Java Persistence API 2.0

Zu neuen Ufern

Das Java Persistence API geht in die nächste Version und hält eine Vielzahl an Ergänzungen bereit. Bleibt zu betrachten, welche Vorteile diese bringen und wie Sie sie praktisch einsetzen. **Frank Schwarz**

Auf einen Blick

Inhalt
Erfolgreich eingeführte Programmierschnittstellen wie das JPA werden nicht nur permanent gepflegt und fehlerbereinigt, sondern entsprechend den Praxisanforderungen erweitert. Das JPA 2.0 steht kurz vor der Fertigstellung und bietet eine Vielzahl an Erweiterungen, die sich sinnvoll in eigene Projekte integrieren lassen. Dieser Artikel gibt einen Überblick über die wesentlichen Neuerungen.

Plattform
Java

Technik/Anwendung
Neuerungen zum Java Persistence API

Voraussetzungen
JPA 2.0

Autor
Frank Schwarz ist Mitinhaber der buschmais GbR, eines Zusammenschlusses von Softwarearchitekten, deren Schwerpunkte in den Bereichen Systemintegration, Verteilung, Skalierbarkeit und Persistenz liegen und die im Kundenverbund Lösungen im Umfeld von Java-EE- und .NET-Technologien entwerfen und implementieren.

Die Fertigstellung der Java Enterprise Edition 6 steht kurz bevor. Nach mehrmaliger Terminverschiebung soll nun bis zum Jahresende alles in „trockenen Tüchern" sein. Ziel der Neuauflage ist eine bessere Verzahnung der enthaltenen Standards und eine Erweiterung der angebotenen Funktionalitäten. Im Zuge dessen hat auch der Persistenz-Standard JPA wesentliche Ergänzungen erfahren. Die Spezifikation des Java Persistence API wurde im Mai 2006 fertiggestellt und liegt damit mehr als drei Jahre zurück. Die Einfachheit des Programmiermodells und die Ausdrucksstärke der Abfragesprache haben diese Mapping-Technologie seither zum Erfolg geführt. In manchen Projekten war allerdings festzustellen, dass einige Features nur unter Umgehung des Standards zu erzielen waren. Zweifel an einer einfachen Austauschbarkeit der Mapper-Implementierung entstanden. Die zweite Version des Java Persistence API kann diese Bedenken weitgehend zerstreuen. Die Neuerungen sind so zahlreich, dass viele hier nur kurz angerissen werden. Nachfolgend werden die Erweiterungen der Mapping-Möglichkeiten, die Ergänzungen der Abfragesprache und die Änderungen des API näher beleuchtet.

Das Mapping

Eine empfindliche Einschränkung des bisherigen JPA-Standards stellte bei Beziehungsmappings die enge Kopplung an die Art der Assoziation dar. Der Standard legte beispielsweise für 1:n-Beziehungen fest, dass unidirektionale Assoziationen eine Join-Tabelle verwenden sollten, bidirektionale Assoziationen eine Join-Spalte. Nun ist es leider so, dass in bestehenden Datenbankschemas 1:n-Beziehungen fast immer durch eine Fremdschlüssel-Spalte in der Detailtabelle definiert werden – unabhängig davon, wie sich die Beziehung Java-seitig abbildet. JPA 2.0 hebt diese Einschränkung ausdrücklich auf. Bild 1 zeigt die zusätzlichen Möglichkeiten, eine 1:n-Beziehung darzustellen. Der Standard geht sogar so weit, selbst Entitäten einer 1:1-Beziehung über eine Join-Tabelle assoziieren zu lassen.

