ABOUT HEBERT COELHO

Senior Java Development, with 4 certifications and a published book about JSF (portuguese only). Founder of the blog uaiHebert.com visited from more than 170 different countries.

Ultimate JPA Queries and Tips List - Part 1

Posted by: Hebert Coelho in Enterprise Java July 9th, 2012

There are several JPAs "how to" that we can find on the internet, here in this blog, that teaches how to do several tasks with JPA.

Usually I see some people asking questions about Queries with JPA; usually to answer this kind of questions several links are provided trying to find a solution to the question.

Until today I could not find a blog post that puts together a good subject about queries with JPA, tips about performance/utilization, source code to download...

Today we will see:

- Model classes and a class that will generate database data
- Find method; Use the getReference method to get a better performance, displaying query parameters in the console with the log4j
- JPQL: Queries with simple parameters or objects, Joins, Order By, Navigating through relationships
- JPQL: Functions: AVG, COUNT, MAX, MIN, TRIM, SUM, UPPER, LOWER, MOD, LENGHT, SQRT; Using HAVING, GROUP BY
- JPQL: Filtering Conditions: LIKE, IN, DISTINCT, EMPTY, BETWEEN, NULL, MEMBER OF, EXISTS (Subqueries), ANY, ALL, SOME, CONCAT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCATE, SIZE, SUBSTRING
- JPA: NamedQuery, querying with dates, warnings about the getSingleResult method
- JPA: NativeQuery, NamedNativeQuery
- JPA: Complex Native Queries
- JPA: Optimizing queries with EJB
- JPA: Pagination
- JPA: Database Hints
- JPA: Creating a object from a query
- JPQL: Bulk Update and Delete
- JPA: Criteria

You will see that in every main class we will invoke the method "*CodeGenerator.generateData()*". This class method only creates data in the database; with this data our queries will find the correct results.

In the last page of this post you will find the link to download the source code of this post.

In this post we will use JPA 2.0 with Hibernate as the provider. The database will be the HSQLDB and will be attached to this project. You can download the source code and run the project without the need of any extra configuration. We will not talk about how to set up the HSQLDB because the focus of this post is how to query data of the database.

This post will not use best practices of development in some points. The focus of this post will be to display how the JPA queries works.

NEWSLETTER

118900 insiders are alrea
weekly updates and complime
whitepapers!
Join them now to ga
access to the latest news ir
as well as insights about Andr
Groovy and other related tech

Email address:

Your email address

Sign up

JOIN US

With **1,**
unique v
500 a
placed a
related s
Constan
lookout
encoura
So If yo
unique and interesting content th
check out our **JCG** partners prog
be a **guest writer** for Java Code
your writing skills!

Model classes and a class that will generate database data

```
01 package com.model;
02
03 import java.util.ArrayList;
04 import java.util.List;
05
06 import javax.persistence.CascadeType;
07 import javax.persistence.Entity;
08 import javax.persistence.GeneratedValue;
09 import javax.persistence.GenerationType;
10 import javax.persistence.Id;
11 import javax.persistence.JoinColumn;
12 import javax.persistence.OneToMany;
13 import javax.persistence.OneToOne;
14
15 @Entity
16 public class Person {
17
18  @Id
19  @GeneratedValue(strategy = GenerationType.AUTO)
20  private int id;
21
22  private String name;
23  private int age;
24
25  public Person() {
26
27  }
28
29  public Person(String name, int age) {
30   this.name = name;
31   this.age = age;
32  }
33
34  @OneToMany(mappedBy = 'person', cascade = CascadeType.ALL)
35  private List<Dog> dogs;
36
37  @OneToOne(cascade = CascadeType.ALL)
38  @JoinColumn(name='address_id')
39  private Address address;
40
41  public int getId() {
42   return id;
43  }
44
45  public void setId(int id) {
46   this.id = id;
47  }
48
49  public String getName() {
50   return name;
51  }
52
53  public void setName(String name) {
54   this.name = name;
55  }
56
57  public int getAge() {
58   return age;
59  }
60
61  public void setAge(int age) {
62   this.age = age;
63  }
64
65  public List<Dog> getDogs() {
66   if (dogs == null) {
67    dogs = new ArrayList<Dog>();
68   }
69
70   return dogs;
71  }
72
73  public void setDogs(List<Dog> dogs) {
74   this.dogs = dogs;
75  }
76
77  public Address getAddress() {
78   return address;
79  }
80
81  public void setAddress(Address address) {
82   this.address = address;
83  }
84
85  @Override
86  public int hashCode() {
87   return getId();
88  }
89
90  @Override
91  public boolean equals(Object obj) {
92   if (obj instanceof Person) {
93    Person person = (Person) obj;
94    return person.getId() == getId();
95   }
96
97   return false;
98  }
99 }
```

```
01 package com.model;
02
03 import java.util.Date;
04
05 import javax.persistence.Entity;
06 import javax.persistence.GeneratedValue;
07 import javax.persistence.GenerationType;
```

```
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Dog {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;

 private String name;
 private double weight;

 @Temporal(TemporalType.TIMESTAMP)
 private Date dateOfBirth;

 public Dog() {

 }

 public Dog(String name, double weight, Date dateOfBirth) {
  this.name = name;
  this.weight = weight;
  this.dateOfBirth = dateOfBirth;
 }

 public static void main(String[] args) {

  System.out.println(new Date());
 }

 @ManyToOne
 private Person person;

 public int getId() {
  return id;
 }

 public void setId(int id) {
  this.id = id;
 }

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public double getWeight() {
  return weight;
 }

 public void setWeight(double weight) {
  this.weight = weight;
 }

 public Date getDateOfBirth() {
  return dateOfBirth;
 }

 public void setDateOfBirth(Date dateOfBirth) {
  this.dateOfBirth = dateOfBirth;
 }

 public Person getPerson() {
  return person;
 }

 public void setPerson(Person person) {
  this.person = person;
 }

 @Override
 public int hashCode() {
  return getId();
 }

 @Override
 public boolean equals(Object obj) {
  if (obj instanceof Dog) {
   Dog dog = (Dog) obj;
   return dog.getId() == getId();
  }

  return false;
 }
}
```

```
package com.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Address {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;

 private String streetName;
 private int houseNumber;

 public Address() {

