Hibernate in großen Web-Projekten

Thomas Kestler, elevato GmbH

Einleitung

Hibernate hat sich längst als Persistenz-Framework für JAVA-Projekte unterschiedlicher Größen durchgesetzt. Auch in Web-Projekten kommt Hibernate immer öfter zum Einsatz. Hibernate ist sehr leistungsfähig, aber leider auch sehr komplex. Der folgende Beitrag soll eine Hilfestellung zum Einsatz von Hibernate geben, Speziell in Hinblick auf Web-Projekte, die nochmals besondere Aspekte mitbringen.

Hibernate wird oft unterschätzt, weil man sehr schnell eine Hello-World Anwendung damit bauen kann. Für reale Projekte hält Hibernate aber mannigfaltige Überraschungen bereit, die durchaus den Projekterfolg beeinträchtigen können. Gerade, wenn alle Entwickler gleichermaßen mit Hibernate arbeiten sollen („das ist OpenSource, das kapiert doch jeder“), sind Probleme vorprogrammiert. Der nachfolgende Beitrag soll auf mögliche Probleme aufmerksam machen und Lösungswege aufzeigen. Die Ausführungen basieren auf Hibernate 3.2. Eine Aktualisierung des Dokuments auf die neueste Hibernate-Version ist geplant.

Der Einstieg

Der übliche Einstieg über die Online-Dokumentation (www.hibernate.org) ist steinig. Die Beschreibungen sind oft zu knapp und die Beispiele nicht gerade praxisrelevant. Somit liegt der Kauf eines Buches nahe, inzwischen gibt es reichlich Bücher auf dem Markt, auch deutschsprachige. Die aktuelle Referenz ist „Java Persistence with Hibernate“ von Christian Bauer und Gavin King [1]. Auch die ältere Ausgabe „Hibernate in Action“ der gleichen Autoren [2] kann noch gute Dienste leisten, basiert aber noch auf der Version 2 und behandelt JPA und Annotations nicht. Beiden Bücher, wie auch der Online-Doku ist gemeinsam, das sie didaktisch schlecht gemacht sind und etliche Fehler enthalten.

Bei der Durchsicht anderer Bücher zum Thema gab es ebenfalls Licht und Schatten, im Detail muss das jeder für sich beurteilen.

Besonderheiten von großen Web-Projekten

Mit Web-Projekten sind hier keine Mickey-Mouse-Anwendungen gemeint, sondern zentrale, kritische Anwendungen unter hoher Last. Für kleine Anwendungen sind viele der folgenden Aspekte vielleicht nicht relevant, wohl aber für Anwendungen mit hoher Last, hoher Anzahl Benutzer, Clustering, etc. Oftmals existieren viele Applikationsserver, aber nur ein Datenbankserver (evtl. Cluster). Somit können die Datenbankzugriffe schnell zum Flaschenhals werden und die Skalierbarkeit eines Systems begrenzen. Daher wird die frühzeitige Analyse der Datenbankzugriff hier zur unerlässlichen Pflicht, z.B. durch Protokollieren der SQL-Befehle (log4j: `org.hibernate.SQL`)

Web-Projekte bringen einige Besonderheiten mit:

- Lange „Transaktionsdauer“ von Anzeige der Maske bis zum Submit
- Trennung Controller/View (kein DB-Zugriff in View-Schicht)
- Schichtentrennung (Business Logik)
- (oftmals) Multi-Server Betrieb

Lange Transaktionen

Von der Anzeige der Eingabemaske bis zum Submit kann viel Zeit vergehen, evtl. erfolgt dieser gar nie (oder doppelt). Denken wir an das Ausfüllen einer Registrierungsseite oder eines mehrseitigen Wizards beim Online-Kauf (Eingabe Personendaten, Eingabe Zahlungsart, Eingabe Versandweg). Ein solcher Ablauf wird in Hibernate auch als Conversation bezeichnet. Hibernate definiert folgende Szenarien (Patterns):

- session-per-request
- session-per-request with detached objects
- Open-session-in-view
- session-per-conversation

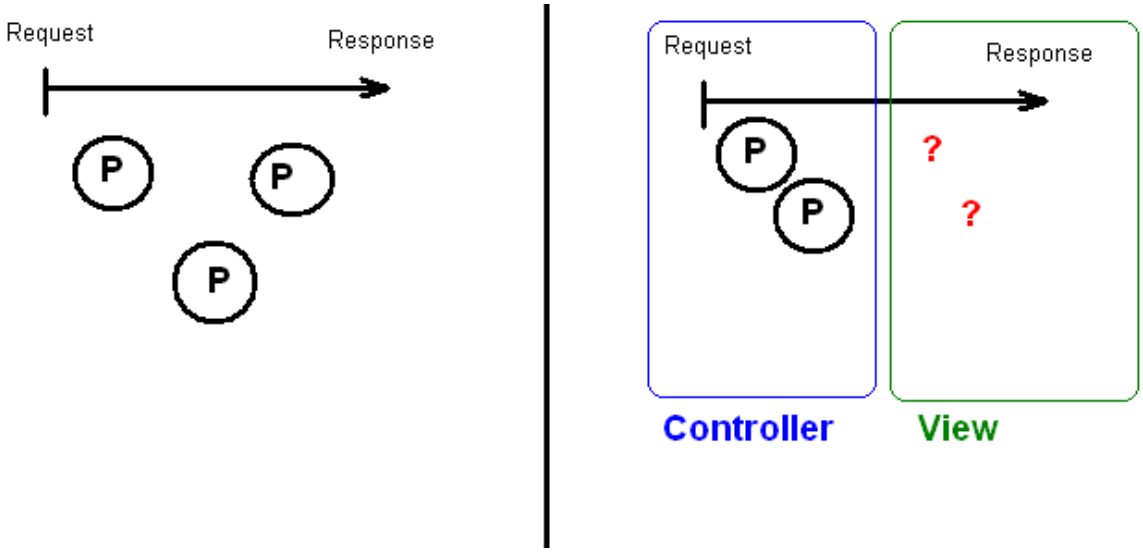
Im folgenden wollen wir uns die einzelnen Szenarien genauer ansehen.

session-per-request

Für jeden Request wird eine Session geöffnet, Daten gelesen oder verändert und dann die Session wieder geschlossen. Beispiel: Benutzer kann seine Stammdaten ändern, er hat die HTML-Seite ausgefüllt und klickt nun den Submit-Button:

```
Session s = factory.openSession();
Transaction tx=null;
try {
 tx = s.beginTransaction();
 User user = s.load(User.class, userId);
 user.setFirstname(request.getParameter("firstname");
 user.setLaststname(request.getParameter("lastname");
 user.setEmail(request.getParameter("email");
 . . .
 tx.commit();
}
catch (Exception e) {
 if (tx!=null) tx.rollback();
 throw e;
}
finally {
 s.close();
}
```

Hübsch, aber etwas kurz gedacht. Eine Web-Anwendung besteht aus Controller und View. Wie weit erstreckt sich der Persistence Context (pardon, die Session) denn nun?



