Limitations of finder methods in spring data jpa

==========

a) Only select operations are possible i.e non-select operations are not possible

b) Aggragate operations are not possible (count(*),max(-),min(-) and etc...)

c) GroupBy operations are not possible

d) Working with scalar operations/Projections is bit complex

are

Select Operations ===> reading data /records Non-Select operations ===> insert, update, delete operations

e) While selecting data by using multiple properites and multiple conditions the method names becoming really very lengthy f) Method names must be taken by following some conventions.. That process kills the readability...

g) we can not call PL/SQL procecedure and functions..

SQL

(orginal queries)

note1:: To overcome all the above problems.. take the support of @Query methods using JPQL/HQL or native SQL queries note2:: Use finder methods to perform single property/col condition based select operations.

Working with

of

@Query methods

=>On the top custom methods declared in our Repository interface, we need to add @Query annotation either having JPQL/HQL or native SQL query.. Method can have flexible signature and no need of following any naming conventions..

syntax:

our

In Repository interface

@Query("<HQL/JPQL> or <native SQL>")

<return type> <method name>(params .....)

Projections=Scalar Operations =

selecting specific single or multiple

col values of the db table

Entity Operations mean selecting all the col values of the Db table

JPQL is the specification given PA for ORM softwares to create Object based Query Language ..

In hibernate JPQL is given as HQL

In Eclipse Link JPQL is given as ELQL

=>native SQL queries means original SQL Queries

=>JPQL :: Java Persistence Query Lanaguge

JPQL (specification)

hibernate

(HQL)

EJB entityBeans (EJBQL)

Eclipse Link (ELQL)

=> HQL :: Hibernate Query Language => EJBQL :: EJB Query Language

=> ELQL Eclipse Link Query Lanaguage

DB s/w dependent Queries

DB s/w independent Queries

@Query method advantages

========

==========

=>Support both select and non-select operations (except insert operation) =>Can work with either HQL/JPQL or Native SQL queries

=> Method names and signatures can be taken having flexibility

=> Can be used to call PL/SQL Procedures and functions

=> Supports aggregate

select operations

=> supports group by, order by clauses

=> supports to work with joins (To get data from Two db tables having

=> supports aggregate operations and etc...

implicit conditions) HQL/JPQL

Why @Query methods do not support INSERT Queries?

=>HQL/JPQL queries are DB s/w independent Queries =>Native SQL queries are DB s/w dependent Queries =>HQL/JPQL querie are written using Entity class and its properties where as the Native SQL queries are written using db table name and its col names.

HQL:: Hibernate Query Language JPQL :: Java Persistence Query Language

Ans) HQL /JPQL INSERT Query can not work with generators configured in the Id Proeprty of Entity class but in spring data jpa the id value must be generated using one or another generator.. So use repo.save(-) or repo.saveXxx(-) methods for insert operation which can internally work with generators.

How to perform Record insertion with partial values?

(AUTO,SEQUENCE,...)

Ans) use repo.save(-) method having Entity obj with partial values (some properties will have NULL values) or keep @Transient on the top certain properties in the Entity class.

native SQL Queries =DB s/w specific SQL Queries

note:: @Transient properites do not participate in any kind of CURD Persistence operations)

HQL/JPQL

=========

note:: JPQL is specification where as HQL is Hibernate Impl of JPQL

=>HQL/JPQL

=>It is objects based DB s/w independent Query language

=>These Queries basead persistence is portable across the multiple DB s/ws => Supports both single row and bulk operations with our choice conditions..

=> Supports both Entity and scalar (Projections) select operations =>Supports both named (:<name>) and positional params (?1,?2,?3,...) =>Supports all where caluse conditions

=>Supports joins

=>HQL/JPQL keywords are not case-sensivite but the Entity class names

Cin

and proeprty names used the same queries are case-sensitive

and etc..

query

|-->create table, alter table,... Queries based DDL operations are not possible =>HQL/JPQL does not support insert Queries (There is no HQL/JPQL Insert Query) not =>HQL/JPQL based PL/SQL programming is possible

note: every HQL/JPQL will be converted in to underlying DB s/w specific SQL query.. note:: Learning curve of HQL/JPQL is very less becoz it is very much similar to SQL

