Ultimate JPA Queries and Tips List – Part 1

1

JPA: NamedQuery, querying with dates, warnings about the getSingleResult method

To avoid the repetition of queries codes, upgrade the performance and make easier to maintain the queries we can use the NamedQueries. A NamedQuery uses JPQL as syntax and it is declared in the entity class. It is easier to edit the query after an update in the class code.
If you want to do a query using a date as parameter you can send only the date object or you can pass an enum that will describe the date type (recommended).
Bellow you will see how to create and use a @NamedQuery and how to query with date:

```
@Entity
@NamedQuery(name='Dog.FindByDateOfBirth', query='select d from Dog d where
d.dateOfBirth = :dateOfBirth')
public class Dog {

 public static final String FIND_BY_DATE_OF_BIRTH = 'Dog.FindByDateOfBirth';

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

 // getter and setter
```

```
@Entity

@NamedQueries({
  @NamedQuery(name='Person.findByName', query='select p from Person p where
p.name = :name'),
  @NamedQuery(name='Person.findByAge', query='select p from Person p where p.age
= :age')})
})
public class Person {

 public static final String FIND_BY_NAME = 'Person.findByName';
 public static final String FIND_BY_AGE = 'Person.findByAge';

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

        // get and set

}
```

```
EntityManager em = CodeGenerator.getEntityManager();
```

Ultimate JPA Queries and Tips List – Part 1

2

```
  int age = 70;
  List<Person> personByAge = getPersonByAge(em, 70);
  System.out.println('Found ' + personByAge.size() + ' person(s) with the age
of: ' + age);

  SimpleDateFormat formatter = new SimpleDateFormat('dd/MM/yyyy');
  Date dateOfBirth = null;
  try {
   dateOfBirth = formatter.parse('10/1/1995');
  } catch (ParseException e) {
   e.printStackTrace();
  }

  List<Dog> dogsByDayOfBirth = getDogsByDayOfBirth(em, dateOfBirth);
  System.out.println('Found ' + dogsByDayOfBirth.size() + ' dog with birth date
of ' + formatter.format(dateOfBirth));

  /*
   * This queries will raise Runtime Exceptions
   *
   * em.createQuery('select p from Person p').getSingleResult(); //
NonUniqueResultException
   *
   * em.createQuery('select p from Person p where p.name =
'JJJJ'').getSingleResult(); //NoResultException
   */

  CodeGenerator.closeConnection();
 }

 @SuppressWarnings('unchecked')
 private static List<Dog> getDogsByDayOfBirth(EntityManager em, Date
dateOfBirth) {
  Query query = em.createNamedQuery(Dog.FIND_BY_DATE_OF_BIRTH);
  query.setParameter('dateOfBirth', dateOfBirth, TemporalType.DATE);

  return query.getResultList();
 }

 @SuppressWarnings('unchecked')
 private static List<Person> getPersonByAge(EntityManager em, int age) {
  Query query = em.createNamedQuery(Person.FIND_BY_AGE);
  query.setParameter('age', age);

  return query.getResultList();
 }
}
```

About the code above:

- If you have only one query you can use the @NamedQuery annotation; if you have more than one query you can use the @NamedQueries annotations.
- When you do a query using a date object you can also use the TemporalType enum to detail the type of the date. For date queries you can use “java.util.Date” or “java.util.GregorianCalendar”.

Ultimate JPA Queries and Tips List – Part 1

3

getSingleResult()
Be careful when you use this method. It has a special way to handle two behaviors that it is easy to happen and both behaviors will raise an exception:

- Find more than one object from the query result: NonUniqueResultException
- Find no result: NoResultException

You always need to use a try/catch to avoid these exceptions to be thrown in the production environment. If you want to see this exceptions in real time, in the code above you can find two commented queries; it will raise the exceptions bellow:

```
/**

  * Returns the name of the first person using a native sql
  */
 private static String getFirstPersonName(EntityManager em) {
  Query query = em.createNativeQuery('select top 1 name from person');
  return (String) query.getSingleResult();
 }

 /**
  * Return an object using a native sql
  */
 private static Dog getTopDogDescending(EntityManager em) {
  Query query = em.createNativeQuery('select top 1 id, name, weight from dog
order by id desc', Dog.class);
  return (Dog) query.getSingleResult();
 }
}
```

About the code above:

- Notice that we use a native query instead JPQL. We can have an entity as result of a native query. The method “getTopDogDescending” returns a Dog object after a native query invocation.

You also can create your native query as @NamedNativeQuery. The difference between NamedNativeQuery and NativeQuery is that the NamedNativeQuery is defined on its entity class, and you can have only one with that name.
The advantages of using a @NamedNativeQuery are:

- Easy to maintain the code: every query is on the class, if a class update an attribute it will be easier to update the query.
- Helps on a better performance: Once your query is already declared the JPA will map it and keep its syntax on the memory. The JPA will not need to “parse” you query every time that your project use it.
- Raises your code reutilization: Once you declare a @NamedNativeQuery you must provide the “name” parameter with a value and this name must be unique to the Persistence Unit scope (you set this value in the “persistence.xml” and it is used by the EntityManager).

