14.–17. 09. 2009
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Standleitung

JPA trifft auf die reale Welt

Werner Eberling

MATHEMA Software GmbH

Fakten und Versprechen

Java Persistence API

aktuelle produktive Version: 1.0

standardisierte, leicht erlernbare ORM-Schnittstelle

Einfaches Persistenz-Modell

„Convention over Configuration“

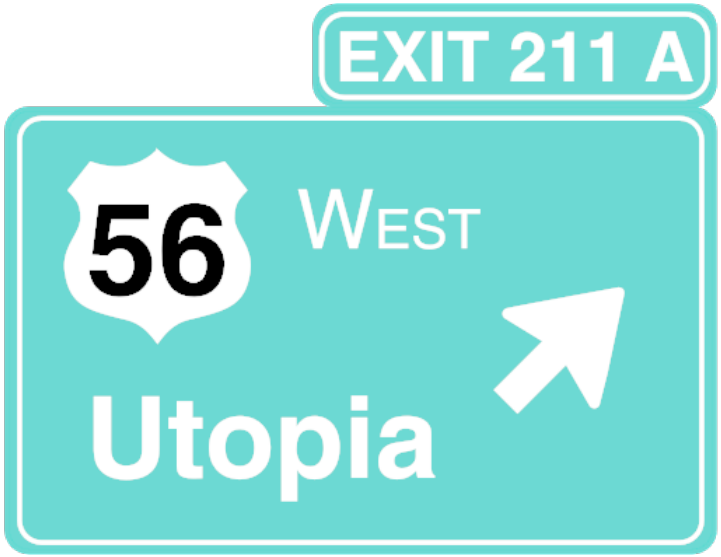
Wäre das alles schön (einfach)...

...wenn die Welt

da draussen nicht

so wäre wie sie

nunmal ist



Quelle: http://deadwhitemales.net/pictures/blogs/utopia.png

“Gewachsene” Datenmodelle

Quelle: http://www.jan-kretschmer.de/photo/photolog07/gebueckbaum_0907.jpg

fehlende
Normalisierung

„generische“
Tabellen

„verteilte“
Datenhaltung

. . .

Der Auftrag

Aufträge

- werden in zentraler Tabelle gehalten
- werden in unterschiedlichen Werkstätten verarbeitet
- sollen u.a. nach Werkstätten gefiltert werden

Ist doch ganz einfach

- select a from Auftrag a where werkstattKZ='R'

ABER

- Es gibt kein Feld werkstattKZ
- (dritte Stelle der Auftragsnummer)

Entity als Konverter

Schaffung eines „sauberen“ Objektmodells

Entity als „Datenkonverter“

Weiter Anwendungsfälle:

- Abbildung von String- bzw. Zahlen-konstanten in Java-enums

ABER

- Was ist mit JPA-Queries?

Entity als Konverter - JPA-Queries

Konvertierungslogik muss in Queries abgebildet werden

Variante 1: Query-Builder-Methoden

- Für einfache Konvertierungen
- Teil der Entity-Klasse

Variante 2: Kapselung in Data Access Object

- Für komplexere Konvertierungen
- Hält Persistenzlogik zusätzlich an zentraler Stelle

Variantenbildung einmal anders

„Generische“ Tabellen zur Abbildung von Varianten

- Tabelle: Produktionseinheit
- Felder: Prop1, Prop2, ..., Prop42, Technologie

Quelle: http://www.pollux.franken.de/fileadmin/user_upload/images/eier-legende-wollmilchsau.jpg

@Inheritance sei Dank

Abbildung über Vererbungshierarchie

- Abstrakte Basisklasse enthält alle (generischen) Attribute
- Konkrete Subklassen bieten konkret benamte Attribute

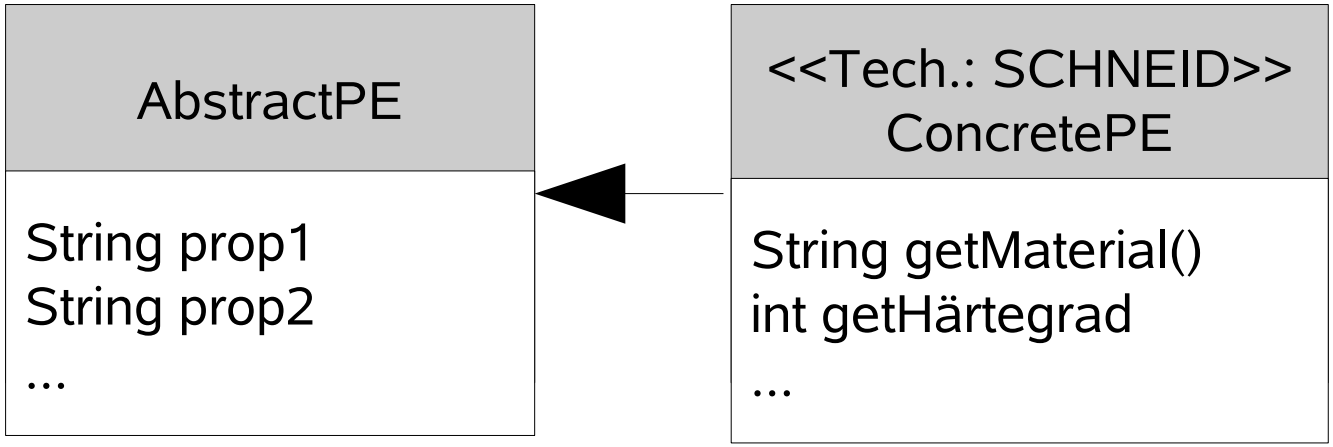
Wieder Konvertierungslogik in der Entity

