Referent:
Marko Modsching

Vortrag:
JPA mit Hibernate

Datum:
04.01.2011

Deutsche Software
Engineering & Research GmbH

Steinstraße 11
02826 Görlitz – Germany
Telefon: +49 35 81 / 374 99 – 0
Telefax: +49 35 81 / 374 99 – 99
E-Mail: info@dser.de
Internet: www.dser.de

Agenda

- Warum Object Relational Mapping
- Schichtenmodell
- JPA allgemein
- EntityManager
- Annotationen
- Beziehungen
- Datenabfragen
- Optimierungsmöglichkeiten
- Designempfehlungen
- Demo-Projekt

Warum Object Relational Mapping (ORM)?

- Fördert Einheit von Code und Daten
  - Relational Datenbank => Daten
  - Objektorientierung => Code
- Abbildung von Objekten in Datenbanktabellen
- Einfaches „handling“ von Daten
- Einheitliche Weiterverarbeitung
- Leichteres Refactoring vom Applikationen

Systemarchitektur – Schichtenmodell

Frontend -Schicht

Service-Schicht (Biz-Layer)

Data-Access-Schicht

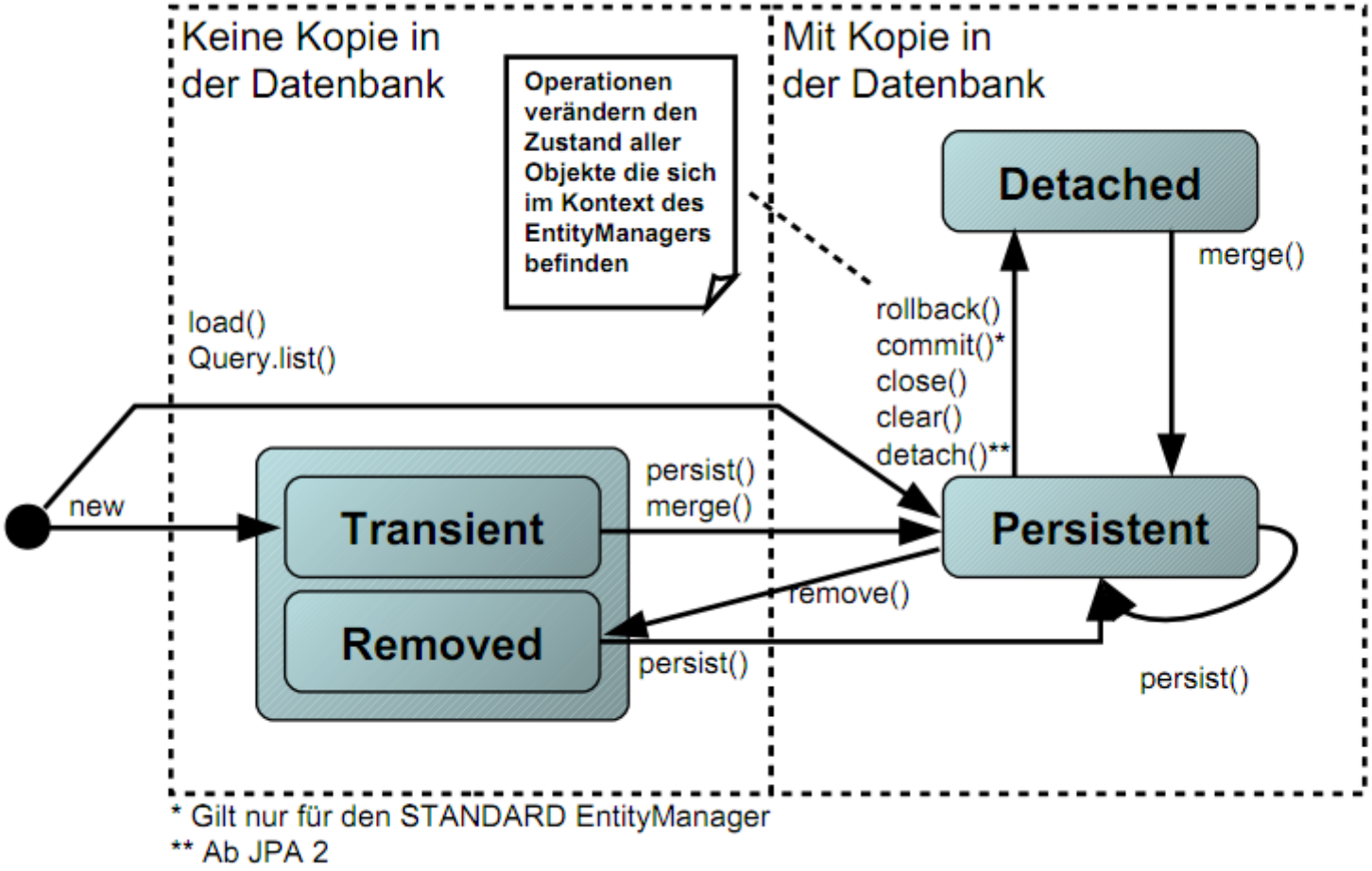
Java Persistence API (JPA)

