Agenda

Query

JPAQL

Criteria API

SQL

Batch-Updates

Performance

Cache

Abfragemöglichkeiten

- Java Persistence Query Language (JPQL, JPAQL)
- Hibernate Query Language (HQL)
- Criteria API seit JPA 2.0
- Hibernate QBC (Query by Criteria)
- Hibernate QBE (Query by Example)
- SQL nativ

JPAQL: Query Language

- Die **Java Persistence Query Language (JPA-QL**, auch **JPQL)** ist an SQL angelehnt und unterstützt
  - Anfragen über Entities und ihre persistenten Zustände,
  - Updates oder
  - Lösch-Anweisungen.
- Bei der EJB QL in EJB 2.x habe es keine Operationen für UPDATE und DELETE.
- In JPA-QL sieht die einfachste Anfrage so aus:

```
select c from Customer c
```

JPAQL: Ergebnisse

- getResultList(): liefert Liste von Entitäten
- getSingleResult(): liefert einzige Entität
  - wirft Exception, falls nicht eindeutig

```
Query query = em.createQuery("select s from Spieler s")
List<Spieler> list = query.getResultList();
Spieler spieler =  (Spieler) query.getSingleResult();
```

Bedeutung von SELECT und FROM

- SELECT und FROM haben eine besondere Bedeutung
  - **SELECT** bestimmt den Typ der Objekte oder Werte, die zurückgeliefert werden
  - **FROM** gibt die so genannte Domäne einer Suche an

Anfrage	**Ergebnisliste enthält Objekte vom Typ**
select o from Customer o	com.jpa.Customer
select i from Invoice i	com.jpa.Invoice

GrOß- KlEiN-Schreibung

- Die GrOß- KlEiN-Schreibung **für Schlüsselwörter** ist ega

```
select c from Customer c
```

führt zum gleichen Ergebnis wie

```
Select c FROM Customer c
```

- Da JPA-QL eine getypte Sprache ist, spielt die Groß-/Kleinschreibung der Entities und Properties eine Rolle.

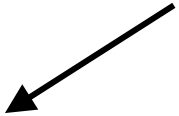
```
select c from CUSTOMER c
```
Fehler!

- Der Name der Entity bestimmt das Element name der Annotation @Entity.
  - Standardmäßig ist das der Name der Klasse.

Typ der Rückgabe von getResultList()

- Bei Anfragen der Bauart „select e from Entity e“ ist das Ergebnis eine Menge mit Elemente vom Typ der Entity.

```
Query query = em.createQuery(
 "select c from Customer c" );
```



```
List<Customer> resultList = query.getResultList();
```

- Es gibt aber auch SELECT-Anfragen, die keine Sammlungen bestimmter Entities geben.
  - Aggregat-Funktionen, wie COUNT(), liefern nur genau einen numerischen Wert.

Alias

- Ein Alias ist eine Referenz auf da zu erfragende Objekt, um es für andere Teile der Anfrage zu verwenden.

```
select c from Customer c
```

- Das ist eine Abkürzung für

```
select c from Customer AS c
```

- Der Alias ist dann die Variable, die in der WHERE-Klausel weitere Einschränkungen ermöglicht:

```
String jpaql = "select c from Customer c " +
               "where c.firstname = 'Bill'";
Query query = em.createQuery( jpaql );
System.out.println( query.getResultList() );
```

→ [Bill Karsen, Bill Clancy, Bill Ott, Bill Sommer, Bill King]

Allgemeine Form für Anfragen

- Die allgemeine Form für Anfragen ist die Folgende, wobei alles in eckigen Klammern optional ist.

```
select_statement :: =
    select_clause
    from_clause
    [where_clause]
    [groupby_clause] [having_clause]
    [orderby_clause]
```

- SELECT und FROM ist zwingend!
  - Bei Hibernates Sprache HQL kann SELECT entfallen.

