Using the Criteria API to Create Queries

The Criteria API is used to define queries for entities and their persistent state by creating query-defining objects.

Criteria queries are written using Java programming language APIs, are typesafe, and are portable.

Such queries work regardless of the underlying data store.

The following topics are addressed here:

- Overview of the Criteria and Metamodel APIs
- Using the Metamodel API to Model Entity Classes
- Using the Criteria API and Metamodel API to Create Basic Typesafe Queries

Overview of the Criteria and Metamodel APIs

Similar to JPQL, the Criteria API is based on the abstract schema of persistent entities, their relationships, and embedded objects.

The Criteria API operates on this abstract schema to allow developers to find, modify, and delete persistent entities by invoking Java Persistence API entity operations.

The Metamodel API works in concert with the Criteria API to model persistent entity classes for Criteria queries.

The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries.

Developers familiar with JPQL syntax will find equivalent object-level operations in the Criteria API.

The following simple Criteria query returns all instances of the `Pet` entity in the data source:

```
EntityManager em = ...;
CriteriaBuilder cb =
em.getCriteriaBuilder();
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q =
em.createQuery(cq);
List<Pet> allPets =
q.getResultList();
```

The equivalent JPQL query is:

```
SELECT p
FROM Pet p
```

This query demonstrates the basic steps to create a Criteria query:

1. Use an `EntityManager` instance to create a `CriteriaBuilder` object.

2. Create a query object by creating an instance of the `CriteriaQuery` interface.

This query object's attributes will be modified with the details of the query.

3. Set the query root by calling the `from` method on the `CriteriaQuery` object.

4. Specify what the type of the query result will be by calling the `select` method of the `CriteriaQuery` object.

5. Prepare the query for execution by creating a `TypedQuery<T>` instance, specifying the type of the query result.

6. Execute the query by calling the `getResultList` method on the `TypedQuery<T>` object.

Because this query returns a collection of entities, the result is stored in a `List`.

The tasks associated with each step are discussed in detail in this chapter.

To create a `CriteriaBuilder` instance, call the `getCriteriaBuilder` method on the `EntityManager` instance:

```
CriteriaBuilder cb =
em.getCriteriaBuilder();
```

The query object is created by using the `CriteriaBuilder` instance:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
```

The query will return instances of the `Pet` entity, so the type of the query is specified when the `CriteriaQuery` object is created to create a typesafe query.

The `FROM` clause of the query is set, and the root of the query specified, by calling the `from` method of the query object:

```
Root<Pet> pet = cq.from(Pet.class);
```

The `SELECT` clause of the query is set by calling the `select` method of the query object and passing in the query root:

```
cq.select(pet);
```

The query object is now used to create a `TypedQuery<T>` object that can be executed against the data source.

The modifications to the query object are captured to create a ready-to-execute query:

```
TypedQuery<Pet> q =
em.createQuery(cq);
```

This typed query object is executed by calling its `getResultList` method, because this query will return multiple entity instances.

The results are stored in a `List<Pet>` collection-valued object.

```
List<Pet> allPets =
q.getResultList();
```

Using the Metamodel API to Model Entity Classes

The Metamodel API is used to create a metamodel of the managed entities in a particular persistence unit.

For each entity class in a particular package, a metamodel class is created with a trailing underscore and with attributes that correspond to the persistent fields or properties of the entity class.

The following entity class, `com.example.Pet`, has four persistent fields: `id`, `name`, `color`, and `owners`:

```
package com.example;...
@Entity
public class Pet {
@Id
protected Long id;
protected String name;
protected String color;
@ManyToOne
protected Set<Person> owners;
...
}
```

The corresponding Metamodel class is:

```
package com.example;...
@Static Metamodel(Pet.class)
public class Pet_ {
public static volatile
SingularAttribute<Pet, Long> id;
public static volatile
SingularAttribute<Pet, String>
name;
```

