open source library

Bernd Müller
Harald Wehr

Java-Persistence-API mit Hibernate

Standardisierte Persistenz

ADDISON-WESLEY

3 Objektlebenszyklus

Die vorangegangenen Kapitel gaben bereits kurze Einblicke in das Laden und Speichern von Objekten. Zentraler Anlaufpunkt ist, wie aus den bisherigen Beispielen ersichtlich, eine Instanz der Klasse `EntityManager`. Diese Instanz verwaltet eine ihr zugewiesene Menge von Java-Objekten und sorgt dafür, dass diese entsprechend ihrer definierten Primärschlüssel einmalig sind. Diese Rolle übernimmt in der Hibernate-Welt die Klasse `Session`. Instanzen dieser Klasse besitzen eine Reihe weiterer Methoden, die wir an geeigneter Stelle ebenfalls darstellen werden. Will man jedoch entsprechend des JPA-Standards implementieren, sollte der Persistenzkontext immer über die Klasse des Entity-Managers angesprochen werden, wie bereits oben beschrieben.

Der Persistenzkontext stellt unsere Menge von Objekten dar, die entsprechend der konfigurierten Identität des zugrunde liegenden Entity genau einmal mit ihrem Schlüssel innerhalb dieser Menge vorkommen. Der Entity-Manager verwaltet diese Menge von Objekten, nimmt neue Objekte auf, überwacht Änderungen von Objekten, die dieser Menge bereits angehören, entfernt Objekte aus dieser Menge oder verbindet ehemals verwaltete Objekte mit einem neuen Kontext. Er stellt im Prinzip die Schnittstelle zu dieser Menge von verwalteten Objekten dar.

Bei der Erzeugung und Verwendung des Entity-Managers muss unterschieden werden, ob es sich um eine Java-SE- oder Java-EE-Anwendung handelt. Auf Erstere gehen wir im nächsten Abschnitt ein. Die Konfiguration im Rahmen eines Java-EE-Servers zeigen wir in Abschnitt 3.2.

3.1 JPA in Java-SE-Anwendungen

Eine Java-SE-Anwendung ist eine gewöhnliche Applikation mit einer Main-Methode oder aber eine Servlet-Anwendung, bei der der Servlet-Container keinerlei Funktionalität zur Persistenz von Objekten bereitstellt. Die JPA-Spezifikation spricht in diesem Fall von einem Entity-Manager, der unter Verwaltung der Applikation steht („Application managed“). Der Anwendungsentwickler muss selbst die Initialisierung des Entity-Managers, den Beginn und das Ende einer Transaktion sowie das Schließen des Entity-Managers übernehmen. Die Beispiele der vorangegangenen Kapitel setzten diese Umgebung immer voraus. Nachfolgender Programmausschnitt zeigt noch einmal schematisch den gesamten Vorgang:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("kundenTest");
EntityManager em = emf.createEntityManager();

// verwalte den Persistenzkontext ...

em.close();
emf.close();
```

Im Folgenden wollen wir zeigen, wie unsere erzeugten Java-Objekte diesem Persistenzkontext hinzugefügt oder aus diesem entfernt werden. Des Weiteren gehen wir darauf ein, welche Auswirkung das Ändern von Attributen bei Objekten hat, die bereits einem Persistenzkontext angehören.

3.1.1 Speichern von Objekten

Die JPA-Spezifikation klassifiziert Java-Objekte hinsichtlich ihres Zustands auf folgende Art und Weise:

- **Neue** Objekte wurden noch nicht in die Obhut des Entity-Managers übergeben und besitzen noch keinen Eintrag in der Datenbank.
- Dagegen sind **verwaltete** Objekte (managed) zum Zeitpunkt der Betrachtung unter der Kontrolle des Entity-Managers und in der Datenbank persistent.
- **Losgelöste** Objekte (detached) sind augenblicklich nicht an einen Entity-Manager gebunden, besitzen aber im Unterschied zu den neuen Objekten ein Äquivalent in der Datenbank. Sie wurden bereits mit einem Entity-Manager in der Datenbank gespeichert, wurden von diesem aber wieder getrennt.
- Sind Objekte dagegen mit einem Entity-Manager bzw. dessen Persistenzkontext verbunden, in der Datenbank gespeichert, aber zum Löschen vorgemerkt, werden sie im Rahmen der Spezifikation als **gelöschte** (removed) Objekte bezeichnet [URL-EJB].

Die Abbildung 3.1 zeigt die möglichen Zustandsübergänge, die die Methoden der Klasse `EntityManager` auslösen. Diese Übersicht ist noch nicht ganz vollständig, da auch Methoden anderer Klassen Zustandsübergänge ermöglichen können. Dies passiert z.B. dann, wenn ein Persistenzkontext an die Reichweite einer Transaktion und nicht an den Entity-Manager gebunden ist. Der Einfachheit halber wollen wir diese Punkte aber zunächst unberücksichtigt lassen und uns im Rahmen der Einführung auf die Zustandsübergänge in einer Java-SE-Umgebung konzentrieren.

Speichern transienter Objekte

Um transiente Objekte, also neu erzeugte, noch nicht persistente Objekte, in der Datenbank zu speichern, stellt die Klasse `EntityManager` die Methode `persist()` bereit:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("kundenTest");

Kunde kunde = new Kunde();
kunde.setVorname("Heidi");
kunde.setNachname("Mustermann");
EntityManager em = emf.createEntityManager();
em.persist(kunde);
```

