Day15: Spring boot Data JPA

In order to develop the Data Access Layer we have 2 approaches:

1. JDBC approach
2. ORM based approach (JPA with Hibernate)

Spring Data JPA is not an ORM software, it is just a framework that adds an extra layer of abstraction on the top of the ORM s/w like the hibernate.

Spring Boot internally uses by default Hibernate engine(ORM software) and it provides an abstraction on the top of the hibernate implementation and provides the **CrudRepository** and the **JpaRepository** interfaces to simplify the Data Access Layer logic by significantly reducing the amount of boilerplate code.

CrudRepository(I)
|
JpaRepository(I)

JpaRepository:

JpaRepository is a JPA-specific extension of the Repository. It contains the full API of **CrudRepository**. because **JpaRepository** is the child interface of the **CrudRepository**. So it contains API for basic CRUD operations and also API for pagination and sorting and some extra functionalities.

Steps to develop Spring Boot Rest application using Spring Data JPA with MySQL.

Step 1: Create a spring boot starter project by adding the following dependencies:

a. Spring Web (To get the support of the tomcat server and to generate the Rest API)
b. Spring Data JPA: (spring boot starter data JPA) to add the JPA capabilities.
c. Spring Devtools: To reload the application when we do any changes (optional)
d. ****MySQL JDBC driver**

step 2: Make the following entries inside the **application.properties** file of our application:

```
#changing the server port
server.port=8088


#db specific properties
spring.datasource.url=jdbc:mysql://localhost:3306/dbName
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root
```

#ORM s/w specific properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

Note: Make sure username, password and database names are correct.

Note: Once we add the spring data JPA dependency in our application, the above configuration is mandatory, because with the above configuration, at the time of application startup only our database connection, and tables will be created and ready for us.

Step 3: Create an Entity class by applying at least following 2 annotations:

@Entity
@Id

Example: Create a Student bean class inside the com.masai.entities package:

```
package com.masai.entities;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO) // to autogenerate ID.
  private int roll;

  private String name;

  private int marks;

  public Student() {

  }

  public Student(int roll, String name, int marks) {
    super();
    this.roll = roll;
    this.name = name;
    this.marks = marks;
  }

  public int getRoll() {
    return roll;
  }

  public void setRoll(int roll) {
    this.roll = roll;
  }

  public String getName() {
    return name;
  }

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

  public int getMarks() {
    return marks;
  }

  public void setMarks(int marks) {
    this.marks = marks;
  }
```

```
}
```

Step 4: Create a **StudentDao** interface by extending the **JpaRepository** interface inside **com.masai.repository** package.

Example:

```
package com.masai.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.masai.entities.Student;

public interface StudentDao extends JpaRepository<Student, Integer> {

}
```

Here **Student** is an Entity/Domain class that repository manages and **Integer** is the type of the @Id annotation applied filed type.

Note: Here developer need not implement this interface, Spring-Data-JPA internally implements this interface for us as a spring bean (this class is already registered with the spring container).

Step5: Inject this StudentDao reference inside the **@RestController** class or **@Service** class and by using this reference perform the CRUD operation by taking the help of methods of JpaRepository.

Example: Saving a Student in the table:

Create a StudentController class inside the com.masai.controller package. and make this class as a Rest controller by applying the @RestController annotation on the top of this class.

and inject the StudentDao dependency using @Autowire annotation to this class.

```
package com.masai.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.masai.entities.Student;
import com.masai.repository.StudentDao;

@RestController
@RequestMapping("/jpaApp")
public class StudentController {

  @Autowired
  private StudentDao studentDao;

  @PostMapping(value = "/saveStudent")
  public Student saveStudent(@RequestBody Student student) {

  Student insertedStudent=studentDao.save(student);
  //here we passed Studen obj without roll and we get the inserted Student obj with the autogenerated roll.

    return insertedStudent;

  }
}
```

From the Postman call the above API by :

POST: **http://localhost:8088/jpaApp/saveStudent**

And in the request body pass the following JSON data (here don't give the roll number)

```
{
"name": "Ramesh",
"marks": 795
}
```

Note: In Real-time applications, Controller layer classes should not directly communicate with Data Access Layer classes like Repository, Controller should communicate with the Repository via Service layer classes.

But in our example, we will directly communicate the Controller layer with Data Access Layer. So that we can focus only on Spring Data JPA.