```

```
 }

 public Address(String streetName, int houseNumber) {
  this.streetName = streetName;
  this.houseNumber = houseNumber;
 }

 public int getId() {
  return id;
 }

 public void setId(int id) {
  this.id = id;
 }

 public String getStreetName() {
  return streetName;
 }

 public void setStreetName(String streetName) {
  this.streetName = streetName;
 }

 public int getHouseNumber() {
  return houseNumber;
 }

 public void setHouseNumber(int houseNumber) {
  this.houseNumber = houseNumber;
 }

 @Override
 public int hashCode() {
  return getId();
 }

 @Override
 public boolean equals(Object obj) {
  if (obj instanceof Address) {
   Address address = (Address) obj;
   return address.getId() == getId();
  }

  return false;
 }
}
```

We got some basic classes with relationships unidirectional and bidirectional. These relationships will help us to manipulate all type of queries that we will be executing.

To generate the database data we have the class bellow:

```
001 package com.main;
002
003 import java.text.ParseException;
004 import java.text.SimpleDateFormat;
005 import java.util.Date;
006
007 import javax.persistence.EntityManager;
008 import javax.persistence.EntityManagerFactory;
009 import javax.persistence.Persistence;
010
011 import com.model.Address;
012 import com.model.Dog;
013 import com.model.Person;
014
015 public class CodeGenerator {
016  private static EntityManagerFactory emf;
017  private static EntityManager em;
018
019  public static final String PERSON01_NAME = 'John';
020  public static final String PERSON02_NAME = 'Mary';
021  public static final String PERSON03_NAME = 'Anna';
022  public static final String PERSON04_NAME = 'Joseph';
023  public static final String PERSON05_NAME = 'Mark';
024  public static final String PERSON06_NAME = 'I will not have any relationship';
025
026  public static void startConnection() {
027   emf = Persistence.createEntityManagerFactory('JpaQuery');
028   em = emf.createEntityManager();
029   em.getTransaction().begin();
030  }
031
032  public static void closeConnection() {
033   em.getTransaction().commit();
034   emf.close();
035  }
036
037  public static void generateData() {
038   int year = 1995;
039   int month = 1;
040   int day = 10;
041
042   Dog dog01 = new Dog('Yellow', 3.5d, createNewDate(day, month, year));
043   Dog dog02 = new Dog('Brown', 8.5d, createNewDate(++day, ++month, ++year));
044   Dog dog03 = new Dog('Dark', 15.5d, createNewDate(++day, ++month, ++year));
045   Dog dog04 = new Dog('Kaka', 4.3d, createNewDate(++day, ++month, ++year));
046   Dog dog05 = new Dog('Pepe', 8.2d, createNewDate(++day, ++month, ++year));
047   Dog dog06 = new Dog('Casillas', 6.1d, createNewDate(++day, ++month, ++year));
048   Dog dog07 = new Dog('Fish', 6.7d, createNewDate(++day, ++month, ++year));
049   Dog dog08 = new Dog('Lion', 3.1d, createNewDate(++day, ++month, ++year));
050   Dog dog09 = new Dog('Cat', 5.5d, createNewDate(++day, ++month, ++year));
051   Dog dog10 = new Dog('Java', 21.7d, createNewDate(++day, ++month, ++year));
052   Dog dog11 = new Dog('JSF', 23.65d, createNewDate(++day, ++month, ++year));
053   Dog dog12 = new Dog('VRaptor', 24.0d, createNewDate(++day, ++month, ++year));
054   Dog dog13 = new Dog('Ferrari', 3.7d, createNewDate(++day, ++month, ++year));
055   Dog dog14 = new Dog('Porshe', 1.33d, createNewDate(++day, ++month, ++year));
056   Dog dog15 = new Dog('Bike', 4.44d, createNewDate(++day, ++month, ++year));
057   Dog dog16 = new Dog('Rambo', 5.44d, createNewDate(++day, ++month, 2015));
```

```
  Dog dog17 = new Dog('Terminator', 3.88d, createNewDate(++day, ++month, 2016));
  Dog dog18 = new Dog('John McClan', 3.88d, createNewDate(++day, ++month, 2016));

  Person person01 = new Person(PERSON01_NAME, 33);
  person01.getDogs().add(dog01);
  person01.getDogs().add(dog02);
  person01.getDogs().add(dog03);
  person01.setAddress(new Address('Street A', 30));
  dog01.setPerson(person01);
  dog02.setPerson(person01);
  dog03.setPerson(person01);

  Person person02 = new Person(PERSON02_NAME, 27);
  person02.getDogs().add(dog04);
  person02.getDogs().add(dog05);
  person02.getDogs().add(dog06);
  person02.setAddress(new Address('Street B', 60));
  dog04.setPerson(person02);
  dog05.setPerson(person02);
  dog06.setPerson(person02);

  Person person03 = new Person(PERSON03_NAME, 7);
  person03.getDogs().add(dog07);
  person03.getDogs().add(dog08);
  person03.getDogs().add(dog09);
  person03.setAddress(new Address('Street B', 90));
  dog07.setPerson(person03);
  dog08.setPerson(person03);
  dog09.setPerson(person03);

  Person person04 = new Person(PERSON04_NAME, 43);
  person04.getDogs().add(dog10);
  person04.getDogs().add(dog11);
  person04.getDogs().add(dog12);
  person04.setAddress(new Address('Street C', 120));
  dog10.setPerson(person04);
  dog11.setPerson(person04);
  dog12.setPerson(person04);

  Person person05 = new Person(PERSON05_NAME, 70);
  person05.getDogs().add(dog13);
  person05.getDogs().add(dog14);
  person05.getDogs().add(dog15);
  person05.getDogs().add(dog16);
  person05.setAddress(new Address('Street D', 150));
  dog13.setPerson(person05);
  dog14.setPerson(person05);
  dog15.setPerson(person05);
  dog16.setPerson(person05);

  Person person06 = new Person(PERSON06_NAME, 45);

  em.persist(person01);
  em.persist(person02);
  em.persist(person03);
  em.persist(person04);
  em.persist(person05);
  em.persist(person06);

  em.persist(dog17);
  em.persist(dog18);

  em.flush();
 }

 private static Date createNewDate(int day, int month, int year) {
  SimpleDateFormat formatter = new SimpleDateFormat('dd/MM/yyyy');
  try {
   return formatter.parse('' + day + '/' + month + '/' + year);
  } catch (ParseException e) {
   e.printStackTrace();
   return null;
  }
 }

 public static EntityManager getEntityManager() {
  return em;
 }
}
```

