SPRING DATA JPA(practical)

Important points:

We can use only predefined methods;

Ex:order-findby-coursename-containing.

If you define by your own then yo need query for that

\*\* if you want to create relation for a table then

1,create a object of other table

Ex:teacher in student.

2,use join column such that give id of that existing table to create foreign key.

For many to many need joincolumns inside joincolumn used.

Later table created with with that key.

\*\*as it is in course we should create test methods in courserepository only.

Before creating application,

1.create controller and add a function through post mapping which returns string make sure it is accessible in browser in that api call using restcontroller and postmapping.

2.now create student entity and extend it in repository after that create a save function in service using student repository as dependency injection.

3.now run and make sure student entity is created in h2 databas and try to send data and view it.

STEP-1:

\*spring initializer-h2-database

Springweb

Lambok

Spring-data jpa

Step-2:(entity with jpa annotations)

Create student table in entity.

While creating entity use some constraints.

Code:

/\*package com.sandeep.springdatapa1.Entity;  
  
import jakarta.persistence.\*;  
import jakarta.validation.constraints.Email;  
import jakarta.validation.constraints.NotBlank;  
import lombok.AllArgsConstructor;  
import lombok.Builder;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Entity  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@Table(  
 name="student",  
 uniqueConstraints= @UniqueConstraint(  
 name = "email\_address\_unique",  
 columnNames = "email\_address")  
  
)  
public class Student {  
  
 @SequenceGenerator(  
 name="studentsequence",  
 sequenceName="studentsequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.SEQUENCE,  
 generator = "studentsequence"  
 )  
 @Id  
 private Long Id;  
 @Column(name="firstname")  
 private String Firstname;  
 private String Lastname;  
 @Column(name = "email\_address",nullable = false)  
 private String Email;  
 @Embedded  
 private Guardian guardian;  
}\*/  
package com.sandeep.springdatapa1.Entity;  
  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Builder;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Entity  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@Table(  
 name = "student",  
 uniqueConstraints = @UniqueConstraint(  
 name = "email\_address\_unique",  
 columnNames = "email\_address")  
)  
public class Student {  
  
 @SequenceGenerator(  
 name = "studentsequence",  
 sequenceName = "studentsequence",  
 allocationSize = 1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "studentsequence"  
 )  
 @Id  
 @Column(name = "id")  
 private Long id; // Use lowercase 'id' to match typical conventions  
  
 @Column(name = "firstname")  
 private String firstname; // Use lowercase 'firstname' to match repository method  
  
 @Column(name = "lastname")  
 private String lastname; // Use lowercase 'lastname'  
  
 @Column(name = "email\_address", nullable = false)  
 private String email; // Use lowercase 'email'  
  
 @Embedded  
 private Guardian guardian;  
}

**1. Annotations**

* @Entity: Marks the class as a JPA entity, meaning it's mapped to a database table.
* @Data: Lombok annotation that generates getters, setters, toString(), equals(), and hashCode() methods automatically.
* @AllArgsConstructor: Lombok annotation that generates a constructor with all fields as parameters.
* @NoArgsConstructor: Lombok annotation that generates a default no-argument constructor.
* @Builder: Lombok annotation that provides a builder pattern for object creation.
* @Table: Specifies the table name as student and sets a unique constraint on the email\_address column to ensure no two students have the same email.
* @UniqueConstraint: Ensures the uniqueness of the email\_address field in the student table.
* @SequenceGenerator: Defines a sequence generator named studentsequence to automatically generate unique IDs for new students. It uses the database sequence studentsequence with an allocation size of 1.
* @GeneratedValue: Specifies that the id field is auto-generated by using the studentsequence generator with the SEQUENCE strategy.
* @Id: Marks the id field as the primary key for the table.
* @Column: Used to specify column properties for fields in the database table.

**2. Fields**

* id: The primary key of the student table. It is auto-generated using the sequence defined by @SequenceGenerator and mapped to the id column in the table.
* firstname: Mapped to the firstname column in the table. This field holds the student's first name.
* lastname: Mapped to the lastname column in the table, holding the student's last name.
* email: Mapped to the email\_address column in the table, which stores the student's email. It is marked as nullable = false, meaning this field must have a value and cannot be null.
* guardian: An @Embedded object representing another class (not shown here) called Guardian, which is embedded into the student entity. This means the fields of the Guardian class will be included as columns in the student table.