JPAQL: Einfache Queries


```
select p from de.example.Person p
select pers from de.example.Person as pers
select pers from Person pers
```

JPAQL: Parameter

- Man kann in einer Query Parameter einsetzen
  - entweder mittels Indexparameter ?1, ?2
  - oder benannte Parameter (bevorzugt)

```
String jpql =  "select c from Customer c "
                + "where c.firstname like :name";
   Query query = em.createQuery( jpql );
   String s    = "Bill";
   query.setParameter( "name", s );
   System.out.println( query.getResultList() );
```

JPAQL: Parameter (2)

- bei mehrdeutigen Parametern kann Mapping angegeben werden
  - z. B. Java.util.Date
  - standardmäßig TimeStamp

```
String q1 =
"select p from Person p where p.birthday <= ?1";
   Query query = em.createQuery(q1) ;
// nur Datum setzen
query.setParameter (1, new Date(),
   TemporalType.DATE);
return query.getResultList();
```

JPAQL: Ergebnismenge einschränken

- Da die Ergebnismenge groß werden kann, lässt sie sich zwei Methoden einschränken:
  - Query setFirstResult(int startPosition)
  - Query setMaxResults(int maxResult)
- Da die Methoden die aktuelle Query liefern, kann man sie gut kaskadieren:

```
List<Customer> resultList = query.setFirstResult( 1 )
                                 .setMaxResults( 2 )
                                 .getResultList();
```

JPAQL: Named Queries

- **Named Queries können vorab definiert werden**

```
@Entity
@NamedQuery( name = "customer.findAll",
             query = "select c from Customer c" )
public class Customer
{ ...      }
```

- **Abfrage an anderer Stelle**

```
Query namedQuery = em.createNamedQuery("customer.findAll" );
println( namedQuery.getResultList() );
```

@NamedQueries

- Mehrere @NamedQuery-Elemente werden in einen Container @NamedQueries gesetzt. Auch dies ist auf Klassen- oder Paketebene möglich.

```
@NamedQueries( {
 @NamedQuery( name = "customer.findAll",
              query = "select c from Customer c" ),
 @NamedQuery( name = "customer.findByName",
              query = "select c from Customer c" +
                      " where c.firstname = :name" )
} )
```

Query mit Parameter


```
Query namedQuery = em.createNamedQuery(

"customer.findByName" );

namedQuery.setParameter( "name",
"Laura" );

println( namedQuery.getResultList() );
```

- Ohne zugewiesenen Parameterwert für „name“ gibt es bei der Ausführung einen Fehler!

JPAQL: WHERE Clause

- WHERE-Bedingung kann Literale enthalten:
  - Strings: 'Meier' ,'Wolf' ,'w',
  - Ganzzahlige Werte: 13, -46
  - Kommazahlen:-24.876, 5E3
  - Wahrheitswerte: true, false

- Beispiele

```
select s from Spieler s where s.tore > 30
select s from Spieler s where s.position = 'Torwart'
```

JPAQL: Operatoren

- Mathematische Operatoren: +, -,*,/.
- Binäre Operatoren: =, >=, <=, <>, !=, like
- Logische Operatoren: and, or, not
- in, not in, between, is null, is not null, is empty, is not empty, member of, not member of, exists, not exists

```
select p.name from Person p
   where p.city IN  ('München', 'Stuttgart')
select p.name from Person p
   where p.alter between 30 AND  40
```

JPAQL: Subqueries

- Unterabfrage, eingebettet in WHERE oder HAVING

```
select p from Person p
   where p.city IN
        (select o.city from Orte o
              where o.plz between '12345'  and '54321')
```

JPAQL: Funktionen

- String Funktionen:
  - LOWER, UPPER, TRIM, CONCAT, LENGTH, LOCATE, SUBSTRING
- Berechnung:
  - Abs, SQRT, mod (Rest bei Division)
- Zeit / Datum:
  - CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP
- Aggregation:
  - COUNT, MAX, MIN, AVG, SUM

