Neue Features für EclipseLink

Über JPA hinaus

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 stellt EclipseLink kurz vor und betrachtet einige der wichtigsten Neuerungen der aktuellen Version 2.1.

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

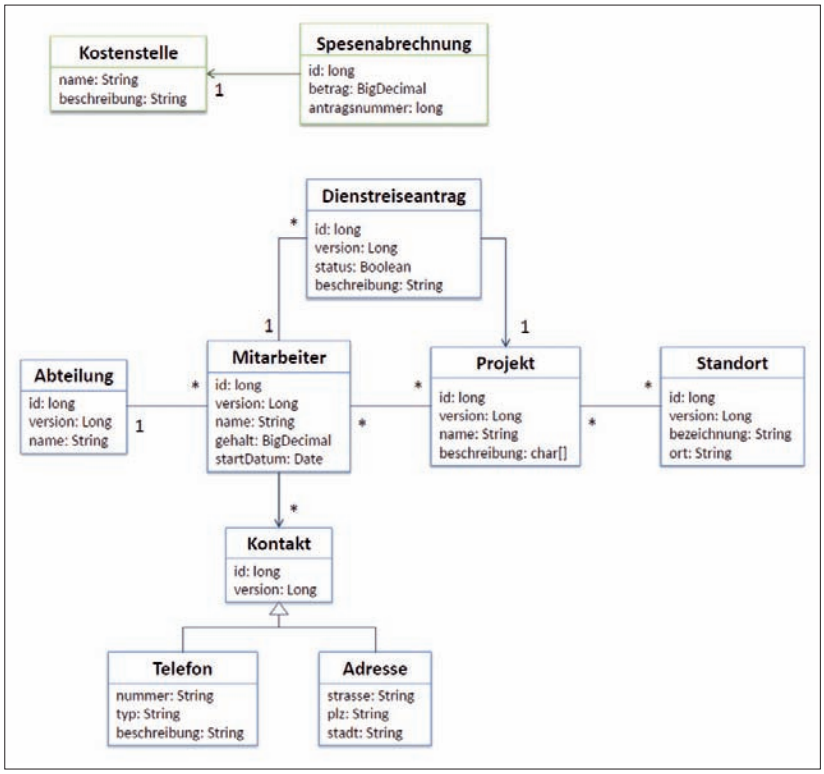
Michael Bräuer arbeitet in der Systemberatung bei Oracle Deutschland und beschäftigt sich mit serverseitiger Anwendungsentwicklung im Java-Umfeld. Sie erreichen ihn unter michael.braeuer@oracle.com.

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.

Über die JPA-Standardkonformität (Java Persistence API) hinaus buhlen die bekannten O/R-Persistenz-Frameworks mit zusätzlichen Features um die Gunst der Java-Entwickler. Für die Version 2.1 griff das EclipseLink-Team verschiedene interessante Entwicklerwünsche auf und setzte sie JPA-konform um. Einige dieser Erweiterungen sollen hier vorgestellt werden.

Als Ende 2009 die Version 2.0 des Java Persistence API freigegeben wurde, war das Echo durchweg positiv. Die neue Version harmonisierte die bereits vorhandenen Features der JPA-1.0-Implementierungen. Trotzdem werden an die Frameworks auch weiterhin Anforderungen gestellt, die der Standard nicht abdeckt – etwa Performance-Optimierungen, datenbankseitig implementierte Funktionen in Abfragen oder das Feintuning des Fetch-Verhaltens beim Laden von Entitäten. Auch Erleichterungen im Formulieren von Abfragen – beispielsweise über virtuelle Felder oder über Downcasts – gehören heute in den Bereich der JPA-Erweiterungen. Immer wieder nachgefragt wird auch eine Möglichkeit, die Modellklassen samt Mappings erst zur Laufzeit festzulegen. EclipseLink 2.1 bringt diese Möglichkeiten nun mit [1], [2].

Das Objektmodell (Bild 1)



Das Objektmodell

Zur Veranschaulichung der JPA-Erweiterungen in EclipseLink wird das Objektmodell in Bild 1 herangezogen. Keine Rolle spielt dabei das Datenbankschema; das Mapping kann als vollständig JPA-2.0-konform angenommen werden.

Für einige Beispiele wird es erforderlich sein, zur Laufzeit in die Mapping-Beschreibungen einzugreifen. Hierfür bietet EclipseLink verschiedene Einstiegspunkte an; zwei davon werden im Kasten „EclipseLink erweitern“ beschrieben.