- Aktuell JPA 2.0
- Standard Schnittstelle für ORM Implementierungen
  - JSR 220, JSR 303 (Bean-Validation)
- Definiert Persistence Entity
  - PoJo Klasse ⇔ Tabelle
  - Objekt ⇔ Zeile in Tabelle
- Mappingvorschirft in Form von Annotation oder per XML
- Definiert Funktionsumfang des EntityManagers
  - Objektzugriff
  - HQL
  - Criteria-API

Entity Manager

- Leichtgewichtige Session zur Datenbank
- Definiert Methoden zum Datenzugriff
  - Persist, merge, remove, flush, lock
  - Find, load
  - createCriteria, createQuery, createSQLQuery
- First-Level Cache
- Nicht Threadsafe
- Fehler via Runtime Exceptions
- Im Exception-Fall ist die session „dirty“! Muss gerollbackt werden

Entity-Life-Cycle



Entity - Annotationen

- @Entity
  - Markiert die Klasse als Entität. Damit kann sie in den Persistenzkontext aufgenommen werden
- @Table
  - Definiert die primäre Tabelle für die Entity
  - Parameter: name, catalog, schema, uniqueConstraints
  - Fachlich motivierte Attributkombinationen können unabhängig vom technischen Schlüssel auf unique gesetzt werden.
- @Column
  - Definiert die Tabellenspalte für das markierte Feld
  - Parameter: name, unique, nullable, updateable, insertable, columnDefinition, table, length, precision, scale
- @Transient
  - Markiert ein nicht persistentes Feld

Beziehungen zwischen Objekten

- 1:1 Beziehung (@OneToOne)
- 1:n Beziehung (@OneToMany)
- n:n Beziehung (@ManyToMany)
- Unterscheiden zwischen Unidirektional und Bidirektional
- Bsp für 1:n
  - JavaUserGroup -> Member ➔ Unidirektional von JavaUserGroup
  - JavaUserGroup <- Member ➔ Unidirektional von Member
  - JavaUserGroup <-> Member ➔ Bidirektional

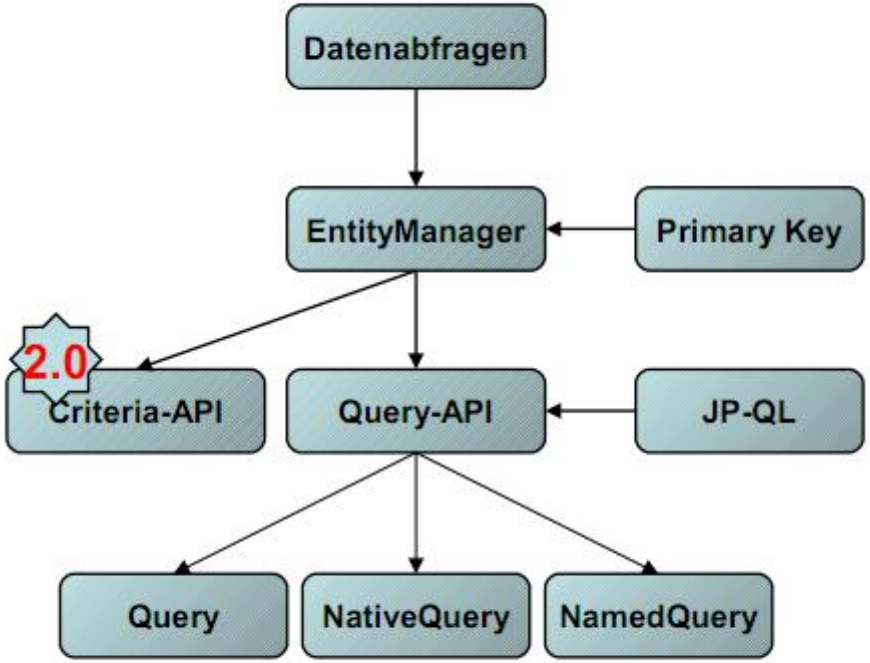
Bidirektionale Beziehungen

- „mappedBy“
  - verweist auf die zuständige Seite der Beziehung (owner). Dies erfolgt bei unidirektionalen Beziehungen implizit. Objektreferenzen beider Seiten müssen gepflegt werden.
- **Achtung!!!**
  - Bei bidirektionalen Beziehungen können aufgrund der Standardladestrategie „EAGER“ der Rückwärtsreferenz ungewollt ganze Objektnetze aus der Datenbank gelesen werden.
- **Allerdings:**
  - Lassen sich durch Einbeziehen der Rückwärtsreferenzen Datenabfragen flexibler formulieren.
  - Können Objekt-Teilnetze kontrolliert durch Aufruf der Rückwärtsreferenzen nachgeladen werden.

