Agenda

Query

JPAQL

Criteria API

SQL

Batch-Updates

Performance

Cache

Native SQL

- Queries können nativ in SQL angegeben werden
- mehrere Varianten
  - createNativeQuery(String sql)
    - **liefert Skalar**
  - createNativeQuery(String sql, Class entityClass)
    - **muss alle Spalten der Entity liefern**
  - createNativeQuery(String sql, String mapping)
    - **liefert beliebige Kombination von Entities und einzelnen Skalaren**
    - **Mapping wird als @SqlResultSetMapping definiert**

```
String sql = "select * from Topic";
Query query = em.createNativeQuery(sql,  Topic.class);
```

ResultSetMapping

- Wird in der dritten Variante angegeben
- Wenn bspw. Reihenfolge oder Bezeichnungen des Resultset nicht mit den erwarteten übereinstimmt

```
@Entity
@SqlResultSetMapping(name="simple",
   entities={@EntityResult(entityClass=kurs.Topic.class,
   fields={    @FieldResult (name="name", column="name") ,
               @FieldResult (name="topicID",  column="ID" ) }
   ) })
@NamedNativeQuery(name="allTopics",  resultSetMapping ="simple",
query="select name, topicID as ID from Topic")
   public class Topic implements Serializable{   ...
```

JPAQL: Bulk Operationen

- JPAQL unterstützt:
  - Bulk Updates und Deletes
    - Nur für einen Entity-Typ
    - Sollten in einer eigenen Transaktion ablaufen
    - Versionsnummer muss manuell gepflegt werden

```
em.createQuery("DELETE FROM Customer c"
  + " WHERE c.status = 'inactive '").executeUpdate
  () ;
em.createQuery("UPDATE customer c"
  + " SET c.status = 'outstanding'"
  + " WHERE balance < 10000"
  + " AND 1000 > (SELECT COUNT(o) FROM customer ...)")
  .executeUpdate() ;
```

Agenda

Query

JPAQL

Criteria API

SQL

Batch-Updates

Performance

Cache

Performance-Probleme mit JPA/Hibernate?

- **Nachteile von JPA/Hibernate**
  - Overhead durch OR-Mapping, Dirty-Checking, DB-Unabhängigkeit,...
- **Vorteile von JPA/Hibernate**
  - z. B. Caches
    - ggf. nutzlos durch falsche Verwendung
- **Wie reagieren?**
  - Nutzung von JPA/Hibernate Optimierungsmöglichkeiten
  - Stored Procedures oder direkt SQL aufrufen
  - Alternativen zu JPA/Hibernate:
    - JDBC, iBatis,...

Werkzeuge Performance-Suche

- **Monitoring Tools**
  - Profiler: z. B. JProfiler, JProbe, PerformaSure
  - JMX-Tooling: z. B. JConsole, AdventNet, MC4J, ...
- **Hibernate Statistics** (JConsole, Statsviewer)
- **DB-Analyzer**
  - Tracer: z. B. P6Spy, IronTrack SQL, Elvyx
  - SQL-Clients: z. B. Toad, SQL-Developer (für Oracle DB)
- **Lasttests**
  - The Grinder

Objektnavigation über Relationen

- **Navigationsgeschwindigkeit verbessern**
  - vollständiges Objektnetz in Speicher laden (speicherintensiv!!!)
- **Speicherverbrauch optimieren**
  - Relationen bei Bedarf nachladen (aber: N+1 Problem!!!)
- **Workarounds**
  - Batch Fetching
  - Subselect Fetching
  - Join Fetching (Achtung: Kartesisches Produkt)
  - HQL (für explizite Abfragen)
  - Paging
  - Filter
  - Caching

Select n + 1 Problem

- Szenario:
  - Spieler mit 1 :n-Beziehung zu Adresse
  - 1.000.000 Spieler, jeweils 1 bis 5 Adressen

```
List<Spieler> players = s.createQuery("from Spieler").list();
foreach  (Spieler s : players) {
   // lädt Adressen für s nach
   System.out.println(s.addresses.size());
}
```

- Problem: 1 Abfrage für alle Spieler + 1.000.000 Abfragen für Adressen

Lazy vs. Fetchtype

- **Wann laden?**
  - Lazy
  - Eager
- **Wie laden?**
  - Join Fetching
  - Select Fetching
  - Batch Fetching
  - Subselect Fetching
- beliebig kombinierbar
  - außer: Lazy und Join Fetching funktioniert nicht