Find method; Use the getReference method to get a better performance, displaying query parameters in the console with the log4j

The find method usually is invoked before we execute some change in the database like update some object attribute, relationship or to delete it.

Bellow you will find codes that uses the find method:

```
package com.main;

import javax.persistence.EntityManager;

import com.model.Address;
import com.model.Person;

public class Page03 {
 public static void main(String[] args) {
  CodeGenerator.startConnection();

  CodeGenerator.generateData();

  EntityManager em = CodeGenerator.getEntityManager();

  Person person = em.find(Person.class, 1);

  int addressId = 2;

  // usually we send an id or a detached object from the view
  setAddressToOtherPerson(em, person, addressId);
```

```
22
23   int personId = 4;
24
25   // usually we send an id or a detached object from the view
26   deletePerson(em, personId);
27
28   CodeGenerator.closeConnection();
29  }
30
31  private static void setAddressToOtherPerson(EntityManager em, Person person, int addressId) {
32   Address address = em.find(Address.class, addressId);
33   person.setAddress(address);
34   em.merge(person);
35   em.flush();
36  }
37
38  private static void deletePerson(EntityManager em, int personId) {
39   Person savedPerson = em.find(Person.class, personId);
40   em.remove(savedPerson);
41   em.flush();
42  }
43 }
```

Notice that the methods "*setAddressToOtherPerson*" and "*deletePerson*" uses the find method only to update a reference or to delete an object.

The find() method has an optimized query functionality that will search for an object in the Persistence Context, if it does not find the object it will query the database to bring the data. If you got a relationship annotated with EAGER (e.g.: "*@OneToMany(fetch=FetchType.EAGER)*") the find method will bring these objects from the database. Notice that there is no need to bring all this data from the database for simple tasks like a delete of reference update.

The EntityManager has a specific method that helps with these simples tasks. The EntityManager will do a simple query like "*select id from Person p where p.id = :personId*". We will have a faster and smaller query.

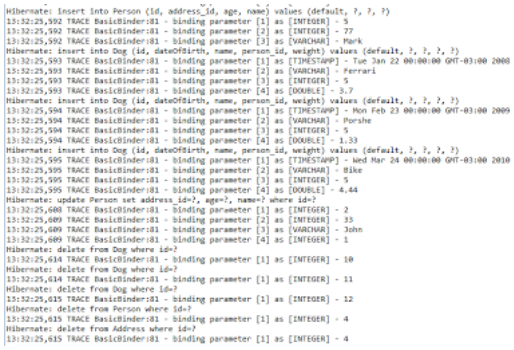
Bellow you can see how we will use the getReference:

```
01 package com.main;
02
03 import javax.persistence.EntityManager;
04
05 import com.model.Address;
06 import com.model.Person;
07
08 public class Page03 {
09  public static void main(String[] args) {
10   CodeGenerator.startConnection();
11
12   CodeGenerator.generateData();
13
14   EntityManager em = CodeGenerator.getEntityManager();
15
16   Person person = em.find(Person.class, 1);
17
18   int addressId = 2;
19
20   // usually we send an id or a detached object from the view
21   setAddressToOtherPerson(em, person, addressId);
22
23   int personId = 4;
24
25   // usually we send an id or a detached object from the view
26   deletePerson(em, personId);
27
28   CodeGenerator.closeConnection();
29  }
30
31  private static void setAddressToOtherPerson(EntityManager em, Person person, int addressId) {
32   Address address = em.getReference(Address.class, addressId);
33   person.setAddress(address);
34   em.merge(person);
35   em.flush();
36   System.out.println('Merged');
37  }
38
39  private static void deletePerson(EntityManager em, int personId) {
40   // usually is find or merge
41   Person savedPerson = em.getReference(Person.class, personId);
42   em.remove(savedPerson);
43   em.flush();
44   System.out.println('Deleted');
45  }
46 }
```

With the method "*getReference*" you will query only for the object ID, you will save some database traffic.

