Endlich dabei – die Criteria Query API in der JPA 2.0

Oliver Kaluza
ORDIX AG
Paderborn

Schlüsselworte:

Criteria Query API, Java Persistance Query Language, JPQL, JPA 2.0

Einführung

Viele Entwickler kennen die Criteria Query API von Hibernate und haben sich gefragt, wann sie endlich in den Standard der JPA übernommen wird. Mit der Version JPA 2.0 ist dies nun geschehen. Die Criteria Query API ist aber nicht als Ersatz für die Java Persistence Query Language (JPQL) gedacht, sondern als Ergänzung für bestimmte Anwendungsfälle. Man kann mit ihr sehr gut Queries in objektorientierter Weise zur Laufzeit zusammensetzen und dabei die Vollständigkeit und die Typkorrektheit weitgehend sicherstellen. Geeignet ist diese API z. B. für die Auswahl von verschiedenen Filtern für die Einschränkung der Ergebnismenge von Projekten, Mitarbeitern, Aufträgen etc.

Fangen wir mit einem einfachen Beispiel an. Eine Query mit JPQL sieht folgendermaßen aus:

```
SELECT w FROM worker w WHERE w.name = 'Oliver'
```

Dies würde mit der Criteria Query API wie folgt aussehen:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Worker> cq = cb.createQuery(Worker.class);
Root<Worker> w = cq.from(Worker.class);
cq.select(w);
cq.where(cb.equal(w.get("name"),"Oliver")));
TypedQuery<Worker> q = entityManager.createQuery(cq);
List<Worker> result = q.getResultList();
```

Wie man feststellt, ist das Schreiben von Queries mit der Criteria Query API nicht unbedingt kürzer, aber das ist auch nicht die Absicht. Mit dem Befehl `cb.createQuery(...)` erzeugen wir eine Criteria Query für den Ergebnistyp `Worker`. Mit `cq.from(...)` erzeugen wir eine Projektionsvariable (`Root`) und mit `cq.select(...)` legen wir die Selektion fest. Mit `cq.where(...)` wird dann die Bedingung beschrieben. Es tauchen also mit 'from', 'select' und 'where' alle Elemente auf, wie man sie von einer normalen JPQL-Query kennt. Der Aufbau ist also der gleiche, man benutzt nur eine andere Schreibweise.

Metamodell-Klassen

Diese Schreibweise stellt leider immer noch nicht die vollständige Typsicherheit und die Tatsache sicher, ob das verwendete Attribut der Entity existiert. Wir arbeiten hier immer noch mit Strings, bei denen es zu Schreibfehler kommen kann. So kommt es beispielsweise erst zur Laufzeit zu einem Fehler, wenn es kein Attribut "`name`" in der Entity `Worker` gibt, oder es vom falschen Typ ist. An dieser Stelle kommen die Metamodell-Klassen ins Spiel. Möchte man diese einsetzen, sollte zu jeder Entity-Klasse eine Metamodell-Klasse existieren. Diese kann man theoretisch manuell erstellen. Viel sinnvoller ist aber die automatische Generierung mit Hilfe von Annotations-Prozessoren, wie z. B. dem Hibernate Metamodel Generator.

In unserem Beispiel haben wir eine Entity-Klasse `Worker`, die wie folgt aufgebaut ist:

```
@Entity
public class Worker {
      @ID
      private Integer id;
      private String name;
      private Integer age;
      @ManyToOne
      private Department department;
      @OneToMany
      List<Project> myProjects;
      ...
}
```

Die Metamodell-Klasse würde dann folgendermaßen aussehen:

```
@StaticMetamodel(Worker.class)
public class Worker_ {
    public static volatile SingularAttribute< Worker, Integer> id;
    public static volatile SingularAttribute< Worker, String> name;
    public static volatile SingularAttribute< Worker, Integer> age;
    public static volatile SingularAttribute< Worker, Department >
department;
    public static volatile ListAttribute< Worker, Project>
myProjects;
}
```

Diese Klasse können wir jetzt einfach in unserer Criteria Query benutzen und sind so vollkommen Typsicher. Fast alle Fehler können so schon zur Kompilierungszeit festgestellt werden:

```
cq.where(cb.equal(w.get(Worker_.name),"Oliver")));
```

Pfad-Ausdrücke

Aus der JPQL kennen wir bereits Pfad-Ausdrücke wie:

```
SELECT w FROM worker w WHERE w.department.location = 'Paderborn'
```

Dieser selektiert alle Arbeiter, deren Abteilung sich in Paderborn befindet. Auch dies ist natürlich mit der Criteria Query API möglich. Der Pfad (Path) kann z. B. direkt in der `where`-Methode eingebaut oder separat definiert werden. Damit ist er an verschiedenen Stellen nutzbar, wie z. B. in der `where`- oder auch der `select`-Methode, um den Rückgabewert zu definieren (dazu später mehr).

Direkte Benutzung:

```
cq.where(cb.equal(w.get(Worker_.department)
           .get(Department_.location), "Paderborn")));
```

Mit separater Definition:

```
Root<Worker> w = cq.from(Worker.class);
Path<Department> dep = w.get(Worker_.department);
Path<String>     loc = dep.get(Department_.location);
cq.select(w).where(cb.equal(loc, "Paderborn");
TypedQuery<Worker> q = entityManager.createQuery(cq);
List<Worker> result = q.getResultList();
```

Dieses Beispiel kann auch mit nur einem Schritt, ohne den Zwischenschritt mit "**dep**" durchgeführt werden. Die Path-Angabe kann, wie bereits beschrieben, auch im `select`-Teil benutzt werden. Das folgende Beispiel liefert alle Standorte, an denen Arbeiter unter 18 Jahren arbeiten (nebenbei zeigt es noch die Verwendung des `distinct`-Befehls):