Die Abbildung zeigt links eine Darstellung ähnlich wie in [1], ein bisschen idealisiert vielleicht, denn wo ist dort die Trennung von Controller und View? Das rechte Teilbild deutet das an. Was passiert mit den Objektinstanzen, die im Controller gelesen wurden und dann z.B. per Model an die View übergeben wurden? Um es kurz zu machen, im obigen Code-Beispiel würden die Persistenzobjekte *detached* werden, also ihren Bezug zum Persistence Context verlieren. Kann funktionieren, muss aber nicht. Ich habe oben absichtlich die Daten mit load() gelesen, da diese Methode (anders als get()) Proxies liefern kann und es beim Zugriff in der View zu einer LazyInitializationException kommen kann.

Eine Möglichkeit, das Problem zu umgehen, wäre es, im Controller alle Daten des User-Objektes in das Model zu kopieren und dann in der view nur noch auf das Model zuzugreifen:

```
model.setFirstname(user.firstname);
model.setLastname(user.lastname);
...
```

Eine andere Möglichkeit besteht darin, die Session bis in die View offen zu halten, so dass die View arglos auf `model.getUser().getFirstName()` zugreifen kann. Das birgt aber wiederum die Gefahr, dass die View auch Veränderungen am Userobjekt vornimmt und das wollen wir nicht (haben wir eingangs gesagt). Außerdem müssen wir sicherstellen, dass die View auch am Ende die Session ordentlich schließt (wie war dass dann mit Fehlerbehandlung in der View?).

session-per-request with detached objects

Das bringt uns geradewegs zum zweiten Pattern. Unsere HTML-Seite zur Eingabe der Stammdaten des Benutzers war ja nicht vom Himmel gefallen. Über den Link

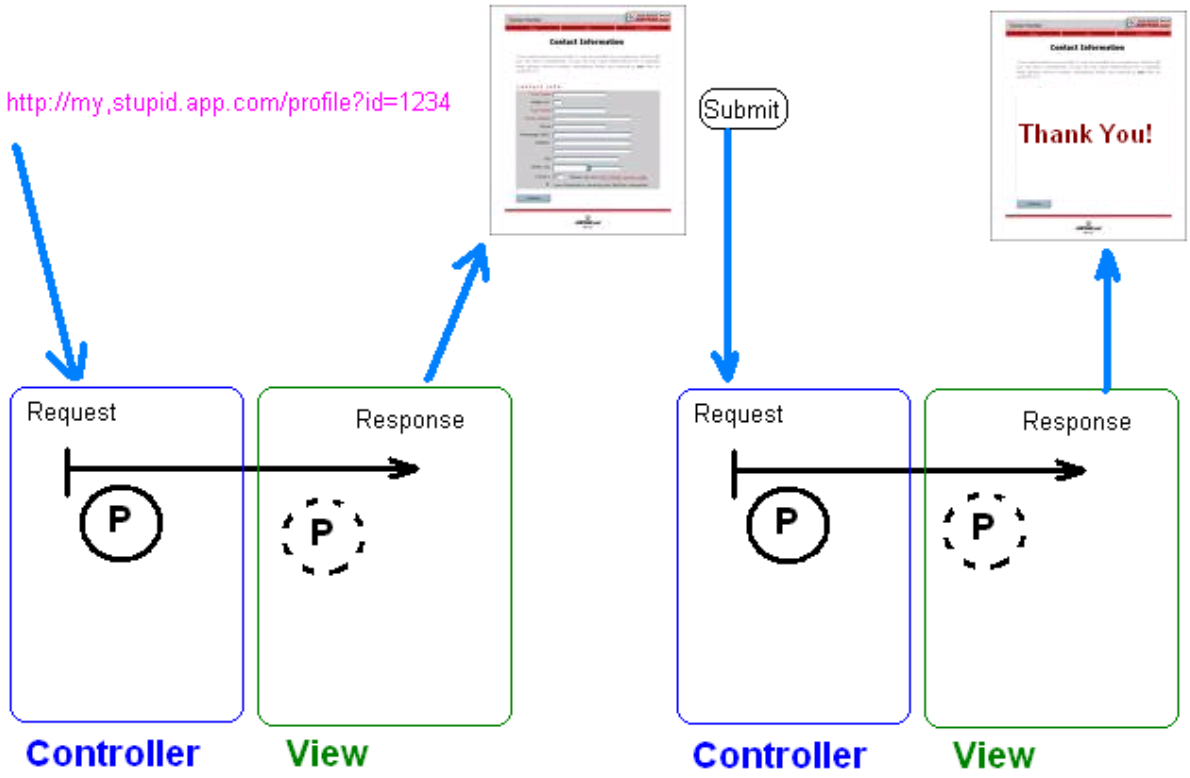
http://my.stupid.app.com/profile?id=1234

gelangt der Benutzer auf die Seite mit seinen Stammdaten (wir unterstellen, er ist bereits eingeloggt). Dieser (GET-)Request führt zum Lesen der Daten:

```
Session s = factory.openSession();
User user = null;
Transaction tx=null;
try {
 tx = s.beginTransaction();
 user = s.load(User.class, userId);
 tx.commit();
}
catch (Exception e) {
 if (tx!=null) tx.rollback();
 throw e;
}
finally {
 s.close();
}
model.setUser(user);
```

Ooops, der Controller übergibt das `User`-Objekt an das Model und damit an die View. Aber die Session ist schon geschlossen, also ist `user` detached. Detached heißt, ein Persistence Object (PO, jetzt kürze ich das ab) hat keinen Bezug mehr zu seinem Persitence Context (sagen wir ruhig Session dazu). Wird das PO verändert und soll dann gespeichert werden, muss es wieder an einen PC (richtig, Persistence Context) gebunden werden, man spricht vom merge oder re-attachment.

