9 Persistenzabbildung

»We want a completely dynamic system, where absolutely no single component is required to stay in memory at all times, where speed of the running applications is of prime importance, and where the programmers have total control over the machine.«

The Unununium project team
http://unununium.org

9.1 Kurz gefasst

Der EJB 3.0 Expert Group ist 2006 das Kunststück gelungen, die Persistenzabbildung in ihren Möglichkeiten zu erweitern, gleichzeitig aber den Umgang mit ihr zu vereinfachen. Da eine solch elegante Persistenzabbildung viel zu schade ist, um sie nur im Rahmen von EJB-Projekten zu nutzen, wurde sie so offen ausgelegt, dass sie nicht nur im gesamten Java-EE-Umfeld zum Einsatz kommen kann, sondern ebenso für Java-SE-Applikationen verwendbar ist. Aus diesem Grund wurde die Persistenzabbildung in eine eigene Spezifikation namens *Java Persistence API* (JPA) ausgelagert. Die EJB-Spezifikation nimmt mit der aktuellen Version 3.1 nur noch Bezug auf die nunmehr eigenständige JPA-Spezifikation. Auf den folgenden Seiten geben wir einen möglichst vollständigen und tiefgehenden Überblick über die JPA im Kontext von EJB, sodass Sie direkt mit der Nutzung von Persistent Entities loslegen können.

Nach der Beschreibung der grundlegenden Annotationen zur Definition der Persistenzeigenschaften von *Persistent Entities* (ehemals *Entity Beans*) geht es ans Eingemachte: Mapping auf mehrere Datenbanktabellen, Vererbungsstrategien und Polymorphie, Abbildung von Objektbeziehungen, Fetching-Strategien und Transaktionssteuerung: Nichts ist unmöglich.

9.2 Der Blick zurück

Den Blick zurück haben wir in diesem Fall in ein eigenes Kapitel ausgelagert – und das nur, um denjenigen, die bereits mit den EJB-Vorgängerversionen gearbeitet haben, den bekannten Begriff *Entity Beans* noch einmal präsentieren zu können. In Kapitel 8 haben wir die Entity Beans, um die sich die Welt der Persistenzabbildung bis zur EJB-Version 2.1 drehte, noch einmal kurz Revue passieren lassen. Im folgenden Abschnitt, der Ihnen einen ersten Einblick in die Welt der Persistenzabbildung gibt, wird auch auf die entsprechenden Konzepte in EJB 2.x eingegangen.

9.3 Persistenz? Abbildung?

Nun haben wir den Begriff »Persistenzabbildung« schon mehrfach verwendet, ohne ihn zuvor definiert zu haben. Das holen wir jetzt schleunigst nach!

Dauerhafte Speicherung der (fachlichen) Daten

Neben den prozessbeschreibenden EJB-Typen (Session Bean bzw. Message-Driven Bean) gibt es in EJB 3.1 (und auch schon in den Vorgängerversionen) bzw. JPA einen Objekttyp, der die fachlichen Objekte anwendungsfallbezogen als Datenstruktur beschreibt. Zur Laufzeit einer verteilten objektorientierten Applikation werden im Rahmen der implementierten Geschäftsprozesse solche fachlichen Objekte erzeugt, sondiert, modifiziert und auch wieder gelöscht. Natürlich sollen einmal erzeugte fachliche Objekte auch nach der Beendigung der Applikation (bzw. dem Herunterfahren des Applikationsservers) beim nächsten Neustart wieder zur Verfügung stehen. Deshalb lautet eine Grundanforderung an derartige Systeme, dass fachliche Objekte, aber auch technische Informationen wie beispielsweise Konfigurationsdaten, persistent gemacht (d.h. dauerhaft gespeichert) werden können. Auch heute noch werden für diese Aufgabe hauptsächlich relationale Datenbankmanagementsysteme (RDMBS) eingesetzt. In solchen Systemen werden die Daten in (relationalen) Tabellen gespeichert (siehe z.B. [Kemper06] für eine Beschreibung des *Entity-Relationship-Modells*, das den RDBMS zugrunde liegt).