- Diesmal in Basis und Ableitung verteilt

ABER

- Was ist mit JPA-Queries?

@Inheritance sei Dank



Getrennte Datenhaltung

„Breite“ Tabellen
unbeliebt

Schnitt kann sinnvoll
oder beliebig erfolgen

Beispiel:

- Wieder der Auftrag
- Trennung in
  - Auftrag
  - ????

Quelle: http://hollenberger.files.wordpress.com/2007/11/tate-riss-2.jpg

@SecondaryTable oder anders?

@SecondaryTable ist genau dafür gedacht

ABER

- Was ist mit lazy secondary entries?

In diesem Fall besser

- Technisch als Relation abbilden
- Entity versteckt dieses Detail

ABER

- Was ist im zweiten Fall mit JPA-Queries?

Aus eins mach zwei

Luftverkehrsgesellschaften und Agenten

Zwei Entitäten in einer Tabelle

- Problem: Eindeutigkeit

Lösung

- Zusätzlicher View

ABER

- Was ist bei schreibenden Zugriffen?

Schnell, schneller ... Optimierungen

„Naive“ Implementierungen gehen auf Kosten der Laufzeit

JPA 1.0 kennt nur 1st Level Caching

- Suche über PrimaryKey schnell
- Query-Strategien entscheidend

VendorLockIn?

Quelle: http://www.freetagger.com/wp-content/uploads/2007/06/stoppuhr_s.jpg

Alles was Recht ist

Zugriffsfilter für die Suche nach Flugdaten

Filter u.a. nach

- Liste von Luftverkehrsgesellschaften
- Liste von Agenten
- Liste von Callsigns

Problem

- Sofortiges Laden der 1:n Relationen

Lösung

- Provider spezifischen Fetching-Strategie

Ein großes Ganzes hat viele Teile

Ein Auftrag besitzt eine Menge an Arbeitsgängen

- Klassische 1:n Relation

„Naiv“ implementierte Suche

- select a from Auftrag a where nummer=1234
- Performanzprobleme durch ungeschickte Lade-Strategie

Lösung

- Einführung eines „Fetching-Queries“:
  - select ag from Arbeitsgang ag where ag.auftrag.nummer=1234
  - select a from Auftrag a where nummer=1234

Warum dem Standard folgen?

Der Standard gewährleistet Herstellerunabhängigkeit

ABER

In der Realität wird der Provider doch nie gewechselt!

Oder vielleicht doch?

Das Lama zieht weiter...

Anwendung zum Logomanagement

Ursprüngliche Entwicklung für Tomcat/Hibernate

Aktuelle Planung
- Betrieb im Glassfish Application Server (Toplink)

Eine sauberer Entwicklung macht es möglich
- Hibernate als „Sonderprovider“ eine schnelle Alternative

Probleme
- Kurzschreibweisen bei JPA-Queries
- Unsauberkeiten in persistence.xml

JPA-Queries aber richtig

em.createQuery(„from flug f where knr=123“)

- Wird von Hibernate akzeptiert
- Findet Toplink nicht wirklich schön

em.createQuery(„select f from flug f where knr=123“)

- Besser, aber immer noch nichts für Toplink

em.createQuery(„select f from flug f where f.knr=123“)

- So soll es sein ,)

Persistenz wirklich als Objekteigenschaft?

Persistenz mit JPA meist im Code nicht sichtbar

- Sehr elegante Lösung
- Gering „Verschmutzung“ des Applikationscodes

ABER

- Wirkt oft etwas „magisch“
- Klare Grenzen zwischen managed/detached Zustand wichtig!

Expliziter Laden/Ändern/Speichern Zyklus in der Realität einfache zu verstehen

Vielen Dank!

Werner Eberling

MATHEMA Software GmbH