SQL> SELECT * FROM CORONA_VACCINE

?,?,?,?> positional params

?1,?2,?3------> Ordinal Positional params :name,:age,:addrs ----> named params

db table name

HQL/JPQL> FROM Corona Vaccine (or)

HQL/JPQL>

Entity class name FROM com.nt.entity.Corona Vaccine fully qualified Entity class (or)

name

HQL/JPQL> FROM Corona Vaccine cv

(or) HQL/JPQL> SELECT cv

FROM

alias name Corona Vaccine CV col name

dib table

L

Writing SELECT Keyword in HQL/JPQ Queries is optional if u r selecting all colum/property values from Db table

col name

SQL> UPDATE CORONA_VACCINE_SET PRICE=PRICE+? WHERE COMPANY=? Entity class name property name HQL/JPQL> UPDATE Corona Vaccine SET_price=price+?1 WHERE company=?2 property name

(or)

For scalar or projections (select operation that select specific single or multiple col values) the select keyword is mandatory

(SQL supports jdbc type positional params)

ordinal positional params HQL/JPQL> UPDATE Corona Vaccine SET price-price+:addOnPrice WHERE company=:manifacture

=>?,?,... are called positional params

[named parameters are good]

naměed parameter

named parameters

for

(?)

:<name> :::: named parameter

=>?1,?2,?3,.. are called ordinal positional params From Hibernate 5.2 there is no support positional params,So we can use only ordinal nositional params given by JPQL or named params (P1,72,73,...)

:<name>

recomanded

db table name

col name

SQL> SELECT_REG_NO,NAME,COMPANY FROM CORONA_VACCINE WHERE COMPANY IN(?,?)

db table col names

Entity class name

HQL/JPQL> SELECT regNo, name, company FROM Corona Vaccine WHERE company IN(?1,?2)

(or) property names

positional params

→orinal positional params

>named params

HQL/JPQL> SELECT regNo, name, company FROM Corona Vaccine WHERE company IN(:comp1,:comp2) (or) HQL/JPQL> SELECT cv.regNo, cv.name, cv.company FROM Corona Vaccine as cv WHERE cv.company IN(:comp1,:comp2) (or) alias name HQL/JPQL> SELECT cv.regNo, cv.name, cv.company FROM Corona Vaccine as cv WHERE cv.company IN(?1,?2) property names

alias name

named params

(orinal positional params)

Exampl code

=> HQL/JPQL keywords are not case sensitive, but the entity class names and its properites used in the HQL/JPQL Queries are case-sensitive.

Repository Interface

public interface IDoctorRepository extends JpaRepository<Doctor,Integer> { //@Query("FROM Doctor WHERE income>=?1 AND income<=?2")

//@Query("FROM com.nt.entity.Doctor WHERE income>=?1 AND income<=?2")

//@Query("FROM Doctor doc WHERE doc.income>=?1 AND doc.income<=?2")

//@Query("FROM Doctor doc WHERE doc.income>=? AND doc.income<=?") //---> Throws the IllegalArgumentException

/* @Query("SELECT doc FROM Doctor doc WHERE doc.income>=?1 AND doc.income<=?2")

```
public List<Doctor> search DoctorsByIncomeRange(double startRange, double endRange);*/
/*@Query(" FROM Doctor WHERE income>=:start AND income<=:end")
public List<Doctor> search DoctorsByIncomeRange(@Param("start") double startRange, @Param("end") double endRange);
*/
@Query(" FROM Doctor WHERE income>=:start AND income<=:end")
public List<Doctor> search DoctorsByIncomeRange(double start, double end);
}
```

The method parameter values will be bound with namea pararn values automatically if their names are matching otherwise we need to Param explicitly.. if u r getting exception though names are matching then either

use @Param or enable following settings in the Eclipse Project.

service inteface

```
public interface IDoctorMgmtService {
}
public List<Doctor> showDoctorsByIncomeRange(double startRange, double endRange);
```

service Impl class

```
@Service
public class DoctorMgmentServiceImpl implements IDoctorMgmtService {
@Autowired
private IDoctorRepository doctorRepo;
@Override
public List<Doctor> showDoctorsByIncome Range(double startRange, double endRange) {
return doctorRepo.search DoctorsByIncome Range(startRange, endRange);
}
```