Lazy vs. Fetchtype

- **Join Fetching**
  - 1 SELECT für ALLES
  - Assoziiertes Objekt oder Collection wird im gleichen SELECT geladen mittels OUTER JOIN.
- **Select Fetching**
  - 1 SELECT für Masterobjekte
  - 1 SELECT pro Masterobjekt wird benutzt um assoziiertes Objekt oder Collection zu laden.
  - Ist eager eingestellt (Hibernate lazy="false")
    - zweites SELECT sobald Master-Objekt geladen wird
  - Ist lazy eingestellt (Hibernate lazy="true")
    - zweites SELECT erst wenn auf Assoziation zugegriffen wird

Lazy vs. Fetchtype

- **Batch Fetching**
  - Optimisierungsstrategie für Select Fetching.
  - 1 SELECT für Masterobjekte
  - 1 SELECT für Assoziation des Masters, wobei Assoziationen der weiteren (Batchgröße-1) Master mit geladen werden
- **Subselect Fetching**
  - 1 SELECT für Masterobjekte
  - 1 SELECT für alle Assoziationen aller Masterobjekte
  - Ist eager eingestellt (Hibernate lazy="false")
    - zweites SELECT sobald Master-Objekt geladen wird
  - Ist lazy eingestellt (Hibernate lazy="true")
    - zweites SELECT erst wenn auf Assoziation zugegriffen wird

Beeinflussungsmöglichkeiten des Nachlade

- Mapping
- Query
- Manuell (Test über PersistenceUnitUtil)
- Entity Graph (neu ab JPA 2.1)

Beeinflussungsmöglichkeiten des Nachladens

- Nachladen beeinflussen über Mapping
  - fetch-Attribut an
    - @ManyToOne (default EAGER)
    - @OneToOne (default EAGER)
    - @OneToMany (default LAZY)
    - @ManyToMany (default LAZY)
- **Nachladen beeinflussen über Query**
  - Angabe von **`join fetch`** in JPQL
  - Angabe von **`root.fetch(...)`** in Criteria

Beeinflussungsmöglichkeiten des Nachladens

- **Manuell**
  - Zugriff auf Objekt führt zu Nachladen
  - Überprüfen des Ladezustandes mit `PersistenceUnitUtil`
    - Über EntityManagerFactory
    - Zusätzlich Möglichkeit, die Id zu erhalten

Beeinflussungsmöglichkeiten des Nachlade

- **Entity Graph**
- Spezifikation des Fetch-Verhaltens
  - Find-Operationen (via **`Map`**)
  - Queries (via **`setHint`**)
- Angabe über
  - **`javax.persistence.fetchgraph`** →Alle nicht enthaltenen Attribute lazy
    - only the attributes specified by the entity graph will be treated as FetchType.EAGER. All other attributes will be lazy.
  - **`javax.persistence.loadgraph`** →Alle nicht enthaltenen Attribute default
    - all attributes that are not specified by the entity graph will keep their default fetch type.

Beeinflussungsmöglichkeiten des Nachlade

- **Named Entity Graph and Named SubGraph**

```
@Entity
@NamedEntityGraphs({
        @NamedEntityGraph(name = "graph.AuthorBooks",
                          attributeNodes = @NamedAttributeNode("books")),
        @NamedEntityGraph(name = "graph.AuthorBooksReviews",
                          attributeNodes = @NamedAttributeNode(value = "books", subgraph = "books"),
                          subgraphs = @NamedSubgraph(name = "books", attributeNodes = @NamedAttributeNode("reviews"))) })
public class Author implements Serializable {
```

```
EntityGraph graph = em.getEntityGraph("graph.AuthorBooks");

// Requires additional DISTINCT
List<Author> authors = em
        .createQuery("SELECT DISTINCT a FROM Author a", Author.class)
        .setHint("javax.persistence.fetchgraph", graph).getResultList();
```

Beeinflussungsmöglichkeiten des Nachlade

- **Named Entity Graph and Named SubGraph**

```
@Entity
@NamedEntityGraphs({
        @NamedEntityGraph(name = "graph.AuthorBooks",
                          attributeNodes = @NamedAttributeNode("books")),
        @NamedEntityGraph(name = "graph.AuthorBooksReviews",
                          attributeNodes = @NamedAttributeNode(value = "books", subgraph = "books"),
                          subgraphs = @NamedSubgraph(name = "books", attributeNodes = @NamedAttributeNode("reviews"))) })
public class Author implements Serializable {
```

```
EntityGraph graph = em.getEntityGraph("graph.AuthorBooksReviews");

// Requires additional DISTINCT
List<Author> authors = em
        .createQuery("SELECT DISTINCT a FROM Author a", Author.class)
        .setHint("javax.persistence.loadgraph", graph).getResultList();
```