```
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Worker> w = cq.from(Worker.class);
Path<String> loc = w.get(Worker_.department).get(Department_.location);
cq.select(loc).distinct(true);
cq.where(cb.lessThan(w.get(Worker_.age),18);
TypedQuery<String> q = entityManager.createQuery(cq);
List<String> result = q.getResultList();
```

Join-Ausdruck

Wie bei SQL und JPQL gibt es natürlich auch bei der Criteria Query API Joins, bei denen ein Join-Typ angegeben werden kann. Gibt man keinen Typ an, so handelt es sich um einen Inner-Join. Das folgende Beispiel soll alle Arbeiter liefern, die in Projekten tätig sind, die sich verzögern:

```
Root<Worker> w = cq.from(Worker.class);
Join<Worker, Project> p = w.join(Worker_.myProjects, JoinType.Inner);
cq.select(w).distinct(true);
cq.where(cb.equal(p.get(Project_.status), "delayed");
TypedQuery<Worker> q = entityManager.createQuery(cq);
List<Worker> result = q.getResultList();
```

Bei den Join-Ausdrücken gibt es wie bei den Path-Ausdrücken die Möglichkeit, diese zu verketten.

Parameter

Die bisherigen Beispiele haben stets feste Werte bei den Abfragen. In der Praxis möchte man diese in der Regel parametrisieren. Dies erfolgt in der Criteria Query API mit Hilfe von `ParameterExpressions`. Im folgenden Beispiel führen wir auch gleich die `Predicates` mit ein. Dies sind separat definierte Bedingungen, die man im `where`-Befehl benutzen kann. Das Beispiel soll alle Mitarbeiter liefern, die ein bestimmtes Alter haben und an einem bestimmten Ort arbeiten.

```
CriteriaQuery<Worker> cq = cb.createQuery(Worker.class);
Root<Worker> w = cq.from(Worker.class);
Path<String> l = w.get(Worker_.department).get(Department_.location);

ParameterExpression<String> pLoc = cp.parameter(String.class, "loc");
Predicate criteria1 = cb.equal(l, pLoc);
ParameterExpression<Integer> pAge = cp.parameter(Integer.class, "age");
Predicate criteria2 = cb.equal(w.get(Worker_.age), pAge);

cq.select(Worker).distinct(true);
cq.where(cb.and(criteria1, criteria2);
TypedQuery<Worker> q = entityManager.createQuery(cq);
q.setParameter("loc", location);
q.setParameter("age", age);
List<Worker> result = q.getResultList();
```

In den beiden Variablen `location` und `age` stehen dann die eigentlichen Werte, die übergeben wurden.

In-Ausdruck

Häufig möchte man ein Attribut auf mehrere Werte abfragen. Dazu dient der `in`-Befehl. Das folgende Beispiel ermittelt alle Arbeiter, die in Paderborn, Wiesbaden oder Köln arbeiten:

```
Root<Worker> w = cq.from(Worker.class);
Path<String> loc =w.get(Worker_.department).get(Department_.location);
cq.select(w)
cq.where(cb.in(loc).value("Paderborn).value("Wiesbaden").value("Köln "));
```

Die Schreibweise ist erst einmal etwas ungewöhnlich und bei vielen Werten auch unpraktisch. Daher gibt es eine Kurzform, die allerdings nicht so viele Optionen erlaubt:

```
cq.where(loc.in("Paderborn,"Wiesbaden","Köln");
```

construct Befehl

Manchmal möchte man die Ergebniswerte aus der Query dazu nutzen, um eine neue Klasse zu instanziieren. Dies kann man mit der Criteria Query API direkt in der Abfrage machen. Dazu benötigt man den `construct`-Befehl. Für das folgende Beispiel haben wir eine Klasse `WorkerDetails`, die nur Namen und Alter enthält. Ferner hat sie einen Konstruktor mit diesen beiden Parametern als Eingabe.

```
CriteriaQuery<WorkerDetails> cq = cb.createQuery(WorkerDetails.class);
Root<Worker> w = cq.from(Worker.class);
Path<String> name = w.get(Worker_.name);
Path<Integer age  = w.get(Worker_.age);
cq.select(cb.construct(WorkerDetails.class, name, age);
TypedQuery<WorkerDetails> q = entityManager.createQuery(cq);
List<WorkerDetails > result = q.getResultList();
```

Fazit

Die Criteria Query API steht der JPQL in ihrem Funktionsumfang in nichts nach. Durch ihre doch recht ausschweifende Schreibweise gegenüber der JPQL, bietet sie sich nicht für jeden Zweck als Alternative an. Für das dynamische Zusammensetzen von Queries zur Laufzeit ist sie aber erste Wahl.

Auch wer auf Typsicherheit steht ist – im Zusammenhang mit den Metamodell-Klassen – mit der Criteria Query API gut bedient. In Summe kann man also von einer guten und sinnvollen Ergänzung des JPA-Standards sprechen.

Kontaktadresse:

Oliver Kaluza

ORDIX AG

Westernmauer 12-16

D-33098 Paderborn

Telefon: +49 (0) 5251-10630

Fax: +49 (0) 180-1673490

E-Mail: info@ordix.de

Internet: www.ordix.de