Bellow you will find the lo4j.properties configuration needed to display the JPA queries parameters in the console. Usually when we invoke a query using the Hibernate, the Hibernate will format the query with "?" instead of using the real value. With the code bellow you will be able to see the query parameters:

```
01 # Direct log messages to stdout
02 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
03 log4j.appender.stdout.Target=System.out
04 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
05 log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
06
07 # Root logger option
08 log4j.rootLogger=ERROR, stdout
09
10 # Hibernate logging options (INFO only shows startup messages)
11 log4j.logger.org.hibernate=ERROR
12
13 # Log JDBC bind parameter runtime arguments
14 log4j.logger.org.hibernate.type=TRACE
```



If you want to deactivate the log you just need to comment the lo4j.properties last line with the # symbol, and to set the show_log configuration in the "persistence.xml" to false.

JPQL: Queries with simple parameters or objects, Joins, Order By, Navigating through relationships

To do a basic query you just need to run a command like this: "select d from Dog d". One thing that you always need to keep in your mind is that: to do this kind of query we use *JPQL* and not the regular *SQL*.

The advantage of using JPQL is that it is very similar to SQL and it is portable. You may use the same query in every database without a problem.

Never concatenate your query with a string. If you do a query like this: "select p from Person p where p.name" + person.getName(), you can be sure that hackers will love it. They use this kind of code to do an attack named "SQL Injection" (or JPQL Injection). The way to avoid this kind of attack is adding parameters to your query like we will see bellow.

You will see bellow several ways to do a query:

```
001 package com.main;
002
003 import java.util.List;
004
005 import javax.persistence.EntityManager;
006 import javax.persistence.Query;
007
008 import com.model.Dog;
009 import com.model.Person;
010
011 public class Page04 {
012  public static void main(String[] args) {
013   CodeGenerator.startConnection();
014
015   CodeGenerator.generateData();
016
017   EntityManager em = CodeGenerator.getEntityManager();
018
019   List<Dog> dogs = listAllDogs(em);
020
021   for (Dog dog : dogs) {
022    System.out.println(dog.getName());
023   }
024
025   Person person03 = findPersonByName(em, CodeGenerator.PERSON03_NAME);
026   System.out.println(person03.getName());
027
028   Person person01 = new Person();
029   person01.setId(1);
030
031   Person savedPerson = findPersonByPersonObject(em, person01);
032   System.out.println(savedPerson.getName());
033
034   List<Dog> dogsByWeight = listAllDogsOrderingByWeight(em);
035
036   for (Dog dog : dogsByWeight) {
037    System.out.println(dog.getWeight());
038   }
039
040   String addressName = findAddressNameOfPerson(em, CodeGenerator.PERSON04_NAME);
041   System.out.println('Person 04 address is: ' + addressName);
042
043   Person person02 = findPersonByNameWithAllDogs(em, CodeGenerator.PERSON02_NAME);
044
045   for (Dog dog : person02.getDogs()) {
046    System.out.println('Person 02 Dog: ' + dog.getName());
047   }
048
049   Person person05 = findPersonByNameThatMayNotHaveDogs(em, CodeGenerator.PERSON06_NAME);
050   System.out.println('Is the list of the Dogs from the Person 05 empty? ' +
person05.getDogs().size());
051
052   CodeGenerator.closeConnection();
053  }
054
055  /**
056   * Easiest way to do a query
057   */
058  @SuppressWarnings('unchecked')
059  private static List<Dog> listAllDogs(EntityManager em) {
060   Query query = em.createQuery('select d from Dog d', Dog.class);
061
062   return query.getResultList();
063  }
064
065  /**
066   * Easiest way to do a query with parameters
067   */
068  private static Person findPersonByName(EntityManager em, String name) {
069   Query query = em.createQuery('select p from Person p where name = :name', Person.class);
070   query.setParameter('name', name);
```

```
071   return (Person) query.getSingleResult();
072  }
073
074  /**
075   * Executes a query that has as parameter an object
076   */
077  private static Person findPersonByPersonObject(EntityManager em, Person person) {
078   Query query = em.createQuery('select p from Person p where p = :person');
079   query.setParameter('person', person);
080   return (Person) query.getSingleResult();
081  }
082
083  /**
084   * Query that will list all dogs with an order
085   */
086  @SuppressWarnings('unchecked')
087  private static List<Dog> listAllDogsOrderingByWeight(EntityManager em) {
088   Query query = em.createQuery('select d from Dog d order by d.weight desc', Dog.class);
089
090   return query.getResultList();
091  }
092
093  /**
094   * Query that get only a field instead a complete class object
095   */
096  private static String findAddressNameOfPerson(EntityManager em, String name) {
097   Query query = em.createQuery('select p.address.streetName from Person p where p.name = :name');
098   query.setParameter('name', name);
099   return (String) query.getSingleResult();
100  }
101
102  /**
103   * Query that will fetch a lazy relationship Be carefull, with this kind of
104   * query only those who have the relationship will come in the result
105   */
106  private static Person findPersonByNameWithAllDogs(EntityManager em, String name) {
107   Query query = em.createQuery('select p from Person p join fetch p.dogs where p.name = :name',
Person.class);
108   query.setParameter('name', name);
109   return (Person) query.getSingleResult();
110  }
111
112  /**
113   * With this query will will bring results that may not have arelationship
114   */
115  private static Person findPersonByNameThatMayNotHaveDogs(EntityManager em, String name) {
116   Query query = em.createQuery('select p from Person p left join fetch p.dogs where p.name =
:name', Person.class);
117   query.setParameter('name', name);
118   return (Person) query.getSingleResult();
119  }
120 }
```

About the code above:

- Each query is invoked like "*em.createQuery("HHH", HHH.class)*" with the specific query text and return class. You can define a return class, e.g. Person.class. The Person.class parameter will indicate to the JPA which is the return object.
- We can use basic attributes as query parameters like "*p.name = :name*" or an object "*p = :person*". If you use an object the JPA will compare by its @ID.
- If you want to order your query you just need to do: "*order by d.weight desc*". The default order value is asc and you do not need to write it.
- About the join you must pay attention to the two kinds of joins that we used. In the "*findPersonByNameWithAllDogs*" method we just used "*... Person p join fetch p.dogs ...*" to bring the dogs list. We needed to use the join fetch because the dog list is annotated with the "lazy" attribute; if we did not included the join fetch and executed a command like "*person.getDogs()*", other "*trip*" to the database would be required. If you use this query to find a Person that has no dogs, the JPA will found no data in the database no matter if your database has a Person without dogs. If you want to do a query that brings a fetch dog collection and Persons that has or not dogs you will need to use "*...Person p left join fetch p.dogs...*" like we did in the method: "*findPersonByNameThatMayNotHaveDogs*". The "*left join fetch*" will bring Persons that has an empty dog list.

JPQL: Functions: AVG, COUNT, MAX, MIN, TRIM, SUM, UPPER, LOWER, MOD, LENGHT, SQRT; Using HAVING, GROUP BY

JPQL also has a lot of functions that help us with the queries. Bellow you can see their description:

- AVG – Does a number average
- COUNT – Counts the records amount found by the query
- MAX – Gets the higher value of a column
- MIN – Gets the lower value of a column
- TRIM – Removes the white space at the begging/end of the text
- SUM – Sums all the values of a column
- UPPER – Modifies all the column text to UPPER CASE
- LOWER – Modifies all the column text to lower case
- MOD – Returns the modulus of a column
- LENGTH – Returns the size of a String
- SQRT – Returns the square root of a number

Bellow you will see how to use these functions:

```
001 package com.main;
002
003 import java.util.List;
004
005 import javax.persistence.EntityManager;
006 import javax.persistence.Query;
007
```

```
import com.model.Person;

public class Page05 {
 public static void main(String[] args) {
  CodeGenerator.startConnection();

  CodeGenerator.generateData();

  EntityManager em = CodeGenerator.getEntityManager();

  Number average = getPersonsAgeAverage(em);
  System.out.println(average);

  List<Object[]> personsFilteredByDogsWeight = getPersonsWithDogsWeightHigherThan(em, 4d);

  for (Object[] objects : personsFilteredByDogsWeight) {
   Person person = (Person) objects[0];
   Long count = (Long) objects[1];
   System.out.println('The person : ' + person.getName() + ' has ' + count + ' dogs with the weight > 4');
  }

  List<Object[]> dogsMinAndMaxWeightList = getDogMinAndMaxWeight(em);
  Object[] dogMinAndMaxWeightResult = dogsMinAndMaxWeightList.get(0);
  System.out.println('Min: ' + dogMinAndMaxWeightResult[0] + ' Max: ' + dogMinAndMaxWeightResult[1]);

  Number sumOfAllAges = getTheSumOfAllAges(em);
  System.out.println('All summed ages are: ' + sumOfAllAges);

  String loweredCaseName = getLoweredCaseNameFromUpperCase(em, CodeGenerator.PERSON03_NAME);
  System.out.println(loweredCaseName);

  Number personAgeMod = getPersonAgeMode(em, CodeGenerator.PERSON05_NAME, 6);
  System.out.println('Person modulus age: ' + personAgeMod);

  Number personAgeSqrt = getPersonAgeSqrtUsingTrim(em, '        ' + CodeGenerator.PERSON04_NAME + '        ');
  System.out.println('Person modulus age: ' + personAgeSqrt);

  List<Object[]> personsByDogsAmount = getPersonByHavingDogAmountHigherThan(em, 3);

  for (Object[] objects : personsByDogsAmount) {
   Person person = (Person) objects[0];
   Long count = (Long) objects[1];
   System.out.println(person.getName() + ' has ' + count + ' dogs');
  }

  CodeGenerator.closeConnection();
 }

 /**
  * Uses the AVG sql database function
  */
 private static Number getPersonsAgeAverage(EntityManager em) {
  Query query = em.createQuery('select avg(p.age) from Person p');
  return (Number) query.getSingleResult();
 }

 /**
  * This query will use the count database function
  *
  * @return List<Object[]> where object[0] is a person, object [2] is a Long
  */
 @SuppressWarnings('unchecked')
 private static List<Object[]> getPersonsWithDogsWeightHigherThan(EntityManager em, double weight) {
  Query query = em.createQuery('select p, count(p) from Person p join p.dogs d where d.weight > :weight group by p');
  query.setParameter('weight', weight);
  return query.getResultList();
 }

 /**
  * This query will use the min and max sql database function
  *
  * @return List<Object[]> where object[0] is the min, object [2] is the max
  */
 @SuppressWarnings('unchecked')
 private static List<Object[]> getDogMinAndMaxWeight(EntityManager em) {
  Query query = em.createQuery('select min(weight), max(weight) from Dog');
  return query.getResultList();
 }

 /**
  * This query will use the sum sql database function
  */
 private static Number getTheSumOfAllAges(EntityManager em) {
  Query query = em.createQuery('select sum(p.age) from Person p');
  return (Number) query.getSingleResult();
 }

 /**
  * Method that uses the UPPER and LOWER database functions
  */
 private static String getLoweredCaseNameFromUpperCase(EntityManager em, String name) {
  Query query = em.createQuery('select lower(p.name) from Person p where UPPER(p.name) = :name');
  query.setParameter('name', name.toUpperCase());
  return (String) query.getSingleResult();
 }

 /**
  * Method that uses the mod database function
  */
 private static Number getPersonAgeMode(EntityManager em, String personName, int modBy) {
  Query query = em.createQuery('select mod(p.age, :modBy) from Person p where p.name = :name');
  query.setParameter('modBy', modBy);
  query.setParameter('name', personName);
  return (Number) query.getSingleResult();
 }

