Hibernate Kochbuch

Von Thomas Kestler, elevato GmbH

Einleitung

Dieses „Kochbuch“ ist eine kompakte Zusammenfassung von Themen, die einem bei Hibernate immer wieder begegnen. Es erhebt keinen Anspruch auf Vollständigkeit, sondern ist als kleines Nachschlagewerk (nicht zuletzt für den Autor selbst) gedacht. Die Ausführungen basieren auf Hibernate 3.2. Eine Aktualisierung des Dokuments auf die neueste Hibernate-Version ist geplant.

XML oder Annotations

Hibernate unterstützt derzeit mehrere Mappings:

- .hbm.xml, die althergebrachten XML-Files von Hibernate
- Annotations (JPA und Hibernate-spezifische)
- JPA XML Files

Wer von Hibernate 2 kommt, kennt die .hbm.xml Files sicher bereits. Sie bieten alle Möglichkeiten von Hibernate und stellen wohl den durchgängigsten Weg zum Mapping dar.

Die JPA-Annotationen folgen dem EJB 3.0 Standard und sind bequem zu verwenden, außerdem sind Annotations gerade in Mode. Leider benötigt man Hibernate-spezifische Annotationen, da der Standard nicht alle Möglichkeiten von Hibernate abdeckt. Dummerweise haben die teilweise gleiche Namen, so dass Hibernate-Annotationen unbedingt mit voller Qualifikation angegeben werden sollten:
`@org.hibernate.annotations.Cache(usage=CacheConcurrencyStrategy.READ_ONLY)`
Damit wird aber auch klar, dass der Wunsch nach Portabilität nicht erfüllt wird.

JPA XML Mappings folgen dem Standard, aber da etliche Hibernate Features damit nicht oder nur umständlich abbildbar sind, macht das wenig Spaß. Für mich derzeit kein Favorit.

To be Lazy Or Not to be

Das Lazy Loading ist ein mächtiges Feature von Hibernate, kann aber auch Kopfzerbrechen bereiten, wenn man sich nicht vorher ein paar Gedanken darüber macht. Ein Vorgehen, das sich bewährt hat besteht darin, prinzipiell alles als lazy zu mappen und bei Lesen durch Setzen von FetchMode die benötigten Daten ausdrücklich mitzulesen:

```
public class OrderDAOImpl {

 public Order findById(long id) {
  DetachedCriteria crit = DetachedCriteria.forClass(Order.class);
  crit.setFetchMode("customer", FetchMode.JOIN);
  crit.setFetchMode("customer.country", FetchMode.JOIN);
  crit.setFetchMode("items", FetchMode.JOIN);
  crit.setFetchMode("items.product", FetchMode.JOIN);
  crit.setResultTransformer(CriteriaSpecification.DISTINCT_ROOT_ENTITY);
  crit.add(Restrictions.eq("id", id));
```

```
  ListCustomer> list = getHibernateTemplate().findByCriteria(crit);
  if (list.isEmpty()) {
   return null;
  }
  Customer customer = list.get(0);
  return customer;
 }

}
```

! Lazy Loading hat vor allem Bedeutung bei den ManyToOne-Beziehungen und es sollte im Projekt die generelle Regelung gelten, dass alle ManyToOne-Beziehungen explizit als Lazy gekennzeichnet werden:

```
@ManyToOne (  targetEntity = Halter.class, fetch = FetchType.LAZY)
 @JoinColumn (name = "halter_id")
 private Halter halter;
```

Beim Lesen, z.B. in der DAO-Klasse muss das Laden dann explizit ausgeführt werden (über FetchMode.JOIN wie oben oder über Hibernate.initialize()). Wird der FetchType nicht angegeben, so gilt per Default `FetchType.EAGER`.

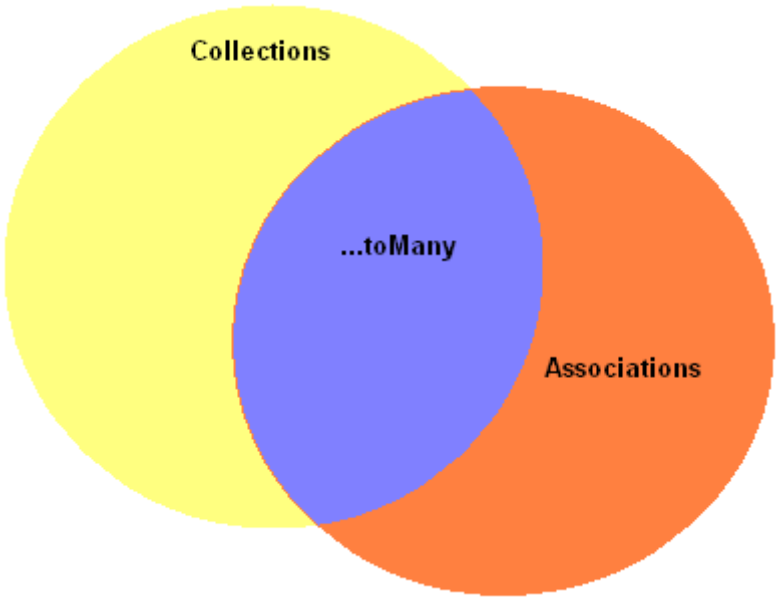
Pitfall Collections / Associations

Viele Programmier stolpern ständig über den Themenkomplex Collections und Associations. Sowohl das Reference Manual (online), als auch die Bücher von Bauer/King [1] tragen Ihren Teil zur Verwirrung bei. Die Ursache des Problems scheint zu sein, dass Hibernate die Sicht aus JAVA auf die Datenbankabbildung hat: in JAVA gibt es Collections, wie können wir die auf Datenbankaspekte mappen (und nicht: in der Realität gibt es folgende Anforderungen, wie können diese modelliert werden).