Datenbanknahe Programmierung

In klassischen datenbankbasierten Applikationen wurde das Datenmodell eng an das Datenbankmodell angelehnt. Das erleichterte die Abbildung der in der jeweiligen Programmiersprache implementierten komplexen Datentypen in die relationale Welt der Datenbank. Den Programmcode für die Speicherung hat der Applikationsentwickler selbst geschrieben – zunächst spezifisch für das verwendete Datenbanksystem (gegebenenfalls unter Nutzung der herstellerspezifischen

Eigenheiten), später über standardisierte Schnittstellen wie zum Beispiel [JDBC]. Es gab sogar Programmiersysteme, die direkt mit einem bestimmten RDMBS gekoppelt waren. Ein prominentes Beispiel ist [OracleForms].

Der »datenbank-verschmutzte« Code

Der Hauptkritikpunkt, der gegen diese Art der Programmierung erhoben wurde, war die Orientierung der fachlichen Modellierung an den (datenbank)technischen Möglichkeiten. Im Extremfall wirkte sich diese Abhängigkeit sogar auf die Benutzungsoberfläche der Applikation aus. Damit mutete man dem Benutzer ein »Kompott« aus fachlichen und technischen Modellelementen zu, das mit den Gegenständen seiner (Arbeits-)Welt nicht mehr allzu viel zu tun hatte – SAP lässt grüßen ...

Objektorientierte Applikationen

Das objektorientierte Programmierparadigma prangerte diese Nähe zur Datenbank offen an: Warum soll sich eine fachliche Applikation, die mit fachlichen Objekten arbeitete, dem Diktat der Datenbank unterwerfen? Das Design der fachlichen Objekte hat sich einzig und allein an den fachlichen Gegebenheiten zu orientieren. Die Art und Weise, wie bzw. wo (d.h. in welchen Tabellen) ein solches Objekt dauerhaft in einer Datenbank gespeichert wird, soll den Entwickler der Applikation nicht interessieren. Der Benutzer darf von alledem nichts mitbekommen: Er arbeitet mit fachlichen Objekten, die im Aufbau und Verhalten den realen Konzepten und Objekten seines Arbeitsumfelds nachempfunden sind.

Objektrelationale Abbildung

Aber auch diese »reinen fachlichen Objekte« mussten irgendwie gespeichert werden. Da man bewusst auf eine strukturelle Nähe zum Datenbankmodell verzichtet hat, ist diese sogenannte *objektrelationale Abbildung* (*Object-Relational Mapping*, *O/R-Mapping*) in den meisten Fällen komplexer als bei der datenbanknahen Programmierung. Deshalb ersann man schon recht früh Mechanismen, um diesen Abbildungsprozess zu automatisieren.

Persistenzabbildung in EJB 2.x

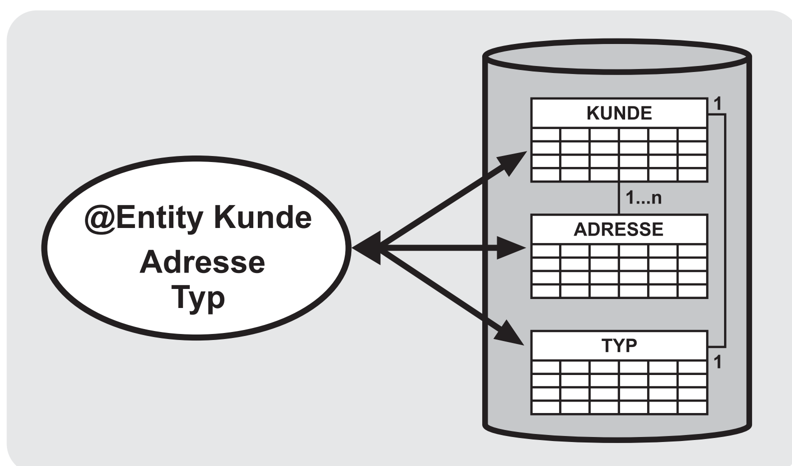
Enterprise JavaBeans 2.x kannte zwei verschiedene Modelle, um die Persistenz der fachlichen Objekte (Entity Beans) sicherzustellen: Bei der *Container-Managed Persistence* (*CMP*) wurde das relationale Abbild der Objekte in Deployment-Deskriptoren beschrieben. Hier mussten alle für die Persistenzabbildung wesentlichen Informationen hinterlegt werden:

- In welcher Datenbankinstanz sollen die Objekte gespeichert werden?
- In welcher Datenbanktabelle wird ein Objekt eines bestimmten Typs gespeichert?
- In welcher Tabellenspalte wird ein bestimmtes persistentes Feld gespeichert?
- Welche Felder umfasst der Datenbankschlüssel des Objekts?