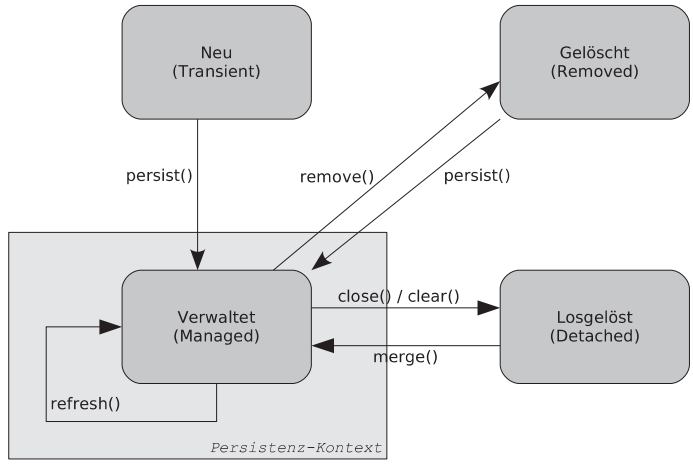

Abbildung 3.1: Objektlebenszyklus eines Entity

```
// ID erhalten ?
System.out.println("Id des Kunden: " + kunde.getId()); // null
```

Damit ist der neue Kunde aber noch nicht in der Datenbank gespeichert. Die Ausgabe auf der Konsole lautet:

```
Id des Kunden: null
```

Unser Objekt besitzt also noch keine eindeutige ID, obwohl dies in der Klasse entsprechend konfiguriert ist:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

Da wir uns im Moment nicht in einer Umgebung befinden, die von einem Java-EE-Container verwaltet wird, muss sich die Applikation selbst um den Beginn und das Ende der Transaktion kümmern. Die Spezifikation spricht in diesem Fall von einer *Resource-Local*-Transaktion.

Erst wenn die Transaktion des Entity-Managers abgeschlossen wird, findet man die Instanz in der Datenbank bzw. besitzt das Objekt eine eindeutige ID.

```
EntityTransaction tx = em.getTransaction();
tx.begin();
tx.commit();

// ID erhalten ?
System.out.println("Id des Kunden: " + kunde.getId()); // 991...

em.close();
```

Die Konsolenausgabe nach dem erfolgreichen Commit lautet nun:

```
Id des Kunden: 991
```

Hinweis

Grundlage dieses Kapitels ist das Beispielprojekt *objektlebenszyklus*. Nach dem Starten der HSQLDB sind bereits eine Reihe von Kunden in der Datenbank vorhanden. Ein neu eingefügter Kunde bekommt also nicht die Id 1.

Die Objektinitialisierung sowie das Verändern der Objekteigenschaften muss nicht erst nach dem Öffnen einer Transaktion bzw. nach der Initialisierung des EntityManagers erfolgen. Diese Arbeiten können ohne Berücksichtigung der JPA durchgeführt werden. Erst wenn das Objekt tatsächlich in der Datenbank abzulegen ist, muss die JPA aufgerufen werden.

Die Klasse `EntityManager` unterscheidet zwischen zwei verschiedenen sogenannten Flush-Modi. Sie werden mit der Methode `setFlushMode(FlushModeType f)` entsprechend konfiguriert. Voreingestellt ist der Modus `FlushModeType.AUTO`. Er gewährleistet bei einer Abfrage der Datenbank mittels der JPA, dass zunächst **alle** Änderungen der vorhandenen Java-Objekte, die die Abfrage beeinflussen könnten, in die Datenbank geschrieben werden. Erst dann wird die eigentliche Abfrage durchgeführt. Sollte keine Abfrage in der Transaktion erfolgen, wird die Synchronisation mit der Datenbank spätestens bei `commit()` der Transaktion durchgeführt.

`FlushModeType.COMMIT` lässt den `EntityManager` Änderungen erst bei Beendigung der Transaktion in die Datenbank bringen. Dieser Modus kann dazu führen, dass bei einer Abfrage Daten gelesen werden, die nicht aktuell sind, weil Änderungen noch im Persistenzkontext vorhanden sind, aber noch nicht mit der Datenbank synchronisiert wurden.

Des Weiteren ist es möglich, die Methode `flush()` selbst aufzurufen. In diesem Zuge wird der bisherige Zustand des Persistenzkontext mit den Inhalten der Datenbank synchronisiert.

Die Implementierung von Hibernate kennt weitere Flush-Modi, die in Tabelle 3.1 dargestellt sind.

Hinweis

Stehen Objekte über Assoziationen in Beziehung, muss die Methode `persist()` nicht auf jedem Objekt einzeln aufgerufen werden. Das Konzept der transitiven Persistenz (Persistence by Reachability) berücksichtigt bei Ausführung der Methode `persist()` auch die referenzierten Objekte. Näheres dazu findet der Leser in Abschnitt 4.1.2.

Name	Bedeutung
ALWAYS	Die Synchronisation wird vor jeder Abfrage durchgeführt.
AUTO	Die Synchronisation wird immer dann vor einer Abfrage durchgeführt, wenn die Abfrage von den noch nicht in der Datenbank befindlichen Änderungen beeinflusst sein könnte.
COMMIT	Die Synchronisation wird erst bei `commit()` der Transaktion durchgeführt.
MANUAL	Die Synchronisation wird nicht von Hibernate durchgeführt, sondern muss vom Entwickler selbst mittels `Session.flush()` angestoßen werden.

Tabelle 3.1: Hibernates FlushMode

Speichern persistenter Objekte

Zur Erinnerung an den vorangegangenen Abschnitt: Persistente Objekte haben bereits einen äquivalenten Datenbankeintrag und unterliegen der Kontrolle eines EntityManagers, sprich wurden mit einer Instanz dieser Klasse geladen oder durch diese bereits persistent gemacht. Erfolgen nun Änderungen an diesem *überwachten* Objekt, führt die JPA diese Änderungen automatisch in der Datenbank aus. Dies wird aus Effizienzgründen nicht sofort passieren, spätestens aber wenn die Methode `flush()` auf der `EntityManager`-Instanz ausgeführt wird bzw. die Transaktion mit `commit()` beendet wird. In der Literatur wird dieser Vorgang als *Automatic Dirty Checking* bezeichnet. Der Persistenzprovider überwacht alle Objekte des Kontext und stellt Änderungen automatisch fest.