Man muss diese Sichtweise berücksichtigen, um das Thema zu verstehen. Sieht man sich das Mapping mit XML an, so findet man, dass zunächst die Collection als Property beschrieben und darin(!) dann die Verwendung, bzw. Abbildung:

```
  <bag name="bids"  inverse="true"
    cascade="all">
   <key column="item"/>
   <one-to-many class="Bid"/>
  </bag>
```

Hibernate kennt mehrere Arten von Collections: `<list>`, `<set>`, `<bag>`, `<idbag>`. Hibernate kennt auch mehrere Arten von Assoziationen: `<one-to-one>`, `<one-to-many>` `<many-to-one>`, `<many-to-many>` . Die *toMany-Assoziationen werden technisch durch Collections modelliert (abgebildet). Somit gibt es eine Schnittmenge zwischen Collections und Assoziationen:



Häufige Anwendungsfälle

Das Mapping des Objektmodels macht vielen Programmierern immer wieder Kopfzerbrechen, nicht zuletzt, weil die Konzepte von Hibernate nicht unbedingt intuitiv sind und die verfügbare Dokumentation nicht gerade zielführend. Dabei kann man das Thema mit ein paar einfachen Faustregeln einfach meistern. Im folgenden wird daher Vorgehen beschrieben, das auf 95% aller Anwendungsfälle passen sollte.

Internes Attribut (Embedded, <component>)

Wenn Attribute einer Entität im Objektmodell als eigene Klasse modelliert sind, können diese ebenfalls persistiert werden. Es handelt sich logisch um eine 1:1 Beziehung wie zwischen `User` und `Address`. Die Klasse `Address` wird dazu als `embeddable` gekennzeichnet:

```
@Embeddable
public class AddressEmbeddable {

 @Column (name = "adr_country", nullable = false)
 private String country;

 @Column (name = "adr_city", nullable = false)
 private String city;

 @Column (name = "adr_street", nullable = false)
 private String street;

 @Column (name = "adr_houseno", nullable = true)
 private String houseno;
```

In der umgebenden Klasse `User` wird die Property dann wie folgt annotiert:

```
@Entity
@Table (name = "UserTable") // USER ist reserviertes Wort in ORACLE
public class UserEmbed {
 @Id
 @GeneratedValue(strategy=GenerationType.AUTO)
 private long id;

 private String userName;

 private String password;

 private String email;

 @Embedded
 private AddressEmbeddable address;
```

Alle Attribute von `User` gehen in die Tabelle USERTABLE, also auch die Adressfelder.

Externes Attribut (OneToOne)

Manchmal will man eine 1:1 Beziehung in einer eigenen Tabelle speichern. Beispiel: ein Benutzer hat immer eine (Rechnungs-)Adresse und die soll in extra Tabelle ADRESSEN gehalten werden, z.B: weil man die Adresse auch noch von anderen Tabellen referenzieren will (shared reference). Für diesen Fall existiert die `@OneToOne Annotation`, die aber leider etwas schwer verdaulich ist. Zunächst stellt sich die Frage nach der ID der Entität `Address`: normalerweise hat die Adresse die gleiche ID wie der Benutzer. Hibernate nennt dies einen Shared Primary Key. Wenn Hibernate nun ein Adressobjekt speichern will, muss die ID befüllt werden. Hierzu existiert ein extra ForeignKey-Generator:

```
@Id
 @GeneratedValue (generator = "fkUserGenerator")
 @org.hibernate.annotations.GenericGenerator(name="fkUserGenerator",
strategy = "foreign", parameters={
     @Parameter(name="property", value="user")
     })
 @Column (name = "userId")
 private long userId;

 ... // kommt gleich ...
 private User user;
```

Der PrimaryKey der Tabelle ADDRESS ist nämlich ein Fremdschlüssel (ForeignKey) auf die Tabelle USERTABLE.

Nun muss noch die Beziehung zwischen den beiden Entitäten formuliert werden. Aus Sicht des `User` besteht eine 1:1 Beziehung zu `Address` (genauer: 1:C, denn die Adresse könnte auch fehlen). Wie ist es aber aus Sicht von `Address` zu `User`? Eigentlich müsse die Adresse ja nichts von dem User wissen, aber der ForeignKeyGenerator benötigt nun mal den User, um sich dessen ID zu holen. Also (und nur deswegen) modellieren wir auch die 1:1-Beziehung von `Address` nach `User`.