Client app

```
@Component
public class QueryMethodsTestRunner implements CommandLineRunner {
@Autowired
private IDoctorMgmtService service;
@Override
public void run(String... args) throws Exception {
service.showDoctorsByIncomeRange(20000.0, 300000.0).forEach(System.out::println);
}
}
```

project properties --->java compiler-->

Store information about method parameters (usable via reflection)

In HQL/JPQL Queries we can keep only two types of params

a) JPA style ordinal positional params (?1,?2,?3,....)

b) Named Params (:<name>) (best)

Important observations about HQL/JPQL Query parameters

a) Jdbc style plain positional parameters(?,?,..) are not allowed.

b) Only jpa style ordinal positional (?1,?2,?3,...) are allowed

c) Ordinal Positional parameter index must start with 1 and should continue in increment sequence with out any gap.

d) We can take parmater in the HQL/JPQL Query only representing input values i.e

we should not take representing HQL/JPQL keywords, Entity class name and entity property names and etc..

e) if HQL/JPQL Query's named parmaeters (:<name>) names and custom method parameter names are matching ..then no need of placing @Param annotaitons

before java method parameters..otherwise placing them is mandatory.

f) prefer working with named parameters more .. compare to the ordinal positional params

more positional params makes to give sequence index for the parameters

ordinal

g) we can use both poistional params and named params togather at once in a HQL/JPQL query, but named params must be placed after the positional params

FROM Corona Vaccine WHERE price>=?1 AND price<=?2 // valid

FROM Corona Vaccine WHERE price>=?1 AND price<=?3 // invalid (sequence shoud not miss) FROM Corona Vaccine WHERE price>=?0 AND price<=?1 // invalid (index must start with 1)

FROM CoronaVaccine WHERE price>=?2 AND_price<=?1 // valid java method arg values goes to query params in reverse order FROM Corona Vaccine WHERE price>=?1 AND price<=:max // valid

FROM Corona Vaccine WHERE price>=:min AND price<=?1 // invalid (we can not place ordinal positional param after named param) FROM Corona Vaccine WHERE price>=? AND price<=? // invalid (jdbc style positional params are not allowed)

FROM ?1 WHERE price>=?2 AND price<=?3 //invalid (we can not take entity class name as the param)

FROM Corona Vaccine WHERE price>=?1 AND price<=?1 // technically valid,but writing this kind query is meaning leass FROM Corona Vaccine ?1 price>=?2 AND price<=?3 // invalid (taking the property name as the param is not allowed) FROM Corona Vaccine WHERE :prop>=:min AND :prop<=:max // invalid (we can not take property name through

HQL/JPQL Select Queries

[Giving 0 or More Records]

named parameter)

Entity query [Selecting all col values]

List<T> (return type)

Entity class obj

Entity class obj

Entity class obj

Scalar-Projection

query (specific multiple col. values) List<Object[]>(return type)

0

1

scalar-Projection qury(specific single col) ↓

List<Property Type> (return type)

Entity class obj

2

0

if the selected single property

type is int then List collection

contains Integer wrapper objs as the

eleements.. if the selected single property

2

is String then the List collection

contains String wrapper objs as the elements

note ::No need of taking seperate type interfaces for scalar/Projection operations.

note:: arrays are objects in Java, So we can place the arrays as the elements of the List collection

Code in repository Interface

-----Select-- Entity Query

```
@Query("FROM Doctor WHERE specialization IN(:sp1,:sp2) ORDER BY specialization") public List<Doctor> search DoctorsBySpecializations(String sp1,String sp2);
```

//

Select -- Projection Query with specific multiple col values

```
@Query("SELECT docId,docName,income FROM Doctor WHERE income between :start and :end") public List<Object[]> searchDoctorDataByIncome(double start, double end);
```