JPAQL: Wildcards

- Bei Stringvergleichen sind folgende spezielle Zeichen erlaubt:
- % jede Folge von Zeichen
- _ ein einzelnes Zeichen
- \%, _ Escapesyntax

```
select p from Person p where p.name like '%meier'
select s from Spieler s where s.position like  'Mit%'
```

JPAQL: Sortieren

- Defaultist aufsteigend

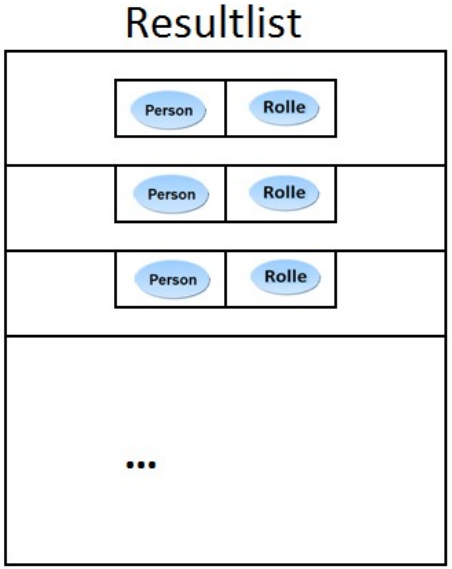
```
...  order by p.name
...  order by p.name DESC
...  order by p.nachname ASC,  p.vorname DESC
```

JPAQL: Joins

- liefern Kombinationen der beteiligten Entities
- **verschiedene Möglichkeiten:**
  - inner join / join
    - 1:1-Zuordnung. Elemente ohne Zuordnung sind ausgeschlossen
  - left outer join / left join
    - 1:1-Zuordnung. inklusive Elemente ohne Zuordnung der linken Tabelle
  - right outer join / right join
    - inklusive Elemente ohne Zuordnung der rechten Tabelle
  - full join
    - kartesisches Produkt
  - fetch join in der from Klausel
  - theta join in der where Klausel
  - implizite Assoziation join

JPAQL: Join / Inner Join

- JOIN (alternativ INNER JOIN)
- bei Tupeln enthält Resultlist Object-Arrays (Object[])
- IN-Operator (aus EJBQL): Zugriff auf Elemente einer Collection

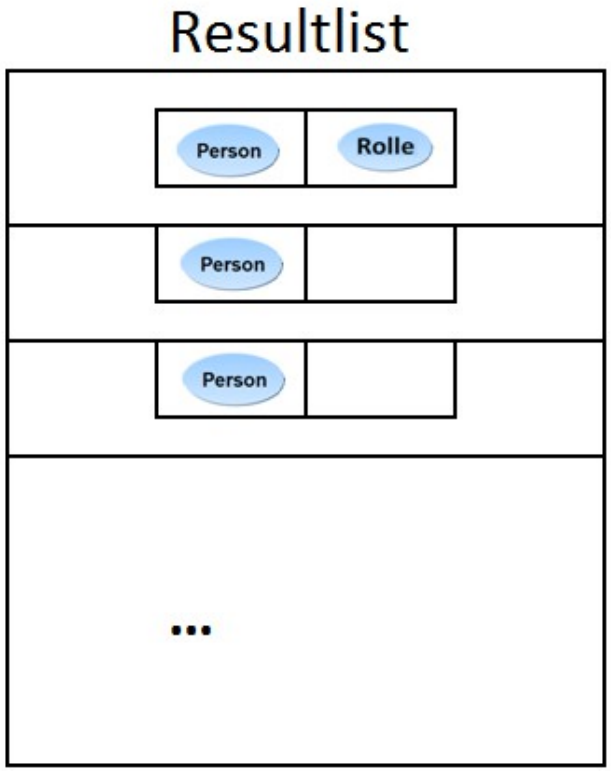



```
select p, r from Person p
  join p.roles r
select p, r from Person p
  inner join p.roles r
// von EJBQL
select p, r from Person as p,
  in(p.roles) r
```