Listing 3.1: Speichern eines bereits persistenten Objekts

```
Kunde kunde = new Kunde();
kunde.setVorname("Heidi");
kunde.setNachname("Mustermann");

EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
em.persist(kunde);

// Aenderungen am persistenten Objekt, da in derselben Session
kunde.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy")
  .parse("11.07.1960"));

// commit fuehrt die Aenderungen automatisch aus, wir brauchen
// nichts weiter tun
tx.commit();
```

An dieser Stelle möchten wir einen Blick auf das tatsächlich auf der Datenbank ausgeführte SQL-Kommando richten. Uns interessiert dabei nicht so sehr der Befehl zum Einfügen des Objekts, sondern vielmehr das Update-Statement. Da in unseren Beispielen die Datenbank immer im Debug-Modus gestartet wird, sehen wir hier, unab-

hängig davon, wie bei Hibernate das Logging eingestellt ist, die Ausgabe der Datenbank bequem auf der Kommandozeile. Sie lautet für die eigentliche Änderung:

```
update Kunde set vorname=?, nachname=?, geburtsdatum=? where id=?
```

Wie zu erkennen ist, werden alle Spalten des zugrunde liegenden Eintrags geändert, obwohl nur das Geburtsdatum tatsächlich geändert wurde. Möchte man dieses Verhalten ändern, muss die Klasse `Kunde.java` verändert werden:

```
@Entity
@org.hibernate.annotations.Entity(
      dynamicUpdate = true)
public class Kunde implements Serializable {
```

Mit dem Eintrag `dynamicUpdate` wird Hibernate angewiesen, nur die Spalten in der Datenbank zu ändern, die sich tatsächlich geändert haben. Das Update-Statement nach Aktivierung sieht dann wie folgt aus:

```
update Kunde set geburtsdatum=? where id=?
```

Weitere Beispiele bzw. Hibernate-spezifische Konfigurationsmöglichkeiten findet man in der Tabelle 2.5 auf Seite 53. Diese Konfigurationsmöglichkeiten verlassen aber den Standard der JPA. Dies erkennt man daran, dass der Annotation-Typ `Entity`, an dem die Konfiguration vorgenommen wird, nicht dem JPA-Package `javax.persistence` entstammt, sondern dem Hibernate-Package `org.hibernate.annotations`.

Ein weiterer Hinweis zum Debuggen der von Hibernate abgesetzten SQL-Statements: Da Hibernate intern sogenannte Prepared Statements benutzt, sind die eigentlichen Werte nicht direkt in das SQL-Statement eingebunden. Dies kann man daran erkennen, dass selbst die Ausgaben von Hibernate Fragezeichen statt der eigentlichen Werte enthalten. Möchte man wissen, welche Werte tatsächlich zur Datenbank geschickt werden, hilft die Debug-Option `log4j.logger.org.hibernate.type=debug` bei Verwendung von `log4j`. Damit erscheint dann folgende Ausgabe:

```
DEBUG SQL:393 - update Kunde set geburtsdatum=? where id=?
DEBUG TimestampType:133 - binding '1960-07-11 00:00:00' to parameter: 1
DEBUG IntegerType:133 - binding '3' to parameter: 2
```

Speichern von losgelösten Objekten

Warum gibt es losgelöste Objekte? Persistente Objekte bleiben doch unter Kontrolle des Entity-Managers und ihr Zustand bzw. die Änderung ihrer Eigenschaften werden automatisch erkannt und bei einem `commit()` der zugrunde liegenden Transaktion in die Datenbank überführt. Reicht es nicht aus, während der Laufzeit eines Programms oder einer Webanwendung zu Beginn den Persistenzkontext aufzubauen und ihn dann, wann immer benötigt, zu benutzen?

Die Antwort auf diese Frage lautet: Es kommt darauf an, welcher Art die Anwendung ist bzw. auf welche Art und Weise Informationen vom Benutzer erfragt und anschließend in die Datenbank gebracht werden sollen. Natürlich spielen dabei auch Überlegungen zum konkurrierenden Zugriff mehrerer Benutzer auf dieselben Datensätze

eine Rolle. Für alle Lösungen muss stets gelten, dass Verbindungen zur Datenbank nicht dauerhaft offen gehalten werden sollten. Transaktionen, die möglicherweise auch Sperren auf bestimmte Datensätze halten, müssen von möglichst kurzer Dauer sein, um andere Benutzer nicht bei ihrer Arbeit zu beeinträchtigen.

Stellen Sie sich eine web-basierte Home-Banking-Anwendung vor, bei der Sie eine Überweisung vornehmen möchten. Nachdem Sie ein leeres Überweisungsformular geladen haben, beginnen Sie mit Ihren Eingaben. Machen dies hunderte Benutzer gleichzeitig und wird während dieser Vorgänge jeweils eine Transaktion in der Datenbank offen gehalten, ist die Datenbank allein durch diese offenen Verbindungen extrem belastet. Um das zu vermeiden, kann bei länger anhaltenden Interaktionen die Verbindung zur Datenbank vorübergehend geschlossen werden. Verschiedene Lösungswege für dieses Problem stellen wir in Abschnitt 6.5 vor. Die Möglichkeit, Objekte vom Persistenzkontext zu lösen und erneut anzuhängen, ist dabei eine Variante. Da sie thematisch zum Objektlebenszyklus eines Entity gehört, erklären wir deren Verwendung in diesem Abschnitt.

Losgelöste Objekte entstehen durch die `close()`-Methode des Entity-Managers. Sie sind damit aber nicht „verloren", solange das Java-Programm entsprechende Referenzen auf die Objekte hält. Sehen wir uns das nachstehende Beispiel an. Die Methode `merge()` verbindet das losgelöste Objekt wieder mit einem Persistenzkontext:

Listing 3.2: Die Methode merge() im Einsatz

```
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("kundenTest");

Kunde kunde = new Kunde();
kunde.setVorname("Max");
kunde.setNachname("Mustermann");

EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(kunde); // (1) persistentes Objekt
tx.commit();
em.close(); // (2) mit close() werden alle verwalteten Objekte detached

// (3) Benutzer-Interaktion, Objekt ist losgeloest
kunde.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy")
        .parse("11.07.1960"));
em = emf.createEntityManager();
tx = em.getTransaction();
tx.begin();
Kunde kunde2 = em.merge(kunde); // (4) Achtung, Rueckgabewert!
kunde.setVorname("Jens"); // (5) Transient, keine Beruecksichtigung
kunde2.setVorname("Christian"); // (6) Persistent, wird gespeichert
tx.commit();

em.close();
emf.close();
```

(1) `persist()` macht das zuvor transiente Objekt `kunde` persistent. Alle Änderungen bis zu diesem Punkt werden bei `tx.commit()` in die Datenbank überführt.