Hier nun die beiden Klassen :

```java
@Entity
@Table (name = "UserTable") // USER ist
reserviertes Wort in ORACLE
public class User {
 @Id
 @GeneratedValue(strategy=GenerationType.AUTO)
 private long id;

 private String userName;

 private String password;

 private String email;

 @OneToOne (cascade = CascadeType.ALL)
 @PrimaryKeyJoinColumn (name = "userId")
 private Address address;
...
```

```java
@Entity
public class Address {

 @Id
 @GeneratedValue (generator = "fkUserGenerator")
 @org.hibernate.annotations.GenericGenerator(name=
"fkUserGenerator", strategy = "foreign", parameters={
     @Parameter(name="property", value="user")
     })
 @Column (name = "userId")
 private long userId;

 @OneToOne (targetEntity = User.class)
 @JoinColumn (name = "userId")
 private User user;
 ...
```

Zwei Dinge müssen erwähnt werden:

1. die Cascade-Einstellung ist nötig, damit beim Speichern des Benutzers auch die Adresse gespeichert wird.
2. die @JoinColumn-Annotation ist nötig, weil sonst Hibernate den Namen der Join-Column rät: USER_ID und sich das so bemerkbar macht:
   `ORA-00904: "USER_ID": ungültiger Bezeichner`

Wenn nun ein User erzeugt wird, muss die bidirektionale Abhängigkeit zwischen den Objekten beachtet werden:

```java
  User u1 = new User();
   u1.setUserName("Hans im Glück");
   u1.setEmail("hans@glueck.de");

   Address a2 = new Address();
   a2.setCountry("Germany");
   ...
   a2.setUser(u1);
   u1.setAdresses(a2);

   s.save(u1);
```

Referenz (ManyToOne)

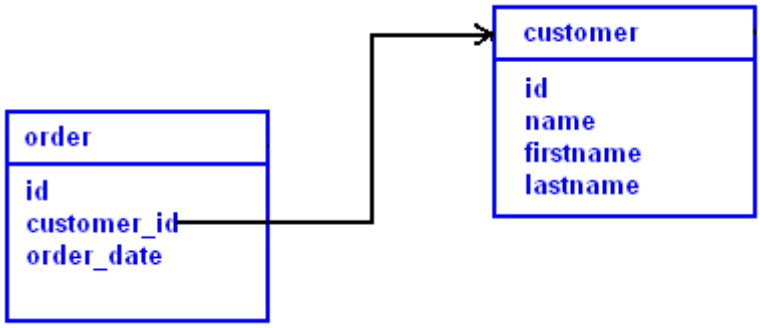
Ein Objekt (Order) hat eine Referenz auf ein anderes Objekt (Customer), genauer, die Enität Order referenziert die Entität Customer. In Hibernate kann dies mit @ManyToOne abgebildet werden: Mehrere Aufträge können zu einem Customer gehören.

```java
public class Order {
 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @ManyToOne (targetEntity = Customer.class, fetch = FetchType.LAZY)
 @JoinColumn (name = "customer_id")
 private Customer customer;

 // ...
}
```

Wer es lieber mit XML mappt:

```
<many-to-one name="customer" column="customer_id"  fetch="join" lazy="true"
cascade="none"/>
```

Abbildung in der Datenbank



Unidirektional

Die gezeigte Assoziation ist unidirektional und zwar von Order nach Customer. Es wäre hier nicht sinnvoll, die Rückwärtsrichtung zu modellieren.

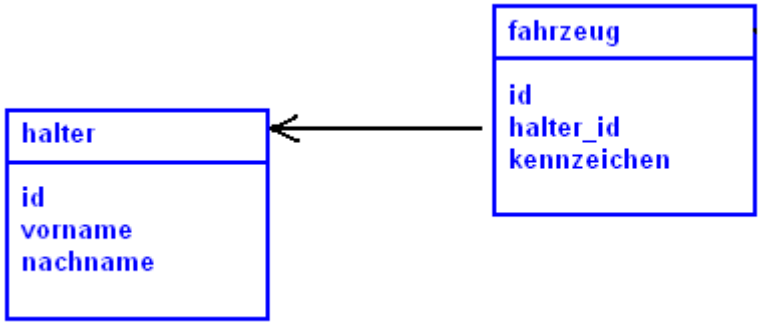
Assoziation (OneToMany)

Ein typischer Fall für eine Assoziation sind ein Halter und seine Fahrzeuge. Ein Halter kann mehrere Fahrzeuge halten, aber jedes Fahrzeug habt genau einen Halter. Im Klassenmodell wird diese Assoziation als Listentyp (List, Set, ...) modelliert. Welcher Listentyp verwendet wird, hängt zudem von der Semantik ab: kann ein Fahrzeug mehrfach in der Liste vorkommen (hier: nein), muss die Liste sortiert vorliegen, etc.

```
@Entity
public class Halter {
 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @OneToMany (targetEntity = Fahrzeug.class, cascade = CascadeType.ALL,
fetch = FetchType.LAZY)
 @JoinColumn (name = "halter_id")
 private Set<Fahrzeug> fahrzeuge;
}

@Entity
public class Fahrzeug {
 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @Column (name = "halter_id")
 private long halter;

 @Column (name = "kennzeichen")
 private String kennzeichen;
}
```

Die Beziehung wird durch `@OneToMany` ausgedrückt, der CascadeType zeigt an, dass die Fahrzeuge in der Liste automatisch mit angelegt, geändert oder gelöscht werden, wenn eine dieser Operationen auf dem Halter erfolgen.