- Wie werden Beziehungen zu anderen Objekten in der Datenbank gespeichert (beispielsweise als Fremdschlüsselspalte oder separate Tabelle)?

Das CMP-Modell hatte einige Beschränkungen. So ließen sich Objekte nur auf genau eine Datenbanktabelle abbilden. Ein Multi-Table-Mapping, wie es insbesondere bei der Abbildung von Vererbungsbeziehungen und Polymorphie zum Einsatz kommt, ließ sich mit den Ausdrucksmöglichkeiten von CMP nicht beschreiben. Auch die für den Zugriff auf die in der Datenbank gespeicherten Entity Beans verwendete Abfragesprache namens EJB QL eignete sich nur für einfache Abfragen. Da viele EJB-Systeme auf einem existierenden Datenmodell aufsetzten, musste es aber möglich sein, solche Spezialfälle abzubilden und abzufragen.

Abb. 9–1
Abbildung einer Entität auf mehrere Tabellen



Mithilfe der *Bean-Managed Persistence* (*BMP*) war dies tatsächlich möglich – allerdings zu einem gewissen Preis: Der Entwickler muss bei der BMP die gesamte Kommunikation mit der Datenbank für alle Operationen (`SELECT`, `INSERT`, `UPDATE` und `DELETE`) selbst programmieren. Das lief dem erklärten Ziel zuwider, die Persistenzabbildung so weit wie möglich zu automatisieren. Außerdem existierten am Markt bzw. im Open-Source-Umfeld bereits Frameworks zur Persistenzabbildung, die den gewünschten Automatisierungsgrad boten. Als prominentes Beispiel sei hier [*Hibernate*] genannt, dessen Konzepte die Java Persistence API maßgeblich beeinflussten.

9.4 Persistent Entities

Da die Java Persistence API sowohl für den Einsatz im Java-EE-Kontext als auch für Java-SE-Applikationen entworfen worden ist, war der Name *Entity Bean* (siehe Kapitel 8) nicht mehr passend, da man im »klassischen« Java-SE-Umfeld nicht unbedingt mit Beans zu tun hat. Deshalb hat man sich für einen neuen Namen entschieden: Aus Entity Beans werden *Persistent Entities*.

Die JPA-Spezifikation definiert die Persistent Entities wie folgt:

Definition Persistent Entity

> An entity is a lightweight persistent domain object.

Der »richtige« Name

Warum steht in der Definition »entity« und nicht »Persistent Entity«? Nun, die Benennung dieser Elemente ist in der Spezifikation nicht eindeutig geregelt. »Entity« allein, wie es auch in der JPA-Spezifikation vornehmlich verwendet wird, ist aus unserer Sicht zu allgemein – es fehlt der Bezug zur Persistenz. Deshalb haben wir uns für den Begriff »Persistent Entity« entschieden, der auch in der JPA-Spezifikation sporadisch verwendet wird. Allerdings müssen Sie damit rechnen, in der Literatur nach wie vor auch den Begriff »Entity Bean« anstelle von »(EJB 3.x|Persistent)? Entity«[1] zu finden.

Anhand obiger Definition wollen wir nun die wesentlichen Neuerungen der Persistent Entities im Vergleich zu den Entity Beans kurz skizzieren.

9.4.1 Lightweight

Persistente POJOs

Persistent Entities sind einfache Java-Objekte (*Plain Old Java Objects*, *POJOs*). Ihre Persistenzeigenschaften werden in Form von Annotationen festgelegt, anstatt sie – wie bei den Entity Beans – in komplexen Deployment-Deskriptoren beschreiben zu müssen (was allerdings auch mit JPA 2.0 durchaus noch erlaubt ist) beziehungsweise händisch zu implementieren (Bean-Managed Persistence).

Die Zugriffe auf die Felder einer Persistent Entity sind klar geregelt: Entweder verwendet man `get()`- und `set()`-Methoden entsprechend dem Bean-Konzept von Java, oder die Persistent Entity stellt spezielle Geschäftsmethoden zur Verfügung.

