# Assignment: JPQL

In this exercise, you will practice JQPL queries in combination with an H2 SQL database.

The tests implemented for this assignment are merely used for checking the results from the database after each query.

Difficulty: ☆☆☆

Estimated time: 30min

Learning objectives:

* I understand what is JPQL
* I understnd the problem of derived queries
* I can use JPQL @Query annotation to query a database
* I can use native queries along JPQL
* I can use named parametes in a JPQL query
* I can use JPQL to make aggregated queries

## Step 1: Download the startup project

Download the startup Java project and open it in IntelliJ (or another IDE). It contains example code on how to use JPQL queries.

The project is configured to use H2 as the database. The great thing about H2 is that you don’t need much to run it, just a simple dependency in build.gradle, and some setup options in the application.properties. It’s great for prototyping and testing.

## Step 2: First JPQL queries:

In the previous assignment, you have worked with **JPA derived queries**, they are really handy because you can simply specific in the method’s name your query. However, they can make the name of the method quite long, and you cannot make more elaborated queries with it. So, for that we can use Java Persistence Query Language (JPQL), which is the query language provided by JPA. JPQL has a syntax very similar to SQL, however it does not have all the functionalities SQL provides. The main difference is that JPQL operates on Java classes and objects, instead of database tables and columns.

So, let’s take a look on the following example in the CountryRepository:

package fontys.sem3.school.repository;  
  
import fontys.sem3.school.repository.entity.CountryEntity;  
import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;  
  
public interface CountryRepository extends JpaRepository<CountryEntity, Long> {  
  
 CountryEntity findByCode(String code);  
  
 @Query("select c from CountryEntity c where c.code = ?1")  
 CountryEntity getCountryByCode(String code);  
  
}

The first method, findByCode(String code), is a derived query. The second method, getCountryByCode(String code), is anotahed with the @Query with a JPQL query on it “select c from CountryEntity c where c.code = ?1”. As you can imagine, both methods are doing the same query: returning a CountryEntity by the code. Of course this JPQL query example is just for ilustration purposes, if you can make your query using derived query, there is no need to write a JPQL for it.

In this JPQL query, you can observe that it is operating on the class CountryEntity, and not on the database table (which is called COUNTRY). Also, the parameter of the query is provided in order: the first one “?1” will be the first argument of the method. You can test this method calling the test testGetCountryByCode() in the CountryRepositoryTest class.

Now it is your turn. We need to have a JPQL query to return a StudentEntity when the PCN is given. Do not forget to create the test for it in the class StudentRepositoryTest.

In the previous query we select the whole Java object, but we can also return some atributes of it:

public interface CountryRepository extends JpaRepository<CountryEntity, Long> {  
  
 @Query("select s.name from CountryEntity s where s.code = ?1")  
 String getCountryNameByCode(String code);

}

Now we also need a JPQL query to return only the name of the StudentEntity. Also recall to create the test for it.

Finally, we can also use more than one parameter in the JPQL query. For example, in the query below we are searching for Country Code and Name:

public interface CountryRepository extends JpaRepository<CountryEntity, Long> {  
  
 @Query("select s.name from CountryEntity s where s.code = ?1 and s.name = ?2")  
 String getCountryNameByCodeAndName(String code, String name);

}

This approach is called ordered parameters, where you need to follow the numbers used in the query when defining the method arguments.

## Step 3: JPQL query with named parameters:

Using ordered parameters can be confusing sometimes. JQPL provides a way to name the parameters, so it improves readability. For that, we simply use the semicolon followed by the name of the parameter in the query. Thus, to use it in the method’s argument, we use the @Param() annotation:

package fontys.sem3.school.repository;  
  
import fontys.sem3.school.repository.entity.CountryEntity;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
import org.springframework.data.repository.query.Param;  
  
public interface CountryRepository extends JpaRepository<CountryEntity, Long> {  
  
 @Query("select s.name from CountryEntity s where s.code = :countryCode")  
 String getCountryNameByCodeNamedParam(@Param("countryCode") String code);

}

Now it is your turn again. We need a JPQL query with named param to get a Sudent by PCN.

## Step 4: Native queries:

JPQL also allow us to use SQL queries (called native queries), then we can use regular SQL syntax. For that, we just need to add the “nativeQuery = true” attribute to the @Query annotation. This way we are informing JPA that we will use a native SQL query:

public interface CountryRepository extends JpaRepository<CountryEntity, Long> {  
  
 @Query(

value = "SELECT \* FROM COUNTRY WHERE NAME = :countryName",

nativeQuery = true

)  
 CountryEntity getCountryByNameNamedParam(@Param("countryName") String name);

}

Now it is your turn once again. We need a native JPQL query with named param to get a Sudent by name.

## Step 5: Making aggregated queries with JPQL:

JPQL can be also used to make more elaborated queries such as using aggregation (e.g., by using “GROUP BY”). For example, let’s say we need to report the highest grades per course. In SQL, this query would look like this:

|  |
| --- |
| SELECT COURSE\_ID, MAX(GRADE)  FROM COURSE\_GRADE  GROUP BY COURSE\_ID; |

As you can imagine, in our application there is class or object which represents what is being returned by that select, i.e., the tuple COURSE\_ID, MAX(GRADE). This is not a problem, because JPA will return an array of Object: *Object[]*. But, handling with a list of Object can be messy and error prone. Thus, JPA allows us to customize the results in a object-oriented approach. So, first, let’s create a class MaxCourseGrades to represent the return of the aggregated query:

package fontys.sem3.school.repository.entity;  
  
import lombok.AllArgsConstructor;  
import lombok.Getter;  
  
@AllArgsConstructor  
@Getter  
public class MaxCourseGrades {  
  
 private CourseEntity course;  
 private Double maxGrade;  
}

It does not need to be annotated with @Entity, but you do need to provide the constructor and getters so you can use it properly.

Then, the JQPL query be written as:

package fontys.sem3.school.repository;  
  
import fontys.sem3.school.repository.entity.CourseGradeEntity;  
import fontys.sem3.school.repository.entity.MaxCourseGrades;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.data.jpa.repository.Query;  
  
import java.util.List;  
  
public interface CourseGradeRepository extends JpaRepository<CourseGradeEntity, Long> {  
  
 @Query("SELECT new fontys.sem3.school.repository.entity.MaxCourseGrades(c.course, MAX(c.grade)) "  
 + "FROM CourseGradeEntity AS c "  
 + "GROUP BY c.course")  
 List<MaxCourseGrades> getMaxGradePerCourse();  
   
}

Here we need to bind the output for the SELECT to the MaxCourseGrades class. You can check the output on the testing class.

Finally, your last step. We need an aggregated JPQL query to report the avarage of the grades per course. Also create a simple test for it.