Abbildung in der Datenbank



Unidirektional / Bidirektional

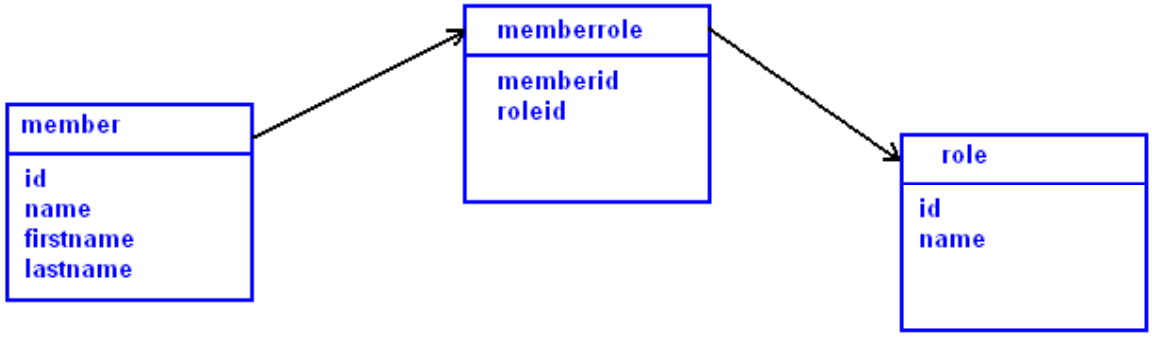
Diese Assoziation ist unidirektional, d.h. von Halter nach Fahrzeug. Wir könnten diskutieren, ob auch die Rückrichtung von Fahrzeug nach Halter relevant sein könnte, z.B. bei einer Abfrage nach Kennzeichen wollen wir die Daten des Halters anzeigen. In diesem Fall müssten wir die Assoziation als bidirektional mappen:

```
@Entity
public class Halter {
 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @OneToMany (targetEntity = Fahrzeug.class, mappedBy = "halter", cascade =
CascadeType.ALL, fetch = FetchType.LAZY)
 private Set<Fahrzeug> fahrzeuge;
}

@Entity
public class Fahrzeug {
 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @ManyToOne (targetEntity = Halter.class, fetch = FetchType.LAZY)
 @JoinColumn (name = "halter_id")
 private Halter halter;

 @Column (name = "kennzeichen")
 private String kennzeichen;
}
```

Im obigen Beispiel ist der Rückwärtsverweis an zwei Stellen ersichtlich: einerseits bei Fahrzeug durch die `@ManyToOne Annotation` und andererseits bei Halter durch das `mappedBy` Attribut (nun weis Hibernate den Namen der Spalte `halter_id` durch Befragung der `@Column` Annotation bei halter.

Relation (ManyToMany)

Ein Beispiel für eine Relation ist die Beziehung zwischen einem Benutzer (Member) und dem ihm zugeordneten Rollen:



Betrachten wir einen sehr gebräuchlichen Fall: ein Benutzer (Member) hat eine oder mehrere Rollen zugeordnet. Diese Zuordnung in eine ManyToMany-Assoziation, in der Datenbank wird sie über die Relationentabelle memberrole abgebildet. Im ER-Jargon wird diese Beziehung als n:m-Beziehung bezeichnet. In einfacheren Fall ist die Assoziation unidirektional, also es interessiert uns nicht, welche Member zu einer bestimmten Rolle gehören – uns interessiert nur, welche Rollen hat ein Member.

Mapping mit Annotations

```
@ManyToMany (targetEntity = Role.class, cascade = CascadeType.ALL)
@JoinTable (name = "memberrole")
private Set<Role> roles = new HashSet<Role>();
```

Auszug aus dem Log

```
create table Member (id number(19,0) not null, name varchar2(255), primary key (id))
create table Role (id number(19,0) not null, name varchar2(255), primary key (id))
create table memberrole (Member_id number(19,0) not null, roles_id number(19,0) not null,
primary key (Member_id, roles_id))
alter table memberrole add constraint FKB01D2E10AE97CCF5 foreign key (roles_id) references
Role
alter table memberrole add constraint FKB01D2E1020E6519C foreign key (Member_id) references
Member
```

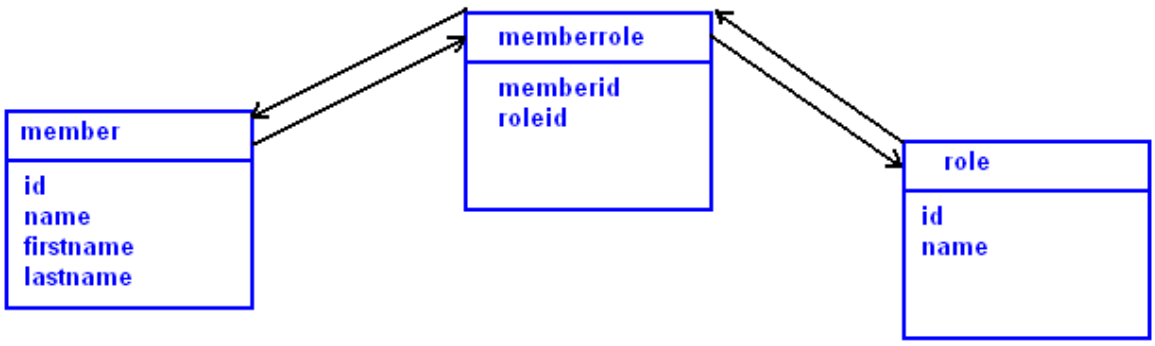
Die Namen der Spalten in `memberrole` wurden generiert, falls diese anders lauten, können Sie mit `@JoinColumns` auch explizit angegeben werden.

```
@ManyToMany (targetEntity = Role.class, cascade = CascadeType.ALL)
 @JoinTable (name = "member2role",
     joinColumns = { @JoinColumn (name = "memberid") },
     inverseJoinColumns = { @JoinColumn (name = "roleid") })
 private Set<Role> roles = new HashSet<Role>();
```

