Virtuelle Felder in Abfragen, virtuelle Objektmodelle

Neu in EclipseLink

EclipseLink erlaubt die Interaktion mit diversen Datensystemen und unterstützt das Java Persistence API (JPA). Version 2.1 bringt neue Features. **Michael Bräuer, Frank Schwarz**

Auf einen Blick

Dieser Artikel ergänzt den Beitrag aus database pro 6/2010 und beleuchtet virtuelle Felder in Abfragen sowie virtuelle Objektmodelle. Beides sind Neuerungen in der EclipseLink-Version 2.1.

- **Plattform**
  Unabhängig
- **Autoren**

Michael Bräuer arbeitet in der Systemberatung bei Oracle Deutschland und beschäftigt sich mit serverseitiger Anwendungsentwicklung im Java-Umfeld. Sie erreichen ihn unter michael.braeuer@oracle.com.

Frank Schwarz ist IT-Berater mit Schwerpunkt auf den Java-Enterprise-Technologien. Er ist Mitinhaber der buschmais GbR, eines Beratungshauses mit Sitz in Dresden. Sie erreichen ihn unter frank.schwarz@buschmais.com.

Schon in der vorangegangenen Ausgabe hatte **database pro** vier neue Features von EclipseLink vorgestellt [1]. Diesmal kommen noch zwei hinzu: virtuelle Felder in Abfragen sowie virtuelle Objektmodelle.

Bei der Konzeption von Anwendungen wird oft gefordert, dass sich die fachliche Modularisierung auch in den Programmstrukturen wiederfinden solle. Im Idealfall entsteht dann ein Datenmodell, welches aus Gruppen von *Entity*-Klassen besteht, die eine Beziehung nur zu anderen Gruppenmitgliedern eingehen. Existieren aber Anforderungen, die modulübergreifende Abfragen nach sich ziehen, ist man mit JPA-Standardmitteln sehr schnell am Ende – meist muss die Modularisierung an dieser Stelle aufgegeben werden.

Um die fachliche Abgrenzung im Datenmodell beibehalten zu können, existiert in EclipseLink das Konzept „Query-Key". Ein Query-Key entspricht einem Attribut der *Entity*-Klasse, ohne dass dafür ein Feld in der Klasse existiert. Dieses virtuelle Attribut kann sich auf Tabellenspalten beziehen, aber auch einwertige oder mehrwertige Beziehungen zu anderen Entitäten abbilden. Neu in EclipseLink 2.1 ist die Möglichkeit, diese virtuellen Attribute wie normale Attribute in JPQL-Abfragen zu verwenden – zuvor hätte man hier auf das native Expressions-API ausweichen müssen.

Auf das Beispielmodell aus dem vorangegangenen Heft angewendet, besteht Java-seitig keine Beziehung zwischen *Dienstreiseantrag* und *Spesenabrechnung*. In einer Spesenabrechnung wird lediglich die Nummer des Dienstreiseantrags vermerkt, auf den sich die Abrechnung bezieht. Mit EclipseLink 2.1 kann man dennoch folgende JPQL-Abfrage formulieren:

```
SELECT s FROM Spesenabrechnung s WHERE
s.dienstreiseantrag.projekt.name =
'DesignOne'
```

Hier ist *dienstreiseantrag* ein Query-Key in der Klasse *Spesenabrechnung*. Er kann in der Abfrage wie ein normales Attribut verwendet werden – also auch innerhalb längerer Navigationspfade. Ein Query-Key gehört zu den Mapping-Metadaten und wird einmalig, am besten in einem *SessionCustomizer*, deklariert, siehe Listing 1.

Virtuelle Objektmodelle

Neben der Möglichkeit, einzelne Attribute zu virtualisieren, gestattet EclipseLink es auch, das gesamte Objektmodell – oder Teile daraus – zu virtualisieren. Davon profitieren Anwendungen, die im Wesentlichen nur CRUD-Funktionalitäten anbieten, aber auf keinem festen Datenbankschema operieren. Ändert sich das Datenbankschema, muss eine solche Anwendung nicht angepasst und neu kompiliert werden – ein Neustart würde ausreichen. Listing 2 zeigt, wie die Klassen *Spesenabrechnung* und *Kostenstelle* zur Laufzeit erzeugt und zwischen ihnen eine unidirektionale N-zu-1-Beziehung aufgebaut werden kann. Auch hier kommt wieder ein *SessionCustomizer* zum Einsatz.

Ansatzpunkt für die Erzeugung virtueller *Entity*-Klassen zur Laufzeit ist der *DynamicClassLoader*. Mit seiner Hilfe lassen sich *DynamicEntity*-Klassen definieren. Sind die Klassen erzeugt, kann mithilfe eines *DynamicTypeBuilder* das Mapping dieser Klassen spezifiziert werden. Reichen die Möglichkeiten des *DynamicTypeBuilder* nicht aus, kann ohne Weiteres auch auf das EclipseLink-Mapping-API zurückgegriffen werden. Zum Abschluss werden die neu erzeugten *DynamicEntity*-Klassen am *DynamicHelper* registriert und stehen damit der Anwendung wie traditionelle JavaBean-Modellklassen zur Verfügung.