Datenbankfunktionen in JPQL-Queries

Oft steht man vor der Herausforderung, große Datenmengen verarbeiten oder auf Basis von Berechnungen das Abfrageergebnis einschränken zu müssen. Eine naive Art der Vorgehensweise bestände darin, alle für die Berechnung benötigten Daten mittels JPQL abzufragen, um dann im Java-Adressraum alle gewünschten Berechnungen vorzunehmen. Dieser Ansatz ist jedoch im Allgemeinen hinsichtlich der Kosten, die durch Objekterzeugung und Garbage-Collection entstehen, sehr ineffizient, besonders dann, wenn Anzahl und Größe der Objekte hoch sind. Dann werden Objektstrukturen erzeugt, die unter Umständen für eine weitere Verarbeitung gar nicht benötigt werden.

Zur Demonstration der Problematik soll folgendes Beispiel dienen: Sucht man etwa alle Projekte, deren Projektbeschreibungen (CLOB-Spalte in der Datenbank) eine gewisse Zeichenkette enthalten, und es existieren 1.000 Projekte mit einer durchschnittlichen Größe der Projektbeschreibung von 500 KByte, so lässt sich Folgendes feststellen: Treffen nur zwei Projekte das Kriterium, so lädt man 998 Projekte umsonst und verschwendet mit dieser Abfrage mindestens 487 MByte im Heap. Hinzu kommen Ressourcen für die eigentliche Auswertung und die anschließende Garbage-Collection.

Eine Möglichkeit, dieses Problem zu lösen, besteht darin, die Logik für Berechnung und Einschränkung auf Datenbankseite durch Datenbankfunktionen zu implementieren. Das hat den Vorteil, dass nur die Informationen in den Java-Adressraum gelangen, die dort auch benötigt werden. Für den Aufruf solcher Funktionen in JPQL steht mit EclipseLink 2.1 der *FUNC-*

Operator zur Verfügung. Das eben genannte Beispiel ließe sich dann unter Verwendung einer Datenbankfunktion *CLOB_CONTAINS* (siehe Listing 1 und unter [3]) in JPQL wie folgt lösen:

```
SELECT p FROM Projekt p
WHERE FUNC(▷
  'CLOB_CONTAINS', p.beschreibung,
  'Abgeschlossen'▷
) = 'TRUE'
```

Diese Abfrage wird in folgendes SQL-Statement umgewandelt:

```
SELECT ID, PROJEKT_NAME, BESCHREIBUNG,
  AUFTRG_ID FROM Projekte
WHERE (▷
  CLOB_CONTAINS(BESCHREIBUNG, ?) = ?▷
) bind => [Abgeschlossen, TRUE]
```

Gerade für datenbankintensive Berechnungen, für die schon fertige Datenbankfunktionalität existiert, sind Berechnungen auf Datenbankseite sehr effizient möglich. Der *FUNC*-Operator kann im *SELECT*-Teil wie auch im *WHERE*-Teil einer JPQL-Abfrage vorkommen.

Downcasts in Queries

Mit JPA 2.0 gibt es die Möglichkeit, bei Abfragen auf Typen einer Klassenhierarchie einzuschränken. Fragt man etwa nach Mitarbeitern, zu denen auch eine Adresse gespeichert ist, kann eine JPQL-Abfrage wie folgt definiert werden:

```
SELECT m FROM Mitarbeiter m
JOIN m.kontakte ktkt
WHERE TYPE(ktkt) = Adresse
```

Dabei ist zu beachten, dass im Domänenmodell *Adresse* ein Subtyp der Klasse *Kontakt* ist und die Abfrage somit auf einen Subtyp beschränkt wird. Die Abfrage ist polymorph, das heißt, wären weitere Subtypen unter *Adresse* definiert, würden diese ebenfalls in der Ergebnismenge berücksichtigt. Will man jedoch weitere Einschränkungen auf Attributen eines Subtyps vornehmen – etwa die Suche auf Mitarbeiter einschränken, deren Adresse eine spezielle Postleitzahl aufweist –, ist Kreativität gefragt. EclipseLink 2.1 schließt diese Lücke.

In JPQL kann der Operator *TREAT(Y AS X)* dazu benutzt werden, die Konvertierung eines Pfadausdrucks *Y* in den Typ *X* innerhalb der *FROM*-Klausel durchzuführen. Dies würde für eben genanntes Beispiel so aussehen:

```
SELECT m FROM Mitarbeiter m
JOIN TREAT(m.kontakte AS Adresse) adr
WHERE adr.plz = '15469'
```

Sobald eine Konvertierung in den Subtyp durchgeführt wurde, können auch dessen Attribute benutzt werden.

