Brainwash your Developers – Richtiges Java für PL/SQLer

Wolf G. Beckmann, TEAM

Bei neuen Entwicklungen ist JavaEE im Backend als Standard gesetzt. Als PL/SQL-Entwickler kennt man sich gut mit der Verarbeitung und Speicherung von Daten aus, hat aber typischerweise eine datenbankorientierte Sichtweise. Java geht da einen anderen Weg, den es lohnt, mitzugehen.

In Java hat sich die Art, wie auf persistente Daten, also Daten, die in der Datenbank gespeichert werden, zugegriffen wird, über die Jahre stark verändert. Am Anfang sind die „Datenbank-Entwickler“ genauso vorgegangen, wie es in PL/SQL auch gemacht wurde. Über JDBC (Java Database Connectivity) wurden native SQL-Abfragen, Updates und auch DDL-Statements an die Datenbank gesendet. Dabei wurde das SQL-Statement als String an JDBC übergeben, gegebenenfalls noch Parameter gesetzt und ausgeführt. Das Ergebnis wurde in einem speziellen ResultSet abgearbeitet *(siehe Listing 1)*.

```
public void jdbcExample() throws SQLException {
   Connection con = getConnection();

   PreparedStatement stmnt = con.prepareStatement(
          "select * from Location where code = ?");
   stmnt.setString(1, "PB");
   stmnt.execute();
   ResultSet res = stmnt.getResultSet();

   while (res.next()) {
       System.out.println(res.getString("CITY"));
       System.out.println(res.getString("COUNTRY"));
   }
   stmnt.close();
}
```

Listing 1: Select via JDBC

```
public class Department {
private String name;
private String city;
private List<Employee> employees;

public String getName() {
       return name;
   }
   public void setName(String name) {
       this.name = name;
   }
   …
}

public class Employee {
   private String name;
   private String job;
   private Float salary;

   public String getName() {
       return name;
   }
   public void setName(String name) {
       this.name = name;
   }
   …
}
```

Listing 2: Beispiel POJOs

```
@Entity
public class Department {
   @Id
   private String name;
   private String city;
   @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
   private Set<Employee> employees;

   public void addEmployee(Employee employee) {
       if (employees == null) {
           employees = new HashSet<Employee>();
       }
       employees.add(employee);
       employee.setDepartment(this);
   }

   public void remEmployee(Employee employee) {
       getEmployees().remove(employee);
       employee.setDepartment(null);
   }
   …
}

public class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String name;
   private String job;
   private Float salary;
   @ManyToOne
   private Department department;

   public String getName() {
       return name;
   }
   public void setName(String name) {
       this.name = name;
   }
   …
}
```

Listing 3: Beispiel Entities

Einige Nachteile dieser Vorgehensweise waren, dass relativ viel Code notwendig war, das Statement erst zur Laufzeit auf seine Korrektheit geprüft wurde und die Daten umkopiert werden mussten.

Als nächster Schritt wurde ein Pattern, also ein Muster für die Entwicklung, eingeführt: das DAO (Data Access Object) Pattern. Das DAO kapselt die Persistierung in eine eigene Klasse und stellt Methoden zum Laden, Abfragen, Verändern und Speichern eines Objektes bereit, wobei das Objekt eine Repräsentation einer Datenbank-Tabelle ist. Durch diese Kapselung wurde schon viel erreicht: Der eigentliche Business-Code arbeitet mit Objekten und ist aufgeräumter, da die umständlichen JDBC-Aufrufe in der DAO-Klasse liegen. Dennoch gibt es unter anderem die Nachteile, dass die Verwaltung der Objekte explizit im Code vorgenommen werden muss, das Durchlaufen von Hierarchien aufwendig ist und sich der Entwickler weiter um die Datenbank kümmern muss.

Das Java Persistence API (JPA)

Dann wurde das Thema Persistierung noch einmal völlig neu überdacht. Java und die Verwendung der Objekte sollten im Vordergrund stehen und nicht mehr die Speicherung der Daten.

Das Ergebnis war JPA. JPA ist ein definiertes API, das von verschiedenen OR-Mappern (objektrelationalen Mappern) wie Hibernate oder EclipseLink implementiert wird. Diese OR-Mapper sind dann beispielsweise Bestandteil des jeweiligen Application-Servers.

JPA versucht, den Entwickler so wenig wie möglich von der Persistierung spüren zu lassen. In Java werden die Daten in Objekten gehalten und diese sollten lediglich persistiert werden. Es sollte die Rückkehr zu den „Plain Old Java Objects“ (POJO) sein. Ein POJO besteht nur aus den Attributen sowie den dazugehörigen Gettern und Settern *(siehe Listing 2)*.

