ABOUT ANUJ KUMAR

Anuj is working as a Senior S/W Engineer in Netherlands. He is a Sun Certified Java Professional with experience in design and development of mid/large scale Java applications. He is the creator of EasyTest Framework(https://github.com/EaseTech) which is a Data Driven Testing Framework for Java.

JPA 2 | Dynamic Queries Vs Named Queries

Posted by: Anuj Kumar in Enterprise Java June 18th, 2013 0 401 Views

JPA has its own Query language called JPQL. JPQL is very similar to SQL, with one major difference being that JPQL works with entities as defined in the application whereas SQL works with table and column names as defined in the database. JPA provides us with a variety of options when it comes to defining the JPA queries that will perform CRUD operations on our defined Entity classes. These options are dynamic queries, Named Queries and Criteria queries. This post tries to look at each of the option in slight detail with the focus on when to use each type of query definition, what the performance problems might be and also some security threat associated with Dynamic Queries.

compliant query string to the createQuery method of the EntityManager class. Defining a dynamic query has its advantages and disadvantages. Lets look at each one of them in turn.

Advantages

The main advantage of using dynamic queries is in situation where you dont know how the query will look like until runtime and when the structure of the query is dependent on the user inputs or other conditions.

Disadvantages

The main disadvantage of dynamic queries is the cost associated with translating a JPQL query to SQL every time it is invoked. Most of the providers may try to cache SQL query generated from dynamic queries, but may not always be successful in doing that. Case where the query will not be cached or provider will find it difficult to cache is when concatination is used to bind parameter values directly into the query string. Hers an example where the query will not be cached because every time the JPQL is invoked a new query may be generated because of dynamic parameters.

```java
@Stateless
public class DynamicQueriesExample implements DynamicQuery {
    @PersistenceContext(unitName="dq")
    EntityManager em;

    public long queryFinancialRecordsOfDept(String deptName, String companyName) {
        String query = "SELECT d.records " +
                       "FROM Department d " +
                       "WHERE d.name = '" + deptName +
                       "' AND " +
                       "      d.company.name = '" + companyName + "'";
        return em.createQuery(query, Long.class).getSingleResult();
    }

}
```

In the above query, we are concatinating the deptName and companyName values in the query String. As a result, every time the method queryFinancialRecordsOfDept is invoked, a new query is generated. This new query is difficult to cache because the variables make the String unique almost every time. Thus if you have many dynamic queries in your application, and they are being called quite often, then you will have performance problems to solve.

The second problem with dynamic queries written like above is the actual concatenation operation. Since you are using simple String concatenation, and since Strings are immutable, the JVM will generate many String objects, most of which will be discarded in the end and will

placed among the top Java related sites around. Constantly being on the lookout for partners; we encourage you to join us. So If you have a blog with unique and interesting content then you should check out our **JCG** partners program. You can also be a **guest writer** for Java Code Geeks and hone your writing skills!

The third problem with dynamic queries, written like above, is that of security. In the above query for example, a hacker can easily pass in a value for companyName to alter the query to his advantage. Finding out what query in an application is expecting is easier than we think. Simple stack trace from the application reveals more than one can imagine.

So, if an application is expecting the company name to be specified at runtime by its user, a hacker in the above case can pass the value for companyName parameter as a GET or POST request with value of companyA OR d.company.name = companyB:

```
"SELECT d.records " +
"FROM Department d " +
"WHERE d.name = 'deptA' +
   "AND d.company.name = 'companyA'
       OR d.company.name = 'companyB';
```

This type of security risk can easily be reduced by using named/positional Parameters feature of JPA. Named parameters help us in binding the values to the query at a much later stage in the query processing. By using named parameters, the query is not altered every time for different parameters. Thus the query remains same and becomes easy for the provider to cache it.