Beeinflussungsmöglichkeiten des Nachlade

- **Dynamic entity graph**

```
EntityGraph graph = em.createEntityGraph(Author.class);
Subgraph<Book> bookSubGraph = graph.addSubgraph(Author_.books);
bookSubGraph.addSubgraph(Book_.reviews);

// Requires additional DISTINCT
List<Author> authors = em
        .createQuery("SELECT DISTINCT a FROM Author a", Author.class)
        .setHint("javax.persistence.fetchgraph", graph).getResultList();
```

Eager und Join Fetching: 1 Select

- **globaler Fetch-Plan (*.hbm.xml):**

```
<set name="addresses" lazy="false" fetch="join"  ... /
```

- **globaler Fetch-Plan (JPA):**

```
@OneToMany
@Fetch(value=FetchMode.JOIN)
private Set<Address> addresses = ...
```

- HQL/JPAQL:

```
from Player p left join fetch p.addresses
```

Probleme bei Join: Kartesisches Produkt

PID	NAME	...	AID	STR	...	RID	ROLE	...
	Tabelle			Tabelle			Tabelle	
	Person			Adresse			Rolle	

Subselect Fetching: 1 + 1 Selects

- **globaler Fetch-Plan (*.hbm.xml):**

```
<set name="addresses" fetch="subselect"  ... />
```

- **globaler Fetch-Plan (JPA):**

```
@OneToMany
@Fetch(value=FetchMode.SUBSELECT)
private Set<Address> addresses =  ...
```

- entspricht:

```
select * from Address where spieler_fk
in  (select id from Spieler)
```

Batch Fetching:
n + 1 / Batching Selects

- **globaler Fetch-Plan (*.hbm.xml):**

```
<set name="addresses" batch-size="8"  … />
```

- **globaler Fetch-Plan (JPA):**

```
@OneToMany
@BatchSizeFetch(size=8)
private Set<Address>    addresses = ...
```

- entspricht:
  - 1 Select für alle Spieler
  - 1 Select für Adressen zum ersten Spieler, lädt die Adressen der nächsten 7 Spieler gleich mit
  - erst wieder ein Select beim der 9. Spieler

Laden mit expliziten Queries

- **Alternative Abfrage für spezifisch Use Case**

```
List<Person> allPersons =
session.createQuery("from Person p left join fetch
  p.Addresss").list ();

List<Person> allPersons =
  session.createCriteria(Person.class)
  .setFetchMode("addresses",  FetchMode.JOIN)
  .list () ;
// Iterate through the collections...
```

- **Führt zu folgender Datenbankanfrage**

```
select p.*, a.*
  from Person p
left outer join Address a on p.Person_ID = a.Person_ID
```

Laden mit expliziten Queries

root.fetch(...) in Criteria


```
EntityManager em =
  entityManagerFactory.createEntityManager();
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery criteriaQuery =
  criteriaBuilder.createQuery(Person.class);
Root person = criteriaQuery.from(Person.class);
person.fetch(Person_.adresses);
criteriaQuery.select(person);

List resultList =
  em.createQuery(criteriaQuery).getResultList();
```

- **Äquivalent zu**

  select p from Person p join fetch p.adresses

Optimierungsstrategie

- **Einstieg mit Lazy Default-Fetch-Plan**
  - 1-n/m-n-Relationen - lazy
  - n-1/1-1-Relationen - eager (lazy=false)
- **Es folgt Szenarien-bezogenes Aufzeichnen**
  - der auslösenden Hibernate-Abfragen
  - der resultierenden DB-Abfragen
- **Danach Optimierung der Zahl und Komplexität der resultierenden DB-Anfragen**
  - spezifische Optimierung einer Anfrage
  - seltener Optimierung des globalen Fetch-Plans