Persistieren von Daten

Um diese POJOs persistieren zu können, werden sie mit Annotationen zu Entities erweitert *(siehe Listing 3)*. Dabei müssen sie auch einigen Anforderungen genü-

gen. Sie brauchen beispielsweise eine eindeutige Id. Des Weiteren müssen die Beziehungen näher beschrieben werden. Um (vernünftig) eine Liste von Unterobjekten (in *Listing 3* die Employees) zu persistieren, wird typischerweise im Unterobjekt eine Rückreferenz benötigt. Um diese Rückreferenz einfach zu bearbeiten, werden zwei Hilfsmethoden im Entity hinzugefügt (hier addEmployee() und remEmployee()).

Über die Annotation @Id wird das Attribut mit dem Primärschlüssel angegeben, wobei die Verwaltung auch von JPA über die Angabe @GeneratedValue übernommen werden kann. Bei der @OneToMany-Annotation wird in dem Beispiel zum einen angegeben, in welchem Attribut die Rückreferenz gespeichert ist (mappedBy), und zum anderen, wie mit den Unterentities umgegangen werden soll. Dazu später im Beispiel mehr.

Die Entities können nun vom EntityManager (in den Beispielen in em instanziiert) verwaltet werden. Der EntityManager überwacht dazu alle ihm anvertrauten Entities und wenn diese sich im Rahmen einer Transaktion (JPA-Transaktion, nicht Datenbank-Transaktion) verändern (anlegen, ändern, löschen), werden diese Veränderungen in der Datenbank persistiert. Oder anders ausgedrückt: Es werden die entsprechenden Inserts, Updates und Selects gegen die Datenbank ausgeführt.

In *Listing 4* wird gezeigt, wie ein Entity gespeichert wird.

Wie man sieht, muss man sich auch nicht selbst um die Transaktion kümmern, sondern braucht, dank JTA (Java Transaction API), nur den Einstiegspunkt der Transaktion zu annotieren. Die Methode und auch alle von der Methode aufgerufenen Methoden verwenden eine gemeinsame Transaktion, die mit dem Verlassen beendet wird.

```
@PersistenceContext
EntityManager em;

@Transactional
public void departmentAnlegenBeispiel() {

   // Department erstellen
   Department dep = new Department();
   dep.setName("Sales");
   dep.setCity("Paderborn");

   // Mitarbeiter erstellen
   Employee emp = new Employee();
   emp.setName("Peter Müller");
   emp.setJob("Salesman");
   emp.setSalary(7000f);
   dep.addEmployee(emp); // und hinzufügen

   // Mitarbeiterin erstellen
   emp = new Employee();
   emp.setName("Andrea Meier");
   emp.setJob("Saleswoman");
   emp.setSalary(8000f);
   dep.addEmployee(emp); // und hinzufügen

   // persistieren
   em.persist(dep);

}
```

Listing 4: Persistieren von Objekten

```
// Department über Id laden
Department dep = em.find(Department.class, "Sales");

// Department suchen
Query q = em.createQuery(
   "select d from Department d where name = :name",
   Department.class);
q.setParameter("name", "Sales");
List<Department> depList = q.getResultList();
```

Listing 5: Laden oder Suchen von Entities

```
@Transactional
public void increaseDepartmentSalary(String depName,
                                     Float percent,
                                     Float maxSal) {
   Department dep = em.find(Department.class, depName);
   for (Employee emp: dep.getEmployees()) {
       Float newSal = emp.getSalary() * (1f+(percent/100));
       if (newSal <= maxSal) {
           emp.setSalary(newSal);
       }
   }
}
```

Listing 6: Verändern von Entities

Mit persistierten Daten arbeiten

Um ein Entity aus der Persistierung wieder zu restaurieren, kann man ein Entity anhand seiner Id finden oder über einen Query suchen *(siehe Listing 5)*.

Auf den ersten Blick könnte man den Query mit SQL verwechseln. Das ist auch gewollt. In einem solchen Query wird die Sprache JPQL (Java Persistence Query Language), die stark an SQL angelehnt ist, verwendet. In *Listing 5* wird von Department selektiert. Dabei ist nicht die Tabelle, sondern die Klasse (oder konkret das Entity) gemeint, die mit d „gealiast" wird und durch die d (also das Entity Department) komplett als Liste zurückgegeben wird. Queries werden allerdings normalerweise nicht wie in

Listing 5 als String übergeben, sondern als NamedQuery per Annotation in der Klasse des Entitys definiert. Das hat zum einen den Vorteil, dass der Query schon zur Compile-Zeit geparst und auf syntaktische Korrektheit geprüft wird, und zum anderen, dass er an einer zentralen Stelle gehalten wird. Statt em.createQuery() wird dann em.createNamedQuery() aufgerufen.