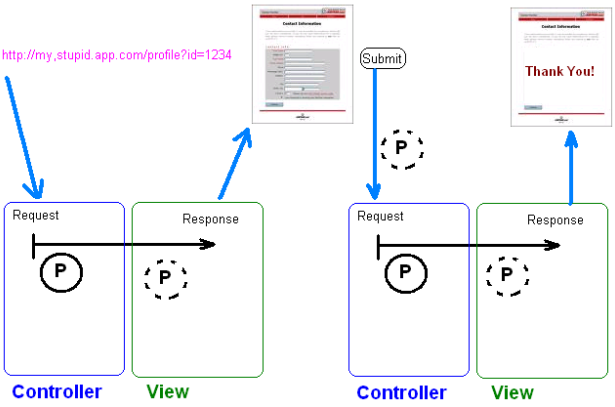
Sehen wir uns jetzt noch mal den gesamten Ablauf der Conversation im Schaubild an (Anwender klickt auf Link, bekommt seine Stammdaten, ändert diese, z.B. Anschrift und sendet Änderungen mit dem Submit-Button ab):



Der Übersichtlichkeit halber habe ich in diesem Bild nur ein PO, nämlich das `User`-Objekt dargestellt und den detached Zustand durch die gestrichelte Linie

angedeuted. Übrigens hätte die Conversation noch länger gehen können, z.B. wenn der Anwender ungültige oder unvollständige Angaben gemacht hätte.

Eigentlich ist das Bild nicht ganz korrekt, denn das `User`-Objekt wird beim Submit ja als Detached Object hereingereicht und im Controller (bzw in der DAO) re-attached:




```
@Override
 protected void doSubmitAction(Object
command) throws Exception {
  AuctionModel model = (AuctionModel)
command;

  User user = model.getUser();

user.setChangedDate(Calendar.getInstance());
 auctionDAO.update();
}
```

Im obigen Code-Beispiel erfolgt das Re-Attachement in `auctionDAO.update()`:

```
getHibernateTemplate().saveOrUpdate(user);
```

Für die Diskussion über die verschiedenen Methoden zum Re-.Attachement verweise ich auf [2, 9.3.2], in der Praxis hat sich `saveOrUpdate()` als sehr elegant erwiesen. Solange man immer sicher sein kann, in welchem Zustand sich ein detached Object befindet, kann man die Operationen auch auf `save()` und `update()` verteilen und gewinnt damit Verständlichkeit.

Open-session-in-view (OSIV)

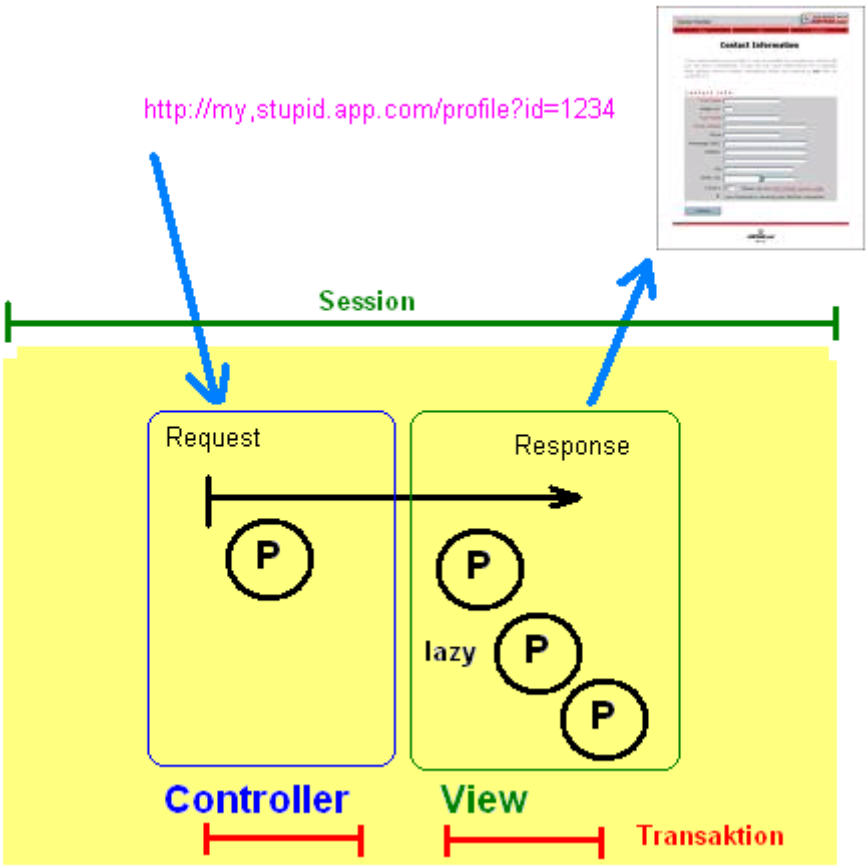
Dieses Pattern ist auf der Hibernate WebSite beschrieben [5] und ermöglicht es, die Session bis in die View auszudehnen. Weitere Beschreibungen in [2] und [4]. Der Vorteil liegt auf der Hand: während des Renderings können Daten auf einfache Art geladen oder (lazy) nachgeladen werden. Die Nachteile überwiegen jedoch:

- Session und damit Connection bleiben länger offen (Resource-Problem)
- Commit/Rollback in View
- Nachladen (lazy) kann zu ungünstigen SELECT-Befehlen führen
- Exception Handling bei Fehlern in View ist schwierig
- HttpStatus evtl. bereit ausgeliefert (flush)

Generell sollte die Logik bei MVC im Controller liegen und nicht in der View. Die View bekommt im Model alles was sie braucht und hat das zu rendern. In sehr komplexen Anwendungen kann das aber aufwändig werden und OSIV erhält dann seine Berechtigung, dennoch muss sichergestellt sein, dass in der View maximal lesende Zugriffe erfolgen

Um die Session bis in die View ausdehnen zu können, muss die Session bereits vor Aufruf des Controllers geholt und nach Ende der View zurückgegeben werden. Dies ist möglich mit Hilfe eines Servlet-Filters oder Interceptors (z.B. mit Hilfe von Spring).

Die DAO-Klassen müssen dann aber die für diesen Request gültige Session auch zugreifen können, hierzu stellt Hibernate mit `HibernateUtils.getCurrentSession()` bereits einen Mechanismus bereit. Transaktionen des Controllers sollten unbedingt bei Verlassen des Controllers abgeschlossen sein, damit nicht eine Exception in der View die Änderungen des Controllers verwerfen kann. Prinzipiell ist es möglich, die Session auch nach Schließen der Transanktion weiter zu nutzen (weil Hibernate eine neue JDBC-Connection holt, wenn die Session keine offene mehr hat), aber empfohlen wird dieses Verhalten nicht. Statt dessen sollte die View eine eigene (readonly) Transkation in der (gleichen) Session öffnen.



session-per-conversation

Wir fragen uns nun so langsam, warum wir nicht gleich die Session während der gesamten Conversation offen halten. Das hätte etliche Vorteile, so gäbe es keine detached Objects und wir könnten endlich mal vom First Level Cache der Session profitieren. Gerade bei komplexeren Konversationen wie einem Mehrstufigen Bestell-Wizard würde das erhebliche Vorteile bringen. Das haben sich viele Entwickler gedacht und JBoss hatte dieses Pattern in JBoss Seam ja auch massiv befördert. Nachdem Hibernate nun unter dem JBoss-Dach weiter entwickelt wird ist dieses Pattern mittlerweilen auch das von Hibernate propagierte Best-Practice-Pattern.