Agenda

Query

JPAQL

Criteria API

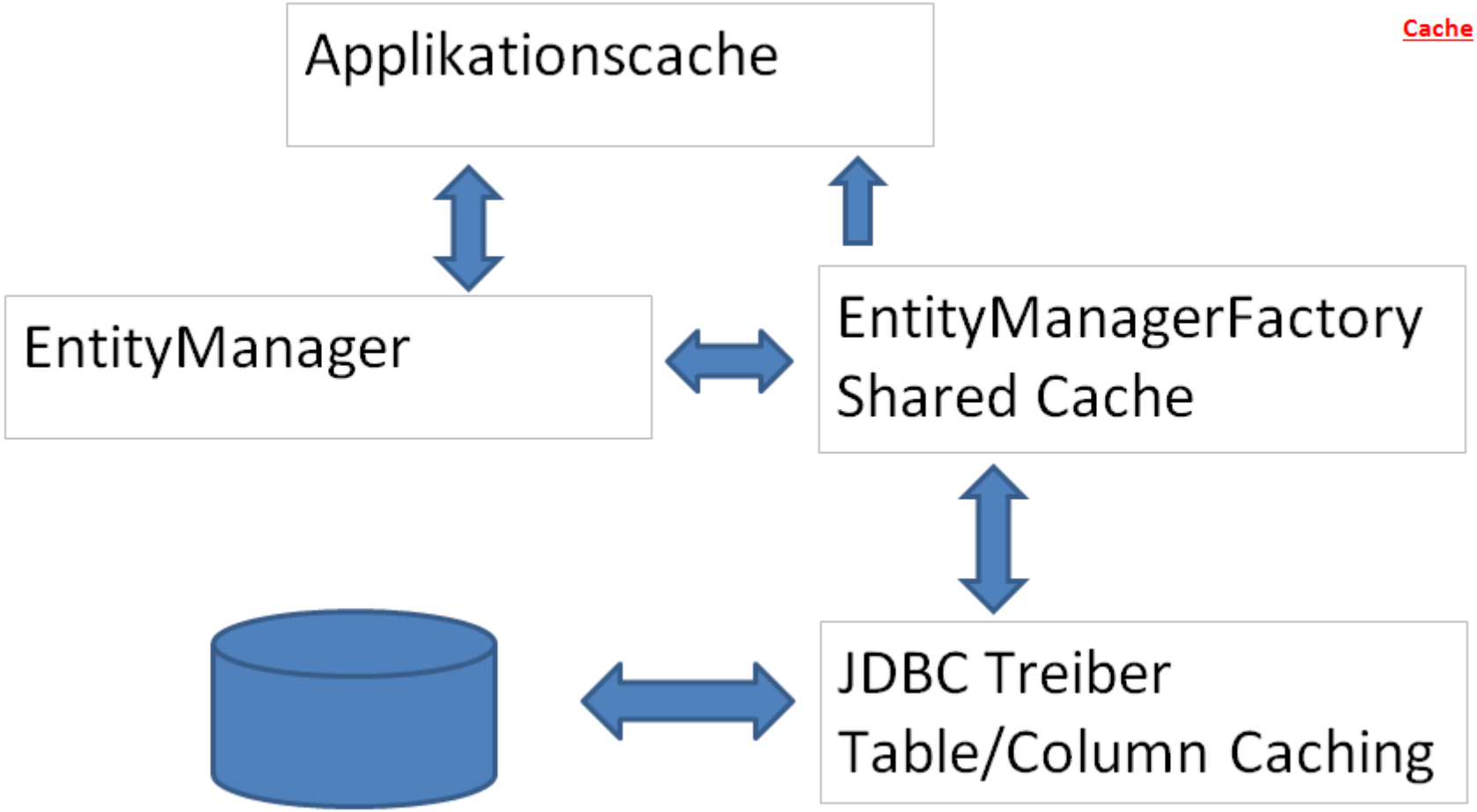
SQL

Batch-Updates

Performance

Cache

Cache Schichten



Caches

- Arten von Caches:
  - **First Level Cache.** Ist mit jeweiligen EntityManager /Hibernate Session verbunden.
  - **Second Level Cache.** Ist mit EntityManagerFactory/ Session Factory verbunden.
  - **Query Cache.** Es werden Anfragen werden gespeichert.
  - Nur der First Level Cache arbeitet standardmäßig. Die beiden anderen Caches sind explizit zu aktivieren.

First Level Cache

- Wenn Objekte aus DB geladen
  - automatische Ablage in First Level Cache
- First Level Cache lässt sich nicht abschalten.
- Wird benutzt, um die Anzahl der einzelnen SQL-Operationen innerhalb einer Transaktion zu minimieren.

```
// in Cache legen
                          session. load()
  em.persist(o);
  em.merge(o);
// aus Cache entfernen
                          session.evict()
  em.clear() ;
  em.detach(o);
```

Second Level Cache

- Objekte sind der ganzen Applikation zugänglich, nicht nur der jeweiligen Session.
- Das reduziert die Datenbankoperationen
  - Problem: **stale data**)
- Hibernate unterstützt diverse Open-Source Bibliotheken:
  - EHCache, OSCache, SwarmCache, JBoss TreeCache
- muß über Provider konfiguriert werden