```
public static volatile
SingularAttribute<Pet, String>
color;
public static volatile
SetAttribute<Pet, Person> owners;
}
```

The metamodel class and its attributes are used in Criteria queries to refer to the managed entity classes and their persistent state and relationships.

Using Metamodel Classes

Metamodel classes that correspond to entity classes are of the following type:

```
javax.persistence.metamodel.
EntityType<T>
```

Metamodel classes are typically generated by annotation processors either at development time or at runtime.

Developers of applications that use Criteria queries may generate static metamodel classes by using the persistence provider's annotation processor or may obtain the metamodel class by either calling the `getModel` method on the query root object or first obtaining an instance of the `Metamodel` interface and then passing the entity type to the instance's `entity` method.

The following code snippet shows how to obtain the `Pet` entity's metamodel class by calling `Root<T>.getModel`:

```
EntityManager em = ...;
CriteriaBuilder cb =
em.getCriteriaBuilder();
CriteriaQuery cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

```
EntityType<Pet> Pet_ =
pet.getModel();
```

The following code snippet shows how to obtain the `Pet` entity's metamodel class by first obtaining a metamodel instance by using `EntityManager.getMetamodel` and then calling `entity` on the metamodel instance:

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
```

Using the Criteria API and Metamodel API to Create Basic Typesafe Queries

The basic semantics of a Criteria query consists of a `SELECT` clause, a `FROM` clause, and an optional `WHERE` clause, similar to a JPQL query.

Criteria queries set these clauses by using Java programming language objects, so the query can be created in a typesafe manner.

Creating a Criteria Query

The `javax.persistence.criteria.CriteriaBuilder` interface is used to construct

- Criteria queries
- Selections
- Expressions
- Predicates
- Ordering

To obtain an instance of the `CriteriaBuilder` interface, call the `getCriteriaBuilder` method on either an `EntityManager` or an `EntityManagerFactory` instance.

The following code shows how to obtain a `CriteriaBuilder` instance by using the `EntityManager.getCriteriaBuilder` method.

```
EntityManager em = ...;
CriteriaBuilder cb =
em.getCriteriaBuilder();
```

Criteria queries are constructed by obtaining an instance of the following interface:

```
javax.persistence.criteria.
CriteriaQuery
```

`CriteriaQuery` objects define a particular query that will navigate over one or more entities.

Obtain `CriteriaQuery` instances by calling one of the `CriteriaBuilder.createQuery` methods.

For creating typesafe queries, call the `CriteriaBuilder.createQuery` method as follows:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
```

The `CriteriaQuery` object's type should be set to the expected result type of the query.

In the preceding code, the object's type is set to `CriteriaQuery<Pet>` for a query that will find instances of the `Pet` entity.

In the following code snippet, a CriteriaQuery object is created for a query that returns a String:

```
CriteriaQuery<String> cq =
cb.createQuery(String.class);
```

Query Roots

For a particular `CriteriaQuery`object, the root entity of the query, from which all navigation originates, is called the query root.

It is similar to the `FROM` clause in a JPQL query.

Create the query root by calling the `from` method on the `CriteriaQuery` instance.

The argument to the `from` method is either the entity class or an `EntityType<T>` instance for the entity.

The following code sets the query root to the `Pet` entity:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

The following code sets the query root to the `Pet` class by using an `EntityType<T>` instance:

```
EntityManager em = ...;
Metamodel m = em.getMetamodel();
```

```
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet_);
```

Criteria queries may have more than one query root.

This usually occurs when the query navigates from several entities.