Bei der Definition von 1:1- und 1:n-Beziehungen ist die Option *orphanRemoval* hinzugekommen. Durch dieses Schlüsselwort wird der O/R-Mapper angewiesen, deassoziierte Entitäten automatisch zu löschen:

```
@Entity
public class Kunde {
  @OneToMany(orphanRemoval=true)
  private List<Adresse> adressen;
}
```

Dieses Mapping hat zur Folge, dass alle Adress-Instanzen, die aus der Liste *adressen* per *kundeX.getAdressen().remove(Y)* entfernt werden, automatisch beim nächsten Commit auch aus der Datenbank verschwinden. Ohne *orphanRemoval* würde die Assoziation zwar aufgelöst, die Adress-Entität bliebe in der Datenbank aber bestehen. Eine weitere Neuerung beim Mapping von Beziehungen ist, eingebettete Objekte zu schachteln. Ein eingebettetes Objekt (*Embeddable*) ist Java-seitig ein eigenständiges Objekt, das datenbankseitig keine eigene Identität besitzt. Ein Embeddable mit der Kardinalität 1 wird in zusätzlichen Spalten der Tabelle des Master-Objekts abgelegt. Beispielsweise könnte die Entität *Lieferung* mit einem Embeddable *Lieferungsdetail* verbunden werden. Mit JPA 2.0 kann das *Lieferungsdetail* wieder Embeddables für Gewicht und Volumen assoziieren, wobei Gewicht und Volumen jeweils wieder aus Größe und Maßeinheit bestehen könnten. Auf diese Weise ist es recht einfach möglich, zusammengesetzte Datentypen zu modellieren und modellweit anzuwenden. Trotz aller Versuche, die relationale Welt so objektorientiert wie möglich zu repräsentieren, ist bisher ein Merkmal des relationalen Modells immer wieder durchgeschlagen: Das Ergebnis einer Abfrage ist immer eine Menge oder Multimenge. Bestenfalls bei Anwendung eines Sortierkriteriums ließe sich das Abfrageergebnis als Liste auffassen. Wenn ein *Kunde*-Objekt mehrere *Adress*-Objekte als Liste assoziiert, kann die Reihenfolge der *Adress*-Objekte innerhalb der Liste bei jedem Neu-Laden eine andere sein. JPA 2.0 führt zur Lösung dieses Problems die Annotation *@OrderColumn* ein:

```
@Entity
public class Kunde {
 @OneToMany
 @OrderColumn
 private List<Adresse> adressen;
}
```

Dieses Mapping enthält die Anweisung an den O/R-Mapper, in der Tabelle *ADRESSE* eine

Spalte *ADRESSEN_ORDER* zu pflegen und darin die Reihenfolge der assoziierten *Adress*-Objekte pro Kunde als Index abzulegen. Sobald die *Adress*-Liste wieder geladen wird, wird automatisch in das SQL-Statement ein *ORDER BY ADRESSEN_ORDER* eingefügt, sodass die ursprüngliche Reihenfolge der *Adress*-Liste rekonstruiert wird. Mit JPA 2.0 können Sie auch Listen von primitiven Datentypen auf ein relationales Schema mappen:

```
@Entity
public class Adresse {
 @ElementCollection
 @OrderColumn
 private List<String> telNr;
}
```

Hier entsteht ein Datenbankschema, wie es in Bild 2 illustriert ist. Mit den Annotationen *@Column* und *@CollectionTable* ist die Tabellenstruktur weiter anpassbar. Neben primitiven Datentypen lassen sich auch Embeddables in einer Element-Collection ablegen. Auch die Abbildung von Map-Strukturen hat an Flexibilität gewonnen. Waren bisher nur primitive Felder des assoziierten Objekts als Map-Key erlaubt, sind demnächst auch primitive Typen, Enumerationen, Embeddables und sogar Entitäten als Map-Key zulässig. Ist beispielsweise ein Produkt über verschiedene Hersteller beziehbar, so können die Lieferkonditionen mit JPA 2.0 modelliert werden als:

```
@Entity
public class Produkt {
 @OneToMany
 @MapKeyJoinColumn(name="HID")
 @JoinTable(name="PROD_LIEFERKOND",
  joinColumns=@JoinColumn(name="PID"),
  inverseJoinColumns=
   @JoinColumn(name="LID"))
 private Map<Hersteller,
  Lieferkondition> lieferkonditionen;
}
```