JPAQL: Left Join

- LEFT JOIN liefert auch Elemente der Masterseite, für die keine Werte der Detailseite existieren
  - null wird gesetzt

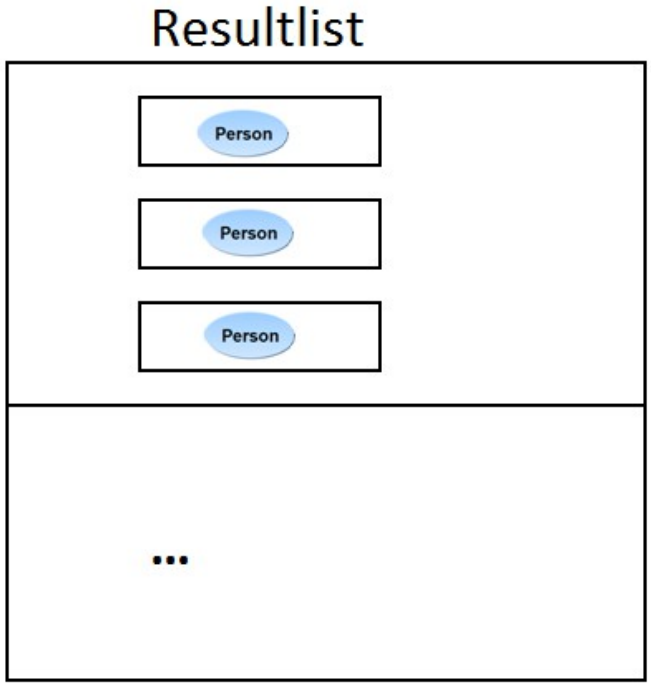
```
// jeweils 2-Tupel
  select p,  a from Person p
   left join p.addresses a
```



JPAQL: Eager Fetching

- Abhängige Objekte werden durch Query initialisiert
- Resultlist enthält Personen mit initialisierten Adressen

```
select p from Person p
left join fetch p.addresses
  where p.id = 5
```



JPAQL: Theta-style join

- Theta-style join
  - Kartesisches Produkt mit join in der where Klausel
  - Join für zwei Klassen, die keine direkte Assoziation besitzen

```
select pers, tr from Person pers, Trainer tr
  where pers.name = tr.name
```

JPAQL: Distinct

- Um doppelte Einträge zu entfernen, nutzt man **distinct**:

```
select distinct c.firstname
from   Customer c
where  c.firstname like '%a'
```

→ [Julia, Laura, Sylvia]

```
select distinct p FROM Person p
select distinct p.name FROM Person p
  LEFT JOIN p.adressen a
```

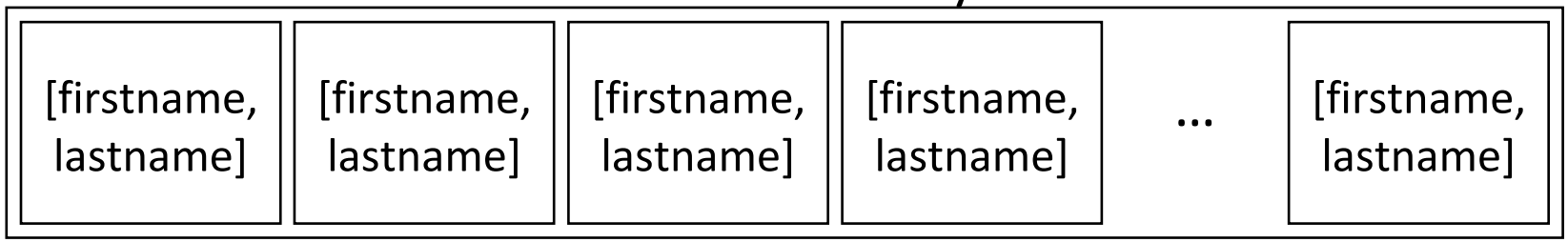
JPAQL: Report Queries

- Der JPA-QL-Ausdruck

  ```
  select c.firstname from Customer c
  ```

  liefert eine Liste mit einzelnen Strings. (Kein Caching, kein Overhead)
- JPA-QL erlaubt aber auch eine Liste von Properties:

  ```
  select c.firstname, c.lastname from Customer c
  ```
- Das Ergebnis von getResultList() ist ein Sammlung vieler kleiner Arrays.
  - Die Anzahl Elemente in diesem Array in unserem Fall 2.