The following code has two Root instances:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet1 =
cq.from(Pet.class);
Root<Pet> pet2 =
cq.from(Pet.class);
```

Querying Relationships Using Joins

For queries that navigate to related entity classes, the query must define a join to the related entity by calling one of the `From.join` methods on the query root object or another join object.

The `join` methods are similar to the `JOIN` keyword in JPQL.

The target of the join uses the Metamodel class of type `EntityType<T>` to specify the persistent field or property of the joined entity.

The `join` methods return an object of type `Join<X, Y>`, where `X` is the source entity and `Y` is the target of the join.

In the following code snippet, `Pet` is the source entity, and `Owner` is the target:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner =
pet.join(Pet_.owners);
```

Joins can be chained together to navigate to related entities of the target entity without having to create a Join<X, Y> instance for each join:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
```

```
EntityType<Owner> Owner_ =
m.entity(Owner.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Owner, Address> address =
cq.join(Pet_.owners).
join(Owner_.addresses);
```

Path Navigation in Criteria Queries

`Path` objects are used in the `SELECT` and `WHERE` clauses of a Criteria query and can be query root entities, join entities, or other `Path` objects.

The `Path.get` method is used to navigate to attributes of the entities of a query.

The argument to the `get` method is the corresponding attribute of the entity's Metamodel class.

The attribute can either be a single-valued attribute, specified by `@SingularAttribute` in the Metamodel class, or a collection-valued attribute, specified by one of `@CollectionAttribute`, `@SetAttribute`, `@ListAttribute`, or `@MapAttribute`.

The following query returns the names of all the pets in the data store.

The `get` method is called on the query root, `pet`, with the `name` attribute of the `Pet` entity's Metamodel class, `Pet_` as the argument:

```
CriteriaQuery<String> cq =
cb.createQuery(String.class);
Metamodel m = em.getMetamodel();
```

```
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet.get(Pet_.name));
```

Restricting Criteria Query Results

The results of a query can be restricted on the `CriteriaQuery` object according to conditions set by calling the `CriteriaQuery.where` method.

Calling the `where` method is analogous to setting the `WHERE` clause in a JPQL query.

The `where` method evaluates instances of the `Expression` interface to restrict the results according to the conditions of the expressions.

`Expression` instances are created by using methods defined in the `Expression` and `CriteriaBuilder` interfaces.

The `Expression` Interface Methods

An `Expression` object is used in a query's `SELECT`, `WHERE`, or `HAVING` clause.

Table 35-1 shows conditional methods you can use with `Expression` objects.

Table 35-1 Conditional Methods in the `Expression` Interface

Method	Description
`isNull`	Tests whether an expression is null
`isNotNull`	Tests whether an expression is not null
`in`	Tests whether an expression is within a list of values

The following query uses the `Expression.isNull` method to find all pets where the `color` attribute is null:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where
 (pet.get(Pet_.color).isNull());
```

The following query uses the Expression.in method to find all brown and black pets:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).in
("brown", "black");
```

The `in` method also can check whether an attribute is a member of a collection.

Expression Methods in the `CriteriaBuilder` Interface

The `CriteriaBuilder` interface defines additional methods for creating expressions.

These methods correspond to the arithmetic, string, date, time, and case operators and functions of JPQL.

Table 35-2 shows conditional methods you can use with `CriteriaBuilder` objects.

Table 35-2 Conditional Methods in the `CriteriaBuilder` Interface

Conditional Method	Description
`equal`	Tests whether two expressions are equal
`notEqual`	Tests whether two expressions are not equal
`gt`	Tests whether the first numeric expression is greater than the second numeric expression
`ge`	Tests whether the first numeric expression is greater than or equal to the second numeric expression

`lt`	Tests whether the first numeric expression is less than the second numeric expression
`le`	Tests whether the first numeric expression is less than or equal to the second numeric expression
`between`	Tests whether the first expression is between the second and third expression in value
`like`	Tests whether the expression matches a given pattern

The following code uses the `CriteriaBuilder.equal` method:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal
(pet.get(Pet_.name)), "Fido");
...
```

The following code uses the `CriteriaBuilder.gt` method:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where
(cb.gt(pet.get(Pet_.birthday)),date);
```

The following code uses the `CriteriaBuilder.between` method:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get
(Pet_.birthday)), firstDate,
secondDate);
```