Wie könnten wir erreichen, dass die Session während der gesamten Conversation geöffnet bleibt? Eine Möglichkeit besteht darin, die Hibernate-Session als Attribut an die http-Session zu hängen. Der Controller würde dann einfach prüfen, ob sie bereits da ist und verwenden oder zuerst neu erzeugen und anhängen.

Das führt uns aber zu einigen Problemen:

- Es kann sehr viele (http-)Sessions geben (zu viele JDBC-Connections)
- (http-)Sessions existieren sehr lange (Anwender geht Kaffee trinken), JDBC-Connections bleiben viel zu lange offen
- Bei Clusterbetrieb wechselt die (http-)Session evtl. den Rechner, kommt die Hibernate-Session da noch mit?
- Nach Auftreten eines Fehlers (Exception) muss die Session geschlossen und nicht weiterverwendet werden.
- Wie erkennt das Framework (Interceptor) das Ende einer Transaktion.

Hibernate hat hier im Vergleich zu früheren Versionen viel Aufwand betrieben um diese Probleme zu lösen. So werden die JDBC-Connections von der Sessoin abgekoppelt, wenn sie nicht mehr benötigt werden und können so zurück in den Connection-Pool gegeben werden.

Gerade die Anzahl der offenen JDBC-Connections würde sonst schnell zum Problem führen. Erst wenn man den FlushMode der Session auf `FlushMode.MANUAL` ändert, wird dieses Pattern machbar. In [2, 11.2.3] ist dies ausführlich beschrieben und Sie werden beim Durchlesen dort bemerken welche Abgründe sich auftun: ob Veränderung in der Datenbank bei tx.commit() ausgeführt werden, hängt u.a. davon ab, welche IdGenerator-Strategy gewählt wurde.

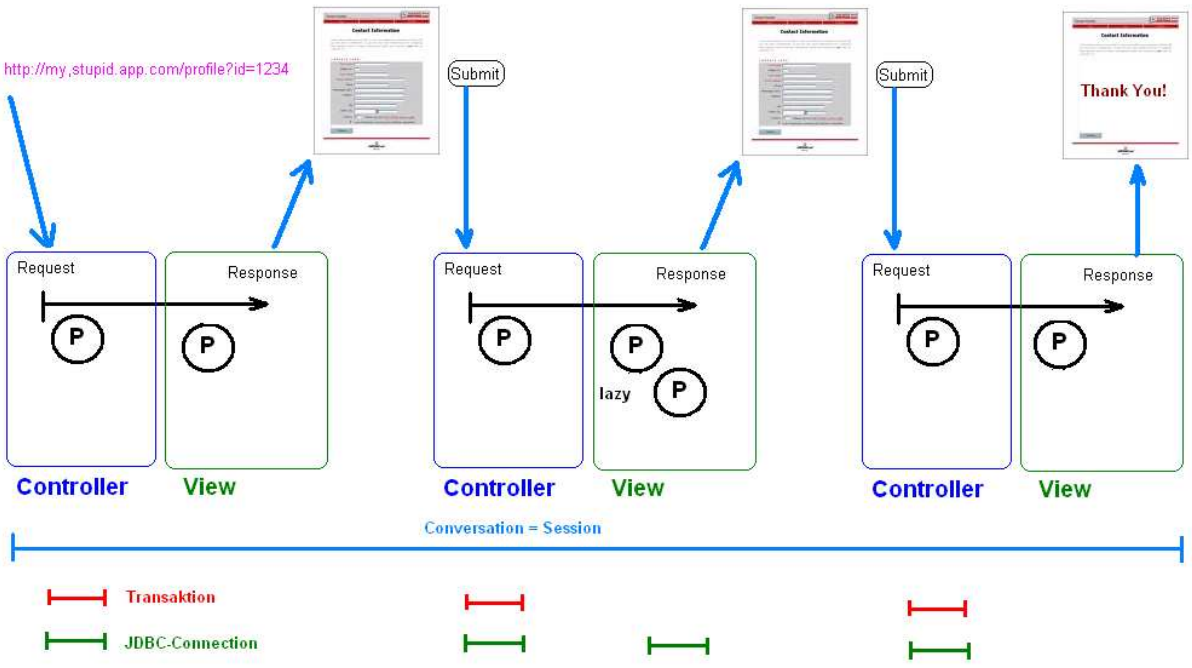
Die Grundlage ist die, dass die SQL-Befehle erst mit dem Flush der Session an die Datenbank übertragen werden. Wird der automatische Flush deaktiviert, erfolgt das erst, wenn `session.flush()` aufgerufen wird (wird die session mit `close()` geschlossen, ohne `flush()` passiert gar nix). Mit tx.commit() wird eine evtl. offene JDBC-Connection geschlossen und somit eine Resource Contention vermieden. Der Persistence Context bleibt hingegen offen, d.h. der First Level Cache der Session bleibt verfügbar. Erfolgt, auch nach einem tx.commit(), ein Zugriff auf ein Objekt, der ein Nachladen erfordert, so wird hierfür einen neue JDBC-Connection angefordert und die Daten nachgeladen (also keine LazyInitializationException).

```
Session s = sessionFactory.openSession();
  s.setFlushMode(FlushMode.MANUAL); //1
  Transaction tx = s.beginTransaction();

  Auction auction = new Auction();
  ...
  s.save(auction); // 2
  long id = auction.getId();
  tx.commit();     // 3

  tx = s.beginTransaction();
  Auction aa = (Auction) s.load(Auction.class, id);       // 4
  tx.commit();  // 5

  Auction aa2 = (Auction) s.load(Auction.class, id); //6

  s.flush();  //7

  Auction aa2 = (Auction) s.load(Auction.class, id); // 8

  for (Bid bid : aa2.getBids()) {
   System.out.println("Bid" + bid.getId());  // 9
```

```
}

s.close();  // 10
```

Zur Verdeutlichung soll das obige Trivial-Beispiel dienen: die Session bekommt den manuellen FlushMode verpasst (1), das Speichern (2) führt noch nicht zum INSERT in die Datenbank, auch der `commit()` (3) ändert daran nichts, es wird maximal die JDBC-Connection freigegeben. Der Persistence Context ist während der gesamten Session offen, also liest (4) nicht aus der Datenbank (sondern aus First Level Cache). Der `commit()` bei (5) macht nichts, da ja keine JDBC-Connection nötig war. Das Lesen der Auction kann auch ausserhalb der Transaktion erfolgen (6), wenn gleich das nicht empfohlen wird. Nun wird die Session geflusht (7), es erfolgt der INSERT in die Datenbank, die Session bleibt weiterhin offen. Auch das nächste Lesen (8) erfolgt aus dem Cache der Session. Wird nun auf die Lazy-Collection zugegriffen, muss nachgeladen werden (9). Dafür holt sich Hibernate wieder eine JDBC-Connection und lädt die Bids nach. Erst mit dem Schließen der Session endet der Persistence Context.