Innerhalb des Criteria-API wurde die *as*-Methode von *javax.persistence.criteria.Expression* für diesen Zweck erweitert. Die analoge Criteria-API-Abfrage schreibt sich so:

```
CriteriaBuilder cb =
  em.getCriteriaBuilder();
CriteriaQuery<Mitarbeiter>
   cq = cb.createQuery(▷
   Mitarbeiter.class);
Root<Mitarbeiter> m =
   cq.from(▷
   Mitarbeiter.class);
Expression<String> plzExpression =
  ((Path<Adresse>) m
    .join(Mitarbeiter_▷
    .kontakte).as(Adresse.class))
    .get(Adresse_.plz);
cq.select(m).where(▷
  cb.equal(plzExpression, "15469"));
```

Listing 1: Suche in CLOB

```
CREATE OR REPLACE FUNCTION CLOB_CONTAINS
( p_beschreibung IN CLOB,
  p_kriterium IN VARCHAR2
) RETURN VARCHAR2 AS
v_occurence  INTEGER := 1;
v_offset     INTEGER := 1;
v_return     INTEGER;
BEGIN
  v_return := DBMS_LOB.INSTR(p_beschreibung,
     p_kriterium, v_offset, v_occurence);
  IF v_return = 0 THEN
     RETURN 'FALSE';
  ELSE
    RETURN 'TRUE';
  END IF;
END CLOB_CONTAINS;
```

Fetch-Gruppen

Häufig will man bei einer Abfrage steuern, welche Inhalte von Objekten oder eines Objektgraphen geladen werden sollen. Dies ist insbesondere dann der Fall, wenn man Strukturen serialisieren und an einen entfernt arbeitenden Client schicken möchte.

Grundsätzlich folgt das Laden der Attribute einer Entität festen Konventionen: Es werden alle Attribute geladen, die nicht per Default oder explizit mit der Mapping-Information *fetch=FetchType.LAZY* versehen sind. Nun taucht jedoch in der Praxis die Anforderung auf, dieses Verhalten bei Abfragen zu überschreiben, das heißt also:

- Entitäten sofort zu laden, die in Beziehung – auch über Dritte – mit den Entitäten der Rückgabemenge stehen;
- gewisse Felder solcher Entitäten beim Laden einzubeziehen oder auszuschließen.

Eine Lösung wäre die Verwendung von Projektionen in der Abfrage; EclipseLink 2.1 bietet aber einen eleganteren Weg an. Hierfür wurde die Funktionalität der Klasse *org.eclipse.persistence.queries.FetchGroup* erweitert. Eine Fetch-Gruppe dient dazu, solche Attribute zu definieren, die beim Ladevorgang gefüllt werden sollen. Dabei ist zu beachten, dass gewisse Attribute, die vom Persistenz-Provider für die Verwaltung benötigt ▶

Listing 2: SQL-Statement zur Query

```
SELECT t1.ID, t1.PROJEKT_NAME,
  t1.BESCHREIBUNG, t1.VERSION,
  t0.MITARBEITER_ID
FROM KONTAKTE t3, MITARBEITER t2,
  PROJEKTE t1,
  PROJEKTE_MITARBEITER t0
WHERE (
  (((t0.MITARBEITER_ID = t2.ID)
  AND (t1.ID = t0.PROJEKT_ID))
  AND (t3.PLZ = ?))
  AND ((t3. MITARBEITER_ID = t2.ID)
  AND (t3.KTYPE = ?))
) bind => [15469, ADRESSE]
```

Listing 3: Mit Projektstandorten

```
SELECT t1.ID, t1.BEZEICHNUNG,
  t1.ORT, t1.VERSION, t0.PROJEKT_ID
FROM
  KONTAKTE t5, MITARBEITER t4,
  PROJEKTE_MITARBEITER t3, PROJEKTE
  t2, STANDORTE t1,
  PROJEKTE_STANDORTE t0
WHERE (
  (((t0.PROJEKT_ID = t2.ID)
  AND (t1.ID = t0.STANDORT_ID))
  AND (((t3.MITARBEITER_ID = t4.ID)
  AND (t2.ID = t3.PROJEKT_ID))
  AND (t5.PLZ = ?)))
  AND ((t5.MITARBEITER_ID = t4.ID)
  AND (t5.KTYPE = ?))
) bind => [15469, ADRESSE]
```

werden, automatisch zur Fetch-Gruppe gehören, dazu zählen zum Beispiel das *@Id*- oder das *@Version*-Feld.