---

1. Diese merkwürdige Zeichenfolge sollten Perl-Gurus und »Unixoide« schnell als regulären Ausdruck identifiziert haben. Allen anderen hilft `http://de.wikipedia.org/wiki/Regulärer_Ausdruck` beim Decodieren.

9.4.2 Persistent

Entity-Manager

Wie aber kann man ein POJO persistent machen? Aus sich heraus ist das Persistent-Entity-Objekt nicht dazu in der Lage. Hier kommt der Entity-Manager ins Spiel. Diese Komponente, die sowohl in Java-EE- als auch in Java-SE-Umgebungen verwendet werden kann, stellt die Verbindung zu einer Datenbank her und erledigt das Speichern, das Aktualisieren, das Löschen sowie das Sondieren der Persistent Entities (siehe Abschnitte 9.6.2 und 9.8).

Persistenzkontext

Der Entity-Manager verwaltet auch den sogenannten Persistenzkontext (vgl. Abschnitt 9.6.4), innerhalb dessen sich eine Persistent Entity bewegen kann. Solange die Persistent Entity einem Persistenzkontext angehört, wird sie vom Entity-Manager verwaltet. Man spricht dann von einer *Managed Entity* oder *Attached Entity*. Wenn die Verbindung zum Entity-Manager gelöst wird, ist die Persistent Entity frei (*detached*, *unmanaged*).

Vererbung und Polymorphie

Neu ist auch, dass die Persistent Entities Vererbung und Polymorphie beherrschen. Musste der EJB-2.1-Entwickler bei Vererbungshierarchien seiner persistenten Objekte die Persistenzabbildung von Hand implementieren (Bean-Managed Persistence), so bietet JPA passende Annotationen, um die Objektbeziehungen deklarativ zu hinterlegen.

9.4.3 Domain Object

Fachliche Objekte statt technischer Zwänge

Persistent Entities sind echte Domänenobjekte, d.h. Abbildungen fachlicher Objekte oder Sachverhalte. Als POJOs konzentrieren sie sich auf die Modellierung der fachlichen Schnittstellen und der Geschäftslogik, anstatt sich in technischen Details zu verlieren, die von der Ablaufumgebung zwingend vorgeschrieben werden. Letzteres war bei den EJB-Versionen bis einschließlich Version 2.1 der Fall.

POJOs vereinfachen den Softwaretest.

Die POJO-Eigenschaft der Persistent Entities hat einige signifikante Vorteile. POJOs lassen sich sowohl innerhalb von Ablaufumgebungen wie EJB 3.x als auch in Java-SE-Applikationen nutzen. Die JPA sieht dies explizit vor. Der Vorteil dieser freien Wahl der Ablaufumgebung lässt sich am besten anhand des Softwaretests illustrieren (siehe auch Kapitel 17). Bei EJB 2.1 war man auf den EJB-Container als Umgebung für Enterprise Beans und somit auch für deren Tests angewiesen. Wollte man Tests unabhängig vom Container ausführen, so war für die Enterprise Beans eine entsprechende Umgebung nachzubilden – kein leichtes Unterfangen, angesichts der Komplexität des Enterprise-JavaBeans-Modells. Dank EJB 3.x und JPA ist es deutlich einfacher, eine Testumgebung für Enterprise Beans und Persistent Entities aufzubauen, z.B. mithilfe von [*Mockito*].

Detached Objects

Wenn ein Persistent-Entity-Objekt aus dem Persistenzkontext herausgelöst (detached) wird, kann es wie ein Value Object verwendet werden. Die Implementierung spezialisierter Value Objects gemäß dem Entwurfsmuster kann somit entfallen. Soll das Persistent-Entity-Objekt zu Clients übertragen werden, die außerhalb der Persistenzumgebung »leben«, so muss die Persistent-Entity-Klasse das Flag-Interface `java.io.Serializable` implementieren, da das Objekt in serialisierter Form »über die Leitung geht«.

Aber das ist noch nicht alles: Wird ein solches *Detached Object* wieder in die Obhut eines Entity-Managers gegeben, kann dieser einen Abgleich mit der Datenbank durchführen. Eventuelle Modifikationen im Detached Object werden dann in den persistenten Zustand des Objekts übernommen.