Listing 1: Beispiel für eine Datenabfrage mit dem Criteria-Query-API

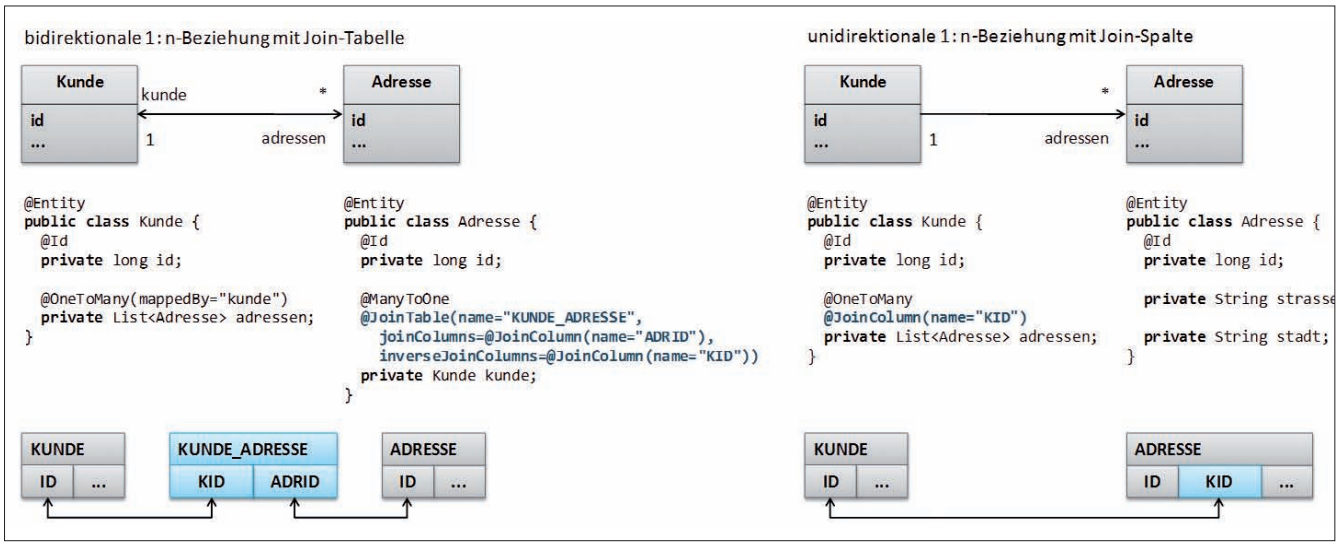
```
QueryBuilder queryBuilder = entityManagerFactory.getQueryBuilder();
CriteriaQuery<Bestellung> criteriaQuery
  = queryBuilder.createQuery(Bestellung.class);
Root<Bestellung> bestellungRoot = criteriaQuery.from(Bestellung.class);
ListJoin bestellposJoin = bestellungRoot.joinList("bestellpositionen")
Path produktPath = bestellposJoin.get("produkt");
Predicate lebensmittelPredicate
  = queryBuilder.equal(produktPath.type(), Lebensmittel.class);
criteriaQuery
  .select(bestellungRoot)
  .where(lebensmittelPredicate);
TypedQuery<Bestellung> query = entityManager.createQuery(criteriaQuery);
List<Bestellung> resultList = query.getResultList();
```

Sind *Hersteller* und *Lieferkonditionen* ebenfalls Entitäten, so ergibt sich ein Datenbankschema, wie es in Bild 3 dargestellt ist. Ähnlich reichhaltig ist der Zuwachs an Varianten, um zusammengesetzte Primärschlüssel zu definieren. Nun dürfen auch 1:1- und n:1-Beziehungen zum Bestandteil des Primärschlüssels erklärt werden. Eine Lieferscheinposition ist deshalb ohne Umschweife so deklarierbar:

```
@Entity
@IdClass(LieferscheinPositionPK.class)
public class LieferscheinPosition {
 @Id
 private long position;
  @Id
  @ManyToOne
  private Lieferschein lieferschein;
}
```

Im Beispiel wird der Fremdschlüssel zur *Lieferschein*-Tabelle automatisch zum Primär- ▶

Die zusätzlichen Mapping-Möglichkeiten von 1:n-Beziehungen (Bild 1)



schlüsselbestandteil der Lieferscheinposition. Dieses Feature nennt die JPA-2.0-Spezifikation auch *Derived Entities* – eine Lieferscheinposition ist existenzabhängig vom Lieferschein.