Mapping mit XML

```
<set name="roles" table="memberrole" lazy="true" cascade="save-update">
    <key column="memberid" />
    <many-to-many column="roleid" class="Role" />
  </set>
```

Unidirektional/bidirektional



Stellen wir uns eine Anwendung vor, welche die bestehenden Rollen auflistet und für jede Rolle die Liste der Member, die diese Rolle innehaben. In diesem Fall brauchen wir nun die Gegenrichtung, wir haben also eine bidirektionale Assoziation.

Relation mit Attributen

Sie haben es längst erkannt, die Tabelle memberrole ist eine Relationentabelle, welche die beide Foreign Keys für member und role enthält. Was aber, wenn diese Relationentabelle weitere Attribute (Properties) enthält, z.B. das Datum, wann diese Zuordnung erfolgte oder wer diese angelegt hat? In diesem Fall bietet es sich an eine zweistufige ManyToOne Beziehung zu mappen (symbolisch):

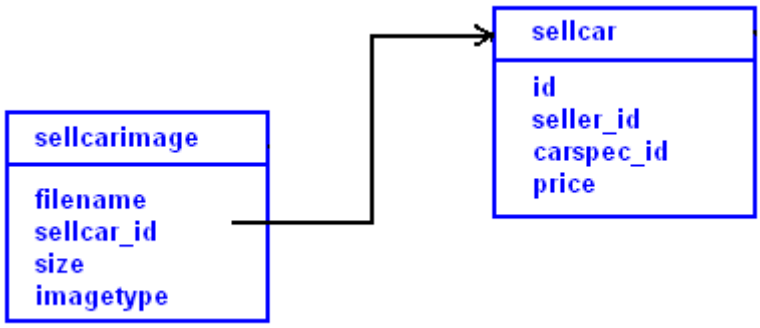
`Member` `<--> ManyToOne <-->` **`MemberRole`** `<--> OneToMany <-->` **`Role`**

Listen (Collections)

Stellen wir uns einen Online-Automarkt vor, in dem Anwender Ihre Autos zum Verkauf feil bieten können. Zu jedem Auto können ein bis zehn Fotos eingestellt werden. Diese Fotos (Images) gehören untrennbar zu dem eingestellten Auto, sind also sicher keine eigenen Entitäten, Ihr LifeCycle ist identisch mit dem der Entität Auto:

```
public class SellCar {

 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @ManyToOne (targetEntity = CarSpecification.class)
 @JoinColumn (name = "carspec_id")
 private CarSpecification spec;

 @org.hibernate.annotations.CollectionOfElements
 @JoinTable(name = "sellcarimages"
 joinColumns = @JoinColumn(name = "sellcar_id"))
 private Collection<Image> images;
}
```

Abbildung in der Datenbank



Es fällt auf, dass die Tabelle sellcarimage keine eine Id hat. Das ist prinzipiell kein Problem, allerdings muss Hibernate bei Änderungen in der Collection alle Elemente löschen und neu anlegen, da eine tupelgenaue Positionierung ohne Id nicht möglich ist. Meist kann man sich diesen Overhaed leisten, falls nicht, muss die Id in sellcarimage Hibernate bekannt gemacht werden:

```
@org.hibernate.annotations.CollectionId(columns = @Column(name = "id",
nullable = false), type = @org.hibernate.annotations.Type(type = "long"),
generator = "hibernate-increment")
 @org.hibernate.annotations.GenericGenerator(name = "hibernate-increment",
strategy = "increment")
```

Zu beachten ist dabei, dass nicht alle IdGenerator Varianten funktionieren, IDENTITY wird hier nicht unterstützt. Der IdGenerator increment ist ja eigentlich Teil des Hibernate-Core, jedoch nicht von JPA, deshalb muss man bei Verwendung von Annotations diesen IdGenerator explizit bekannt geben. Vorsicht: increment ist nicht cluster-fähig!

