JP-QL

Java Persistence Query Language

JPA Query API (JPQL) / Dynamic Queries

Queries werden über 2 verschiedene Interfaces abgebildet:

- Query-Interface (einzige Variante in JPA 1)

```
Query query = em.createQuery("SELECT p FROM Person p");
List results = query.getResultList();
```

- TypedQuery-Interface (seit JPA 2.0)

```
TypedQuery<Person> typedQuery =
        em.createQuery("SELECT p FROM Person p", Person.class);
List<Person> results = typedQuery.getResultList();
```

Abfr

- getSingleResult()
- getResultList()
- executeUpdate()

JPA Query API / Parameters

- **Named Parameters**
  Bezeichner mit vorangestelltem Doppelpunkt

```
TypedQuery<Long> query = em.createQuery(
        "SELECT COUNT(p) FROM Person p WHERE p.name = :name", Long.class);
long personCount = query.setParameter("name", "Muster").getSingleResult();
```

- **Ordinal Parameters** (über Position)
  Positionsangabe mit vorangestelltem Fragezeichen

```
TypedQuery<Long> query = em.createQuery(
        "SELECT COUNT(p) FROM Person p WHERE p.name = ?1", Long.class);
long personCount = query.setParameter(1, "Muster").getSingleResult();
```

- **Criteria Query Parameters**
  folgt später...

JPA Named Queries

- Erlaubt die Definition statischer Queries auf Entity-Ebene
- Trennung von Queries und Sourcecode
- Unterstützt den Einsatz von Parametern

- @NamedQuery (name, query [, lockMode] [,hints])

```
@Entity
@NamedQuery(name=Person.FINDALL, query="SELECT p FROM Person p")
public class Person implements Serializable {
    public static final String FINDALL = "Person.findAll";
```

- (evtl. Entity-Namen als Prefix)

JPA Named Queries

- Sollen mehrere Named Queries definiert werden, müssen diese unter einer @NamedQueries-Annotation gekapselt werden

```
@Entity
@NamedQueries({
  @NamedQuery(name=Person.FINDALL, query="SELECT p FROM Person p"),
  @NamedQuery(name=Person.FINDBYNAME, query="SELECT p FROM Person p WHERE p.name=:name")
})
public class Person implements Serializable {
    public static final String FINDALL = "Person.findAll";
    public static final String FINDBYNAME = "Person.findByName";
```

- Einsatz der NamedQueries

```
TypedQuery<Person> query =
        em.createNamedQuery(Person.FINDALL, Person.class);
List<Person> results = query.getResultList();
```

JPA Native Queries

- Erlaubt den Einsatz von normalen SQL-Query-Statements
- Können ebenfalls managed Entities zurückliefern
- Werden nicht vom Persistence Provider geparsed, deshalb können datenbankspezifische Statements (die nicht JPQL-konform sind) abgesetzt werden.

- Selektion gewisser Spalten in ein Object-Array

```
String sql = "SELECT vorname, nachname FROM Person";
Query query = em.createNativeQuery(sql);
List<Object[]> objects = query.getResultList();

for (Object[] obj : objects) {
    System.out.println(obj[0] + " " + obj[1]);
}
```

JPA Native Queries

- Rückgabe von managed Entities

```
String sql2 = "SELECT * FROM Person";
Query pquery = em.createNativeQuery(sql2, Person.class);
List<Person> persons = pquery.getResultList();

for (Person p : persons) {
    System.out.println(p.getVorname() + " " + p.getNachname());
}
```

JPA Criteria API

- Wird benutzt, um Abfragen objektbasiert aufzubauen, anstatt durch das Parsen von Query-Strings
- Kann nur für dynamische Abfragen verwendet werden → keine NamedQueries
- Interface: CriteriaBuilder

```
// Alternative zu: SELECT p from Person p
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> cquery = cb.createQuery(Person.class);
Root<Person> root = cquery.from(Person.class);
cquery.select(root);

TypedQuery<Person> query = em.createQuery(cquery);
List<Person> persons = query.getResultList();
```