(2) Der Entity-Manager wird geschlossen, um Ressourcen der Datenbank zu sparen. Alle Änderungen gelangen in die Datenbank.

(3) Dieser Punkt simuliert die Interaktion mit dem Benutzer. Zu einem Kunden wird das Geburtsdatum eingegeben. Im Gegensatz zum vorangegangenen Beispiel auf Seite 83 passiert dies nicht während der Laufzeit des Entity-Managers, sondern außerhalb.

(4) Die Methode `merge()` verbindet das veränderte Objekt wieder mit einem Persistenzkontext. Dabei wird zunächst überprüft, ob im angesprochenen Entity-Manager eine Instanz mit gleicher Identität vorhanden ist. Da dies nicht der Fall ist, wird unter der entsprechenden Id in der Datenbank nach dem bereits gespeicherten Objekt gesucht. Zu Demonstrationszwecken weisen wir den Rückgabewert einer neuen Variable zu. `merge()` liefert als Rückgabewert stets das persistente Objekt zurück.

(5) Wir verändern unser altes Kunden-Objekt. Die Änderung des Vornamens in „Jens“wird nicht berücksichtigt, da die Methode `merge()` das persistente (geänderte) Objekt zurückliefert!

(6) Da der Vorname „Christian“ auf dem persistenten Objekt geändert wird, gelangt diese Änderung auch in die Datenbank. Sämtliche Änderungen auf `kunde` bleiben unberücksichtigt. Um hier nicht durcheinanderzukommen, ist es besser, den Rückgabewert von `merge()` der Referenz auf das losgelöste Objekt zuzuweisen. Punkt 4 sollte stattdessen lauten:

```
kunde = em.merge(kunde);
```

Achtung

Bei Verwendung der `merge()`-Methode muss der Leser Vorsicht walten lassen: Der Rückgabewert der Methode ist ein anderes Objekt als das Objekt, das der Methode als Parameter übergeben wird. Dies haben wir bewusst im obigen Beispiel in Schritt 4 demonstriert. Um potenzielle Fehler zu vermeiden, kann das zu übergebene Objekt mit dem Rückgabewert der Methode gleich wieder überschrieben werden.

Die `clear()`-Methode bietet die Möglichkeit, alle im Persistenzkontext vorhandenen Objekte von diesem zu trennen, ohne jedoch wie mit der Methode `close()` den Kontext gleich zu schließen. Nach Aufruf von `clear()` befinden sich alle Objekte im Zustand *losgelöst*, Änderungen an ihren Attributen finden keine Berücksichtigung beim `commit` der Transaktion.

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
```

```
Kunde kunde1 = em.find(Kunde.class, 1);
kunde1.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy")
  .parse("11.07.1960"));

// mit flush() geht uns die Aenderung nicht verloren
//em.flush();

// EntityManager bleibt offen, alle darin enthaltenen Objekte
// werden aus dem Persistenzkontext entfernt
em.clear();

// commit aendert hier nichts
tx.commit();
em.close();
```

Führt man im obigen Beispiel vor der `clear()`-Methode noch `flush()` aus, wird die Änderung des Geburtsdatums noch mit in die Datenbank übernommen.

3.1.2 Einfaches Laden von Objekten

Im Rahmen dieses Kapitels werden wir nur in das Laden von Objekten mit einer gegebenen Id einführen. Eine ausführliche Einführung in die Abfragesprache der JPA, mit der auch mehrere Objekte zugleich geladen werden können, erfolgt in Kapitel 7.

Die JPA stellt zum Laden von Objekten die Methoden `find()` und `getReference()` bereit. Beide verlangen neben dem Typ des zu ladenden Entity als weiteres Argument einen Primärschlüssel. Bei beiden Methoden handelt es sich um generische Methoden:

```
public <T> T find(Class<T> entityClass, Object primaryKey);

public <T> T getReference(Class<T> entityClass, Object primaryKey);
```

Dies ist ein wesentlicher Unterschied zu den klassischen Hibernate-Methoden. Während Hibernate darauf ausgelegt ist, auch mit JDK-Versionen zu arbeiten, die vor der Java-Version 5.0 erschienen sind, setzt die JPA ausdrücklich diese Version voraus und bietet damit eine Reihe von Vorteilen, z.B. die oben erwähnten `Generics`. Der Typ des von den Methoden zurückgelieferten Objekts entspricht dem Typ des ersten Parameters, hier des Parameters `entityClass`. Ein Typecast muss im Gegensatz zu den `Session`-Methoden `get()` bzw. `load()` nicht ausgeführt werden.

Betrachten wir zunächst die Methode `find()`. Sie liefert das gefundene Objekt bzw. `null`, wenn es keinen entsprechenden Datenbankeintrag gibt. Fehler in Form einer `IllegalArgumentException` treten auf, falls der erste übergebene Parameter kein `Entity` im Sinne der JPA ist oder der Typ des übergebenen Primärschlüssels nicht mit dem im `Entity` konfigurierten Primärschlüsseltyp übereinstimmt.

