();
crit.add(example);
List<Product> results = crit.list();
```

Other options on the `Example` object include ignoring the case for strings with the `ignoreCase()` method and enabling use of SQL's `LIKE` for comparing strings, instead of just using `equals()`.

We can also use associations for QBE. In the following sample, we create two `Example` objects: one for the product and one for the supplier. We use the technique explained in the "Associations" section of this chapter to retrieve objects that match both criteria.

```
Criteria prdCrit = session.createCriteria(Product.class);
Product product = new Product();
product.setName("M%");
Example prdExample = Example.create(product);
prdExample.excludeProperty("price");
prdExample.enableLike();
Criteria suppCrit = prdCrit.createCriteria("supplier");
Supplier supplier = new Supplier();
supplier.setName("SuperCorp");
suppCrit.add(Example.create(supplier));
prdCrit.add(prdExample);
List<Product> results = prdCrit.list();
```

We also ignore the price property for our product, and we use `LIKE` for object comparison, instead of equals.

The QBE API works best for searches in which you are building the search from user input. The Hibernate team recommends using QBE for advanced searches with multiple fields because it's easier to set values on business objects than to manipulate restrictions with the Criteria API.

JPA 2 AND THE TYPE-SAFE CRITERIA API

In addition to the Hibernate Criteria API described in this chapter, Hibernate 4.2 includes a new type-safe Criteria API based on the JPA 2 standard. The advantage of this new API is that if you use it, you will not have any errors related to typos in the Criteria restrictions, such as "prduct". Rather than relying on strings in Criteria, there is a new metamodel class for each JPA entity. You would use the references to properties in these metamodel classes in your criteria restrictions, but you would use the javax.persistence criteria classes. Hibernate can generate the metamodel classes for you from the annotations in your existing classes.

The JPA 2 type-safe criteria does not replace the Hibernate Criteria API covered in this chapter. Both criteria APIs will accomplish the same thing for building a query. In this book, we covered the Hibernate way of doing things. Using the JPA 2 Criteria API is similar to the Hibernate Criteria API, but references to fields as strings are replaced with references to fields on generated metamodel classes.

In our example for this chapter, we would run the Hibernate Static Metamodel Generator against our annotated classes to generate the metamodel class files. The Metamodel Generator is a Java 6 annotation processor, so your existing build tools should automatically create the generated metamodel classes. Depending on how you have your source repository and project layout arranged, you will have to decide where to put the generated classes.

Summary

Using the Criteria API is an excellent way to get started developing with HQL. The developers of Hibernate have provided a clean API for adding restrictions to queries with Java objects. Although HQL isn't too difficult to learn, some developers prefer the Criteria Query API, as it offers compile-time syntax checking—although column names and other schema-dependent information cannot be checked until run time.

In the next chapter, we discuss the use of Hibernate filters to restrict the range of data against which queries are applied.