Some of the methods of the JpaRepository interface:

1. **<T> save<T>:** Saves a given entity. Use the returned instance for further operations as the save operation might have changed the entity instance completely.
2. **public long count():** Returns the number of entities available.
3. **Optional<T> findById(id):** Retrieves an entity by its id.
4. **public List<T> findAll();**
5. **public void deleteById(id);**
6. **public void delete(T t);**
7. **public void deleteAll();**

Note: **Optional** is a class introduced in java 8, and it belongs to **java.util** package. this class is given to avoid the **NullPointerException, This class makes the code more readable**

```
if(opt.isPresent()){
    Student s= opt.get();
}
else
    System.out.println("Not found...");
```

Example: **findById**

```
//GET  :- http://localhost:7000/jpaApp/getStudent/10
  @GetMapping("/getStudent/{roll}")
  public Student getStudentByRoll(@PathVariable int roll) {

    Optional<Student> opt=studentDao.findById(roll);

    Student st= opt.get();

    return st;
  }
```

Note: If the Student object is not available for the specified Roll number then it will throw the **NoSuchElementException** with internal server error code 500.

To handle this type of exception and give a proper response to the user we need to create our custom exception class and throw that exception class object explicitly. and handle that exception in the Exception handler method, in our custom format.

Example:

```
StudentException.java:- inside com.masai.exceptions package
----------------------------------------
```

```
package com.masai.exceptions;

public class StudentException extends RuntimeException{

  public StudentException() {
  }

  public StudentException(String message) {
    super(message);
  }
}
```

Throw this **StudentException** when student record not found:

Example:

```
//http://localhost:8088/jpaApp/getStudent/10
  @GetMapping("/getStudent/{roll}")
  public Student getStudentByRoll(@PathVariable int roll) {

    /*Optional<Student> opt=studentDao.findById(roll);

    if(opt.isPresent())
      return opt.get();
    else
      throw new StudentException("Student does not exist...");

    */

  return studentDao.findById(roll).orElseThrow(()-> new StudentException("Student does not exist"));


  }
```

Note: If we don't handle this exception then by default it generate HttpStatus code 500 ("Internal Server Error") with default format error response.

So we need to handle this StudentException in our custom format and send the HttpStatus code 404 "NOTFOUND"

Create a Java bean class, to format our exception handler output inside **com.masai.exception** package.

```
public class MyErrorDetails {

  private LocalDateTime timestamp;
  private String message;
  private String details;
//getters  and setters
}
```

Create separate GlobalException handler class and annotate it with **@GlobalAdvice annotation.**

```
GlobalExceptionHandler.java:- inside com.masai.exceptions package
-------------------------------------


@ControllerAdvice
public class GlobalExceptionHandler {


@ExceptionHandler(StudentException.class)
public ResponseEntity<MyErrorDetails> handleStudentException(StudentException exp,WebRequest req){

MyErrorDetails err=new MyErrorDetails(LocalDateTime.now(), exp.getMessage(), req.getDescription(false));
```

```
return new ResponseEntity<>(err,HttpStatus.NOT_FOUND);

  }
}
```

Example : Getting All the Student details:

```
@GetMapping("/getAllStudents")
  public List<Student> getAllStudents(){

    //List<Student> students= studentDao.findAll();
    //return students;

    return studentDao.findAll();
  }
```

Example: implementing delete method

```
@DeleteMapping("/deleteStudent/{roll}")
public Student deleteStudentByRoll(@PathVariable int roll) {

Student st=studentDao.findById(roll).orElseThrow(() -> new StudentException("Student with Roll "+roll+"does not exit.."));

    studentDao.delete(st);

    return st;

}
```

Example: Implementing update method

```
@PutMapping("/updateStudent")
  public Student updateStudentDetails(@RequestBody Student student) {

    Optional<Student> opt= studentDao.findById(student.getRoll());

    if(opt.isPresent()) {

    //Student existingStudentObj= opt.get();

     dao.save(student);
     //if id field is not found this method will perform as insert operation
     //if id field is found this method will perform as update operation


    }else
      throw new StudentException("Student does not exist..");


    return student;

  }
```

Example: Updating only few fields(only marks of a student):

Giving a grace marks 50 to the student whose roll number 4

```
@PutMapping("/updateStudentMarksbyRoll/{roll}/{graceMarks}")
  public Student updateMarksByRoll(@PathVariable int roll, @PathVariable int graceMarks) {

Student student= dao.findById(roll).orElseThrow(() -> new StudentNotFoundException("Student not found with Roll                    "+ro

    student.setMarks(student.getMarks()+graceMarks);

    //Student updateStudent=dao.save(student);
```

```
    //return updateStudent;

    return dao.save(student);

   }
```

To call the above API:

PUT :- http://localhost:8088/jpaApp/updateStudentMarksbyRoll/4/50

Defining our own custom methods inside the JpaRepository interface extended interface:

In this interface (**StudentDao**) if we define any custom method by following some convention then the Spring Data-JPA framework will provide its implementation automatically.