```

```
114  /**
115   * Method that uses the square root of a person age using the trim function in the name
116   */
117  private static Number getPersonAgeSqrtUsingTrim(EntityManager em, String name) {
118   Query query = em.createQuery('select sqrt(p.age) from Person p where p.name = trim(:name)');
119   query.setParameter('name', name);
120   return (Number) query.getSingleResult();
121  }
122
123  /**
```



```
igherThan(EntityManager em, long
dogAmount) {
128   Query query = em.createQuery('select p, count(p) from Person p join p.dogs group by p.id having
count(p) > :dogAmount');
129   query.setParameter('dogAmount', dogAmount);
130   return query.getResultList();
131  }
132 }
```

About the code above:

- In the method "*getPersonsAgeAverage*" we use the "avg" function to do an average calculation of the age column value.
- In the method "*getPersonsWithDogsWeightHigherThan*" we use the count function to bring the amount of dog with a person object. Notice that we have two distinct results, a number and a person object. These values will come inside an array Object[].
- The LOWER and UPPER functions will change your string case and you can use it to change the result of your query (after the select) or in the where condition. The "*getLoweredCaseNameFromUpperCase*" method uses the LOWER and UPPER functions in both ways.
- The "*getPersonAgeMode*" uses a parameter after the word select. With JPA we can use a parameter in any place of the query, you just need to add the ":" with a variable. You can have the same parameter several times and pass the value with the query.setParameter method.
- In the method "*getPersonByHavingDogAmountHigherThan*" the "*having*" function is invoked with the "*count*" function. We can use the "*having*" function to help us to filter the query data result.

JPQL: Filtering Conditions: LIKE, IN, DISTINCT, EMPTY, BETWEEN, NULL, MEMBER OF, EXISTS (Subqueries), ANY, ALL, SOME, CONCAT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCATE, SIZE, SUBSTRING

Some of these functions have the same purpose but handled differently.