Eine Fetch-Gruppe kann sowohl statisch als Teil der Mapping-Information (Named Fetch-Group) als auch dynamisch zur Laufzeit definiert werden. Dabei lassen sich Fetch-Gruppen auch schachteln. Fetch-Gruppen werden bei einer Abfrage mittels Query-Hint unter Angabe der Fetch-Gruppe benutzt (*QueryHints.FETCH_GROUP*). Das Laden von Beziehungen steuern Sie durch Angabe eines weiteren Query-Hints: *QueryHints.FETCH_GROUP_LOAD*.

Die Funktionalität soll an folgendem Beispiel erläutert werden: Gesucht sind diejenigen Mitarbeiter, deren Postleitzahl innerhalb der Adresse (Attribut *plz*) einem gewissen Wert entspricht. Geladen werden sollen dabei die Namen der Mitarbeiter, deren Abteilungen, die zugewiesenen Projekte ohne Beschreibungen sowie die Projektstandorte. Dafür müssen zwei Fetch-Gruppen definiert werden:

```
FetchGroup projektGr =
  new FetchGroup();
projektGr.addAttribute("name");
FetchGroup mitarbeiterGr =
  new FetchGroup();
mitarbeiterGr.addAttribute("name");
mitarbeiterGr.addAttribute("gehalt");
mitarbeiterGr.addAttribute(▷
  "abteilung"); // Beziehungsattribut
mitarbeiterGr.addAttribute(▷
  "projekte", projektGr);
mitarbeiterGr.addAttribute(▷
  "projekte.standorte");
```

Dann kommt obige Abfrage unter Verwendung der beschriebenen Query-Hints zum Zuge:

```
TypedQuery<Mitarbeiter>
mitarbeiterQuery =
  em.createQuery("SELECT m FROM
  Mitarbeiter m JOIN TREAT(m.kontakte
  AS Adresse) adr WHERE adr.plz =
  '15469'", Mitarbeiter.class);
mitarbeiterQuery.setHint(▷
  QueryHints.FETCH_GROUP,
  mitarbeiterGr);
mitarbeiterQuery.setHint(▷
  QueryHints.FETCH_GROUP_LOAD,
  HintValues.TRUE);
```

Dabei entstehen unter Umständen sehr viele SQL-Statements. Zunächst werden alle Mitarbeiter mit der gesuchten Adresse abgefragt:

```
SELECT t1.ID, t1.ANGEST_NAME,
  t1.GEHALT, t1.VERSION,
  t1.ABTEILUNG_ID FROM
KONTAKTE t0, MITARBEITER t1
WHERE (▷
  (t0.PLZ = ?) AND
  ((t0.MITARBEITER_ID = t1.ID)
  AND (t0.KTYPE = ?))▷
) bind => [15469, ADRESSE]
```

Dann wird für jeden Mitarbeiter durch jeweils ein SQL-Statement die Abteilung ermittelt und durch ein weiteres SQL-Statement herausgefunden, an welchen Projekten der Mitarbeiter beteiligt ist. Für jedes der gefundenen Projekte erfolgt eine weitere Abfrage auf die zugewiesenen Standorte. Die Anzahl der generierten SQL-Statements ist also unmittelbar von der Anzahl der involvierten Objektinstanzen abhängig.