9.4.4 Lebenszyklus

Der Lebenszyklus der Persistent Entities wird in aller Ausführlichkeit in Abschnitt 12.10 beschrieben. Dort werden auch die entsprechenden Callbacks vorgestellt, mit denen der EJB-Entwickler in den Lebenszyklus eingreifen kann.

9.5 Persist my POJO!

Persistent Entities sind also einfache Java-Objekte (POJOs). Aber woher weiß eine Java-SE-Persistenzkomponente (bzw. das entsprechende Modul eines EJB-Containers), wie ein solches Objekt persistent gemacht wird?

9.5.1 Annotation oder Deployment-Deskriptor?

Die Antwort dürfte Sie kaum überraschen: Die Persistenzinformationen werden entweder per Annotationen festgeschrieben oder im Deployment-Deskriptor definiert.

Sind Annotationen »unrein«?

Diese Wahlfreiheit ist zunächst einmal positiv zu bewerten – ruft aber andererseits die Puristen auf den Plan. Nach deren Meinung ist die Beschreibung der Persistenzeigenschaften einer fachlichen Entität mithilfe von Annotationen in der Persistent-Entity-Klasse unangebracht, da die fachlichen Aspekte mit technischen Details über die Art und Weise der dauerhaften Speicherung vermischt werden. Es lässt sich trefflich darüber streiten, ob eine annotierte Persistent Entity noch das Attribut »Plain« verdient. Auf der anderen Seite ist die Verwendung von Annotationen der schnellste Weg, um die Persistenzinforma-

tionen zu notieren. Die Frage, ob eine Klasse mit Annotationen übersichtlich und lesbar ist, lässt sich nicht pauschal beantworten. Die Antwort hängt von der Komplexität der Persistent-Entity-Klasse und insbesondere von der Art und Anzahl der Beziehungen zu anderen Entitäten ab. Man kann aber festhalten, dass für den Softwareentwickler die Annotationsvariante deutliche Vorteile bringt, da er neben den fachlichen Aspekten einer Persistent Entity auch deren Datenmodell in der Persistenzschicht auf einen Blick sieht. Das ist insbesondere dann von Vorteil, wenn man sich in fremden Code einarbeiten muss oder auf Fehlersuche geht. Oft ist es einfacher, direkt in der Datenbank nachzuschauen, anstatt sich durch Objektbäume zu hangeln. Das Wissen um die Persistenzabbildung erschließt dem Entwickler oder Tester eben diese Welt der Persistenzschicht.

Natürlich lassen sich auch Argumente für die Verwendung von Deployment-Deskriptoren finden – beispielsweise die Unabhängigkeit des Entitäten-Modells von der Persistenzschicht, die erst zum Zeitpunkt des Deployments auf das Datenmodell abgebildet wird. Wie so oft, hängt die Entscheidung für Annotationen oder Deployment-Deskriptor vom konkreten Einzelfall ab. Die oben genannten Argumente sollen Ihnen bei der Bewertung Ihres Falls helfen.

9.5.2 Beispiel

Zurück zur eigentlichen Frage – und direkt zu einem einfachen Beispiel. Listing 9–1 zeigt einen Ausschnitt aus der Persistent-Entity-Klasse `News`, die in der Ticket2Rock-Beispielapplikation Nachrichten (z.B. Hinweise auf neue Konzerte im Ticketshop) modelliert.

Listing 9–1
Persistent Entity News

```
@Entity
public class News implements java.io.Serializable {
  private int id;
  private Date datum;
  private String nachricht;

  @Id
  @GeneratedValue
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Date getDatum() {
    return datum;
  }
```

```
    public void setDatum(Date datum) {
      this.datum = datum;
    }

    public String getNachricht() {
      return nachricht;
    }

    public void setNachricht(String nachricht) {
      this.nachricht = nachricht;
    }
  }
```

Drei Annotationen und die Implementierung eines Flag-Interface (`java.io.Serializable`) genügen, um aus dem POJO eine Persistent Entity zu machen.