Die Abfragesprache

Die Abfragesprache JPQL hat einige kleinere Ergänzungen erfahren, um die neuen Mapping-Möglichkeiten besser widerzuspiegeln. Eine ausführliche Diskussion der Abfragesprache finden Sie in [1]. Möchte man alle Kunden samt ihrer ersten, hinterlegten Adresse erhalten, so lässt sich der *INDEX*-Operator anwenden:

```
SELECT k, a FROM Kunde k, IN
(k.adressen) a WHERE INDEX(a) = 0
```

Das Ergebnis dieser Abfrage ist ein Tupel aus *{Kunde, Adresse}*, das Java-seitig als Liste von Object-Arrays zurückgeliefert wird. Auch das erweiterte Map-Mapping ist Abfragen nicht hinderlich. Wollen Sie unter Verwendung des obigen Mappings zwischen Produkt, Hersteller und Lieferkondition alle Produkte erhalten, die vom Hersteller namens *X* bezogen werden können, so lässt sich dies wie folgt abfragen:

```
SELECT p FROM Produkt p, IN
(p.lieferkonditionen) lk WHERE
KEY(lk).name = 'X'
```

JPQL 2 wurde weiterhin um die folgenden Schlüsselwörter ergänzt: *NULLIF, CASE WHEN, COALESCE* und *TYPE*. Mit *NULLIF* sind Werte aus einer Abfrage ausblendbar, wenn sie eine bestimmte Größe besitzen:

```
SELECT MIN(NULLIF(lp.gewicht, -1))
FROM Lieferung l, IN
(l.lieferpositionen) lp W
HERE l.id = 101
```

Diese Abfrage ermittelt die leichteste Lieferposition der Lieferung *101* unter der Annahme, dass ungültige Gewichtsangaben in der Datenbank mit *-1* codiert sein könnten. Mit *CASE WHEN* formulieren Sie eine Bedingung samt Ergebnis:

```
SELECT k, CASE WHEN
SIZE(k.bestellungen) > 10 THEN
'Premiumkunde' ELSE 'Normalkunde' END
FROM Kunde k
```

Besitzt ein Kunde mehr als zehn Bestellungen, liefert die Abfrage das Attribut *Premiumkunde* zurück, anderenfalls *Normalkunde*. Dank dieser Abfrage ist es nicht erforderlich, alle Kunden samt ihrer Bestellung zu laden, nur um dann die Anzahl der Bestellungen pro Kunde nachzuzählen. Das Schlüsselwort *COALESCE* entspricht der PL/SQL-Funktion *NVL*. Es liefert den ersten Wert aus seiner Parameterliste zurück, der nicht *NULL* ist. Wollen Sie alle Bestellungen mit einem Liefertermin in der Zukunft erhalten und dabei auch jene Bestellungen berücksichtigen, deren Liefertermin noch nicht gesetzt ist, lässt sich dies kurz und schmerzlos abfragen:

```
SELECT b FROM Bestellung b WHERE
COALESCE(b.liefertermin, CURRENT_DATE)
>= CURRENT_DATE
```

Mit der Funktion *TYPE* ermitteln Sie den Typ einer Entität. Haben Sie eine Typ-Hierarchie in Java modelliert, die zwischen *Lebensmittel*- und *NichtLebensmittel*-Produkten unterscheidet, so filtern Sie Bestellungen, die Lebensmittel enthalten, mit der folgenden Abfrage heraus:

```
SELECT b FROM Bestellung b, IN
(b.bestellpositionen) pos WHERE
TYPE(pos.produkt) = Lebensmittel
```

Dem erweiterten *IN*-Operator übergeben Sie als Parameter auch eine Java-Collection. Auch Datumsliterale sind nun zulässig, ebenso Berechnungen im *SELECT*-Teil der Abfrage.