Bellow you can see how to use these functions:

```
001 package com.main;
002
003 import java.text.SimpleDateFormat;
004 import java.util.Date;
005 import java.util.List;
006
007 import javax.persistence.EntityManager;
008 import javax.persistence.Query;
009
010 import com.model.Dog;
011 import com.model.Person;
012
013 public class Page06 {
014
015  public static void main(String[] args) {
016   CodeGenerator.startConnection();
017
018   CodeGenerator.generateData();
019
020   EntityManager em = CodeGenerator.getEntityManager();
021
022   List<Person> personByLike = getPersonByNameUsingLike(em, 'oh');
023
024   for (Person person : personByLike) {
025    System.out.println(person.getName());
026   }
027
028   List<Person> personsByAdressNumber = getPersonsByAddressNumberHigherThan(em, 90);
029
030   for (Person person : personsByAdressNumber) {
031    System.out.println(person.getName());
032   }
033
034   List<Person> personsWithoutDogs = getPersonsWithoutDogs(em);
035
036   System.out.println('Total of persons without dogs: ' + personsWithoutDogs.size());
037
038   List<Person> personsWithoutAddress = getPersonsWithoutAddress(em);
039
040   System.out.println('Total of persons without address: ' + personsWithoutAddress.size());
041
042   try {
043    SimpleDateFormat formatter = new SimpleDateFormat('dd/MM/yyyy');
044
045    Date startDate = formatter.parse('01/01/1996');
046    Date endDate = formatter.parse('01/01/1999');
047
048    List<Dog> dogsByBirth = getDogByBirthDate(em, startDate, endDate);
049
050    for (Dog dog : dogsByBirth) {
051     System.out.println(dog.getName() + ': ' + formatter.format(dog.getDateOfBirth()));
052    }
053   } catch (Exception e) {
054    e.printStackTrace();
055   }
056
057   Dog dog = (Dog) em.createQuery('select d from Dog d where d.id = 1',
Dog.class).getSingleResult();
058
059   boolean belongsTo = isThisDogBelongingToAperson(em, dog, CodeGenerator.PERSON01_NAME);
060
```

```
  System.out.println('Is this Dog member of Perons01? ' + belongsTo);

  Person personByConcatedName = getPersonConcatingName(em, 'Ma', 'ry');

  System.out.println('Found the person? ' + personByConcatedName.getName());

  List<Person> personByLocate = getPersonByLocatingStringInTheName(em, 'Mary');

  System.out.println('Amount of persons found by locate: ' + personByLocate.size());

  String personNameBySubstring = getPersonNameBySubstring(em, CodeGenerator.PERSON06_NAME, 12,
18);

  System.out.println('Name substring is: ' + personNameBySubstring);

  List<Person> personsDogWeight = getPersonByDogWeightOnlyHigherThan(em, 20);

  for (Person person : personsDogWeight) {
   System.out.println(person.getName());
  }

  List<Person> distinctPersons = getDistinctPersonsByDogsWeight(em, 2d);
  System.out.println('With the distinct, the result size is: ' + distinctPersons.size());

  List<Person> personsWithDogsAmount = getPersonsWithDougsAmountOf(em, 4);
  System.out.println('Number of persons with 4 dogs: ' + personsWithDogsAmount.size());

  Number numberOfDogsByPerson = getDogAmountByPerson(em, CodeGenerator.PERSON04_NAME);
  System.out.println('The dog amount is to ' + CodeGenerator.PERSON04_NAME + ': ' +
numberOfDogsByPerson);

  List<Dog> dogsBornedAfterToday = getDogsBornAfterToday(em);
  System.out.println('The amount of dogs borned after today is: ' + dogsBornedAfterToday.size());

  CodeGenerator.closeConnection();
 }

 /**
  * This methods compares a value with LIKE
  */
 @SuppressWarnings('unchecked')
 private static List<Person> getPersonByNameUsingLike(EntityManager em, String name) {
  Query query = em.createQuery('select p from Person p where p.name like :name');
  query.setParameter('name', '%' + name + '%');
  return query.getResultList();
 }

 /**
  * This methods show several ways to do a query that checks if a part of a collection is inside
another
  */
 @SuppressWarnings('unchecked')
 private static List<Person> getPersonsByAddressNumberHigherThan(EntityManager em, int
houseNumber) {
  Query query = em.createQuery('select p from Person p where p.address in (select a from Address
a where a.houseNumber > :houseNumber)');
  // Query query = em.createQuery('select p from Person p where (select a from Address a where
a.houseNumber > :houseNumber and p.address = a) > 0');
  // Query query = em.createQuery('select p from Person p where p.address = any (select a from
Address a where a.houseNumber > :houseNumber)');
  // Query query = em.createQuery('select p from Person p where p.address = some (select a from
Address a where a.houseNumber > :houseNumber)');
  // Query query = em.createQuery('select p from Person p where exists (select a from p.address a
where a.houseNumber > :houseNumber)');
  query.setParameter('houseNumber', houseNumber);
  return query.getResultList();
 }

 /**
  * This methods show how to check if a collection is empty
  */
 @SuppressWarnings('unchecked')
 private static List<Person> getPersonsWithoutDogs(EntityManager em) {
  Query query = em.createQuery('select p from Person p where p.dogs is empty');
  return query.getResultList();
 }

 /**
  * This method shows two ways to check if a relationship @OneToOne is empty
  */
 @SuppressWarnings('unchecked')
 private static List<Person> getPersonsWithoutAddress(EntityManager em) {
  Query query = em.createQuery('select p from Person p where p.address is null');
  // Query query = em.createQuery('select p from Person p where p.address is empty');
  return query.getResultList();
 }

 /**
  * Method that uses the between comparation
  */
 @SuppressWarnings('unchecked')
 private static List<Dog> getDogByBirthDate(EntityManager em, Date startDate, Date endDate) {
  Query query = em.createQuery('select d from Dog d where d.dateOfBirth between :startDate and
:endDate');
  query.setParameter('startDate', startDate);
  query.setParameter('endDate', endDate);
  return query.getResultList();
 }

 /**
  * Method that uses the member of comparation to check if an object belogs to a collection
  */
 private static boolean isThisDogBelongingToAperson(EntityManager em, Dog dog, String name) {
  Query query = em.createQuery('select p from Person p where :dog member of p.dogs and p.name =
:name');
  query.setParameter('dog', dog);
  query.setParameter('name', name);

  try {
   return query.getSingleResult() != null;
  } catch (Exception e) {
```

```
    return false;
   }
  }

  /**
   * Methods that concats Strings
   */
  private static Person getPersonConcatingName(EntityManager em, String firstWord, String
secondWord) {
   Query query = em.createQuery('select p from Person p where p.name = concat(:firstWord,
:secondWord)', Person.class);
   query.setParameter('firstWord', firstWord);
   query.setParameter('secondWord', secondWord);
   return (Person) query.getSingleResult();
  }

  /**
   * Method that locates a string inside another
   */
  @SuppressWarnings('unchecked')
  private static List<Person> getPersonByLocatingStringInTheName(EntityManager em, String
valueToBeLocated) {
   Query query = em.createQuery('select p from Person p where locate(p.name, :value) > 0',
Person.class);
   query.setParameter('value', valueToBeLocated);
   return query.getResultList();
  }

  /**
   * Methods that uses the ALL comparator
   */
  @SuppressWarnings('unchecked')
  private static List<Person> getPersonByDogWeightOnlyHigherThan(EntityManager em, double weight)
{
   Query query = em.createQuery('select p from Person p where p.dogs is not empty and :weight <
all (select d.weight from p.dogs d)');
   query.setParameter('weight', weight);

   return query.getResultList();
  }

  /**
   * Method that uses the distinct to remove any repetetition
   */
  @SuppressWarnings('unchecked')
  private static List<Person> getDistinctPersonsByDogsWeight(EntityManager em, double weight) {
   Query query = em.createQuery('select distinct p from Person p join p.dogs d where d.weight >
:weight');
   query.setParameter('weight', weight);
   return query.getResultList();
  }

  /**
   * Method that uses the substring to get just a position of chars inside the string
   */
  private static String getPersonNameBySubstring(EntityManager em, String personName, int
startPosition, int endPosition) {
   Query query = em.createQuery('select substring(p.name, :startPosition, :endPosition) from
Person p where p.name = :personName');
   query.setParameter('personName', personName);
   query.setParameter('startPosition', startPosition);
   query.setParameter('endPosition', endPosition);
   return (String) query.getSingleResult();
  }

  /**
   * Method that checks the size of a collection
   */
  @SuppressWarnings('unchecked')
  private static List<Person> getPersonsWithDougsAmountOf(EntityManager em, int dogAmount) {
   Query query = em.createQuery('select p from Person p where size(p.dogs) = :dogAmount');
   query.setParameter('dogAmount', dogAmount);
   return query.getResultList();
  }

  /**
   * Method that gets the size of a collection
   */
  private static Number getDogAmountByPerson(EntityManager em, String personName) {
   Query query = em.createQuery('select size(p.dogs) from Person p where p.name = :personName');
   query.setParameter('personName', personName);
   return (Number) query.getSingleResult();
  }

  /**
   * Methods that uses the current database server date/time
   */
  @SuppressWarnings('unchecked')
  private static List<Dog> getDogsBornAfterToday(EntityManager em) {
   Query query = em.createQuery('select d from Dog d where d.dateOfBirth > CURRENT_DATE');
   return query.getResultList();
  }
}
```