Dem geschulten Auge ist sicherlich auch aufgefallen, dass weder in *Listing 4* noch in *Listing 5* der EntityManager über die Employees informiert wurde, die als Liste im Department vorhanden sind. Dass sie mit persistiert wurden, liegt an CascadeType.ALL bei der @OneToMany-Annotation *(siehe Listing 3)*. Sie bewirkt, dass unter anderem die Unterentities mit persistiert, aber auch gelöscht etc. werden.

Spannend ist auch, dass beim Restaurieren des Departments aus der Persistenz wirklich nur das Department geladen wird. Für die Liste der Employees wird von JPA ein Proxy-Objekt eingesetzt, dass beim Ansprechen schnell die notwendigen Unterentities nachlädt. Der Entwickler braucht sich also auch hier um nichts zu kümmern.

Wenn die Entities über den EntityManager geladen werden, egal ob über find() oder einen Query, werden sie vom EntityManager überwacht. Das bedeutet, dass die Entities bei Änderungen am Ende der Transaktion gespeichert werden.

Listing 6 zeigt sowohl das Lazy-Loading als auch das Speichern von Änderungen.

Auffällig ist, dass der Code, der sich in *Listing 6* mit der Persistierung beschäftigt, gerade mal zwei Zeilen darstellt: @Transactional und die Zeile mit dem em.find(). Als Entwickler kann ich mich also auf die Fachlichkeit konzentrieren.

Es gibt noch viele weitere Features von JPA, die dem Entwickler die Arbeit mit persistenten Daten erleichtern. An dieser Stelle soll noch auf die Möglichkeit eingegangen werden, Entities gemeinsam zu persistieren. In Java sind Objekte mit vielen Eigenschaften verpönt, da sie dadurch unübersichtlich werden. Es werden lieber Unterobjekte angelegt. Beispielsweise besteht der Name einer Person zumindest aus Titel, Vorname und Nachname. Dennoch ist er so stark an die Person gebunden, dass wohl nur wenige auf die Idee kommen würden, den Namen einer Person in einer eigenen Tabelle halten zu wollen. Es wird auch im Normalfall kein Geschwindigkeitsvorteil zu erwarten sein, wenn sich die Daten verringern und nur bei Bedarf der Name nachgeladen wird. Ein weiterer Punkt für das Auslagern ist, dass eine Namens-Klasse wiederverwendet werden kann. Ein Kunde, Verkäufer, Verantwortlicher etc. hat einen Namen. Wenn ein Namensobjekt verwendet wird, werden alle die gleichen Attribute haben und kein Entwickler braucht darüber nachzudenken, wie die Attribute im konkreten Fall heißen.

Um ein Unterentity mit einer so starken Bindung abzudecken, wird der Dependent beziehungsweise das Embeddable Entity angeboten.

In *Listing 7* wird der Employee dahingehend erweitert, dass der Zeitraum seiner Einstellung (employedSince, hiredUntil) durch das @Embedded Entity employmentPeriod abgebildet wird.

Hinweis: Durch die Angabe @Temporal(TemporalType.DATE) weiß JPA, dass kein Uhrzeitanteil persistiert werden soll. Bei einer MySQL-Datenbank wird durch die Angabe beispielsweise ein anderer Spaltentyp verwendet.

In der Datenbank befinden sich die Spalten employedSince und hiredUntil in der Tabelle Employee genauso wie name, job und salary.

Man muss wissen, was man tut

Obwohl JPA dem Entwickler sehr viel abnimmt, muss man dennoch wissen, wie JPA funktioniert.

Beispielsweise entstehen schnell verwaiste Tabelleneinträge, wenn Unterentities gelöscht werden. Um in unserem Beispiel einen Employee zu löschen, reicht es nicht, den Listeneintrag zu entfernen. Auf den ersten Blick sieht die Methode remEmployee() sehr vielversprechend aus. Wenn man mit ihr ein Entity löscht und danach das Department neu lädt, ist auch der Employee verschwunden. In der Datenbank bleibt der Tabelleneintrag jedoch stehen, lediglich der Verweis auf das Department ist null. Das hat damit zu tun, dass eigentlich nur die Verbindung zwischen den Objekten gekappt wurde. Aus Sicht des EntityManager wird der Employee mit remEmployee() nur verändert, er bleibt aber dem EntityManager weiter bekannt. Er bleibt im sogenannten EntityCache. Um ein Entity zu löschen, muss es wie in *Listing 8* explizit mit em.remove() aus dem Cache entfernt werden.