```
em = emf.createEntityManager();
tx = em.getTransaction();
tx.begin();

// null oder ein Kunde-Objekt
Kunde k = em.find(Kunde.class, 1);
```

```
Kunde k2 = em.getReference(Kunde.class, 10000);

// Fehler, da nicht vorhanden bei Zugriff
System.out.println(k2.getNachname());
tx.commit();
em.close();
```

Im Unterschied dazu kann die Methode `getReference()` ein Objekt zurückgeben, dessen Attribute noch nicht mit den Werten aus der Datenbank gefüllt sind. Man spricht in diesem Fall von einem Proxy-Objekt. Die Werte werden erst dann mit einem zusätzlichen `Select` gelesen, wenn auf die entsprechenden Properties zugegriffen wird. Konnte ein entsprechendes Objekt nicht gefunden werden, tritt spätestens beim Zugriff auf eines der Attribute eine `EntityNotFoundException` auf. JPA erlaubt der jeweiligen Implementierung aber auch, die Exception bereits bei Ausführung von `getReference()` zu werfen.

Bei Verwendung von `getReference()` ist genau darauf zu achten, ob nach dem Beenden der Transaktion die Attribute des losgelösten Objekts komplett initialisiert worden sind. Dies ist erst dann gewährleistet, wenn innerhalb der Transaktion zumindest auf ein (nicht identifizierendes) Attribut zugegriffen wird.

Die Methoden `find()` und `getReference()` lassen sich auch dann verwenden, wenn das Entity einen zusammengesetzten Primärschlüssel verwendet. In diesem Fall werden den Methoden entsprechend initialisierte Instanzen einer Primärschlüssel-Klasse übergeben. Ein Beispiel dazu findet der Leser in Abschnitt 2.5.2.

Achtung

Bei Verwendung von Hibernate tritt im obigen Beispiel die Exception `EntityNotFoundException` erst dann auf, wenn tatsächlich auf ein Property des Proxy-Objekts zugegriffen wird und es keinen entsprechenden Datenbankeintrag gibt. Die Spezifikation weist jedoch ausdrücklich darauf hin, dass eine Implementierung die Exception bereits bei Aufruf der Methode `getReference()` werfen kann. Bei einem Wechsel der JPA-Implementierung kann es also zu Ungereimtheiten kommen, wenn dieser Umstand nicht bereits in der Entwicklung berücksichtigt wird.

Erneutes Laden mit `refresh()`

Die Methode `refresh()` bietet die Möglichkeit, ein verwaltetes Objekt erneut aus der Datenbank zu lesen. Dabei werden die auf dem Objekt (nicht in der Datenbank!) bis zu diesem Zeitpunkt bereits durchgeführten Änderungen mit den Inhalten der Datenbank überschrieben.

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
// Lade Kunden mit Id 1 und 2
Kunde kunde1 = em.find(Kunde.class, 1);
```

```
Kunde kunde2 = em.find(Kunde.class, 2);

// Aenderungen am persistenten Objekt, da in derselben Session
kunde1.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy").parse("11.07.1960"));

kunde2.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy").parse("11.09.1965"));

// kunde1 wird aus DB gelesen und damit zurueckgesetzt,
// die Aenderungen kommen nicht in die Datenbank
em.refresh(kunde1);

// commit aendert nur den zweiten Kunden
tx.commit();
```

3.1.3 Löschen von Objekten

Zum Löschen eines Objekts wird die Methode `remove()` des Entity-Managers verwendet. Die Methode besitzt nur einen Parameter, das zu löschende Objekt. Referenziert das Objekt dagegen weitere Objekte über Assoziationen und ist die Beziehung dieser Entities mit `cascade=REMOVE` bzw. `cascade=ALL` gekennzeichnet, werden diese referenzierten Objekte ebenfalls gelöscht. Näheres zu Objekten und ihren Assoziationen zu anderen Objekten erläutern wir in Kapitel 4.

Losgelöste Objekte verursachen beim Aufruf der Methode `remove()` die Exception `IllegalArgumentException`. Sollen diese Objekte gelöscht werden, müssen sie zunächst mit `merge()` erneut unter die Kontrolle des Persistenzkontext gebracht werden, danach ist der Aufruf von `remove()` zulässig.

3.2 JPA in Java-EE-Anwendungen

Wir wollen uns nun ansehen, wie JPA in einer Java-EE-Umgebung arbeitet. Dabei wollen wir unser Augenmerk vor allem auf die Konfiguration der notwendigen Klassen und Properties-Dateien legen. Uns interessieren hier weniger die Installation und der Betrieb eines kompletten Java-EE-Application-Servers. Sollte der Leser bereits Erfahrung auf diesem Gebiet haben, können unsere Beispiele auf konformen Servern nachvollzogen werden. Die Installation des JBoss-Application-Servers beschreiben wir in Abschnitt A.1.3.

Statt einen kompletten Application-Server zu installieren, verwenden wir hier die eigenständige EJB-3.0-Implementierung von JBoss, den Embeddable-EJB-Container. Dabei handelt es sich nicht um einen vollständigen Application-Server. Diese Implementierung stellt ohne Verwendung eines Containers die notwendigen EJB-Services bereit, die wir für unsere Einführung benötigen. Wir können sie ohne weiteres in einer Java-SE-Umgebung einsetzen.

Die notwendigen Bibliotheken stehen auf der JBoss-EJB3-Seite [URL-JBE] zum Download bereit.

3.2.1 Konfiguration des Eclipse-Projekts

Die Jar-Dateien von JBoss EJB 3.0 müssen sich alle im `lib`-Verzeichnis Ihres Testprojekts befinden. Alternativ dazu können Sie auch das auf der Homepage zum Buch befindliche Eclipse-Projekt *jboss-embedded-ejb30-libs* herunterladen. Es enthält alle benötigten Bibliotheken. Der Vorteil dabei ist, dass bei weiteren Projekten dieses Library-Projekt ohne Kopieren referenziert werden kann.

Passend zu dieser Einführung existiert dort ebenfalls das Projekt *objektlebenszyklus-javaee* mit der in Abbildung 3.2 dargestellten Struktur.