Second Level Cache

- JPA Interface: Cache
- Evict = Entfernen der Instanz aus dem Second Level Cache

```
Cache cache = emf.getCache();
cache.contains(class, id);
cache.evict(class, id);
cache.evict(class);
cache.evictAll();
```

Caching Strategien

- Beim Cachen von Objekten gibt es unterschiedliche Gesichtspunkte, die unterschiedliche Strategien erfordern:
  - Nur Lesen und kein Schreiben (Read-Only)
  - Lesen sowie Schreiben (Read/Write)
  - Zwei Transaktionen können Daten überschreiben (Nonstrict Read/Write)
  - Echte transaktionale Sicherheit (Transactional)
- Die Implementierungen verhalten sich unterschiedlich.
  - JBoss TreeCache ist zum Beispiel transaktional aber EHCache nicht.

JPA Cache-Konfiguration


```
<property name="javax.persistence.sharedCache.mode"
value="ALL"/>
```

- NONE, ENABLE_SELECTIVE, DISABLE_SELECTIVE, ALL, UNSPECIFIED

```
@Cacheable // oder @Cacheable(true)
@Entity public class MyCacheableEntityClass  { }
```

- RetrieveMode, StoreMode
  - RetrieveMode.USE, RetrieveMode.BYPASS
  - CacheStoreMode.USE, CacheStoreMode.BYPASS, CacheStoreMode.REFRESH

```
query.setHint("javax.persistence.cache.retrieveMode",
CacheRetrieveMode.BYPASS);
em.find(Entity.class, 1L, CacheRetrieveMode.BYPASS);
```

Hibernate Cache-Konfiguration

- in Hibernate-Settings aktivieren
  - CacheProvider (EHCache, OSCache, SwarmCache, TreeCache)
  - use_second_level_cache=true
- Cache Regions (angeben für Klassen, Collections oder Queries)
  - Usecase spezifisch optimieren
  - usage: Caching Strategie
    - **read-only:beste Performance**
    - **nonstrict-read-write: nur für nicht konkurrierende Zugriffe**
    - **read-write: gewährleistet read-commited**
    - **transactional: nur bei JTA-Umgebungen**

```
<class name="Person" ...>
    <cache usage="read-write"/>
    . . .
</class>

<set name="addresses">
    <cache usage="read"/>
</set>
```

Second Level Cache löschen

- Evict = Entfernen der Instanz aus dem Second Level Cache
  - sessionFactory.evict(Person.class, id);
  - sessionFactory.evict(Person.class);
  - sessionFactory.evictCollection("Person.addresses", personId);
  - sessionFactory.evictCollection("Person.addresses");

Gefahren bei Second Level Caches

- **andere Applikationen ändern DB**
  - Cache regelmäßig validieren
- **zu viele schreibende Zugriffe**
  - Cache-Miss größer als Cache-Hit: schlechtes Cache-Ratio
- **zu kleine Dimensionierung des Cache**
  - zu groß auch schlecht (längere GC-Laufzeiten, Validierung mit DB)

Query-Cache

- **Query-Cache explizit einschalten**

```
<prop key="hibernate.cache.use_query_cache">true</prop>
```

- **Query-Cache speichert nur Ids**
  - muß mit Entity-Cache zusammen arbeiten
- nur sinnvoll für häufige Abfragen mit immer gleichen Parametern
- darum werden Queries defaultmäßig nicht gecached
  - setCacheable(true) aufrufen!

```
public Object getObjectByName(String name) {
        Query q = session.createQuery("from Object where name= ?");
        q.setParameter(0,  name);
        q.setCacheable(true);
        return q.uniqueResult();
      } _
```

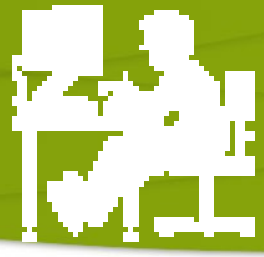
Unwirksamer Cache

- z. B. bei Abfragen mit ständig wechselnden Parametern

```
hql = "from de.example.Firma ... where validFrom <= :date
  and
validTo >= :date";
Query q = session.createQuery(hql);
for(..) {
  q.setDate (. .) ;
  q.list() ;
}
```

- ggf. eigene Caches implementieren
  - insbesondere wenn immer der gleiche Datensatz geholt wird
  - dazu alle abzufragenden Datensätze laden und in Java vergleichen
    - **funktioniert nur bei überschaubarer Anzahl von Datensätzen!**

Aufgabe



- Demo 4: Caching (Projekt: 10-Onlineshop-Gesamt)