Hinweis: Das Konstrukt new ArrayList<Employee>(dep.getEmployees()) ist notwendig, da wir aus einem Set Elemente löschen wollen, während darüber iteriert wird.

```
@Embeddable
public class EmploymentPeriod {
   @Temporal(TemporalType.DATE)
   Date employedSince;
   @Temporal(TemporalType.DATE)
   Date hiredUntil;

   public Date getEmployedSince() {
       return employedSince;
   }
   …
}

@Entity
public class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String name;
   private String job;
   private Float salary;
   @Embedded
   private EmploymentPeriod employmentPeriod;
   @ManyToOne
   private Department department;
   …
}
```

Listing 7: Verändern von Entities

Natürlich ist es auch notwendig, JPA zu verstehen, wenn man auf die Performance achten muss. Natürlich ist es möglich, über NativeQueries wie bei JDBC direkte Datenbank-Abfragen in SQL zu erstellen. Aber das sollte wirklich nur im absoluten Notfall passieren. Am häufigsten stolpert man bei der Performance über das Lazy-Loading-Verhalten. Wenn wir uns *Listing 6* anschauen, werden zum Holen der Entities von JPA zwei Selects ausgeführt. *Listing 9* zeigt die beiden generierten Selects.

Zwei Selects kann man noch verschmerzen, die Anzahl kann sich allerdings schnell potenzieren, wenn der Objektbaum noch eine 3. Ebene bekommt (beispielsweise Locations → Departments → Employees).

Deshalb ist es möglich, JPA anzuweisen, mehrere Ebenen direkt zu selektieren. Das kann permanent bei der Definition der Entities in der @OneToMany-Annotation über fetch = FetchType.EAGER geschehen. Es ist auch möglich, in einem Query, bei dem die Entities explizit über join fetch miteinander verknüpft werden, oder über einen EntityGraph, der dem Query oder find() als Property übergeben wird, beim Laden explizit anzugeben, wann Lazy-Loading verwendet werden soll. Die Verwendung eines EntityGraph ist sehr elegant, da beim gleichen Query oder find() je nach Bedarf unterschiedliche EntityGraphs verwendet werden können. *Listing 10* zeigt die Methode aus *Listing 6* (increaseDepartmentSalary()), um einen EntityGraph erweitert. Durch diese Erweiterung generiert JPA nur noch einen Select, um die Daten zu holen.

Hinweis: Häufig werden in Beispielen von Entities Listen statt Sets für Unterobjekte verwendet. Das macht an vielen Stellen den Code einfacher. Aber gerade, wenn das Lazy-Loading ausgehebelt werden soll, entstehen unerwartete Ergebnisse ab einer Objekthierarchie mit 3 Ebenen.

Datenstrukturen

Zum Abschluss noch ein Hinweis zu den Folgen, wenn mit JPA gearbeitet wird:

Durch JPA wird in Java mit der Datenbank völlig anders umgegangen. Der Fokus liegt auf der Verwendung der Daten, nicht auf der Speicherung. Dadurch erge-

```
@Transactional
public void deleteHighSalaryEmployees(String deptName, Float maxSalary) {
   Department dep = em.find(Department.class, deptName);
   for (Employee emp:new ArrayList<Employee>(dep.getEmployees()))
{
      if (emp.getSalary() > maxSalary) {
          dep.remEmployee(emp);
          em.remove(emp);
      }
   }
}
```

Listing 8: Löschen von Entities

```
Hibernate:
      select
          department0_.name as name1_0_0_,
          department0_.city as city2_0_0_
      from
          Department department0_
      where
          department0_.name=?
Hibernate:
      select
          employees0_.department_name as departme7_1_0_,
          employees0_.id as id1_1_0_,
          employees0_.id as id1_1_1_,
          employees0_.department_name as departme7_1_1_,
          employees0_.employedSince as employed2_1_1_,
          employees0_.hiredUntil as hiredUnt3_1_1_,
          employees0_.job as job4_1_1_,
          employees0_.name as name5_1_1_,
          employees0_.salary as salary6_1_1_
      from
          Employee employees0_
      where
          employees0_.department_name=?
```