Hibernate unterstützt Listen von Wertetypen (wie String) und Listen von Komponenten. Bei Listen von Wertetypen werden Queries nur eingeschränkt unterstützt:

```
@Entity
public class Image {

 @Id
 @GeneratedValue (strategy = GenerationType.AUTO)
 private long id;

 @org.hibernate.annotations.CollectionOfElements
```

```
 @JoinTable(name = "imagekeywords",
 joinColumns = @JoinColumn(name = "image_id"))
 private Collection<String> keywords;

 private String fileName;

}
```

Sollen z.B. Images mit bestimmten Keywords geladen werden, so muss die Query auf sqlRestriction ausweichen, da die einzelnen Keywords nicht direkt referenziert werden können:

```
List<Image> found = s.createCriteria(Image.class, "i")
     .add(Restrictions.sqlRestriction("id IN (SELECT image_id FROM
imagekeywords WHERE element = 'Toll')"))
     .list();
```

Alternativ könnten die Keywords statt als String auch als Component (Klasse Keyword, enthält genau einen String) modelliert werden.

Persistence by Reachability

Ein Objektgraph lässt sich in JAVA schnell aufbauen, doch wie wird dieser Graph in die Datenbank persistiert? Hibernate beschreibt das Verfahren zur Persistierung Persistence by Reachability (PbR), was heißt, dass in einem Graphen alle referenzierten Instanzen mit der Datenbank synchronisiert werden (können). Sehen wir uns folgendes Beispiel an:

```
  Session s = sessionFactory.openSession();
  Transaction tx = s.beginTransaction();

  Halter halter = new Halter();
  Set<Fahrzeug> fahrzeuge = new HashSet<Fahrzeug>();
  Fahrzeug f1 = new Fahrzeug();
  f1.setKennzeichen("HH-XY 1234");
  f1.setHalter(halter);
  fahrzeuge.add(f1);
  Fahrzeug f2 = new Fahrzeug();
  f2.setKennzeichen("HH-ZZ 6789");
  f2.setHalter(halter);
  fahrzeuge.add(f2);
  halter.setFahrzeuge(fahrzeuge);
  s.save(halter);

  tx.commit();
  s.close();
```

Es wird ein Graph aus einem Halter und zwei Fahrzeugen instantiiert. Alle Instanzen sind transitiv, d.h. noch nicht mit der Datenbank (-Session) verbunden. Wird der Halter gespeichert, sollten möglichst auch die Fahrzeuge in der Datenbank persistiert werden. PbR bedeutet, dass der Graph traversiert wird und alle transitiven Instanzen persistiert werden. Damit dies funktioniert muss beim Mapping allerdings der CascadeType entsprechend angegeben werden (im einfachsten Falle ALL). Hier die Debug-Ausgabe der obigen Aktion:

```
11:29:26 DEBUG insert into Halter (id) values (?)
11:29:26 DEBUG insert into Fahrzeug (halter_id, kennzeichen, id) values (?, ?, ?)
11:29:26 DEBUG insert into Fahrzeug (halter_id, kennzeichen, id) values (?, ?, ?)
```

Generelle Hinweise

- Cascade: oftmals ist CascadeType.ALL eine sinnvolle Modellierung. In jedem Fall sollte der CascadeType **immer** explizit angegeben werden, damit man die Absicht des Programmierers sofort erkennen kann

Hinweise zu Annotations

In den obigen Beispielen wurde bereits Annotations eingesetzt. Diese Technologie ist einfach und intuitiv, trotzdem sollten die folgenden Punkte beachtet werden:

- Hibernate-Annotations immer voll qualifizieren (mit Package, also z.B. `@org.hibernate.annotations.Cache`). Für Attribute gilt das gleiche.
- Lazy-Attribut (fetch) immer explizit angeben, immer `FetchType.LAZY` verwenden (`setFetchMode()` in DAO explizit angeben). Dadurch werden Unklarheiten vermieden, was in welcher Klasse als lazy gemappt wurde und was nicht.
- Annotations an Fields schreiben, dadurch Vermeidung unnötiger Methoden wie setId().
- @Column Annotation an jedes Field.
- EntityClass bei Assoziationen immer angeben, schafft Klarheit, vor allem bei Verwendung von Interfaces.

Second Level Cache

Die Konfiguration erfolgt in hibernate.properties oder direkt in der Anwendung:

```
cfg.setProperty(Environment.CACHE_PROVIDER,
"org.hibernate.cache.EhCacheProvider")
```

Wenn EH-Cache verwendet wird, muss dieser ebenfalls konfiguriert werden:

```
<ehcache>
  <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="false"
        />

 <cache name="artikel.Member"
 maxElementsInMemory="100"
 eternal="true"
```

```
 timeToIdleSeconds="0"
 timeToLiveSeconds="0"
 overflowToDisk="false"
  />

  <cache name="artikel.Role"
 maxElementsInMemory="100"
 eternal="true"
 timeToIdleSeconds="0"
 timeToLiveSeconds="0"
 overflowToDisk="false"
  />

</ehcache>
```

Welche Klassen in den SLC gehen, muss pro Klasse definiert werden, z.B. mit Hilfe von Annotations:

```
@Entity
@org.hibernate.annotations.Cache ( usage =
org.hibernate.annotations.CacheConcurrencyStrategy.READ_ONLY )
public class Member {

. . .

 @ManyToMany (targetEntity = Role.class, cascade = CascadeType.ALL)
 @JoinTable (name = "member2role",
     joinColumns = { @JoinColumn (name = "memberid") },
     inverseJoinColumns = { @JoinColumn (name = "roleid") })
 @org.hibernate.annotations.Cache (usage =
org.hibernate.annotations.CacheConcurrencyStrategy.READ_ONLY)
  private Set<Role> roles = new HashSet<Role>();
  . . .
}

@Entity
@org.hibernate.annotations.Cache ( usage =
org.hibernate.annotations.CacheConcurrencyStrategy.READ_ONLY )
public class Role {
  ...
}
```

Query Cache

Der Query Cache puffert die Ergebnisse einer Query, genauer gesagt, die IDs der geladenen Entities (die Entities selbst liegen dann im Entity Cache, sofern die Klasse überhaupt für den SLC gekennzeichnet wurde).