Ob sich dieses Pattern mit dem verwendeten Framework überhaupt nutzen lässt, hängt davon ab, ob man die SessionFactory beeinflussen kann (z.B in Spring eigene SessionFactory von `LocalSessionFactoryBean` ableiten). Weiterhin muss ein Interceptor-Mechanismus (oder Servlet-Filter) die Session über die gesamte Conversation offen halten. Die DAO-Methoden müssen nach DML-Operationen selbst `flush()` ausführen, da sonst die Änderungen erst verspätet oder gar nicht (wenn Interceptor keinen `flush()` vor dem `close()` ausführt) in die Datenbank propagiert werden. Der Interceptor muss aber genau wissen, wann eine Conversation beginnt und endet, damit er am Ende der Conversation den `flush()` ausführen kann.



Fazit

Am einfachsten fährt man wohl zunächst mit dem Pattern *session-per-request-with-detached-objects*. Solange man im Controller alle Daten für die View vorbereiten kann, so dass dort keine Daten nachgeladen werden müssen, hat man ein einfaches und robustes Pattern. Die Patterns *session-per-conversation* und *OSIV* sind sehr komplex und da müsste jemand schon sehr gute Argumente liefern, bevor ich mich dafür entscheiden würde. In komplexen Anwendungen, wo es im Controller nur sehr aufwändig möglich ist, alle nötigen Daten für die View komplett bereit zu stellen, würde dies Vorteile bringen, der Infrastrukturaufwand (Interceptor) und die Komplexität gehen auf der Kostenseite ein.

Software-Design

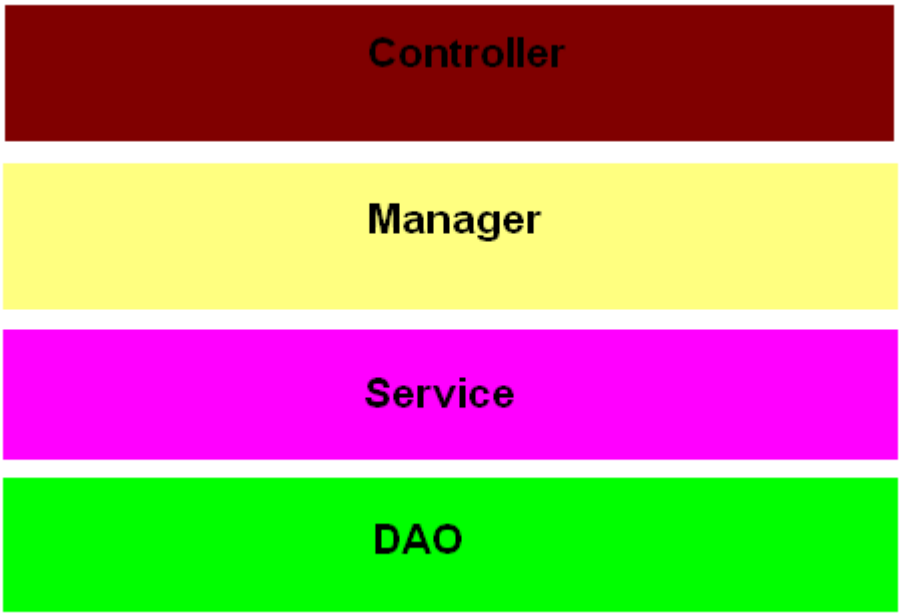
Hibernate wird oft als transparenter Persistenzmechanismus bezeichnet, also stellt sich die Frage, ob wir beim Software-Design überhaupt auf die Tatsache Rücksicht nehmen müssen, dass wir Hibernate einsetzen.

Trennung Controller/View

Viele Web-Anwendungen folgen dem MVC2-Prinzip und verbieten (zumindest dogmatisch) zusätzlich jegliche Datenbankzugriffe aus der View-Schicht (technisch gibt es dafür keinen Grund, aber sicher aus Gründen der Wartbarkeit). Alle DB-Zugriffe müssen also von den Controllern ausgehen.

Schichtentrennung

Eine Aufteilung nach dem SOA-Stack hat sich als sinnvoll und praktikabel erwiesen:



Der DAO-Layer ist der einzige, der direkt mit der Datenbank kommuniziert. Der darüber liegende Service-Layer kann mehrere DAO's nutzen, um die benötigten Daten zu bearbeiten. Der Manager darüber kann wiederum mehrere Services beauftragen. Der Controller selbst kommuniziert nur mit den Managern, welche die Business Logik kapseln und so (sehr) schlanke Controller ermöglichen. Optional kann oberhalb der Manager noch eine Orchestrator-Schicht eingezogen werden, wenn mehrere Manager koordiniert werden müssen.

Die Hibernate-Session wird im Regelfall bis in die Service-Schicht exisiteren, d.h. die Service-Methoden sind (in Spring) mit `@Transactional` ausgezeichnet. Der Service kann so lazy geladene Properties durch Zugriff nachladen (extra SELECT), besser aber durch Aufruf einer entsprechenden DAO-Methode (z.B. mit entsprechendem JOIN). Falls in einer höheren Schicht (Manager, selten Orchestrator) mehrere Service-Methoden zu einer Transaktion zusammengefasst werden müssen, werden diese Methoden mit `@Transactional` ausgezeichnet. Die Session sollte in keinem Fall bis in die Controller-Schicht ausgedehnt werden.

Objektmodell

Beim Design einer Anwendung wird ein Domain-Modell erstellen, welches die Domänen (Enitities) der realen Welt beschreibt (siehe auch [3]). Daraus wird das Objektmodell abgeleitet, diese Objekte werden oft auch als Domainobjekte bezeichnet.

Hibernate propagiert ganz klar die Strategie, die Domainobjekte, welche durchgehend in der gesamten Anwendung verwendet werden, für die Persistierung zu mappen. Dabei muss man sich aber im klaren darüber sein, welche Auswirkungen dies haben kann:

- Lazy Loading kann in höheren Schichten (unerwünscht) stattfinden oder zu Exceptions führen
- Veränderungen an Domainobjekten werden (unbeabsichtigt) persistiert.