Objekttraversierung und Nachladen von Daten

- Verbundene Daten werden „on demand“ automatisch nachgeladen
- Strategie bzgl. dem Laden referenzierter Daten
  - Eager: Gleich alles mitladen.
  - Lazy: Erst dann laden, wenn ein Zugriff darauf erfolgt.
- Bsp. JavaUserGroup => Member
- Achtung!
  - Lazy-Loading führt bei Hibernate zu Proxy-Objekten.
  - Lazy-Loading führt ggf. zu Detached Entities und später ggf. zu Exceptions. (Stichwort: LazyInitializationException)
  - Bei einem merge werden LAZY-Felder, die noch nicht gefetched wurden, auch nicht gemerged.
- Defaults
  - Eager bei Objektattributen und 1:1 und n:1 Beziehungen.
  - Lazy bei ?:n Beziehungen

Datenabfragen



- Erste Anlaufstelle aller Abfragen ist der EntityManager
- Abfragen anhand des Primärschlüssels unterstützt der EntityManager direkt.
- Alle sonstigen Abfragen erfolgen mittels der Query-API und der Datenabfragesprache JP-QL.
- Die Query-API kennt 3 Querytypen
  - Criteria API (neu)
  - Query: JP-QL Abfrage (**Hibernate**: Hibernate Query-Language)
  - NativeQuery: Datenbankabhänige SQL-Abfrage

Optimierungsmöglichkeiten

- Zum Vergleich immer die SQL-Anweisungen heranziehen, die man ohne Hibernate verwenden würde.
- Optimierungen seitens der Datenbank nicht vernachlässigen.
  - Also Datenmodell, Indexe, partitionierte Daten, partitionierte Hash-Indexe
- Die durch Hibernate produzierten SQL-Anweisungen prüfen und kritisch hinterfragen.
- Einige Optimierungsmöglichkeiten sind hibernatespezifisch
- Vermeiden von Kardinalfehlern
  - Laden ganzer Objektlisten, nur um die Anzahl der betreffenden Objekte festzustellen,
  - exzessives Logging von Entity-Objekten
  - überlasteter Connection-Pool.

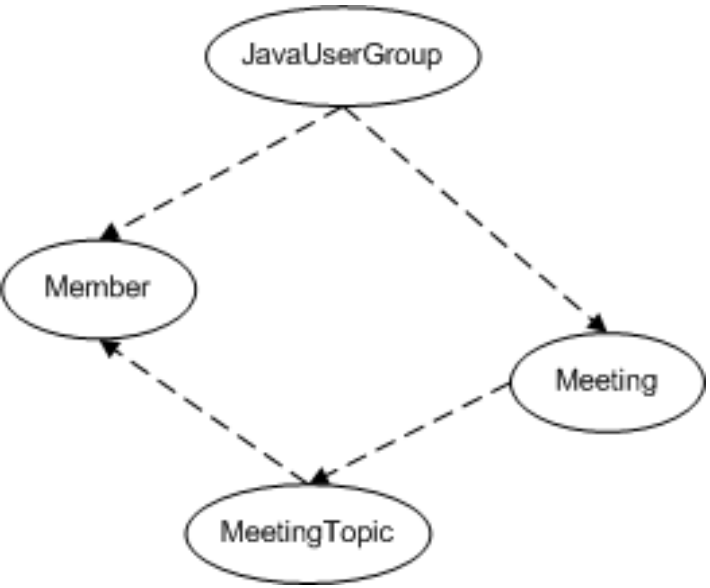
Optimierungsmöglichkeiten II

- Lazy vs. Eager
- hibernate.default_batch_fetch_size Parameter
  - Paketweises Laden mittels 'select…where … in (…)
- Second Level Cache hält Datenobjekte anhand ihrer Primärschlüssel vor
- Query Cache beantwortet identische anfragen mit den selben Ergebnisdaten ohne erneute Datenabfragen
- Vorsicht beim Marshalling von Entity-Graphen
  - Serialisierungslogik traversiert den gesamten Objekt-Graphen
- N+1 Problem bei Update/Insert

Designempfehlungen

- Klassenmodell einfach halten
- Erstellung von Designrichtlinien, für Datenbankobjekte und deren Verhalten
- Definition von Basisklassen
- Kein flush() clear() in der Biz-Logik
- In Memory Datenbanktests, um Mapping zu testen

Demo-Projekt



- Setup JPA EntityManager
- Setup Hibernate Session
- JUG Entity Modell mappen
- Abfragen
  - Criteria
  - HSQL
  - Native SQL (mit Automapping)