JPA Criteria API / SELECT

- Selection von managed Entities (siehe vorige Folie)
- Select von mehreren Spalten

```
// Alternative zu: SELECT distinct p.vorname, p.nachname from Person p
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cquery = cb.createQuery(Object[].class);
Root<Person> root = cquery.from(Person.class);
cquery.select(cb.array(root.get("vorname"), root.get("nachname"))).distinct(true);

TypedQuery<Object[]> query = em.createQuery(cquery);
List<Object[]> objects = query.getResultList();
```

JPA Criteria API / FROM

- Selection von mehreren Tabellen

```
// Alternative zu: SELECT p1, p2 from Person p1, Person p2
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cquery = cb.createQuery(Object[].class);
Root<Person> p1 = cquery.from(Person.class);
Root<Person> p2 = cquery.from(Person.class);
cquery.multiselect(p1, p2);

TypedQuery<Object[]> query = em.createQuery(cquery);
List<Object[]> objects = query.getResultList();
```

- JOIN

```
// Alternative zu: SELECT p.nachname, t.number FROM Person p LEFT JOIN p.numbers t
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cquery = cb.createQuery(Object[].class);
Root<Person> p = cquery.from(Person.class);
Join t = p.join("numbers", JoinType.LEFT);
cquery.multiselect(p.get("nachname"), t.get("number"));

TypedQuery<Object[]> query = em.createQuery(cquery);
  List<Object[]> objects = query.getResultList();
```

JPA Criteria API / WHERE

- Einfache WHERE-Klausel (hier mit Parameter)

```
// Alternative zu: SELECT p FROM Person p WHERE p.nachname LIKE :nn
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> cquery = cb.createQuery(Person.class);
Root<Person> p = cquery.from(Person.class);
cquery.select(p);
ParameterExpression<String> param = cb.parameter(String.class, "nn");
cquery.where(cb.like(p.get("nachname"), param));

TypedQuery<Person> query = em.createQuery(cquery);
query.setParameter("nn", "Muster");
List<Person> objects = query.getResultList();
```

- Verknüpfung von Bedingungen (and/or)

```
// Alternative zu: SELECT p FROM Person p WHERE p.nachname LIKE :nn and p.vorname LIKE :vn
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> cquery = cb.createQuery(Person.class);
Root<Person> p = cquery.from(Person.class);
cquery.select(p);
cquery.where(
        cb.and(
                cb.like(p.get("nachname"), "Muste%"),
                cb.like(p.get("vorname"), "Max%")
        )
);
```

JPA Criteria API / WHERE

Wichtigste Funktionen

- equal, notEqual
- lessThen, lt
- greaterThan, gt
- lessThenOrEqualTo, le
- greaterThanOrEqualTo, ge
- like, notLike
- between
- isNull
- In

Logische Operatoren

- and, or, not
- conjunction (entspr. geklammerter AND-Verkn.)
- disjunction (entspr. Geklammerter OR-Verknüpfung)

JPA Criteria API / GROUP BY und HAVING

```
// Alternative zu: SELECT p.nachname, count(p) FROM Person p
//                 GROUP BY p.nachname HAVING count(p)>2
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cquery = cb.createQuery(Object[].class);
Root<Person> p = cquery.from(Person.class);
cquery.multiselect(p.get("nachname"), cb.count(p));
cquery.groupBy(p.get("nachname"));
cquery.having(cb.gt(cb.count(p), 2));
TypedQuery<Object[]> query = em.createQuery(cquery);
List<Object[]> objects = query.getResultList();
```

JPA Criteria API / ORDER BY

```
// Alternative zu: SELECT p FROM Person p ORDER BY p.vorname DESC
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> cquery = cb.createQuery(Person.class);
Root<Person> p = cquery.from(Person.class);
cquery.select(p);
cquery.orderBy(cb.desc(p.get("vorname")));
TypedQuery<Person> query = em.createQuery(cquery);
List<Person> objects = query.getResultList();
```