About the code above:

- You can add the "NOT" word in your queries. If you use the "*IS EMPTY*" you will search for a collection without values; if you use the "*IS NOT EMPTY*" you will search for a populated collection.
- The "*getPersonsByAddressNumberHigherThan*" shows how to do the same query with different functions. All commented command lines will bring the same result. In/Any/Some/Exists has a close syntax. According to the Pro EJB3 book "*Some*" is an alias to "Any".
- The comparator "*IS EMPTY*" can be used to check a collection (e.g. @OneToMany) or a relationship class (e.g. @OneToOne). The "*IS NULL*" comparator cannot check a collection, but you can use it to check a non collection attribute (e.g. @OneToOne).
- The "*MEMBER OF*" comparator will check if a given parameter belongs to a collection.
- The "CONCAT" function can be used as condition comparator or as a query result. In the code above it was used just as comparator but you could use it like this: "select concat(firstName, lastName) from Person p"
- In the "*getPersonByDogWeightOnlyHigherThan*" method we use the ALL operator. This operator will return true only if all items of the

condition ("*:weight > ALL*") return true. In this method it will return true only if all the person dogs' weight were higher than ":weight", if only one of the dogs got the weight with a smaller value the comparator would return false. You must be aware of, *if the list is empty the comparator will return true*. To avoid this behavior you would need to check if the list is empty like was done in the method: "*p.dogs is not empty*".

- The "*distinct*" function will remove the duplicated objects. In the method "*getDistinctPersonsByDogsWeight*" the "*distinct*" function removes the duplicated persons.
- The "*SUBSTRING*" function extracts a value from a given string. You will set the beginning and the end of the value that will be extracted from the original value. You can use this function as a comparator also.
- The "*SIZE*" function will return the number of elements inside the collection. You can use as comparator or to get the value.
- In the code above we use the "*CURRENTE_DATE*" function to compare the date, you could use also "*CURRENT_TIME, CURRENT_TIMESTAMP*". The JPA specification says that the current date functions can only be used as comparator. JPA does not support any function to retrieve the database current date yet, because this kind of function is not database portable (**4.6.16 Functional Expressions – JSR-000220 Enterprise JavaBeans 3.0 Final Release (persistence)**). If you want to query the database date you can use a NativeQuery to get this value.
- I must always remember that you cannot navigate inside a collection. You cannot do the following command: "person.dogs.name". You can access a dogs name using a command like: *select p from Person p fetch join p.dogs d where d.name = ''*.

Continue to the second part of the series.

Reference: JPA Queries and Tips from our JCG partner Hebert Coelho at the uaiHebert blog.

Tagged with: JPA

9 COMMENTS

Thiago
August 13th, 2012 at 12:54 pm

Eu tenho uma dúvida.. se eu quiser fazer uma query onde os atributos são variáveis, por exemplo se tiver nome cpf projeto e cidade preenchidos pesquisar utilizando todos os campos, porém se não tiver o nome preenchido apenas cpf, projeto e cidade pesquisar apenas por esses campos, e assim sucessivamente com todos os campos, tem que criar uma query para cada opção possível? ou existe uma solução melhor? Obrigado.

Reply

Martin
November 1st, 2012 at 3:58 pm

this is very helpful. Thanks a million!

Reply

Luiz Fábio
February 20th, 2013 at 5:10 pm

simply amazing! thanks for this

Reply

Christian Hessenbruch
March 21st, 2013 at 12:25 am

How come you're using ` not " to surround strings?

Reply

Paul
August 20th, 2013 at 4:44 pm

You saved my day sir, thank you!

Reply

Stéphan
September 5th, 2013 at 4:01 am

This is by far the best tutorial on JPA queries I have seen. Wish I had found it weeks ago!
Thanks a lot

Reply

Julio Cesar Cortorreal
October 23rd, 2013 at 6:49 pm

casi no comento los tutoriales, pero realmente tu codigo esta bien formado y Estructurado.
sigue asi

Reply

Leonel
February 12th, 2014 at 10:33 am

Gran ejemplo, buena explicación, muy completo, muchas gracias…

Reply

anyhowa86
January 1st, 2015 at 12:21 pm

In this when i tried to insert data person_id in dog table was inserted as 0 , whereas it should have been inserted as the person_id which were generated can you please explain me why is this happening.

Reply

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

two + = 11

Post Comment

KNOWLEDGE BASE

- Academy
- Examples
- Library
- Resources
- Tutorials
- Whitepapers

PARTNERS

- Mkyong

THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- Web Code Geeks

HALL OF FAME

- "Android Full Application Tutorial" series
- 11 Online Learning websites that you should check out
- Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
- Android Google Maps Tutorial
- Android JSON Parsing with Gson Tutorial
- Android Location Based Services Application – GPS location
- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused o ultimate Java to Java developers resource center; targeted at the techn technical team lead (senior developer), project manager and junior dev JCGs serve the Java, SOA, Agile and Telecom communities with daily ne domain experts, articles, tutorials, reviews, announcements, code snipp source projects.

DISCLAIMER