Generelle Aktivierung in hibernate.properties:

```
hibernate.cache.use_query_cache = true
```

Damit eine Query auch in den Query Cache kommt, muss `setCacheable()` gesetzt werden:

```
s.createCriteria(Member.class).setCacheable(true).list()
```

Statistiken

Zur Analyse der Effektivität des SLC und des Query Cache können Statistiken erstellt werden:

```
hibernate.generate_statistics = true
hibernate.cache.use_structured_entries = true

Statistics stats = sessionFactory.getStatistics();
```

Criteria Queries

Folgende Criteria Query selektiert Items mit Geboten höher als 1.0:

```
List <AuctionItem> l = s.createCriteria(AuctionItem.class)
      .createCriteria("bids")
      .add(Restrictions.gt("amount", new Float(1.0)))
      .list();

    for (AuctionItem item : l) {
     System.out.println("ITEM-2: " + item.getId() + ", " +
item.getDescription());
     for (Object o : item.getBids()) {
      Bid bid = (Bid) o;
      System.out.println("BID-2:" + bid.getAmount() + " - " +
bid.getBidder().getUserName());
     }
    }
```

Es handelt sich um ein Nested Criteria, weil auf dem Criteria für AuctionItem mit createCriteria() ein inneres Critiera für die Assoziation „bids“ erstellt wurde.

Folgende Daten werden ausgegeben:

```
Hibernate: select this_.id …
ITEM-2: 8, the auction item number 2
Hibernate: select bids0_.item …
BID-2:0.0 - 1E1
Hibernate: select user0_.id as id2_0_, …
BID-2:0.5 - steve
BID-2:1.0 - 1E1
BID-2:1.2 - steve
BID-2:1.5 - steve
ITEM-2: 8, the auction item number 2
BID-2:0.0 - 1E1
BID-2:0.5 - steve
BID-2:1.0 - 1E1
BID-2:1.2 - steve
BID-2:1.5 - steve
```

Zwei Dinge fallen auf: Die Liste enthält zweimal das Item mit Id==8 und die Bids dieses Items sind alle Bids und nicht nur die, welche größer 1.0 sind (die Ausgabe der SQL-Statements habe ich gekürzt).

Für jemanden, der es gewohnt ist, mit SQL zu arbeiten, ist das ungewohnt: folgendes (old-style) SQL würde alle Items mit den zugehörigen Bids mit Geboten höher als 1.0 liefern:

```
select i.id, b.id, b.amount from AuctionItem i, Bid b where b.item = i.id AND
b.amount > 1.0
```

Ergebnis:

```
i.id  b.id  b.amount
8 13 1.2
8 12 1.5
```

D.h., dass die obige Criteria Query sich so verhält, als hätten wir in SQL folgendes programmiert (den Join auf User habe ich hier weggelassen):

```
select * from AuctionItem i, Bid b where b.item = i.id AND i.id IN (SELECT x.item
FROM Bid x WHERE x.amount > 1.0)
```

Also sehen wir uns jetzt doch mal die SQL-Statements genauer an, welche Hibernate freundlicherweise mitprotokolliert hat (`<property name="show_sql" >true </property>`):

```
Hibernate: select this_.id as id0_1_, this_.seller as seller0_1_,
this_.shortDescription as shortDes3_0_1_, this_.description as descript4_0_1_,
this_.ends as ends0_1_, this_."CONDITION" as CONDITION6_0_1_, this_.successfulBid
as successf7_0_1_, bid1_.id as id1_0_, bid1_.item as item1_0_, bid1_.amount as
amount1_0_, bid1_."datetime" as datetime5_1_0_, bid1_.bidder as bidder1_0_,
bid1_.isBuyNow as isBuyNow1_0_ from AuctionItem this_, Bid bid1_ where
this_.id=bid1_.item and bid1_.amount>?
ITEM-2: 8, the auction item number 2
Hibernate: select bids0_.item as item1_, bids0_.id as id1_, bids0_.id as id1_0_,
bids0_.item as item1_0_, bids0_.amount as amount1_0_, bids0_."datetime" as
datetime5_1_0_, bids0_.bidder as bidder1_0_, bids0_.isBuyNow as isBuyNow1_0_ from
Bid bids0_ where bids0_.item=?
Hibernate: select user0_.id as id2_0_, user0_.userName as userName2_0_,
user0_."password" as password3_2_0_, user0_.email as email2_0_, user0_.firstName as
firstName2_0_, user0_."initial" as initial6_2_0_, user0_.lastName as lastName2_0_
from AuctionUser user0_ where user0_.id=?
BID-2:0.0 - 1E1
Hibernate: select user0_.id as id2_0_, user0_.userName as userName2_0_,
user0_."password" as password3_2_0_, user0_.email as email2_0_, user0_.firstName as
firstName2_0_, user0_."initial" as initial6_2_0_, user0_.lastName as lastName2_0_
from AuctionUser user0_ where user0_.id=?
BID-2:0.5 - steve
BID-2:1.0 - 1E1
BID-2:1.2 - steve
BID-2:1.5 - steve
ITEM-2: 8, the auction item number 2
BID-2:0.0 - 1E1
BID-2:0.5 - steve
BID-2:1.0 - 1E1
BID-2:1.2 - steve
BID-2:1.5 - steve
```