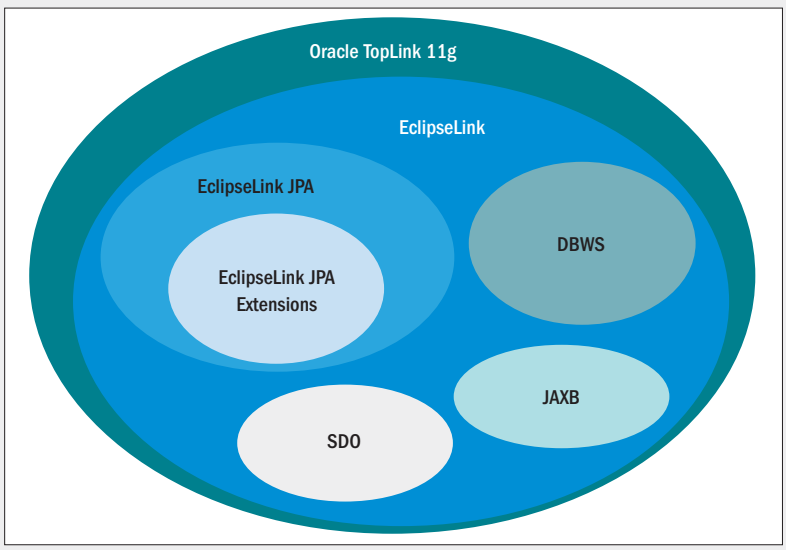
Optimierungen beim Laden

Unabhängig von der Verwendung von Fetch-Gruppen können Ladevorgänge bei Abfragen mittels Join-Fetching und Batch-Fetching optimiert werden. Diese Funktionalität steht schon

Über EclipseLink

Das Eclipse Persistence Services Project (kurz: EclipseLink) wurde 2007 als Eclipse Runtime Project ins Leben gerufen. EclipseLink ist ein Open-Source-Framework und basiert auf Oracle TopLink, das seit mehr als einer Dekade als kommerzielles Persistenz-Framework im Java-Umfeld im Einsatz ist [4]. Oracle brachte die Codebasis der TopLink 11g Preview in das Eclipse-Projekt ein und erweiterte diese um wertvolle Features, wie zum Beispiel eine JPA-1.0-Implementierung.

Wie Oracle TopLink geht EclipseLink über die Funktionalität eines O/R-Mappers hinaus; es ist ein Framework, mit dem sich verschiedenste Persistenz- bzw. Mapping-Anforderungen lösen lassen. Es unterstützt u.a. das Java Persistence API (JPA), Java Architecture for XML Binding (JAXB), Service Data Objects (SDO) und Database Webservices (DBWS), sowohl in Java SE, Java EE oder OSGi-Umgebungen. EclipseLink ist die Referenzimplementierung von JPA 2.0 (Bild 2).



Der Aufbau von EclipseLink (Bild 2)

seit langer Zeit in TopLink und somit auch in EclipseLink zur Verfügung. In beiden Fällen wird das Verhalten der Abfrage über EclipseLink-spezifische Query-Hints gesteuert. Werden Join-Fetching und Batch-Fetching ohne weitere Angabe von Fetch-Gruppen verwendet, kommt die Default-Fetch-Group zum Tragen.

Gerade bei 1:1-Beziehungen lohnt sich Join-Fetching, wenn die an der Beziehung beteiligten Entitäten sofort mitgeladen werden sollen. Dabei wird die ursprüngliche SQL-Abfrage um einen oder mehrere Joins angepasst und der *SELECT*-Teil der Abfrage erweitert.

Zurück zur JPQL-Abfrage *mitarbeiterQuery*. Diese wird mit dem entsprechenden Query-Hint versehen, um einen Join-Fetch der Abteilungen zu erzwingen, also:

```
mitarbeiterQuery.setHint(▷
  QueryHints.FETCH, "m.abteilung");
```

Das Ergebnis ist folgendes SQL-Statement:

```
SELECT t1.ID, t1.ANGEST_NAME,
  t1.GEHALT, t1.STARTDATUM,
  t1.VERSION, t1.ABTEILUNG_ID, t0.ID,
  t0.VERSION, t0.ABT_NAME
FROM ABTEILUNGEN t0, KONTAKTE t2,
  MITARBEITER t1
WHERE (▷
  (t2.PLZ = ?) AND (((t2.
  MITARBEITER_ID = t1.ID) AND
  (t2.KTYPE = ?)) AND (t0.ID =
  t1.ABTEILUNG_ID))▷
) bind => [15469, ADRESSE]
```

Die Objekte vom Typ *Mitarbeiter* und *Abteilung* werden dann sofort erzeugt. Zusätzliche SQL-Abfragen zur Ermittlung der Abteilung für jeden Mitarbeiter sind nicht nötig.

Batch-Fetching ist ein nachgelagertes Fetching, das für mehrwertige Beziehungen interessant ist, um die Anzahl der abgesetzten SQL-Abfragen zu minimieren. Es wird genau ein zusätzliches SQL-Statement ausgeführt, um alle relevanten Objekte des Beziehungstyps zu ermitteln. Wie beim Join-Fetch wird auch hier ein Join über die zugrunde liegenden Tabellenstrukturen hergestellt, jedoch wird der *SELECT*-Teil des Statements auf den Entity-Typ reduziert, der im Query-Hint per Navigation definiert ist.