JPAQL: Report Queries


```
String s =
  "select c.firstname, c.lastname from Customer c";
Query query = em.createQuery( s );
List<Object[]> list = query.getResultList();
for ( Object[] objects : list )
  System.out.printf( "Firstname=%s, Lastname=%s%n",
                     objects[0], objects[1] );
```

Die Ausgabe ist:

Firstname=Laura, Lastname=Steel

Firstname=Susanne, Lastname=King

Firstname=Anne, Lastname=Miller

…

JPAQL: Report Queries
Konstruktor-Ausdrücke in SELECT

- Was für Elemente eine Query liefert, kann
- unterschiedlich sein:
  - Eine Liste von Entities
  - Ein skalarer Wert
  - Eine Liste von Array-Objekten
- Statt der Array-Objekte kann man sich Behälterobjekte zurückgeben lassen.

Ruft Konstruktor SimplePerson(String,String) auf

```
select
new com.tutego.jpa.entitymanager.SimplePerson(
                                c.firstname, c.lastname )
from Customer c
```

JPAQL: Report Queries

Behälterklasse SimplePerson

```
package com.tutego.jpa.entitymanager;

public class SimplePerson
{
  private String name;

  public SimplePerson( String a, String b )
  {
    name = a + " " + b;
  }

  @Override
  public String toString()
  {
    return name;
  }
}
```

JPAQL: Report Queries
Anfrage


```
String s = "select new " +
   "com.tutego.jpa.entitymanager.SimplePerson(" +
   "c.firstname, c.lastname ) " +
   "from Customer c";
Query query = em.createQuery( s );
List<SimplePerson> list = query.getResultList();
for ( SimplePerson p : list )
  System.out.println( p );
```

→

Laura Steel

Susanne King

…

JPAQL: Aggregationen

- Zusammenfassung von Daten und Fakten
  - Gruppierungen mittels group by
  - having by (where-Klausel beschränkt auf group by Felder)
- Folgende Aggregation Funktionen werden unterstützt:

```
avg(...),    sum (...),  min ( . . . ) , max (...)

count(*),    count(...), count   (distinct ...),
   count(all ...)
```

JPAQL: Aggregationen Beispiele


```
Integer count =
(Integer)  entityManager.createQuery(
"select count(*) from Person").getSingleResult() ;


Iterator results = entityManager.createQuery(
   "select person.name,  count(person),  avg(person.age)
   from Person p where p.status > 1
   group by p.name").getResultList().iterator();
while  ( results.hasNext()   ) {
Object[]  row =  (Object[]) results.next();
   Integer avg =  (Integer) row[2];
. . .
}
```

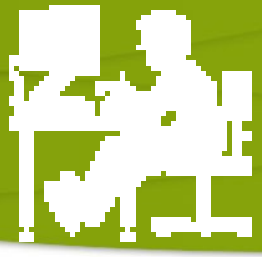
Polymorphie in Queries

- Liefert alle Objekte im Persistenz-Kontext:

```
String s = "select o from java.lang.Object o";
Query query = em.createQuery( s );
System.out.println( query.getResultList() );
```

- In der Praxis wird man eher einen Basistyp wählen, der auch eine Schnittstelle sein kann.