**3. Lowercase Field Names**

* The field names in the second version of the class (e.g., firstname, lastname, email) follow typical Java naming conventions (camel case), where variables start with a lowercase letter. This is crucial because JPA repository methods often rely on field names matching exactly with the column names.

**Key Changes in the Updated Code**

* **Consistency in Naming**: The second version of the class fixes naming conventions, changing fields like Id, Firstname, and Lastname to id, firstname, and lastname for better consistency and adherence to typical Java conventions.
* **Email Field**: The field email is still mapped to the email\_address column, ensuring uniqueness and non-nullability.

Step-3(Save methods for testing):

Declared in studentrepository Test

Below we will create a save method in that we will use  
Student.builder.build() method to save the data.

Below statements are are to show the values.

@Test  
public void savetest() {  
 Student student = Student.*builder*().email("sandeep443@gmail.com").firstname("sandeep").lastname("naidu").build();  
 this.studentrepository.save(student);  
 Long studentId = student.getId();  
 Assertions.*assertNotNull*(studentId, "Student ID should not be null after save.");  
 Optional<Student> savedStudent = this.studentrepository.findById(studentId);  
 Assertions.*assertTrue*(savedStudent.isPresent(), "Saved student should be present.");  
 Assertions.*assertEquals*("sandeep443@gmail.com", ((Student)savedStudent.get()).getEmail(), "Email should match.");  
 System.*out*.println("Saved Student: " + String.*valueOf*(savedStudent.get()));  
}

Step-4:(embedded and embeddable ):

These are used to separate a table of java code into two tables.

Ex:student has guardian name and guardian mobile with this you can create two entities but can maintain relation by using embedded and embeddable.

Student entity:

@Id  
 @Column(name = "id")  
 private Long id; // Use lowercase 'id' to match typical conventions  
  
 @Column(name = "firstname")  
 private String firstname; // Use lowercase 'firstname' to match repository method  
  
 @Column(name = "lastname")  
 private String lastname; // Use lowercase 'lastname'  
  
 @Column(name = "email\_address", nullable = false)  
 private String email; // Use lowercase 'email'  
  
 @Embedded  
 private Guardian guardian;  
}

Guardian entity:

@Embeddable

public class Guardian {  
 private String name;  
 private String mobile;  
 private String email;  
}

both are related using embedded and embeddable methods.

Step-5:creating jpa repository methods in studentrepositorytest):

In this to save data.

Use builder -it is like post mapping

Use .save method()-it stores the data that comes from builder function(just like post mapping api in backeend).

Basic method to test.  
@SpringBootTest don’t use datajpatest data will not be stored.  
@AutoConfigureTestDatabase(  
 replace = Replace.*NONE*)  
class StudentrepositoryTest {  
 @Autowired  
 private Studentrepository studentrepository;  
  
  
 }  
}

\*These are declared in studentrepository.

Rules:ex

public List<Student> findByFirstname(String firstname);

if you are writing function to find firstname:

1.use findBy later extend it to findByFirstaname(here f should be capital even though your first name is small)

2.later use findByFirstname(String firstname);

Here parameter should be in lower case.

By following above rules we declare below ones.

public List<Student> findByFirstname(String firstname);  
  
public List<Student> findByFirstnameContaining(String firstname);  
  
public List<Student> findByGuardian\_name(String guardian\_name);

\*below are declared in in repositorytest.

@Test  
public void findbyname() {  
 List<Student> student1 = this.studentrepository.findByFirstname("nanna");  
 System.*out*.println(student1);  
}

We are returning a list student captured in list and we got the function using studentrepository as it saved in it.  
  
@Test  
public void finfbynamecontaining() {  
 List<Student> student2 = this.studentrepository.findByFirstnameContaining("na");  
 System.*out*.println(student2);  
}

Step-6(query Annatation using getBy):

@Query("select s from Student s where s.email=?1")  
 public Student getByEmail(String email\_address);

Here instead of using predefined methods we can declare a query by our own).

In repository test:

We are returning sinle object so student declared instead of list.