---------- Select -- Projection Query with specific single col values @Query("SELECT docName FROM Doctor WHERE income between :min and :max") public List<String> searchAllDoctorNamesByIncome Range(double min,double max);

code in service Interface

public List<Doctor> searchDoctorsBySpecialization(String sp1, String sp2); public List<Object[]> showDoctors DataByIncome(double start, double end); public List<String> showDoctorosDataByIncome Range(double min,double max);

code in service Impl class

```
@Ovenue
public List<Doctor> searchDoctorsBySpecialization(String sp1, String sp2) {
//use repo
List<Doctor> list=doctorRepo.searchDoctorsBySpecializations(sp1, sp2); return list;
}
@Override
public List<Object[]> showDoctorsDataByIncome(double start, double end) { List<Object[]>
```

list-doctorRepo.searchDoctorDataByIncome (start, end); return list;

}

@Override

public List<String> showDoctorosDataByIncome Range(double min, double max) { List<String> list=doctorRepo.searchAllDoctorNamesByIncomeRange(min, max); return list;

}

code in runner class

select -Entity Query

service.searchDoctorsBySpecialization("physician", "cardio").forEach(System.out::println);

System.out.println("_

__select --- Entity Projection Query_(specific multiple col values)_____");
service.showDoctorsDataByIncome (20000.0, 2000000.0).forEach(row->{

System.out.print(obj+" ");

ग्

represents Object[] in the earch iteration of List Collection

_select --- Entity Projection Query___(specific multiple col value_ _"); service.showDoctorsDataByIncome (20000.0, 2000000.0).forEach(row->{

for(Object obj:row) {

}

System.out.println();

});

System.out.println(".

});

System.out.println("_

}

System.out.println(Arrays.toString(row));

Converts Object[] into String in each iteration of List Collection __select --- Entity Projection Query_____(specific single col value _"); service.showDoctorosDataByIncome Range(40000.0, 5000000.0).forEach(System.out::println);

ing) as the param

When should i use finder methods and when should i use @Query methods for select operation?

single

Ans) if u want to perform SELECT operation by using property/col based condition then go for finder methods.. for remaining all select operation prefer using @Query methods.

S

(findBy methods)

note:: It is always good practice to work with Query methods for

any custom requirement of persistence actvitiy

@Query methods for single Row Operations

Select Operation

(Single Row Operation)

(Entity Query) [Selecting all col values

of a record)

<T> [Entity object]

(return type)

Corona Vaccine obj

(Entity obj)

Scalar Query

[Selecting specific multiple col values] Object class obj pointing to Object[]

Scalar Query [Selecting specific

that property/col value like

single col vlaue]

Object representing

0

wrapper obj/String obj/

other obj.

Object

1

class obj

object[]

object

=> we can place both ordinal positional params and named params in one HQL/JPQL Query but

we place all named parameter only after placing all positional params

eg1: @Query("from Doctor where specialization in(?1,?2,:special3) order by specialization asc") //valid eg2: @Query("from Doctor where specialization in(:special1,?1,:special3) order by specialization asc") //invalid eg3: @Query("from Doctor where specialization in(?1,:special2,:special3) order by specialization asc") //valid

Q) Why should we go for @Query method that give single row when we have direct

findByid(-) or getById(-) or getOne(-) methods in pre-defined Repository interfaces? Ans)The pre-defined methods

or getReferenceById(-)

and etc..

findByid(-) or getById(-) or getOne(-) methods in pre-defined Repository interfaces

take id value (pk col value) as the criteria value/condition value to get that single row, but

we can design the above single row

as the criteria/condition value.

query by taking other unique cols

note :: if these methods found more than 1 record then we get

the pre-defined repository methods

where the custom @Query methods

can be taken for the scalar operations

do not support scalar operations..

of db table

Caused by: javax.persistence.NonUniqueResultException: query did not return a unique result: 2

In Repository Interface

//

--Entity Query giving single record

@Query("FROM Doctor where docName=:name ") //assume that doctor

public Optional<Doctor> showDoctorInfoByName(String name);

---Scalar Query giving single record multiple col values @Query("SELECT docId,docName FROM Doctor where docName=:name ")

public Object showDoctorDataByName(String name);

-Scalar Query giving single record single col value @Query("SELECT specialization FROM Doctor where docName=:name ") public String showSpecializationByName(String name);

To hold single property/col

single value

In service Interface

public Doctor search DoctorByNameDocName(String docName);

public Object search DoctorDataByName(String docName);

public String searchSpecilizationByName(String docName);