Ultimate JPA Queries and Tips List – Part 1

4

NATIVE QUERY @Entity

```
@NamedQueries({
 @NamedQuery(name='Person.findByName', query='select p from Person p where
p.name = :name'),
 @NamedQuery(name='Person.findByAge', query='select p from Person p where p.age
= :age')
})

@NamedNativeQuery(name='Person.findByNameNative', query='select id, name, age
from person where name = :name')
public class Person {

 public static final String FIND_BY_NAME = 'Person.findByName';
 public static final String FIND_BY_AGE = 'Person.findByAge';

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

 // get and set     }
```

You will be able to use the @NamedNativeQuery just like the @NativeQuery: “em.createNamedQuery(“Person.findByNameNative”);”.

Here it comes some bad news. Unfortunately the Hibernate does not have the @NamedNativeQuery implemented yet. If you try to run the code with this annotation you will see the exception bellow:

```
private static void deletePerson(EntityManager em, int personId) {
  Person savedPerson = em.find(Person.class, personId);
  em.remove(savedPerson);
  em.flush();
 }
}

# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

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:

Root logger option

Ultimate JPA Queries and Tips List – Part 1

5

```
log4j.rootLogger=ERROR, stdout

# Hibernate logging options (INFO only shows startup messages)
log4j.logger.org.hibernate=ERROR

# Log JDBC bind parameter runtime arguments
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.

```
/**
  * Easiest way to do a query
  */
 @SuppressWarnings('unchecked')
 private static List<Dog> listAllDogs(EntityManager em) {
  Query query = em.createQuery('select d from Dog d', Dog.class);

  return query.getResultList();
 }

 /**
  * Easiest way to do a query with parameters
  */
 private static Person findPersonByName(EntityManager em, String name) {
  Query query = em.createQuery('select p from Person p where name = :name',
Person.class);
  query.setParameter('name', name);
  return (Person) query.getSingleResult();
 }

 /**
  * Executes a query that has as parameter an object
  */
 private static Person findPersonByPersonObject(EntityManager em, Person person)
{
  Query query = em.createQuery('select p from Person p where p = :person');
  query.setParameter('person', person);
  return (Person) query.getSingleResult();
 }

 /**
  * Query that will list all dogs with an order
  */
```

```
 @SuppressWarnings('unchecked')
 private static List<Dog> listAllDogsOrderingByWeight(EntityManager em) {
  Query query = em.createQuery('select d from Dog d order by d.weight desc',
Dog.class);

  return query.getResultList();
 }

 /**
  * Query that get only a field instead a complete class object
  */
 private static String findAddressNameOfPerson(EntityManager em, String name) {
  Query query = em.createQuery('select p.address.streetName from Person p where
p.name = :name');
  query.setParameter('name', name);
  return (String) query.getSingleResult();
 }

 /**
  * Query that will fetch a lazy relationship Be carefull, with this kind of
  * query only those who have the relationship will come in the result
  */
 private static Person findPersonByNameWithAllDogs(EntityManager em, String
name) {
  Query query = em.createQuery('select p from Person p join fetch p.dogs where
p.name = :name', Person.class);
  query.setParameter('name', name);
  return (Person) query.getSingleResult();
 }

 /**
  * With this query will will bring results that may not have arelationship
  */
 private static Person findPersonByNameThatMayNotHaveDogs(EntityManager em,
String name) {
  Query query = em.createQuery('select p from Person p left join fetch p.dogs
where p.name = :name', Person.class);
  query.setParameter('name', name);
  return (Person) query.getSingleResult();
 }



/**

  * 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();
 }

 /**
  * Method that uses the square root of a person age using the trim function in
the name
  */
```

```
 private static Number getPersonAgeSqrtUsingTrim(EntityManager em, String name)
{
  Query query = em.createQuery('select sqrt(p.age) from Person p where p.name =
trim(:name)');
  query.setParameter('name', name);
  return (Number) query.getSingleResult();
 }

 /**
  * Method that uses the having comparator with count
  */
 @SuppressWarnings('unchecked')
 private static List<Object[]>
getPersonByHavingDogAmountHigherThan(EntityManager em, long dogAmount) {
  Query query = em.createQuery('select p, count(p) from Person p join p.dogs
group by p.id having count(p) > :dogAmount');
  query.setParameter('dogAmount', dogAmount);
  return query.getResultList();
 }
}



/**

  * 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();
 }
}
```

JPA: Optimizing queries with EJB

Every time you do a query inside a transaction scope, the Persistence Context will keep the result “attached”; the Persistence Context will “watch” this objects just in case if any of this “attached” objects receive any update. All “attached” objects updates will be persisted in the database.

```
@PersistenceContext(unitName = 'myPU')
private EntityManager em;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void editPersonName(Integer personId, String personNewName){
```

Ultimate JPA Queries and Tips List – Part 1

12

```
 Person person = em.find(Person.class, personId);
 person.setName(personNewName);
}
```

https://www.javacodegeeks.com/2012/07/ultimate-jpa-queries-and-tips-list-part_09.html