Criteria-API

Neben den Spracherweiterungen innerhalb der JPQL wird mit JPA 2.0 erstmals eine programmiertechnische Abfragemöglichkeit in Form ei-

JSR 317 – Java Persistence API 2.0

Das Java Persistence API 2.0 wird innerhalb des Java Community Process als JSR 317 spezifiziert. Der Java Community Process (JCP) ist eine von Sun Microsystems gegründete Organisation, die sich der zweckmäßigen Weiterentwicklung der Programmiersprache Java verschrieben hat. Innerhalb des JCP engagieren sich zahlreiche namhafte Unternehmen wie IBM, Oracle, Red Hat und SAP, aber auch Organisationen wie die Apache Software Foundation sowie Einzelpersonen. JPA 2.0 wurde 2007 als Java Specification Request (JSR) 317 in den JCP eingebracht und durch das JCP-Direktionskomitee angenommen. Es folgte die Bildung einer Expertengruppe unter Leitung von Linda DeMichiel von Sun. Die vorläufigen Ergebnisse der Expertengruppe wurden mehrfach als Drafts der Öffentlichkeit zur Diskussion vorgestellt. Der aktuelle Proposed Final Draft 2 (PFD2) erschien im September 2009 und stellt mit großer Wahrscheinlichkeit die fertige Spezifikation dar. Die Spezifikation wird jedoch erst freigegeben, wenn in einer formalen Abstimmung die Mitglieder des JCP-Direktionskomitees die Freigabe billigen (Final Approval Ballot). Neben dem Spezifikationstext müssen auch eine Referenzimplementierung (RI) und ein Kompatibilitätstest (Technology Compatibility Kit, TCK) durch die Expertengruppe erarbeitet werden. Wie schon mit TopLink Essentials wird auch diesmal Oracle mit EclipseLink 2.0 die RI liefern. Sun fällt traditionell die Erstellung des TCK zu. RI und TCK müssen noch vor der Freigabe der Spezifikation vorliegen. Implementierungen dürfen sich erst dann als „JPA-2.0-konform“ bezeichnen und damit werben, wenn die Spezifikation verabschiedet ist und das TCK formal bestanden wurde – eine sportliche Herausforderung bis Jahresende für alle Beteiligten.

nes Criteria-Query-API angeboten. Die Zweckmäßigkeit eines solchen API wird jeder Entwickler zu schätzen wissen, der für dynamische Abfragen schon einmal JPQL-Fragmente zusammenpuzzeln musste. Ein programmiertechnischer Ansatz mittels Criteria-API könnte für die obige Lebensmittel-Abfrage wie in Listing 1 dargestellt aussehen.

Die Abfrage sieht auf den ersten Blick überaus komplex aus. Dies ist der Preis dafür, jedes beliebige Detail dieser Abfrage per Programm bestimmen und ändern zu können. Der Nutzen der Abfrage wird deutlich, wenn Sie sich vor Augen führen, wie leicht weitere Selektionskriterien aufnehmbar sind. Wollen Sie beispielsweise Bestellungen abfragen, die darüber hinaus weniger als zehn Positionen besitzen, ließe sich das folgende Prädikat bilden:

```
Expression anzahlBestellposExpression=
 queryBuilder.size(bestellposJoin)
Predicate wenigerAls10PosPredicate
 = queryBuilder.lessThan(
   anzahlBestellposExpression, 10);
```

Dieses Prädikat kann über eine *UND*-Verknüpfung in die Abfrage einbezogen werden:

```
criteriaQuery
 .select(bestellungRoot)
 .where(queryBuilder.and(
  lebensmittelPredicate,
  wenigerAls10PosPredicate));
```

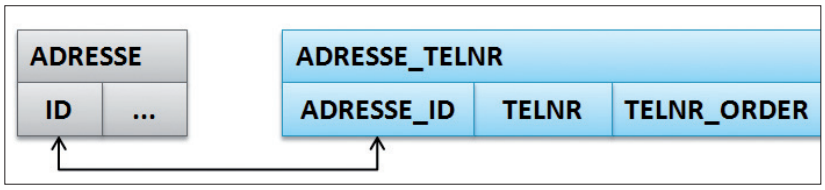
Bei der Vorstellung des Criteria-Query-API wurde von einigen prominenten Stimmen als Kritikpunkt vorgebracht, dass diese Abfragen immer noch nicht an die Typsicherheit von LINQ aus der .NET-Welt heranreichen. Die Spezifikation wurde nachgebessert und beinhaltet jetzt auch die Möglichkeit, typsichere Criteria-Abfragen zu erstellen. Das Ganze sieht jedoch ähnlich sperrig aus. Den Pfad von *Bestellung* über *Bestellposition* zu *Produkt* erhält man typsicher in der folgenden Form:

```
Path<Produkt> produktPath =
 criteriaQuery
  .from(Bestellung.class)
  .join(Bestellung_.bestellpositionen)
  .get(Bestellposition_.produkt);
```