Aufgabe



- Aufgabe 11:
- Abfragen (JPAQL)

HQL: Hibernate Query Language

- Abfragesprache
- nicht case sensitive
- aber : Javaklassen und Properties entsprechend korrekt angeben
  - Objektorientiert
    - Abfragen von Objekten über deren Vererbungsbeziehung
- "from user"
  - HQL : Abfrage der Entities vom Typ User oder einem Subtyp
  - SQL : Abfrage der Daten aus der Tabelle user

```
Query query = session.createQuery("from User");
List result = query.list();
```

HQL: Syntax


```
[select  ...]   from ...   [where  ...]
[group by  ...   [having  . . . ] ]   [order by ...]
```

Einfache HQL Queries

```
from de.example.Person
from de.example.Person as pers
from Person pers
```

HQL Queries mit Parametern


```
Query q = session.createQuery("from User u where
   u.firstName = :fname");
q.setString("fname", "Pan");
List result = q.list();
```

HQL: WHERE Clause


```
from Person as pers WHERE pers.age = 1

from Person as pers WHERE pers.firstname =  'Peter'

from Person as  pers WHERE pers.firstname like 'Pet%'


from Person as pers WHERE lower(pers.firstname)   like
   'pe%'
```

HQL: Operatoren

- Mathematische Operatoren: +, -, * /
- Binäre Operatoren: =, >=, <=, <>, !=, like
- Logische Operatoren: and, or, not
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of .
- Zeit / Datum: current_date () , current_time (),...
- exist(), elements() ...

HQL: Sortieren

- Wie SQL ORDER BY

```
Query query = session.createQuery(
"from User u order by u.name asc,  u.age desc") ;
   List result = q.list();
```

HQL: Joins

- liefern Kombinationen der beteiligten Entities
- **verschiedene Möglichkeiten:**
  - Normales join in der from Klausel
  - fetch join in der from Klausel
  - theta join in der where Klausel
  - implizites join über Assoziation

HQL: Normales (einfaches) Join

- Join in der from Klausel
  liefert Object-Array als Ergebnis

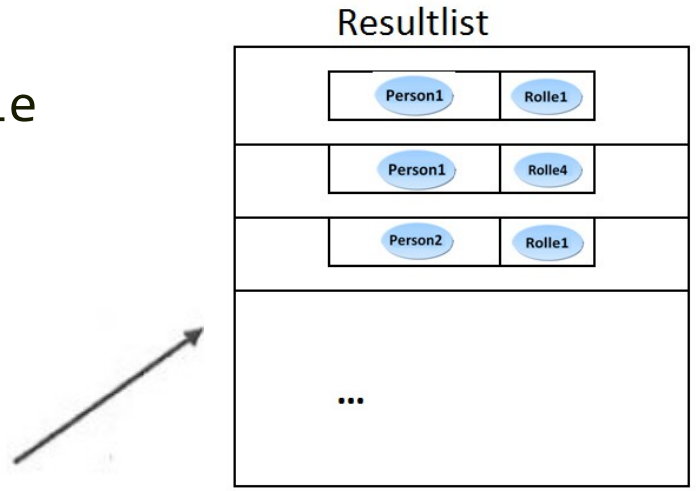
```
from de.example.Person as pers join pers.roles

from de.example.Person as pers

inner join pers.addresses as adr
   left outer join pers.roles as role

from de.example.Person as pers

left join pers.roles as role
```