Gehen wir mal davon aus, dass wir aus o.g. Gründen die Session nicht bis in die View offen halten wollen, dann müssen wir vermeiden, dass es in den oberen Schichten zu LazyInitializationExceptions kommt. Bei einfachen Objektmodellen kann dies noch relativ einfach gewährleistet werden, bei komplexen Anwendungen wird das aber schwierig. Es wird umso schwieriger, je mehr Navigationsmöglichkeiten im Objektgraphen bestehen. Ein Negativbeispiel ist z.B. die Assoziation von einem Kunde zu all seinen Aufträgen. Ein Kunde ist ein Kunde, Punkt. Wenn ich eine Liste der Kunden anzeigen will und dahinter den jeweiligen Umsatz, dann sollte ich besser eine Formula-Property mappen und die Summierung der Umsätze der Datenbank überlassen. Viele Navigationspfade erhöhen die Gefahr von LazyInitializationExceptions. Werden im obigen Beispiel nun wirklich alle Aufträge eines Kunden benötigt, so würde ich dafür eine Service-Methode erstellen:

```
List<Order> findOrdersByCustomer(Customer customer);
```

Aus diesem Gesichtspunkt ist die im [2] als Beispiel gezeigte Modellierung des `User` auch nur exemplarisch zu verstehen und in keinem Fall als Best Practice:

```
public class User extends Persistent {
 private String userName;
 private String password;
 private String email;
 private Name name;
 …
 private List auctions;
 …
```

Um die Lazy-Problematik zu vermeiden, hätten die `User`-Objekte mit einer speziellen Service-Methode `findUsersWithAuctions()` laden können, welche sicherstellt, dass die Auktionen initialisiert sind Das Problem besteht darin, dass wir dem User-Objekt nicht ansehen können, welche Komponenten bereits geladen/initialisiert sind und welche nicht (hier ist das noch trivial, in komplexen Anwendung nicht mehr).

Assoziationen sollten nur da modelliert werden, wo sich auch wirklich von der Anwendung benötigt werden und nicht überall, wo sie theoretisch möglich wären. Dies gilt insbesondere für bidirektionale Assoziationen (wann benötigt man den Rückwärtsverweis wirklich?).

Isolierung durch Interfaces
Eine Möglichkeit besteht darin, mehrere Interfaces zu definieren, die nur die garantiert initialisierten Daten bereitstellen:

`PlainUser` – reine User-Daten ohne Auktionen
`UserWithAuctions` – User-Daten mit Auktionen

Die Service-Methoden liefern dann die jeweiligen Interfaces und es ist klar, was man bekommt (wer dann mit Casts mogelt, ist selber schuld). Man könnte sogar noch weiter gehen und eine Interface-Hierarchie schaffen mit verschieden mächtigen Interfaces für die domain-Objekte (nur lesen, lesen und ergänzen, Vollzugriff) – das dürfte aber nur bei sehr großen Projekten relevant werden.

Isolierung durch Kaspelung
Ein alternativer Ansatz besteht darin, Persistence Objekte (PO) einzuführen, welche nur in einem Teil der Schichten gültig sind und nach oben in Domainobjekte zu wandeln. Aber das führt zu erheblichem Mehraufwand in der Service-Schicht und hat deutliche Nachteile für die Wartbarkeit, erst recht, wenn Objekt-Modell und PO-Modell auseinander laufen. **Ich rate von diesem Ansatz daher ab**, trotzdem will die beiden Varianten zur Verdeutlichung gegenüber stellen. Der folgende Abschnitt ist daher durch die Schriftfarbe abgehoben und kann auch übersprungen werden.

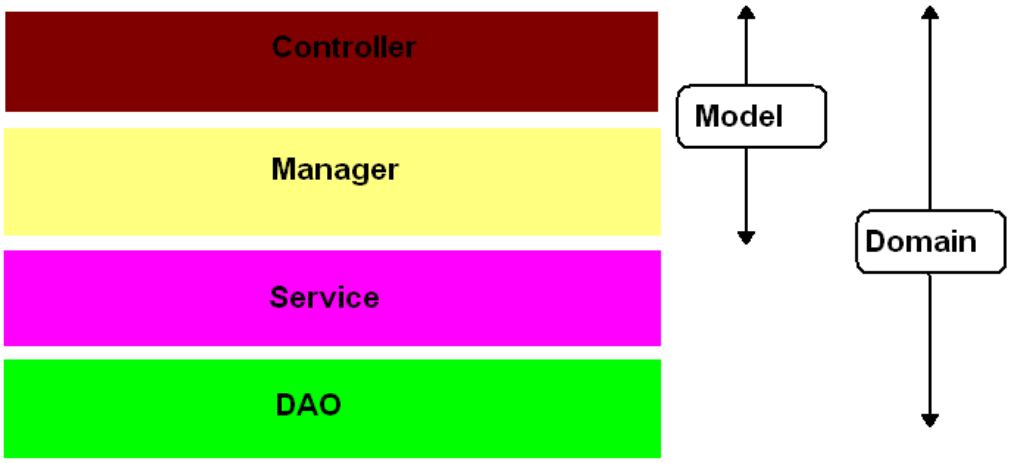
Modellierungsvarianten

Es stellt sich die Frage, bis zu welcher Schicht nun die per DAO gelesenen Persistence Objekte (PO) nach oben durchgereicht werden sollen.

Die Hibernate Autoren vertreten die Ansicht, dass Hibernate deswegen so praktisch ist, weil man die Objekte des Objektmodells direkt auf die Datenbank mappen kann (im Vergleich zu EJB, wo das nicht der Fall ist). Ob man das auch wirklich so tun

muss, ist eine Designfrage. Ich will Vor- und Nachteile dieses Vorgehens aufzeigen und ein alternatives Model vorstellen:

Direktes Mapping der Domainobjekte (Hibernate Style)

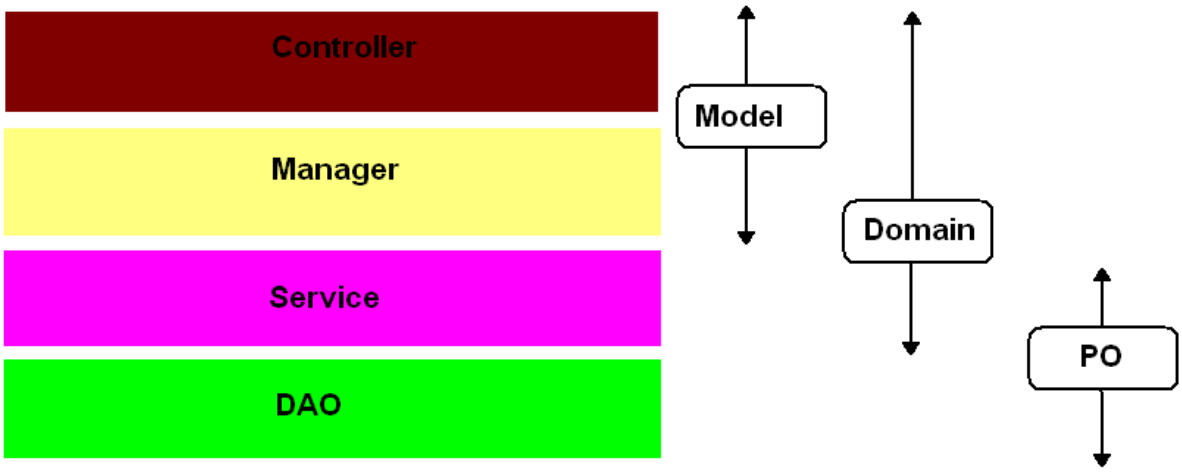


Bei diesem Vorgehen werden die fachlichen Domainobjekte durch die gesamte Anwendung hindurch benutzt. Für die DAO-Schicht wurden Mappings per XML oder Annotations hinzugefügt. Mit Model werden in der Abbildung die Modelobjekte des Controllers (und der View) bezeichnet, die noch in den Manager hereingereicht werden dürfen.