@Entity

`@javax.persistence.Entity` proklamiert diese Klasse zur Persistent Entity. Werden wie in diesem Beispiel keine weiteren Angaben zu Tabellen und Tabellenspalten gemacht, so trifft die Spezifikation die folgenden Grundannahmen für die Persistenzabbildung:

- Der Tabellenname ist identisch mit dem Klassennamen (hier: `NEWS`). Die Spezifikation legt nicht fest, ob die Tabelle gegebenenfalls automatisch angelegt wird oder ob das Fehlen der Tabelle zu einer Exception führt.
- Für jedes persistente Feld wird in dieser Tabelle eine Tabellenspalte mit dem Feldnamen erwartet bzw. angelegt (hier: `ID`, `DATUM` und `NACHRICHT`).

Persistent sind all jene Felder, die nicht mit der Annotation `@javax.persistence.Transient` ausgezeichnet sind. Jedes persistente Feld muss von einem der unterstützten Datentypen sein (siehe Abschnitt 9.8). Dazu zählen beispielsweise die primitiven Typen sowie `java.lang.String` und `java.util.Date`.

@Table, @Column

Natürlich können diese Voreinstellungen auch überschrieben werden. So legt die Angabe von `@javax.persistence.Table(name="TICKET_NEWS")` an der Klassendefinition (üblicherweise nach `@Entity`) eine Abbildung der Persistent Entity `News` auf die Tabelle `TICKET_NEWS` fest. Analog kann am Feld bzw. dessen `get()`-Methode mit der Annotation `@javax.persistence.Column` festgelegt werden, wie die zugehörige Tabellenspalte heißen soll, z.B. `@Column(name="NEWSTEXT")`.

@Id

In Datenbanken muss bekanntlich jeder Datensatz eine unverwechselbare Identifikation haben – auch bekannt unter dem Namen *Primärschlüssel* (*primary key*). Dieser Schlüssel wird über die Annotation `@javax.persistence.Id` festgelegt. Dabei muss es sich nicht – wie im Beispiel – um ein einzelnes Feld handeln. Möglich sind auch zusammengesetzte Schlüssel, die mehrere Felder umfassen oder sogar durch ein Objekt repräsentiert werden. In unserem Fall beschränken wir uns auf die Angabe eines künstlichen Schlüssels. Künstlich deshalb, weil wir aus fachlicher Sicht auf ein Feld namens `id` durchaus verzichten könnten. Auch hier mogelt sich die Persistenzschicht ein wenig in das fachliche Modell hinein. Ein Umstand, der sich nicht einmal durch eine Deklaration im Deployment-Deskriptor beseitigen ließe, denn auch in diesem Fall müsste das fachliche Objekt (mindestens) ein identifizierendes Feld besitzen. Allerdings ist es möglich, den Primärschlüssel aus fachlichen Feldern zusammenzusetzen. Abschnitt 9.11 beschäftigt sich ausführlich mit Primärschlüsseln.

@GeneratedValue

Da unser Primärschlüssel künstlich ist, wollen wir uns nicht um dessen Generierung kümmern. Auch die Einhaltung der Forderung nach Eindeutigkeit dieses Schlüssels soll nicht unsere Aufgabe sein. Glücklicherweise nimmt uns die Persistence API diese Last auf Wunsch ab. Ein einfaches `@javax.persistence.GeneratedValue` in Verbindung mit `@Id` genügt, und schon wird von der Datenbank ein eindeutiger Primärschlüssel generiert. Dabei wird die Art und Weise, wie der Primärschlüssel generiert wird, der Datenbank überlassen. Wir werden uns in Abschnitt 9.11.3 genauer mit diesem Thema befassen.

Das soll alles sein?

Diese wenigen Annotationen reichen tatsächlich aus, um ein POJO zur Persistent Entity zu machen! Für den EJB-2.x-Umsteiger ist dies ein Erlebnis der besonderen Art: keine aufwendigen Angaben im Deployment-Deskriptor und keine Schnittstellenzwänge.

Natürlich hat die Java Persistence API noch wesentlich mehr zu bieten. Die `News`-Klasse ist ein sehr einfaches Beispiel für eine Persistent Entity, die nicht einmal Beziehungen zu anderen Entitäten hat. Deshalb werden in den folgenden Abschnitten die grundlegenden Konzepte und Deklarationsmöglichkeiten des neuen Persistenzmodells systematisch dargestellt.