@Test  
public void printbyemail() {  
 Student student3 = this.studentrepository.getByEmail("sandeep443@gmail.com");  
 System.*out*.println(student3);  
}

Step-7(native query annatation):

This is for complex queries.

@Query(  
 value="select \* from student s where s.email\_address=?1",

In repository test:

@Test  
public void getByEmailNative() {  
 Student student4 = this.studentrepository.getByEmailNative("sandeep443@gmail.com");  
 System.*out*.println(student4);  
}

nativeQuery=true  
)  
 public Student getByEmailNative(String email\_address);

step-8(getemail using params):

@Query(  
 value="select \* from student s where s.email\_address=?1",  
 nativeQuery=true  
 )  
 public Student getByEmailNativebyparams(@Param("email\_address") String email\_address);

@Test  
 public void getByEmailNativebyparams(){  
 Student student5=studentrepository.getByEmailNativebyparams("sandeep443@gmail.com");  
  
 System.*out*.println(student5);  
  
  
}

Using two parameters in update query

step-9(updating email using firstname and using a query):  
 @Modifying  
 @Transactional  
 @Query(  
 value="update Student set firstname=?1 where email\_address=?2",  
 nativeQuery=true  
 )  
 public void updateStudentByEmail(String firstname,String email\_address);

test case:

@Test  
 public void updateStudentByEmail(){  
 studentrepository.updateStudentByEmail("srinuvasrao","sandeep443@gmail.com");  
}

}

Step-9(one to one relation ship):

@OneToOne  
 @JoinColumn(  
 name = "course\_id",  
 referencedColumnName = "courseid"  
 )

It tells about foreign key and type of relationship.

Create two tables using one to one relation ship.

package com.sandeep.springdatapa1.Entity;  
  
import jakarta.persistence.\*;  
import lombok.AllArgsConstructor;  
import lombok.Builder;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class Coursematerial {  
 @Id  
 @SequenceGenerator(  
 name="sequence\_name1",  
 sequenceName = "sequence\_name1",  
 allocationSize = 1  
  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "sequence\_name"  
 )  
 private Long coursematerialid;  
 private String url;  
 @OneToOne  
 @JoinColumn(  
 name = "course\_id",  
 referencedColumnName = "courseid"  
 )  
 private Course course;  
}

step-10: (using cascade).

initially it fails later need to add this in courematerial to run.

@OneToMany(  
 cascade = CascadeType.*ALL*,  
   
)

Above one should be declared in one teacher class

And in other class it should be declared as

Ex:In course material.

@OneToMany(  
  
 mappedBy = "teacher"  
)

@OneToOne(  
 cascade = CascadeType.*ALL*)

Add data by creating test save method.

Use builder to load data data .

Use save method to store data into spring data jpa.

@Test  
public void saveCoursematerial() {  
 Course course = Course.*builder*().coursename("dsa").credited(6).build();  
 Coursematerial coursematerial=Coursematerial.*builder*().url("http//datastructure").course(course).build();  
 coursematerialrepository.save(coursematerial);  
}

Step-11:(fetchin data in one-one relationship):

@OneToOne(  
 cascade = CascadeType.*ALL*,  
 fetch = FetchType.*LAZY*)

Need to add this step then only we can add the data.

Step-11(uni and bidirectional):

When we run findall in courserepositorytest it shows below values.

Course(courseid=2, coursename=dsa, credited=6,

After adding below lines coursematerial values can also be viewed adding the one to one mapping in couses.

@OneToOne(mappedBy =  
"course")  
private Coursematerial coursematerial;

After declaring one to on ecourses below is the answer.

[Course(courseid=2, coursename=dsa, credited=6, coursematerial=Coursematerial(coursematerialid=1, url=http//datastructure))

Step-12:one to many JPA relationship(teacher table);

Keypoints:

Add sequence type.

Add generator value

Add one to many annatation(mention type=cascade in it).

@OneToMany(  
 cascade = CascadeType.*ALL* )

Add column name in join column.

@JoinColumn(  
 name = "teacher\_id",  
 referencedColumnName = "teacherid"  
 )

It alters course and adds a fpreign key teacher is added in couse.

It means a new column with teacher name is created in the courses because of giving a clumn in join column.