The second advantage of using Named/Positional Parameters is that they are marshalled in to the query using the JDBC API and happens at the DB level and at DB level, the DB normally quotes the text passed as parameter. Thus, in the above case, we can alter the query to use named/positional parameters:

WAY 1 : Named Parameters

```
public long queryFinancialRecordsOfDept(String deptName, String companyName) {
        String query = "SELECT d.records " +
                        "FROM Department d " +
                         "WHERE d.name = :deptName +
                          "AND d.company.name = :compName;
        return em.createQuery(query,Long.class)
                 .setParameter("deptName" , deptName)
                 .setParameter("compName" , companyName)
                 .getSingleResult();
    }

}
```

WAY 2 : Positional Parameters

```
public Long queryFinancialRecordsOfDept(String deptName, String companyName) {
        String query = "SELECT d.records " +
                        "FROM Department d " +
                        .setParameter(2 , companyName)
                        .getSingleResult();
        }

}
```

Way 1 of specifying the Parameters uses named variables that can be supplied values using the setParameter method on the Query Object.

Way 2 of specifying the Parameters uses numbers or indexes to bind query parameters to the query string.

Side Note: We can use the same named parameter multiple time in the query, but needs to bind the value only once using the setParameter function.

Named Queries

Named queries are a way of organising your static queries in a manner that is more readable, maintainable as well as performant.

Named queries in JPA are defined using the @NamedQuery annotation. This annotation can be applied only at class level and the Entity on which the Query will work is a good place to define Named Queries. For example, if a named query findAllItemRecords is defined to find all the Item entities in the Database table Item, then the Named query is normally defined on the Item Entity. Here is an example:

```
@NamedQuery(name="Item.findAllItemRecords" ,
                      query="SELECT item " +
                                "FROM Item item")
@Entity
public class Item {

   @Id
   @Column(name="item_id")
   private String itemId;

   @Column(name="item_type")
   private String itemType;

   //.......
 }
```

One thing to notice above is that we are using concatenation operation on our string. But this will not be a performance problem like it is with dynamic queries because the Persistence Provider will convert the named queries from JPQL to SQL at deployment time and will cache them for later use. This means that the overhead of using concatenation is felt only at deployment time and not every time the query is used by the application. And the good thing about concatenating the query like above is that it makes it more readable and thus maintainable.

We can define more than one NamedQueries for a given entity using @NamedQueries annotation. Lets have a look at an example of specifying multiple named queries.

```
@NamedQueries({
@NamedQuery(name="Item.findAllItemRecords" ,
                      query="SELECT item " +
                                "FROM Item item "
                                "WHERE item.itemId=:itemId),
@NamedQuery(name="Item.findItemByType" ,
                      query="SELECT item " +
                                "FROM Item item "
                                "WHERE item.itemType=:itemType)
})
@Entity
public class Item {

   @Id
   @Column(name="item_id")
   private String itemId;

   @Column(name="item_type")
   private String itemType;

   //.......
 }
```

We can use named queries in our method using the createNamedQuery method on the EntityManager.

```
public Item findAllItemRecords(String itemId) {
         return em.createNamedQuery("Item.findAllItemRecords",
                                       Item.class)
                  .setParameter("itemId", itemId)
                  .getSingleResult();
```

Summary

We discussed in this small blog post difference between Dynamic and Named queries in JPA. In the next blog post we will look at Criteria APIs and how they are used.

The contents in this blog post are a result of reading the excellent book Pro JPA 2 . I would recommend it to anyone who is working on JPA related projects.

Tagged with: JPA

(No Ratings Yet) Start the discussion 401 Views

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

Model Serving: Stream Processing vs. RPC / REST with Java, gRPC, Apache Kafka, TensorFlow

July 9th, 2018

6 Log Management Tools You NEED to Know (And How to Use Them)

July 6th, 2018

Apache Camel 2.22 Released with Spring Boot 2 support

July 5th, 2018

Leave a Reply

You are logged in as amitabhmandal | Log out

Start the discussion...

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Subscribe

KNOWLEDGE BASE

- Courses
- Examples
- Minibooks
- Resources
- Tutorials

PARTNERS

- Mkyong

THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- System 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 on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike. JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Geeks are the property of their respective owners. Java is a trademark or registered trademark of Oracle Corporation in the United States and other countries. Examples Java Code Geeks is not connected to Oracle Corporation and is not sponsored by Oracle Corporation.