Listing 9: Zwei Selects durch Lazy-Loading

```
private Map<String, Object> getFetchAllEntityGraphProperty() {
   EntityGraph<Department> fetchAll =
              em.createEntityGraph(Department.class);
   fetchAll.addAttributeNodes("employees");
   Map<String, Object> properties = new HashMap<>();
   properties.put("javax.persistence.fetchgraph", fetchAll);
   return properties;
}

@Transactional
public void increaseFastDepartmentSalary(String depName,
                                         Float percent,
                                         Float maxSal) {
   Department dep = em.find(Department.class,
                            depName,
                            getFetchAllEntityGraphProperty());
   for (Employee emp: dep.getEmployees()) {
       Float newSal = emp.getSalary() * (1f+(percent/100));
       if (newSal <= maxSal) {
           emp.setSalary(newSal);
       }
   }
}
```

Listing 10: find() mit EntityGraph

ben sich auch Änderungen an den Datenstrukturen. Bei der Modellierung der Entities sollte auf Folgendes geachtet werden:

- Kleine, an der Nutzung ausgerichtete, fachliche Strukturen/Hierarchien aufbauen
- Der Kopf der Struktur kann einen fachlichen Schlüssel bekommen, die darunterliegenden Strukturen nur einen technischen.
- Objektstrukturen werden aufgeteilt, Datenstrukturen zusammengefasst (Stichwort Embeddable)
- Bei großen Datenmengen, z.B. BLOBs, Unterstrukturen anlegen

Noch ein allerletzter Hinweis:
Um mit JPA etwas zu spielen, stelle ich ein Demo-Projekt unter *https://gitea.apps.soco.team-pb.de/wb/DoagDemoJPA* zur Verfügung.

Quellen

[1] Ansonsten benötigt JPA eine Zwischentabelle, aber wer will das schon bei einer 1:n-Beziehung?

Über den Autor

Wolf G. Beckmann hat selbst über 10 Jahre PL/SQL entwickelt, bis er als Teamleiter in den Consulting-Bereich gewechselt ist, der primär mit Java arbeitet. Fast alle Projekte im Consulting-Bereich beschäftigen sich mit dem Wechsel aus einer von PL/SQL hin zu einer von Java getriebenen Welt. Auch aus eigener Erfahrung legt er in den Projekten neben der Technik großen Wert darauf, wie die Mitarbeiter bei so einem Wechsel mitgenommen werden.

Wolf G. Beckmann
wb@team-pb.de

Bessere APIs dank GraphQL?

Markus Lohn, esentri

GraphQL wurde ursprünglich von Facebook entwickelt und im Jahre 2015 als Open Source zur Verfügung gestellt. In der Zwischenzeit ist ein regelrechter Hype um die Spezifikation entstanden, ja sogar das Ende von REST wurde ausgerufen. GraphQL ist eine interessante Möglichkeit, APIs zu definieren. Ferner bietet es Lösungen für eine Reihe von Herausforderungen, die typischerweise im Bereich REST auftreten können. Allerdings gibt es nicht nur GraphQL: HAL, JSON:API etc. sind weitere Möglichkeiten, um robuste und flexible APIs zu definieren. In diesem Artikel stelle ich GraphQL vor und versuche, die Unterschiede darzustellen.

APIs haben häufig das Problem mit dem Absetzen von vielen Anfragen an einen Server, um alle benötigten Daten für ein Geschäftsobjekt zu laden. Als Beispiel dient die Anzeige einer Rechnung auf einem mobilen Endgerät oder einer Website *(siehe Abbildung 1)*. Zunächst werden die Basisdaten einer Rechnung wie Rechnungsnummer, Rechnungsdatum, Gesamtbetrag sowie die Referenzen auf den Kunden etc. angefragt. Im nächsten Schritt müssen die Details zu einem Kunden erfragt werden. Danach erfolgt die Abfrage der Rechnungspositionen. Für alle verfügbaren Rechnungspositionen werden unter Umständen weitere Abhängigkeiten, zum Beispiel zu den Produktdaten vom Server, abgeholt. Somit ergibt sich für dieses einfache Beispiel eine Vielzahl von Serveranfragen, die allein für sich nicht komplex sind, aber dennoch eine unnötige Anfragelast bei einem Server erzeugen.

Generell musste die Anbindung von mobilen Applikationen verbessert werden. Die Entwicklung zu GraphQL durch Facebook startete 2012. Im Jahr 2015 erfolgte die Bereitstellung der Spezifikation als Open Source. Für die Weiterentwicklung der GraphQL-Spezifikation zeichnet die GraphQL Foundation verantwortlich. In der Foundation sind neben Facebook weitere namhafte Unternehmen wie Airbnb, AWS, Twitter oder IBM aktiv.

GraphQL definiert eine Abfragesprache für APIs. Auch wenn es der Name vermuten lässt, hat GraphQL nichts mit Graphendatenbanken gemein! Der Name resultiert aus der Tatsache, dass