Ein zentrales Problem hierbei ist das Lazy Loading: da normalerweise nicht alle Properties mitgeladen werden, erfolgt das Lazy Loading in einer höheren Schicht (als der Serviceschicht) und erfordert, dass die Session bis dorthin geöffnet bleibtWurde die Session z.B. bereits im Manager verlassen und im Controller oder der View erfolgt ein Zugriff auf eine lazy Property, wird eine `LazyInitializationException` geworfen. Letztlich fallen solche Fehler aber bei ordentlichen Tests sehr schnell auf.

Vorteile	durchgängiges Objektmodell weniger Aufwand Hibernate Style
Nachteile	Mapping in Domain-Schicht (Annotations) Mapping kann komplex sein und evtl. zum Verbiegen der Domainklassen führen Lazy Loading Problematik

Kapselung der Persistence Objekte



Bei diesem Ansatz sind PO's nur in DAO und Service sichtbar. Das Objektmodell ist rein fachlich modelliert und der Service hat die Aufgabe, die PO's in Domainobjekte zu mappen und umgekehrt. Die Anwendung kann sich immer darauf verlassen, dass alle Daten geladen wurden (keine Lazy Problematik). Das Mapping der PO's kann rein technischen Gesichtspunkten folgen.

Vorteile	keine Lazy Loading Problematik
	Mapping nicht in Domainklassen sichtbar
	Domainklassen rein fachlich modelliert
Nachteile	aufwändiger
	fehleranfälliger
	Optimistic Locking (Version) Feature funktioniert nicht
	oftmals rein relationale Sicht der PO

Das Model in MVC

In beiden Ansätzen wird ein Modelobjekt verwendet, um die Daten zwischen Controller und View zu transportieren.

```
public class MemberEditModel {
 private Member theMember;
 private List<Role> availableRolesList;
 private List<Role> selectedRolesList;
}
```

In einigen einfachen Fällen könnte man auch darüber nachdenken, auf das Model zu verzichten und direkt das Domainobjekt als Model zu verwenden und von einigen Autoren wird das sogar als Best Practice bezeichnet. Das sehe ich anders, denn oftmals kommt es vor, das nun transiente Properties in die Domainklasse eingebracht werden, um den Anforderungen der View zu genügen. Ich rate dringend davon ab und empfehle, immer ein Modelobjekt zu verwenden (unter Umständen Ausnahme, wenn das Model keine zusätzlichen Daten enthalten würde, dann aber konsequent auf Model umstellen, wenn das später doch der Fall wird).

DataBinding

Wenn das Domainobjekt Teil des Models ist, ist es damit auch möglichen Angriffen ausgesetzt. Beim SpringMVC Framework zum Beispiel, überträgt der DataBinder alle Requestparameter direkt in das Model und seine Komponeten. Erfolgt kein Schutz durch `initBinder()`, so wäre es möglich, dass ein Angreifer direkten Zugriff auf das Domainobjekt erhält und so z.B. den Preis manipuliert:

`http://...&theOrder.unitPrice=1.0&theOrder.totalPrice=1.0&...`

Transport

Wie werden die Daten nun zwischen Controller, View und zurück transportiert. Diese einfache Frage führt in der Praxis immer wieder zu Kopfzerbrechen. Der Weg vom Controller zu View ist einfach, in Spring ist das z.B. das Command-Objekt, welches Spring automatisch erstellt oder gezielt in formBackingObject() erstellt werden kann. Diese Objekt-Instanz wird dann in den Request gelegt und ist als Request-Attribut (default: „command“) verfügbar, so dass die View nun darauf zugreifen kann:

```
...
<tr> <td> First Name</td> <td><input name=”user.firstname”
value=”${command.user.firstname}”</td>
...
```

Wenn das Formular mit Submit abgeschickt wird, werden die Eingabefelder als Request-Parameter übertragen und von Spring in das (neu erzeugte) User-Objekt übertragen (Data-Binding). Was dann aber noch fehlt ist die Id des Users. Damit diese auch in das User-Objekt übertragen werden kann, müsste sie von der View ebenfalls als hidden Feld gerendert werden. Bei komplexeren Anwendungen müsste so aber viele hidden Felder gerendert werden, um den Zustand (State) des Detached Objects zu übertragen.

Spring bietet die sogenannte Session-Form, d.h. die Model-Instanz wird in der Session gehalten und beim nächsten Request für das Data-Binding verwendet. Somit behält das Model seinen Zustand (mit Ausnahme der vom Request überschriebenen Eingaben, klar).

Transaktionssteuerung

Viele Frameworks wie z.B. Spring unterstützen Transaktionssteuerung. Spring bietet dafür die `@Transactional` Annotation. Generell sollten Transaktionen jedoch so begrenzt wie möglich sein (zeitlich und räumlich, sprich im Code). Durch die Annotation wird auch die Session ausgedehnt, so dass in oberen Schichten (Service, Manager) uninitialisierte Daten nachgeladen werden können, ohne LazyInitializationException.

Die `@Transactional` Annotation sollte aber nur dort verwendet werden, wo wirklich ein transaktionaler Zusammenhang besteht, z.B: bei Verknüfpung mehrere DAO-Methoden:

```
public class OrderServiceImpl {

 @Transactional
 public void placeOrder(Order order) {
  productDAO.doReservation(order);
  processDAO.insertOrder(order);
  orderDAO.save(order);
 }
}
```

Soll lediglich die Session ausgedehnt werden ohne die Trnsaktion auszudehnen, so sollte die `@Transactional` Annotation mit der Option `readonly = true` verwendet werden.

Transaktionen sollen ganz klar in der Business Logic gekapselt sein, also maximal bis auf Managerebene. Auf Controllerebene hat das `@Transactional` nichts zu suchen. Sind mehrere Manageraufrufe in einer Transaktion nötig, kann ein Orchestrator diese transaktional zusammenfassen.