1

2

Aha! Hibernate liest also wirklich zuerst mit einer SQL-Query nur die Bids (1), welche Amount > 1.0 haben und erhält zwei Tupel als Ergebnis zurück (siehe mein „old-style“ SQL oben). Da bei der Ausgabe der Bids ein Zugriff auf getBids() erfolgt, lädt Hibernate alle(!!) Bids zu dem Item nach(2). Anschließend erfolgt noch ein Lazy Load für die User (grau), aber das hatten wir ja erwartet (warum nur zweimal? Session-Cache!).

Übrigens, die doppelten Einträge in der Liste lassen sich über ein Set leicht wieder eliminieren:

```
Set<AuctionItem> set = new HashSet<AuctionItem>();
    set.addAll(l);
```

Warum ist das so? Naja, weil das Set mit equals() prüft, ob ein gleiches Element bereits enthalten ist. Stimmt, wir erinnern uns, die equals()-Methode, da war doch was: Hibernate erfordert, dass jedes Entity-Objekt die equals-Methode überschreibt.

Equals() und hashCode()

Wir haben gelernt, dass Hibernate die einmal in einer Session gelesenen Objekte puffert (1rst Level Cache). Um nun eindeutig zu erkennen, dass ein Objekt wie User „Steve“ bereit im Cache liegt, werden die equals() bzw. hashCode() Methode der Entity-Klasse verwendet. Zu diesem Thema gab und gibt es viele Diskussionen und reichlich Halbwissen. In [2, S396] wird mit dem Halbwissen aufgeräumt und eine halbwegs saubere Beschreibung geliefert.

In wenigen Worten ausgedrückt: es ist notwendig, für alle PO’s die beiden Methoden zu überschreiben und zwar spezifisch pro Klasse, indem ein sog. Business Key als Vergleichsbasis verwendet wird. Dieser Business Key ist (mehr oder weniger) das, was man als Datenbanker als UNIQUE Key vergeben würde.

Die oftmals genannte Methode mit dem Vergleich der Id führt wie in [2] beschrieben zu dem Problem, dass Hibernate den Key nach dem Insert ändert (von null auf X) und ein Set, welches das Objekt enthält dann aus dem Tritt kommt. Daher wird von der Verwendung der Id als Basis für equals/hashCode auch dringend abgeraten. Eine korrekte Implementierung für User wäre z.B. diese:

```
@Override
 public boolean equals(Object other) {
  if (this==other) {
   return true;
  }
  if (!(other instanceof User)) {
   return false;
  }
  final User otherUser = (User) other;
  return userName.equals(otherUser.getUserName());
 }

 @Override
 public int hashCode() {
  return userName.hashCode();
 }
```

In obiger Implementierung wird davon ausgegangen, dass userName initialisiert ist und nie null sein kann, ansonsten müsste NULL-Check erfolgen. Hier auch ein Kritikpunkt am Hibernate Team: die mit Hibernate ausgelieferten Beispielsourcen enthalten eben **keine** Implementierung für equals/hashCode!

Bleibt noch die Frage, warum funktioniert der Session-Cache dann überhaupt, wenn die User-Klasse die Methoden nicht überschreibt?

Vermeindung kartesisches Produkt

Wenn mehrere Collections zu einer Query gejoint werden, entsteht zwangsläufig ein kartesisches Produkt. Um das in der resultierenden Liste zu vermeiden, dann ein spezieller ResultTransformer dafür eingesetzt werden:

```
DetachedCriteria crit = DetachedCriteria.forClass(Order.class);
crit.add(Restrictions.eq("_id", id));
crit.setFetchMode("_positions", FetchMode.JOIN);
crit.setFetchMode("_createdBy", FetchMode.JOIN);
crit.setResultTransformer(CriteriaSpecification.DISTINCT_ROOT_ENTITY);
```

DeleteOrphan

Oftmals soll sichergestellt werden, dass mit einem Datensatz auch abhänge Datensätze gelöscht werden, z.B. Wenn ein Auftrag gelöscht wird, sollen auch die Auftragspositionen gelöscht werden.

```
/** Positionsliste. */

@OneToMany(targetEntity = Layer.class, cascade = CascadeType.ALL)
@org.hibernate.annotations.Cascade (
org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
@JoinColumn(name = "orderid", nullable = false)
private List<Position> _positions;
```

Mit DELETE_ORPHAN kann dies sichergestellt werden. Es ist zu beachten, dass diese Annotation Hibernate-spezifisch ist. Ob diese Relation als besser OneToMany oder ManyToMany abgebildet wird, hängt vom Einsatzzweck ab.

Thomas Kestler ist Geschäftsführer der elevato GmbH. http://www.elevato.de