Syntax:

```
findBy....
```

Example:

```
public Student findByMarks(int marks);
```

Here **Marks** after **findBy**... is the field name of our Entity class. not the column name of the table.

In the above method if with the marks only one object will come then we need to take "Student" as a return type and if with that marks multiple result will come then we need to change the return type as "List<Student>"

ex:-

```
public List<Student> findByMarks(int marks);
```

Note: If our search condition is on the primary key field then, mention the only one object as the return type, where as if our search condition is on non-primary key field then mention the return type as List.

Example: **StudentDao.java**

```
public interface StudentDao extends JpaRepository<Student, Integer>{

  public List<Student> findByMarks(int marks);
}
```

In **StudentController** class define another method as follows:

```
@GetMapping("/getStudentByMarks/{marks}")
  public List<Student> getStudentByMarksHandler(@PathVariable int marks) {


    return studentDao.findByMarks(marks);

  }
```

To call the above API:

GET :- http://localhost:8088/jpaApp/getStudentByAddress/800

findBy.. multiple fields:

```
public List<Student> findByNameAndMarks(String name, int marks); // here both condition will match(AND)

public List<Student> findByNameOrMarks(String name, int marks); // here any condition will match(OR)
```

Getting some more complex queries:

```
public List<Student> findByMarksLessThan(int marks);

public List<Student> findByMarksLessThanEqual(int marks);

public List<Student> findByMarksGreaterThan(int marks);

public List<Student> findByMarksBetween(int s_marks,int e_marks)
```

Limiting the result:

```
public List<Student> findTop5ByMarks(int marks);
```

Using orderBy:

```
public List<Student> findTop5ByOrderByNameDesc();
```

Note: Even if we need some more complex query like to use joins, subqueries, getting only one or few columns, etc. then we need to use JPQL by using @Query annotation.

Refer the JPQL notes from the JPA-Hibernate session.

Example: define following method inside the StudentDao interface

```
@Query("select s.name from Student s where s.roll=:roll")
public String getStudentNameByRoll(@Param("roll") int roll);
```

Here s.name is the variable name and Student is the class name not the table and column names

Here we have used **named parameter**

Example of using positional parameter:

```
@Query("select s.name from Student s where s.roll=?1")
public String getStudentNameByRoll(int roll);
```

Note: If we will try to project only one column then the return type of our JPQL @Query annotated method will be either String or wrapper class or LocalDate class.

Example:

```
@Query("select s.marks from Student s where s.roll=?1")
  public Integer getStudentMarksByRoll(int roll);
```

Selecting multiple columns using JPQL:

If we will try to project multiple columns then we can take the return type either any one from following:

I**f only one row will come: then**

1. String: Here all the column will come as a comma separated String
2. String[]: each column will come as a String array.
3. any Custom Bean class. (DTO class) (Single Bean object)

If multiple row will come: then

1. List<String>
2. List<String[]>
3. List<Custome bean DTO>

Example: Two column and one row:

```
inside the StudentDao.java:-
----------------------------

  @Query("select s.name,s.marks from Student s where s.roll=?1")
  public String getNameAndMarksfromStudent(int roll);
```

Inside the StudentController class:

```
@GetMapping("/getNameAndMarksByRoll/{roll}")
  public String getStudentNameAndMarksByRoll(@PathVariable int roll) {


    String result=studentDao.getNameAndMarksfromStudent(roll);

    String[] sr= result.split(",");

    System.out.println("Name is "+sr[0]);
    System.out.println("Marks is "+sr[1]);


    if(result == null) {
      throw new StudentException("Student does not exist with the Roll "+roll);
    }
    else
      return result;

  }
```

Example: Two column and multiple row:

```
in StudentDao.java :-
----------------------

  @Query("select s.name,s.marks from Student s where s.marks=?1")
  public List<String> getNameAndMarksfromStudentbyMarks(int marks);
```

Inside the StudentController class:

```
@GetMapping("/getNameAndMarksByMarks/{marks}")
  public List<String> getStudentNameAndMarksByMarks(@PathVariable int marks) {


    List<String> result=studentDao.getNameAndMarksfromStudentbyMarks(marks);



  if(result.size() == 0) {
    throw new StudentNotFoundException("Student does not exist with the marks "+marks);
  }
    else
      return result;


  }
```