Angenommen, *Spesenabrechnung* und *Kostenstelle* sind virtuelle Modellklassen, dann gestaltet sich das Auffinden einer Kostenstelle mit dem Namen *k0815* wie folgt:

```
DynamicHelper helper = new
 DynamicHelper(
 JpaHelper.getServerSession( emf));
Class<? extends DynamicEntity>
 kostenstelleClass =
  helper.getType("Kostenstelle").
  getJavaClass();
DynamicEntity k0815 = em.find(
 kostenstelleClass, "k0815");
```

Die Änderung der Kostenstellen-Beschreibung:

```
k0815.set("beschreibung",
 "allg. Reisekosten");
```

Alle Spesenabrechnungen zu dieser Kostenstelle zu ermitteln, klappt mit JPQL so:

```
TypedQuery<? extends DynamicEntity>
query = em.createQuery("SELECT s FROM
 Spesenabrechnung s WHERE .kostenstelle =
 :param", DynamicEntity.class);
query.setParameter("param", k0815);
List<? extends DynamicEntity>
 spesenabrechnungen =
 query.getResultList();
```

An diesem Beispiel ist sehr gut zu erkennen, dass keine Unterscheidung zwischen „echten" und virtuellen Klassen getroffen wird. Abschließend noch einmal die gleiche Abfrage mithilfe des Criteria-API:

```
EntityManager em =
 emf.createEntityManager();
Class<? extends DynamicEntity>
 spesenabrechnungClass = helper.getType(
 "Spesenabrechnung").getJavaClass();
CriteriaBuilder cb =
 emf.getCriteriaBuilder();
CriteriaQuery<DynamicEntity> query =
 cb.createQuery(DynamicEntity.class);
Root<? extends DynamicEntity>
 spesenabrechnungRoot = query.
 from(spesenabrechnungClass);
query
 .select(spesenabrechnungRoot)
 .where(cb.equal(
  spesenabrechnungRoot.get(
  "kostenstelle"), k0815));
List<? extends DynamicEntity>
 spesenabrechnungen = em.createQuery(
  query).getResultList();
```

Ausblick

Die vorgestellten Beispiele zeigen, welche Anforderungen reale Projekte an die Persistenzschicht stellen können, die sich mit den Mitteln des JPA-Standards nur schwer umsetzen lassen. Nicht nur Performance-Optimierungen, sondern auch funktionale Erweiterungen ergänzen die standardisierten Möglichkeiten.

Alle vorgestellten Erweiterungen gliedern sich in das Java-Persistence-API ein, sodass ein High-Level-Zugriff immer noch über JPA stattfinden kann. Allerdings sind bislang noch nicht alle Kombinationen implementiert. Wer beispielsweise Query-Keys über das Criteria-API nutzen möchte, muss dies zunächst als Feature-Request bei EclipseLink einstellen. Die hier vorgestellten Erweiterungen sind alle auf diese Art entstanden. **[bl]**

[1] Michael Bräuer, Frank Schwarz: Über JPA hinaus; database pro 6/2010, Seite 80 ff.

Listing 1: Virtuelle Objektbeziehungen über Query-Keys

```
public class SpesenabrechnungCustomizer implements SessionCustomizer {

  @Override
  public void customize(Session session) throws Exception {
    ClassDescriptor spesenabrechnungDescriptor =
      session.getDescriptor(Spesenabrechnung.class);
    ClassDescriptor dienstreiseantragDescriptor =
      session.getDescriptor(Dienstreiseantrag.class);

    DatabaseMapping antragsnummerMapping = spesenabrechnungDescriptor
      .getMappingForAttributeName("antragsnummer");
    DatabaseField antragsnummerField = antragsnummerMapping.getField();
    DatabaseField dienstreiseantragPKField = dienstreiseantragDescriptor
      .getPrimaryKeyFields().get(0);
    ExpressionBuilder expr = new ExpressionBuilder();
    Expression joinExpression = expr.getField(dienstreiseantragPKField)
      .equal(expr.getParameter(antragsnummerField));

    OneToOneQueryKey dienstreiseantragKey = new OneToOneQueryKey();
    dienstreiseantragKey.setReferenceClass(Dienstreiseantrag.class);
    dienstreiseantragKey.setJoinCriteria(joinExpression);
    dienstreiseantragKey.setName("dienstreiseantrag");
    spesenabrechnungDescriptor.addQueryKey(dienstreiseantragKey);
  }
}
```

Listing 2: Definition eines virtuellen Objektmodells

```
public class VirtualModelCustomizer implements SessionCustomizer {
  @Override
  public void customize(Session session) throws Exception {
   DynamicHelper helper = new DynamicHelper((DatabaseSession) session);
   DynamicClassLoader dcl = helper.getDynamicClassLoader();

   Class<?> spesenabrechnungClass = dcl.createDynamicClass(
     "app.modell.Spesenabrechnung");
   DynamicTypeBuilder spesenabrechnungBuilder = new JPADynamicTypeBuilder(
     spesenabrechnungClass, null, "SPESENABRECHNUNG");
   spesenabrechnungBuilder.setPrimaryKeyFields("ID");
   spesenabrechnungBuilder.addDirectMapping("id", long.class, "ID");
   spesenabrechnungBuilder.addDirectMapping("betrag",
    BigDecimal.class, "BETRAG");

   Class<?> kostenstelleClass = dcl.createDynamicClass(
     "app.modell.Kostenstelle");
   DynamicTypeBuilder kostenstelleBuilder = new JPADynamicTypeBuilder(
      kostenstelleClass, null, "KOSTENSTELLE");
   kostenstelleBuilder.setPrimaryKeyFields("NAME");
   kostenstelleBuilder.addDirectMapping("name", String.class, "NAME");
   kostenstelleBuilder.addDirectMapping(
     "beschreibung", String.class, "BESCHREIBUNG");

   spesenabrechnungBuilder.addOneToOneMapping(
     "kostenstelle", kostenstelleBuilder.getType(), "KOSTENSTELLE");
   helper.addTypes(false, false,
     spesenabrechnungBuilder.getType(),
     kostenstelleBuilder.getType());
  }
}
```