Report Queries

Hibernate bietet mit Report Queries die Möglichkeit, Teile von Ojekten zu laden, wie wir das aus SQL kennen:

```
SELECT id, name, stadt, strasse, hausnr FROM KUNDE;
```

Hierbei werden nicht komplette Kundenobjekte instantiiert, sondern nur Teile des Kunden gelesen. Aus diesem Grund finden sich diese Daten auch nicht im First Level Cache der Session. Für Report Queries sollten immer dedizierte Klassen definiert werden, die dann mit den Daten der Report Query befüllt werden, z.B. `CustomerForListRQ`. Für die Darstellung einer Liste mit Kunden (samt Link mit Kunden-ID) reicht das völlig aus. Klickt der Anwender auf einen Kunden, so muss der `Customer` ohnehin in diesem (neuen) Request per ID geladen werden.

Best Practice

Im folgenden sind einige Erkenntnisse zusammengefasst der Beachtung helfen kann, die häufigsten Probleme zu vermeiden:

- session-per-request-with-detached-objects Pattern verwenden
- Assoziationen nur da modellieren, wo sie gebraucht werden
- Assoziationen per Konvention immer als lazy mappen, gezielte JOINs in DAO
- Objektsichten durch Interfaces ausdrücken
- Durchgängige Verwendung von Model-Objekten, falls darin Domain-Objekte referenziert werden, Zugriff begrenzen (SpringMVC: `initBinder()`).
- Hibernate-Annotations durch Voranstellen Package von JPA-Annotations abgrenzen.
- Annotations an Fields platzieren (AccessType=field)
- @Column Annotation an jedes Field (erleichtert Wartbarkeit bei Tabellenänderungen)
- HQL nur da wo wirklich nötig, Critiera Queries bevorzugen.
- Report Queries für reine Listen verwenden

Multi-Server-Betrieb

Oftmals laufen Web-Projekte in Clustern, d.h. viele Applikationsserver laufen parallel und greifen auf eine zentrale Datenbank zu. Wie bereits oben erwähnt, wird die zentrale Datenbank somit zur begrenzenden Komponente

Caching

Hibernate hält in der Session einen Cache aller persistenten Objekte, die über diese Session gelesen oder persistiert wurden (First Level Cache). Da in Web-Anwendungen die Session aber nur kurz lebt, kann dieser Cache nur in geringem Maße wirklich ausgenutzt werden. Hibernate kennt auch einen Second Level Cache, der wesentlich langlebiger ist als die Session. Dieser SLC ist sogar plugable, d.h. es können verschiedene Implementierungen eingesetzt werden, die am häufigsten verwendete Implementierung dürfte der EHCache sein.

Als erstes stellt sich natürlich die Frage, ob der SLC nur als ReadOnly-Cache (RO) oder als ReadWrite-Cache (RW) eingesetzt werden soll. Das hängt vor allen von der Art der Daten ab, die gecacht werden sollen. Kritische Daten wie Finanzdaten gehören nicht in den SLC! Statische Daten dagegen können bequem im SLC gepuffert werden, wenn sie sich nur sehr selten oder zu definierten Zeitpunkten ändern. Ein RO-SLC kann die Anzahl der Datenbankzugriffe erheblich reduzieren und somit die Skalierung eines Systems verbessern. In der Praxis sind aber die wenigsten Daten wirklich ReadOnly, d.h. sie ändern sich sehr selten, aber eben doch. Deshalb muss die Anwendung solche Änderungen mitbekommen.

Das Invalidieren eines Cache-Eintrags kann mit Hilfe der API-Methode `SessionFactory.evict()` programmatisch oder direkt durch den Cache selbst (z.B:

JBoss Cache) erfolgen. Im einfachsten Fall prüft die Anwendung in regelmäßigen Intervallen (z.B. alle 60s), ob sich Daten geändert haben (z.B. zentrale Tabelle mit Änderungshinweis). Diese Variante ist trivial, sofern wir es uns leisten können, dass die Änderungen nicht sofort auf alle Server propagiert werden. Andernfalls sollte die Propagation am besten direkt durch den Cache-Provider erfolgen, allerdings bietet derzeit nur JBoss eine Cache-Implementierung mit diesem Feature.

Hibernate erlaubt eine feinkörnige Konfiguration, was im SLC gehalten werden soll: welche Klassen, Collections und Query-Ergebnisse. Für den Cache bieten sich nur solche Objekte an, die sich selten ändern. Der Objekt-Cache ist trivial, wurde bereits ein Objekt der Klasse X mit Id 4711 gelesen und befindet sich im SLC, so wird es nicht mehr aus der Datenbank angefordert. Im Collection-Cache werden die IDs der der Entities dieser Collection gehalten (sofern es sich um Assoziationen handelt, ansonsten die Werte selbst). Die Enities der Assoziation selbst müssen natürlich auch im SLC liegen.

Hier ein kurzes Beispiel:

```
@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

Im Query Cache werden nur die IDs der Entities aus dem ResultSet gehalten, die Entities selbst liegen im Objekt-Cache (Entity-Cache). Ob der Query Cache wirklich Vorteile bringt, hängt davon ab, wie die Queries aufgebaut sind. Wenn nämlich jede Query anders aussieht (auch andere Parameter), dann bringt er nichts (im Gegenteil). Wenn aber stets mit wiederkehrenden Parametern zugegriffen wird, macht der Query Cache Sinn. Insbesondere, wenn die CriteriaQuery verwendet wird, um Assoziationen per JOIN zu einer Entity mitzuladen, die über Ihre Id gelesen wird (sonst könnte man ja auch `get()` oder `load()` nutzen).

Wenn der SLC geräumt wird, so sollte nicht vergessen werden, auch den Query Cache selektiv oder gesamt zu leeren:

```
sessionFactory.evictQueries();
```

Um den Query Cache zu aktivieren muss einerseits die Einstellung

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

gesetzt werden und weiterhin die Query mit `setCacheable()` ausgezeichnet werden:

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

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

Literaturnachweis

[1]Hibernate in Action - Christian Bauer, Gavin King, Manning, 2004, ISBN-10: 193239415X, ISBN-13: 978-1932394153

[2] Java Persistence with Hibernate - Christian Bauer, Gavin King, Manning, ISBN-10: 1932394885, ISBN-13: 978-1932394887

[3] Wikipedia, Domain Model: http://en.wikipedia.org/wiki/Domain_model

[4] Professional Java Development with Spring, Rod Johnson, Jürgen Höller, Alef Arendsen, Wiley & Sons 2005, ISBN-10: 0764574833, ISBN-13: 978-0764574832

[5] Open Session in View, http://www.hibernate.org/43.html