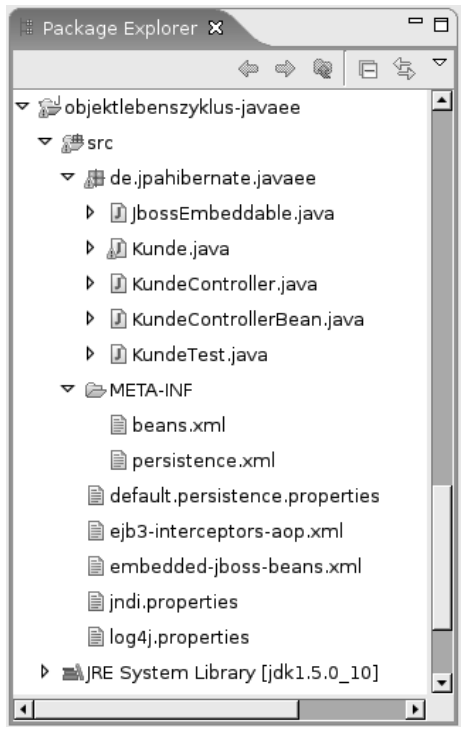

Abbildung 3.2: Struktur eines Java-EE-Projekts mit JBoss EJB 3.0

Das dargestellte Projekt besitzt selbst keine Bibliotheken, referenziert aber das Projekt *jboss-embedded-ejb30-libs* (siehe Abschnitt A.2.1). Die Konfigurationsdateien im `src`-Verzeichnis wurden von uns ohne Anpassungen aus dem JBoss-Paket übernommen. Sie dienen der Konfiguration der Server-Umgebung, auf ihre genauere Darstellung soll hier verzichtet werden. Uns interessieren zunächst die Dateien `beans.xml` und `persistence.xml` im Verzeichnis `META-INF`.

Die Datei `persistence.xml`, dargestellt in Listing 3.3, ist uns als zentraler Bestandteil einer JPA-Konfiguration bereits bekannt.

Listing 3.3: JPA-Konfiguration persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="kundenTest">
    <jta-data-source>java:/HSQLDS</jta-data-source>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="create"/>
    </properties>
  </persistence-unit>

</persistence>
```

Wie der Leser erkennen kann, fehlt die Konfiguration der Datenbank vollständig. Es werden lediglich einige Hibernate-spezifische Properties gesetzt. Hinzugekommen ist dagegen das XML-Element `jta-data-source`. Es verweist auf eine Java-Transaction-API-Datasource mit dem Namen `HSQLDS`. Da die Benutzung eines JTA-Entity-Managers in einer Java-EE-Umgebung der Standardfall ist, muss auch eine entsprechende Datasource konfiguriert sein. Die Thematik der Transaktionen wollen wir an dieser Stelle nicht weiter vertiefen und verweisen auf ihre Einführung in Kapitel 6.

Die referenzierte Datasource konfigurieren wir in der Datei `beans.xml`, dargestellt in Listing 3.4.

Listing 3.4: Konfiguration der JTA-Datasource

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
  xmlns="urn:jboss:bean-deployer:2.0">

  <!-- Enable a JCA datasource available through JNDI -->
  <bean name="HSQLDatasourceFactory"

     class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">
      <property name="jndiName">java:/HSQLDS</property>
      <!-- HSQLDB -->
      <property name="driverClass">org.hsqldb.jdbcDriver</property>
      <property name="connectionURL">
        jdbc:hsqldb:hsql://localhost/datenbank
      </property>
      <property name="userName">sa</property>
      <property name="minSize">0</property>
      <property name="maxSize">10</property>
      <property name="blockingTimeout">1000</property>
```

```
        <property name="idleTimeout">100000</property>
        <property name="transactionManager">
            <inject bean="TransactionManager"/>
        </property>
        <property name="cachedConnectionManager">
            <inject bean="CachedConnectionManager"/>
        </property>
        <property name="initialContextProperties">
            <inject bean="InitialContextProperties"/>
        </property>
    </bean>
    <bean name="HSQLDS" class="java.lang.Object">
        <constructor factoryMethod="getDatasource">
            <factory bean="HSQLDatasourceFactory"/>
        </constructor>
    </bean>
</deployment>
```

3.2.2 Implementierung der EJB-Komponenten

Kommen wir nun zu den eigentlichen Java-Klassen, die wir im Rahmen des Java-Persistence-API betrachten wollen. Dabei legen wir das Interface `KundeController` an, das die Schnittstelle der Client-Anwendung zum EJB-Server darstellt. Es besteht für unser einführendes Beispiel aus nur zwei Methoden.

Listing 3.5: Das Interface KundeController

```
package de.jpahibernate.javaee;

public interface KundeController {

    public void speichereKunde(Kunde k);
    public Kunde ladeKunde(Integer id);

}
```

Im EJB-Container selbst sorgt die Klasse `KundeControllerBean` für die eigentliche Arbeit, hier das Abspeichern eines ihm übergebenen Kunden bzw. das Laden eines Kunden anhand einer bestimmten Id. Diese Klasse ist in Listing 3.6 dargestellt.

Listing 3.6: Die Klasse KundeControllerBean

```
package de.jpahibernate.javaee;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class KundeControllerBean implements KundeController {
```

```
    // diesen bekommen wir per Injektion, wir brauchen ihn nicht
    // zu erzeugen!
    @PersistenceContext
    EntityManager em;

    public Kunde ladeKunde(Integer id) {
        Kunde k = em.find(Kunde.class, id);
        return k;
    }

    public void speichereKunde(Kunde k) {
        em.persist(k);
    }
}
```

Wir verwenden hier eine sogenannte *stateless Session-Bean*. Das bedeutet, dass die Bean keinen für die Anwendung relevanten Zustand besitzt. Sie kann von mehreren Clients parallel benutzt werden. Die Annotation `@PersistenceContext` sorgt dafür, dass bei Verwendung der Variablen `em` der jeweilige EJB-Container eine Instanz der Klasse `EntityManager` bereitstellt und der Variablen vor dem Methodenaufruf zuweist. Des Weiteren bekommt die Methode eine Transaktion zugewiesen, die bei Beenden der Methode automatisch committed wird. Tabelle 3.2 fasst alle Elemente der Annotation `@PersistenceContext` zusammen.