In service Impl class

@Override

To hold single record

Internally points to Object[]

public Doctor search DoctorByNameDocName(String docName) {

Doctor doc=doctorRepo.showDoctorInfoByName(docName).orElse Throw(()-> new IllegalArgumentException("Doctor not found"));

return doc;

}

@Override

public Object search DoctorDataByName(String docName) {

Object obj=doctorRepo.showDoctorDataByName(docName);

return obj;

}

@Override

public String searchSpecilizationByName(String docName) {

```
String result=doctorRepo.showSpecializationByName(docName);
return result;
}
In runner class
System.out.println("============
======== ");
Doctor doctor-service.search DoctorByNameDocName("raja");
System.out.println("Doctor Info ::"+doctor);
System.out.println(".
Object obj=service.search DoctorDataByName("raja");
");
Object data[]=(Object[])obj;
for(Object o:data) {
obj is ponting object[]
System.out.print(o+" ");
obj -------->
1001 raja
}
System.out.println();
System.out.println("Result is ::"+Arrays.toString(data));
System.out.println(".
_");
String result=service.searchSpecilizationByName("raja");
System.out.println("specilization is ::"+result);
HQL/JPQL supports aggragate opreations like count(*), max(-),min(-), avg(-) and etc...
Example code
===========
Code in Repository interface
@Query("SELECT count(distinct docName) FROM Doctor")
public int fetch Doctors NameCount();
@Query("SELECT count(*),max(income), min (income), avg(income), sum (income) from Doctor")
public Object fetchAggregateData();
Code in service interface
public int showDoctorNamesCount();
public Object showAggregateData();
Code in service Impl class
@Override
```

```
public int showDoctorNamesCount() {
internally points to Object[]
int count=doctorRepo.fetchDoctorsNameCount();
return count;
}
@Override
public Object showAggregateData() {
Object obj=doctorRepo.fetchAggregateData();
return obj;
}
```

Code in Runner class

```
System.out.println("unique doctor names count ::"+service.showDoctorNamesCount());
System.out.println("_
Object[] results=(Object[])service.showAggregateData();
System.out.println(" records count ::"+results[0]);
System.out.println("max income value:: "+results[1]);
");
System.out.println("min income value ::"+results[2]);
System.out.println("sumof income ::"+results[3]); System.out.println(" avg of income ::"+results[4]);
```

Object obj

result[]

4

90000 50000 436467

6878.5

0

1

2

3

4

When should i use finder methods and when should i use @Query methods for select operation?

single

Ans) if u want to perform SELECT operation by using property/col based condition then go for finder methods.. for

remaining all select operation prefer using @Query methods.

Performing non-select Operations using HQL/JPQL in @Query methods

==========

=> INSERT HQL/JPQL is not supported, sos.save(-) method for it

=> For other non-select Opeations like DELETE,UPDATE we need to pplace

@Query

+ @Modifying Annotations

=>To indicate the given HQL/JPQL query is non-select HQL/JPQL query

=> if u r not taking seperate service class.. then we need to place @Transactional on the top

of Repository (1) or @Query methods+ @Modifying methods otherwise we need to place on the top

of service class methods or service class itself.

Code in Repository interface

```
@Modifying
@Transactional
@Query("update Doctor SET income-income+(income * :percentage/100.0) WHERE specialization=:sp")
public int hikeDoctorsIncomeBySpecialization(String sp, double percentage);
@Query("DELETE FROM Doctor WHERE income>=:start AND income<=:end")
@Modifying
@Transactional
```

=>Select queries do not modify the data, So we do not need to commit the data

so we do not need @Transactional annotation

=>Non-Select queries modify the data, So we need to commit the modify the data so we need to place @Transactional annotation on the top of the methods at various levels

@Transcational commits data if things are going smoothly otherwise the @Transactional rollback the data (uncommits

the data)

if any exception is raised in the b.method

@Transactional annotation is required in the b.method that performs non-select operations becoz the modified data should be committed if there are no exceptions in the the logics otherwise the modified data will be rolled

```
public int remove DoctorsByIncomeRange(double start, double end);
```