Erweitert man obiges Beispiel um folgenden Query-Hint:

```
mitarbeiterQuery.setHint(▷
  QueryHints.BATCH, "m.projekte");
```

so wird das SQL-Statement aus Listing 2 generiert, um alle für die obige JPQL-Query relevanten Projekte auf einmal abzufragen. Sollen zusätzlich auch die Standorte der Projekte geladen werden, so erzeugen Sie mit dem Query-Hint

```
mitarbeiterQuery.setHint(QueryHints.▷
  BATCH, "m.projekte.standorte");
```

mithilfe einer weiteren Navigation zu den Standorten eine zusätzliche SQL-Abfrage (Listing 3).

Zu beachten ist, dass bei der Navigation entlang vieler Beziehungen viele Datenbanktabellen in Verbindung gebracht werden, also die Datenmenge, die datenbankseitig verarbeitet werden muss, sehr groß werden kann. Daher kann ab EclipseLink 2.1 der Join-Modus durch Subselects mit *IN* oder *EXISTS* ersetzt werden. Welcher der beiden SQL-Operatoren zum Einsatz kommt, wird mit *QueryHints.BATCH_TYPE* und den Ausprägungen *BatchFetchType.EXISTS* und *BatchFetchType.IN* gesteuert. Exemplarisch sei die Nutzung von *BatchFetchType.IN* gezeigt. Fügt man

```
mitarbeiterQuery.setHint(▷
  QueryHints.BATCH_TYPE,
  BatchFetchType.IN);
```

zu den schon definierten Query-Hints hinzu, so erhält man anstelle der letzten beiden SQL-Abfragen folgende SQL-Statements:

```
SELECT t1.ID, t1.PROJEKT_NAME,
  t1.BESCHREIBUNG, t1.VERSION,
  t0.MITARBEITER_ID
FROM PROJEKTE_MITARBEITER t0,
  PROJEKTE t1
WHERE (▷
  (t1.ID = t0.PROJEKT_ID) AND
  (t0.MITARBEITER_ID IN (?,?,?,?,?))▷
) bind => [2, 4, 5, 6, 11]

SELECT t1.ID, t1.BEZEICHNUNG, t1.ORT,
  t1.VERSION, t0.PROJEKT_ID
FROM PROJEKTE_STANDORTE t0,
  STANDORTE t1
WHERE (▷
  (t1.ID = t0.STANDORT_ID) AND
  (t0.PROJEKT_ID IN (?,?,?,?))▷
) bind => [1, 2, 3, 4]
```

An die erste Query werden die gefundenen Mitarbeiter übergeben und deren Projekte geladen. In die zweite Abfrage gehen die Projekte ein, um die Standorte zu laden. Dies vermeidet den Aufbau großer kartesischer Produkte. **[bl]**

[1] www.eclipselink.org
[2] http://wiki.eclipse.org/EclipseLink/Release/2.1.0
[3] http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28419/d_lob.htm#i998546
[4] www.oracle.com/technetwork/topics/history-of-toplink-101111.html
[5] http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)#r7c1-t26

EclipseLink erweitern

Über zwei gut definierte Erweiterungspunkte lassen sich Mappings in EclipseLink erweitern: Der erste Einstiegspunkt wird Descriptor-Customizer genannt.

Ein Descriptor-Customizer ist ein Callback, welcher nach der Initialisierung der Entity-Metadaten aufgerufen wird. Hierzu muss der Customizer das Interface *org.eclipse.persistence.config.DescriptorCustomizer* implementieren. Die Entity-Klasse muss mit *@Customizer(MyDescriptorCustomizer.class)* annotiert werden.

Ein Descriptor-Customizer unterliegt der Einschränkung, jeweils nur die Metadaten einer einzelnen Entity-Klasse manipulieren zu können. Möchte man beispielsweise eine zusätzliche Beziehung zwischen zwei Entity-Klassen etablieren, so empfiehlt sich der Einsatz eines Session-Customizers.

Ein Session-Customizer folgt ebenfalls dem Callback-Muster. Das Interface heißt hier *org.eclipse.persistence.config.SessionCustomizer*. Der Callback wird aufgerufen, nachdem die EntityManagerFactory initialisiert ist. Über den Callback lässt sich auf sämtliche Mapping-Deskriptoren Zugriff nehmen. Ein Session-Customizer wird über den *persistence.xml-Deployment*-Deskriptor aktiviert [5].