`@PersistenceContext(...)`

Option	Typ	Default	Beschreibung
`name`	`String`	–	Name des `EntityManager`, unter dem er mithilfe von JNDI manuell erzeugt werden kann. Das Attribut wird nicht benötigt, wenn der Manager per Injektion initialisiert wird.
properties	`PersistenceProperty[]`	–	Implementierungsspezifische Einstellungen, die dem Container und Persistenz-Framework übergeben werden können
type	`PersistenceContextType`	`TRANSACTION`	Flag, um zwischen einem transaktionsbezogenen und einem erweiterten Persistenzkontext zu wählen
unitName	`String`	–	Name einer Persistenzeinheit, definiert in der Datei `persistence.xml`

Tabelle 3.2: @PersistenceContext-Optionen

Um unser Beispiel zu komplettieren, fehlt noch eine Klasse mit einer `Main`-Methode. Sie ist in Listing 3.7 dargestellt.

Listing 3.7: Der erste Test unserer EJBs

```
1  package de.jpahibernate.javaee;
2
3  import javax.naming.InitialContext;
4  import org.jboss.ejb3.embedded.EJB3StandaloneBootstrap;
5
6  public class KundeTest {
7
8      public static void main(String[] args) throws Exception {
9          startup("META-INF/beans.xml");
10
11         InitialContext initialContext = new InitialContext();
12         KundeController kundeController = (KundeController) initialContext
13                 .lookup("KundeControllerBean/local");
14
15         Kunde k = new Kunde();
16         k.setVorname("Heide");
17         k.setNachname("Mustermann");
18
19         kundeController.speichereKunde(k);
20
21         kundeController.ladeKunde(1);
22
23         shutdown();
24     }
25
26     public static void startup(String xmlResource) {
27
28         EJB3StandaloneBootstrap.boot(null);
29
30         EJB3StandaloneBootstrap.deployXmlResource(xmlResource);
31
32         EJB3StandaloneBootstrap.scanClasspath("bin");
33     }
34
35     public static void shutdown() {
36         EJB3StandaloneBootstrap.shutdown();
37     }
38 }
```

Die Methoden `startup()` und `shutdown()` enthalten spezifische Anweisungen, um unseren JBoss-Embeddable-EJB-Container entsprechend zu initialisieren. Auf diese soll hier nicht weiter eingegangen werden. Entsprechende Informationen kann der Leser der Dokumentation entnehmen. In Zeile 11 instanziieren wir ein Objekt der Klasse `javax.naming.InitialContext`. Dabei wird die Datei `jndi.properties` ausgelesen. Damit ist der `Context` ab sofort in der Lage, die Implementierung des lokalen Interface `KundeController` zu lokalisieren und uns für die weiteren Operationen bereitzustellen. In Zeile 19 benutzen wir unsere `KundeControllerBean`, ohne sie selbst instanziiert zu haben. Diese Form der Objekterzeugung ist Voraussetzung für die Benutzung des Controller-Bean und dessen Entity-Manager, den wir über die sogenannte Injektion erhalten. Hätten

wir die Variable `kundeController` statt über den `InitialContext` mit dem Statement

```
kundeController = new KundeControllerBean();
```

initialisiert, würden die Methoden `ladeKunde()` und `speichereKunde()` mit einer `java.lang.NullPointerException` abbrechen, weil der Entity-Manager nicht korrekt initialisiert wurde.

Wie dieses Kapitel gezeigt hat, müssen Anwendungen in einer Java-EE-Umgebung nicht mit einer Implementierung von `EntityManagerFactory` kommunizieren, um einen Persistenzkontext zu erzeugen oder um auf einen bestehenden zuzugreifen. `EntityManager`-Instanzen lassen sich entweder per Injektion, die Spezifikation spricht von *Dependency-Injection*, oder mithilfe von JNDI aufbauen. Auf das Konzept der Dependency-Injection gehen wir in Kapitel 10 näher ein.

3.3 Entity-Listener und Callback-Methoden

Um individuell auf Veränderungen im Lebenslauf eines Entity reagieren zu können, stellt die JPA Listener und Callback-Methoden zur Verfügung. Methoden dieser Klassen werden vor oder nach der Ausführung von `persist()`, `remove()`, `update()` oder dem Laden eines Objekts aufgerufen. Übernimmt das Entity die Implementierung dieser Methoden selbst, spricht man von Callback-Methoden. Mit Implementierung ist hier aber nicht die Anwendung eines Java-Interface gemeint. Die Möglichkeit, Callback-Methoden anzuwenden, wird über Annotationen ermöglicht. Dafür stehen zur Auswahl:

- `@PrePersist`
- `@PostPersist`
- `@PreRemove`
- `@PostRemove`
- `@PreUpdate`
- `@PostUpdate`
- `@PostLoad`

Die Vorsilbe *Pre* steht für Aktionen, die das JPA-Framework *vor* der eigentlichen Aktion ausführt, *Post* bedeutet *nach* der benannten Aktion. Sehen wir uns zunächst die Callback-Methoden am Beispiel der Klasse `Konto` an. Die Annotation `@PostPersist` ist direkt in der Klasse `Konto` an einer Methode mit beliebigem Namen verankert. Die zugrunde liegende Methode kann `private` sein, um unberechtigten Zugriff außerhalb der Persistenzmechanismen zu verhindern. Die annotierte Methode muss parameterlos sein.

```
@Entity
public class Konto implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```

```
    private Integer nummer;
    private BigDecimal kontostand;

    public Konto() {
    }

    @PostPersist
    private void testCallBack() {
        System.out.println("Nach Persist aufgerufen");
    }
}
```

Default-Listener-Klassen

Listener-Klassen übernehmen die oben beschriebene Funktionalität nicht im Entity selbst, sondern in einer eigenen Klasse. Diese Klasse muss kein bestimmtes Interface implementieren oder von einer bestimmten Klasse erben. Erforderlich ist lediglich der Default-Konstruktor. Die Signaturen der Listener-Methoden müssen dagegen genau einen Parameter vom Typ `Object` enthalten:

```
void methodenName(Object t)
```

Dieser Parameter ist das Entity, auf das eine der oben beschriebenen Persistenzaktivitäten gerade angewendet wird.

Default-Listener gelten im Allgemeinen für alle Entity-Klassen, die der Persistenzeinheit zugrunde liegen. Sie werden in einer separaten Mapping-Datei definiert. In dieser XML-Mapping-Datei kann die gesamte Konfiguration einer JPA-Anwendung auch ohne Annotationen durchgeführt werden. Details dazu beschreiben wir in Kapitel 8.

Default-Listener werden im Abschnitt `persistence-unit-metadata` wie folgt definiert:

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
       <entity-listener class="de.jpahibernate.util.DefaultListener2">
          <post-persist method-name="postPersistDefaultListener"/>
       </entity-listener>
       <entity-listener class="de.jpahibernate.util.DefaultListener">
          <post-persist method-name="postPersistDefaultListener"/>
       </entity-listener>
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Mithilfe der Annotation `@ExcludeDefaultListeners` können die Default-Listener von bestimmten Entities ausgeschlossen werden. Stattdessen lassen sich die nachfolgend beschriebenen Entity-Listener verwenden.

Entity-Listener-Klassen

Entity-Listener werden über die Annotation `@EntityListeners(Listener.class)` oder in der Mapping-Datei festgelegt. Wir beschränken uns hier auf die Konfiguration als

Annotation. Die Implementierung erfolgt analog zu einem Default-Listener. Einziger Unterschied ist die Annotation der jeweiligen Methode, die im obigen XML-Ausschnitt über das Attribut `method-name` realisiert wurde.

```
public class KontoListener {

    @PostPersist
    private void postPersistKontoListener(Object k) {
        System.out.println("postPersistKontoListener");
    }
}
```

Im zu überwachenden Entity muss die Listener-Klasse anschließend noch bekannt gemacht werden:

```
@Entity
@EntityListeners(KontoListener.class)
public class Konto implements Serializable {
    ...
}
```

Treten in den beschriebenen Methoden zur Laufzeit Exceptions auf, wird die Transaktion (sofern vorhanden) mit einem Rollback rückgängig gemacht. Weiter ist es möglich, in einer Entity-Klasse mehrere Listener-Klassen zu definieren, die ihrerseits auf dasselbe Ereignis reagieren können.

Wie die vorangegangenen Betrachtungen zeigen, ist es möglich, auf ein Ereignis im Lebenslauf eines persistenten Objekts auf verschiedene Art und Weise, z.B. durch Callback-Methoden oder Listener, zu reagieren. Diese Arten der Ereignisbearbeitung können auch mehrfach durchgeführt werden. So werden im Rahmen der Ereignisverarbeitung auch Callback-Methoden von Oberklassen sowie deren Definitionen von Entity-Listeners berücksichtigt. Dabei wird immer folgende Reihenfolge eingehalten:

- Zuerst kommen die Default-Listener in der Reihenfolge ihrer Konfiguration in der Mapping-Datei zur Ausführung.
- Als Nächstes gelangen die in den Entity-Klassen definierten Entity-Listener ebenfalls in der Reihenfolge ihrer Anordnung in der `@EntityListeners`-Annotation zur Ausführung. Finden sich in den Oberklassen des Entity weitere Definitionen, gelangen zunächst diese zur Ausführung, beginnend mit der obersten Klasse. `@ExcludeSuperclassListeners` unterbindet diese Aufrufe.
- Danach werden die im Entity definierten Callback-Methoden aufgerufen. Auch hier werden zunächst die Methoden der in der Hierarchie am höchsten stehenden Klasse aufgerufen. Dieses Verhalten kann aber auch durch Überschreiben der Methoden verändert werden.

Folgendes Beispiel verdeutlicht die Zusammenhänge:

```
public class DefaultListener {
    private void postPersistDefaultListener(Object k) {
        System.out.println("postPersistDefaultListener");
    }
}
```

```
...

public class DefaultListener2 {
    private void postPersistDefaultListener(Object k) {
        System.out.println("postPersistDefaultListener2");
    }
}

...

public class KontoListener {
    @PostPersist
    private void postPersistKontoListener(Object k) {
        System.out.println("postPersistKontoListener");
    }
}

...

public class KontoListener2 {
    @PostPersist
    private void postPersistKontoListener(Object k) {
        System.out.println("postPersistKontoListener2");
    }
}

...

@Entity
@EntityListeners(KontoListener.class)
public class Konto implements Serializable {

    @PostPersist
    public void testCallBackKonto() {
        System.out.println("testCallBackKonto");
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private Integer nummer;
    private BigDecimal kontostand;
}

...

@Entity
@EntityListeners( { KontoListener2.class })
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Sparkonto extends Konto {

    @PostPersist
    public void testCallBackSparkonto() {
        System.out.println("testCallBackSparkonto");
    }
    private BigDecimal guthabenzinssatz;
}
```

Wird eine Instanz der Klasse `Sparkonto` mittels `persist()` in der Datenbank gespeichert, werden zunächst die in den Default-Listeners implementierten Listener-Methoden zur Ausführung gebracht. Die Konfiguration des Beispiels orientiert sich am Ausschnitt der Mapping-Datei auf Seite 96. Anschließend werden die Methoden der in der Klassenhierarchie definierten Entity-Klassen beginnend mit der obersten Klasse ausgeführt. Schließlich gelangen die Callback-Methoden nach dem gleichem Prinzip zur Ausführung.

Mit der folgenden Testklasse überprüfen wir unsere Aussagen:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("kundenTest");

Konto konto = new Sparkonto();
konto.setNummer(123456789);
konto.setKontostand(new BigDecimal("12.34"));

EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(konto);

tx.commit();
em.close();

emf.close();
```

Auf der Konsole ergibt sich die folgende Ausgabe:

```
postPersistDefaultListener2
postPersistDefaultListener
postPersistKontoListener
postPersistKontoListener2
testCallBackKonto
testCallBackSparkonto
```

Hinweis

Callback-Methoden können als `public`, `private` oder `protected` deklariert sein. Sie dürfen jedoch nicht `static` oder `final` sein.