Ergebnisliste enthält Object[] als Elemente

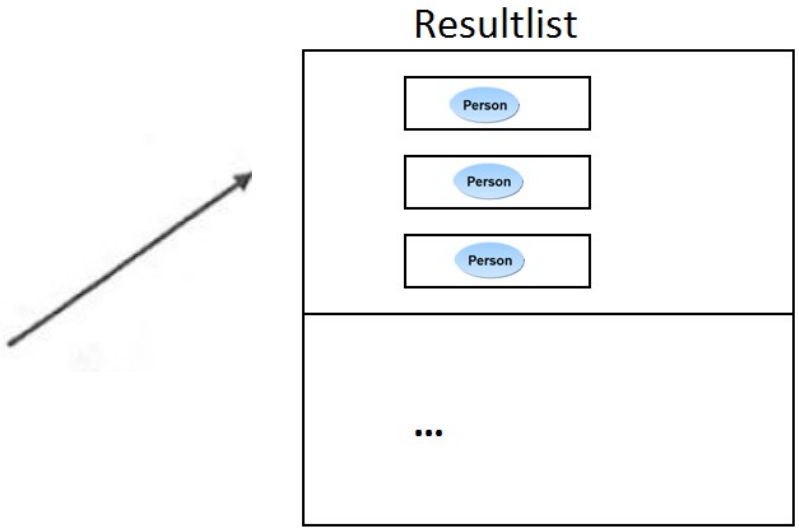
HQL: Eager Fetching

- Abhängige Objekte werden durch Query initialisiert
- Eager Fetching Strategie des Mappings (ob select oder join fetch) wird bei HQL ignoriert
- Explizites fetch join in HQL möglich Resultlist

```
from Person p left join fetch p.addresses
   where p.id
```

- Liste von Personen
- jede Person hat bereits eine initialisierte Collection von Adressen

=



HQL: Theta-style join

- Theta-style join
  - Kartesisches Produkt mit join in der where Klausel
  - Join für zwei Klassen, die keine direkte Assoziation besitzen

```
from Person pers, Trainer tr
  where pers.name = tr.name
```

Join bei impliziten Assoziationen

- Navigation über Zu-Eins-Beziehungen mittels Punktoperator: "."
- Nur bei HQL möglich

```
from Addresse adr
where adr.person.firstname like  '%abd%'
```

- nicht bei Collections (Zu-N-Beziehungen)

~~from Person p where~~
~~p.addresses.street like like 'Haupt%'~~

```
from Person p join p.addresses adr
where adr.street like 'Haupt%'
```

HQL: Distinct


```
select distinct(p)
from Person p
join fetch p.addresses
```

HQL: Select Klausel

- SELECT ist optional
- Lädt nur Objekte die in der Query als Ergebnis angegeben werden

```
select pers.rolle from de.example.Person as pers

select rolle from de.example.Person as pers
   inner join pers.rolle as rolle

select new User(person, rolle)

from de.example.Person person
   join person.rolle as rolle
```

HQL: Report Queries

- Geben sowohl Entitäten, als auch einzelne Eigenschaften zurück
- Ergebnis als Object-Array in java.util.List

```
Iterator i = session.createQuery(
"select pers.firstname,  pers.lastname,
  rolle.roleType "
+ "from Person pers join pers.roles rolle "
+ "where rolle.roleType > 2").list().iterator();
while  ( i.hasNext()) {
Object[]  row =  (Object []) i.next();
  String firstname =  ((String)) row[0];
```

HQL: Aggregationen

- Folgende Aggregation Funktionen werden unterstützt:
  - avg(...), sum(...), min(...), max (...)
  - count(*), count(...), count (distinct ...), count(all . . .)

HQL: Aggregationen Beispiele


```
Integer count =  (Integer)session.createQuery(
        "select count(*) from Person").uniqueResult();
Iterator results = session.createQuery(
   "select person.firstname,  count(person),
   avg(person.age)
   from Person p where p.status > 1
   group by p.firstname").list().iterator();
while  ( results.hasNext()   ) {
   Object[]  row =  (Object[]) results.next();
   Integer avg =  (Integer) row[2];
. . .
}
```

HQL: Ergebnismenge einschränken


```
Query q = s.createQuery("from ...");
q.setFirstResult(20) ;
q.setMaxResults(10);
List result = q.list();
```

HQL: Benannte Queries

- Queries können als Named Queries vorab definiert werden

```
<query name="Person.getByPLZ">
   <![CDATA[
        from Person p join p.addresses adr
        where adr.zipcode like ?
   ]]>
</query>


// Abfrage an anderer Stelle
Query query =
   session.getNamedQuery("Person.getByPLZ") ;
query.setParameter(0,"12345");
```