Hierbei spielen die Modellklassen *Bestellung_* und *Bestellposition_* eine entscheidende Rolle. Diese Klassen werden beim Kompilieren durch einen Annotationen-Prozessor automatisch generiert. Sie haben folgenden Inhalt:

```
@Generated("JPA MetaModel for
 Bestellung")
@StaticMetamodel(Bestellung.class)
public abstract class Bestellung_ {
 public static volatile
  SingularAttribute<Bestellung, Long>
id;
 public static volatile
  ListAttribute<Bestellung,
   Bestellposition> bestellpositionen;
}
```

Datenbankschema für das Beispiel @ElementCollection (Bild 2)



Instanzen dieser Modellklassen sind auch über ein Metamodell-API ermittelbar. Folgendes Beispiel illustriert die Anwendung auf den vorliegenden Fall:

```
Metamodel metamodel =
 entityManagerFactory.getMetamodel();
ManagedType bestellungType =
 metamodel.type(Bestellung.class);
ListAttribute bestellposAttribute
 = bestellungType.getList(
   "bestellpositionen");
Class bestellposJavaType =
 bestellposAttribute.getBindableJava▷
Type();
ManagedType bestellposType =
 metamodel.type(bestellposJavaType);
SingularAttribute produktAttribute
 = bestellposType.▷
    getSingularAttribute("produkt");
```

Ein kleiner Wermutstropfen bleibt: Dem Metamodell fehlen leider jegliche datenbankbezogenen Mapping-Informationen. Es ist deshalb standardkonform nicht zu ermitteln, in welcher Datenbanktabelle sich eine Entität befindet oder von welchem Datenbanktyp ein bestimmtes Attribut ist.

Das API

Ein Design-Ziel von JPA war es, das API so schlicht wie möglich zu halten. Dies wurde auch bei JPA 2.0 beibehalten. Als Entity-Manager-Operation ist lediglich *Detach* hinzugekommen. Mit *Detach* lösen Sie gezielt Instanzen aus einem aktiven Persistence-Context heraus und beenden damit das aktive Management dieser Instanzen (zum Beispiel Änderungsverfolgung oder Nachladen). *DETACH* kann neben *PERSIST, MERGE, REMOVE* und *REFRESH* auch als Cascade-Option in den Beziehungsmappings angegeben werden. Eine weitere interessante ▶

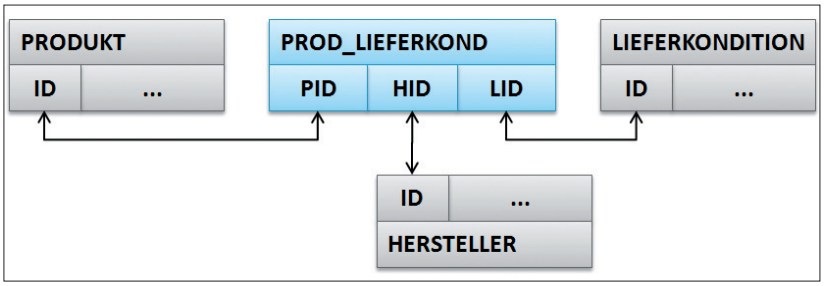
Möglichkeit stellen die *Unwrap*-Methoden am Query- und Entity-Manager-Interface dar. *Unwrap* erlaubt es, an das native Query- und Session-API der Mapper-Implementierung zu gelangen. Auch haben Helfer-Methoden über das Interface *PersistenceUnitUtil* Einzug in den Standard gehalten. Über den Aufruf

```
entitManagerFactory.▷
  getPersistenceUnitUtil().▷
  isLoaded(bestellung,
  "bestellpositionen")
```

lässt sich ermitteln, ob die Liste der Bestellpositionen an einer *Bestellung*-Instanz bereits geladen vorliegt. JPA 2.0 wartet erstmalig mit pessimistischen Sperren auf. Abfrageoperationen erlauben als zusätzlichen Parameter die Angabe eines pessimistischen Lock-Modes. Mithilfe von

```
entityManager.lock(bestellung,
 LockModeType.PESSIMISTIC_READ)
```