The following code uses the CriteriaBuilder.like method:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where
(cb.like(pet.get(Pet_.name)), "*do");
```

Multiple conditional predicates can be specified by using the compound predicate methods of the `CriteriaBuilder` interface, as shown in Table 35-3.

Table 35-3 Compound Predicate Methods in the `CriteriaBuilder` Interface

Method	Description
`and`	A logical conjunction of two Boolean expressions
`or`	A logical disjunction of two Boolean expressions
`not`	A logical negation of the given Boolean expression

The following code shows the use of compound predicates in queries:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ =
m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

```
cq.where(cb.equal
(pet.get(Pet_.name), "Fido")
.and(cb.equal(pet.get(Pet_.color),
"brown"));
```

Managing Criteria Query Results

For queries that return more than one result, it's often helpful to organize those results.

The `CriteriaQuery` interface defines the `orderBy` method to order query results according to attributes of an entity.

The `CriteriaQuery` interface also defines the `groupBy` method to group the results of a query together according to attributes of an entity, and the `having` method to restrict those groups according to a condition.

Ordering Results

The order of the results of a query can be set by calling the `CriteriaQuery.orderBy` method and passing in an `Order` object.

`Order` objects are created by calling either the `CriteriaBuilder.asc` or the `CriteriaBuilder.desc` method.

The `asc` method is used to order the results by ascending value of the passed expression parameter.

The `desc` method is used to order the results by descending value of the passed expression parameter.

The following query shows the use of the `desc` method:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
cq.orderBy
(cb.desc(pet.get(Pet_.birthday)));
```

In this query, the results will be ordered by the pet's birthday from highest to lowest.

That is, pets born in December will appear before pets born in May.

The following query shows the use of the `asc` method:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
```

```
Join<Owner, Address> address =
cq.join
(Pet_.owners).join(Owner_.address);
cq.select(pet);
cq.orderBy(cb.asc
(address.get(Address_.postalCode)));
```

In this query, the results will be ordered by the pet owner's postal code from lowest to highest.

That is, pets whose owner lives in the 10001 zip code will appear before pets whose owner lives in the 91000 zip code.

If more than one `Order` object is passed to `orderBy`, the precedence is determined by the order in which they appear in the argument list of `orderBy`.

The first `Order` object has precedence.

The following code orders results by multiple criteria:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Join<Pet, Owner> owner =
cq.join(Pet_.owners);
cq.select(pet);
cq.orderBy(cb.asc(owner.get(Owner_.
lastName),
owner.get(Owner_.firstName));
```

The results of this query will be ordered alphabetically by the pet owner's last name, then first name.

Grouping Results

The `CriteriaQuery.groupBy` method partitions the query results into groups.

These groups are set by passing an expression to `groupBy`:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
```

This query returns all `Pet` entities and groups the results by the pet's color.

The `CriteriaQuery.having` method is used in conjunction with `groupBy` to filter over the groups.

The `having` method takes a conditional expression as a parameter.

By calling the `having` method, the query result is restricted according to the conditional expression:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.groupBy(pet.get(Pet_.color));
cq.having(cb.in(pet.get(Pet_.color)).
value("brown").value("blonde");
```

In this example, the query groups the returned `Pet` entities by color, as in the preceding example.

However, the only returned groups will be `Pet` entities where the `color` attribute is set to `brown` or `blonde`.

That is, no gray-colored pets will be returned in this query.

Executing Queries

To prepare a query for execution, create a `TypedQuery<T>` object with the type of the query result by passing the `CriteriaQuery` object to `EntityManager.createQuery`.

Queries are executed by calling either `getSingleResult` or `getResultList` on the `TypedQuery<T>` object.

Single-Valued Query Results

The `TypedQuery<T>.getSingleResult` method is used for executing queries that return a single result:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
...
TypedQuery<Pet> q =
em.createQuery(cq);
Pet result = q.getSingleResult();
```

Collection-Valued Query Results

The `TypedQuery<T>.getResultList` method is used for executing queries that return a collection of objects:

```
CriteriaQuery<Pet> cq =
cb.createQuery(Pet.class);
...
TypedQuery<Pet> q =
em.createQuery(cq);
List<Pet> results =
q.getResultList();
```