Code in service Interface

```
public int appraise DoctorsIncomeBySpecialization(String specialization, double hike Percentage);
```

back

```
public int fireDoctorsByIncomeRange(double start, double end);
```

Code in serivice Impl class

```
@Override
public int appraise DoctorsIncomeBySpecialization(String specialization, double hikePercentage) {
int count=doctorRepo.hikeDoctorsIncomeBySpecialization(specialization, hikePercentage);
}
return count;
@Override
```

```
public int fireDoctorsByIncome Range(double start, double end) {
return doctorRepo.removeDoctorsByIncomeRange(start, end);
}
```

Code in Runner class

```
System.out.println(".
non-select operations_
");
int count=service.appraise DoctorsIncomeBySpecialization("cardio", 10.0);
System.out.println("no.of records that are effected::"+count);
System.out.println("delected doctors count::"+service.fire DoctorsByIncome Range (10000.0, 150000.0));
```

Using @Query methods having native SQL queries

=>Native SQL queries means the underlying DB s/w specific SQL queries i.e if the underlying Db s/w is oracle there specific SQL queries ... and etc..

=>Use this native SQL queries to perform those Operations which are not possible with HQL/JPQL Queries like

=> insert Query

=> Date operations (sysdate in oracle, now() in mysql)

=>ddl queries (create table, drop table,alter table and etc..)

=>To call PL/SQL procedure and functions

with

Native SQL Query= DB s/w specific SQL Query

=>Use @Query annotation with "nativeQuery=true" param to work native SQL queries

=>while working with native SQL queries we can place 3 types of parameters

a) jdbc style positional parameters (?,?,?,..)__

b) jpa style ordinal positional parameters (?1,?2,?3,....)

c) named parameters (:<name1>,:<name2>,....)

=>NativeSQL queries will be written using db table names and its column names...

=> Native SQL Queries based persistence logic is DB s/w dependent persistence logic

Example code in Repository(1)

native SQL Queries

```
@Query(value = "INSERT INTO JPA_DOCTOR_INFO
VALUES(DOCID_SEQ.NEXTVAL,:name,:income,:special)",nativeQuery =true)
@Modifying
@Transactional
public int registerDoctor(String name, String special, double income);
```

Code in the Repository

```
Native SQL Queries @Query(value="INSERT INTO JPA_CUSTOMER(CNAME,CADD, BILLAMT)
VALUES(:name,:addrs,:amt)",nativeQuery = true) @Transactional
```

```
@Modifying
public int registerCustomer(String name,String addrs, double amt);
@Query(value="SELECT now() FROM DUAL",nativeQuery = true)
@Query(value="SELECT SYSDATE FROM DUAL",nativeQuery = true) public String showSystem Date();
@Query(value="CREATE TABLE TEMP (col1 number(5))",nativeQuery = true)
@Modifying
@Transactional
public int createTempTable();
```

code in Service interface

```
public String insertDoctor(String name, double income,String specialization);
public String showSystem Date();
public String createTempDBtable();
```

Code in service impl class

```
@Override
public String insertDoctor(String name, double income,String specialization) {
int count=doctorRepo.registerDoctor(name, specialization, income);
return count==0?" Doctor not registered":"Doctor is registered";
}
@Override
public String showSystemDate() {
return doctorRepo.showSystem Date();
}
@Override
public String createTempDBtable() {
int count=doctorRepo.createTempTable();
return count==0?"db table is created":"db table is not created";
}
```

Code in runner class

```
System.out.println("_
native SQL Queries
_");
System.out.println(service.insertDoctor("suresh", 800000.0,"cardio"));
System.out.println("system date time ::"+service.showSystemDate());
System.out.println(service.createTempDBtable());
```

=>JPQL/HQL queries will go to DB s/w as SQL Queries as translated queries i.e every JPQL-HQL Query will be converted

into nativeSQL/SQL Query

=> NativeSQL Queries given to spring data jpa application will go to underlying DB s/w as it is for execution

with out any translation

```
public String showSystem Date();
@Query(value="create table Temp5(col1 integer, col2 varchar(20))", nativeQuery = true)
@Modifying
@Transactional
public int createTempTable();
```

Code in Runnner class

==========

```
custRepo.createTempTable();
System.out.println("--
");
-");
System.out.println(custRepo.showSystem Date());
System.out.println("-
custRepo.registerCustomer("manish","hyd", 5678.00);
```