Datenbankschema für das Lieferkonditionenbeispiel (Bild 3)



kann für eine geladene Entität ein *SELECT FOR UPDATE* auch nachgeschossen werden. Alle JPA-Implementierungen bieten bereits jetzt neben dem obligatorischen Persistence-Context-Cache einen Shared Cache für geladene Datenbankentitäten an. Dieser Cache kann mit JPA 2.0 direkt angesprochen und konfiguriert werden. Ein programmiertechnisches Leeren sieht beispielsweise wie folgt aus:

```
entityManagerFactory.getCache().▷
evictAll()
```

Über die Annotation *@Cacheable* lässt sich für einen Entitätstyp festgelegen, ob Instanzen in den Shared Cache aufgenommen werden dürfen. Die Shared-Cache-Mode-Einstellung in der *persistence.xml* steuert, ob das Caching die Regel oder die Ausnahme darstellen soll.

Über Query-Hints legen Sie fest, ob für eine bestimmte Abfrage der Shared Cache umgangen werden soll. Im Rahmen der Java Enterprise Edition 6 wird ebenfalls ein Standard zur Validierung von Java-Beans erarbeitet [5]. Neben JavaServer Faces 2.0 integriert auch das zukünftige Persistence API diesen Standard. Die Einbindung der Bean-Validierung ist weitgehend konfigurationsfrei gehalten. Es reicht das Vorhandensein einer entsprechenden Bibliothek im Klassenpfad. Die Validierung ist im einfachsten Fall so nutzbar:

```
@Entity
public class Produkt {
 @Id
 private long id;
 @javax.validation.constraints.NotNull
 @javax.validation.constraints.Size(▷
  min=2, max=255)
 private String name;
}
```

Die Validierung wird, sofern nichts anderes konfiguriert ist, vor jedem Einfügen und jedem Update von Datenbankentitäten durchgeführt. Als letzter Punkt noch ein Hinweis auf die Konfiguration via *persistence.xml*: Die Angaben für Datenbank-URL, Treiber, Nutzer und Passwort lassen sich über die Properties *javax.persistence.jdbc.x* für jede Implementierung übereinstimmend festlegen.

Eine erste Beurteilung

Der Spezifikationsgruppe ist mit JPA 2.0 ein großer Wurf gelungen. Die Gemeinsamkeiten der verfügbaren Mapper-Implementierungen sind damit weitestgehend standardisiert. Anders als bei der ersten JPA-Version muss man sich anstrengen, um weitere Features zu finden, bei denen eine ausreichend große Übereinstimmung für eine weitere Standardisierung bestünde.

Die Entwickler von Apache OpenJPA, EclipseLink und Hibernate arbeiten momentan mit Hochdruck an der Anbindung des neuen API (vergleiche [6] bis [8]). Milestones-Builds der zukünftigen Produktversionen sind für erste Tests bereits herunterladbar.

Aufgrund der Fülle von neuen Features besitzen diese Versionen noch einen frühen Alpha-Status. Anfang 2010 ist sicherlich mit den ersten produktiv einsetzbaren JPA-2.0-konformen Versionen zu rechnen. **[am]**

[1] Objektorientiertes SQL – Die Java Persistence Query Language; database pro 1/2009, S. 36 ff.
[2] Java Community Process (JCP); http://jcp.org/
[3] Blog des Specification-Leads; http://blogs.sun.com/ldemichiel
[4] Java Persistence 2.0, JSR 317; http://jcp.org/en/jsr/summary?id=317
[5] Bean Validation, JSR 303; http://jcp.org/en/jsr/summary?id=303
[6] Apache OpenJPA 2.0; http://openjpa.apache.org/jpa-20-roadmap.html
[7] EclipseLink 2.0; http://wiki.eclipse.org/EclipseLink/Development/JPA_2.0
[8] Hibernate 3.5; http://opensource.atlassian.com/projects/hibernate/browse/HHH-4190