@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class Teacher {  
 @Id  
 @SequenceGenerator(  
 name="teachersequence",  
 sequenceName = "teachersequence",  
 allocationSize=1  
 )  
 @GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "teacher\_sequence"  
 )  
 public Long taecherid;  
 public String teachernfirstame;  
 public String teacherlastname;  
 @OneToMany(  
 cascade = CascadeType.*ALL* )  
 @JoinColumn(  
 name = "teacher\_id",  
 referencedColumnName = "teacherid"  
 )  
 private List<Course> courses;  
}

Step-11(many to one relationship):

\*In step level we have one to many in teacher remove that.

\*Add many to one in course and join column in that.

\*now add the data by creating saveteaherfromcourse.

@ManyToOne(  
 cascade = CascadeType.*ALL*)  
@JoinColumn(  
 name = "teacher\_id",  
 referencedColumnName = "teacherid"  
)  
private Teacher teacher;

@Test  
public void coursesaveteacerobject() {  
 Teacher teacher=Teacher.*builder*().teachernfirstame("spandana").teacherlastname("spandinchu").build();  
 Course course=Course.*builder*().coursename("dsaa").credited(8).teacher(teacher).build();  
 courserepository.save(course);  
}

Step12(paging and sorting):

Below describes:

If we have 6 records.

Pageable firstpage= PageRequest.*of*(0,3);-it tells to consider 3 records as a single page..

Long elements=courserepository.findAll(firstpage).getTotalElements();

To see number of pages.

Long elements1= (long) courserepository.findAll(firstpage).getTotalPages();

To see total content.

List<Course> courses=courserepository.findAll(firstpage).getContent();

@Test  
 public void pagenstionfind(){  
 Pageable firstpage= PageRequest.*of*(0,3);  
 Pageable secondpage=PageRequest.*of*(0,2);  
 List<Course> courses=courserepository.findAll(firstpage).getContent();  
 Long elements=courserepository.findAll(firstpage).getTotalElements();  
 System.*out*.println(elements);  
  
 Long elements1= (long) courserepository.findAll(firstpage).getTotalPages();  
 System.*out*.println(elements1);  
 System.*out*.println(courses);  
}

Next:

(sorting the records):

Pageable sortbycreditedandcoursename= PageRequest.*of*(0,2, Sort.*by*("coursename").descending().and(Sort.*by*("credited")));

It stores records and display in println according to the descending order.

@Test  
 public void pagenstionfindallsorting(){  
 Pageable sortbytitle= PageRequest.*of*(0,2, Sort.*by*("coursename"));  
 Pageable sortbycredited= PageRequest.*of*(0,2, Sort.*by*("credited").descending());  
 Pageable sortbycreditedandcoursename= PageRequest.*of*(0,2, Sort.*by*("coursename").descending().and(Sort.*by*("credited")));  
 List<Course> courses=courserepository.findAll(sortbytitle).getContent();  
 System.*out*.println(courses);  
}

Using findbycoursenamecontaining():

We use only single parameter string to find it but we mpdify it by addin a parameter paging.

Ex:

Page<Course> findByCoursenameContaining(String title, Pageable pageRequest );

Findbycourse() used inside the paging:

We can use only predefined methods;

Ex:order-findby-coursename-containing.

If you define by your own then yo need query for that

@Test  
 public void findBytitlepaging(){  
 Pageable firsttenpage=PageRequest.*of*(0,10);  
 List<Course> courses=courserepository.findByCoursenameContaining("D",firsttenpage).getContent();  
 System.*out*.println(courses);  
}

Step-13:

(JPA many to many relationships):

In many to many we need to join both the columns such that it creates a table.

\*For one tableid-

@JoinTable(  
 name = "coursestudentmapping",  
 joinColumns=@JoinColumn(  
 name = "studentcourseid",  
 referencedColumnName = "courseid"  
 )

\*For next table tableid,

inverseJoinColumns=@JoinColumn(  
 name="student\_id",  
 referencedColumnName = "id"  
  
 )

@ManyToMany()  
@JoinTable(  
 name = "coursestudentmapping",  
 joinColumns=@JoinColumn(  
 name = "studentcourseid",  
 referencedColumnName = "courseid"  
 ),  
 inverseJoinColumns=@JoinColumn(  
 name="student\_id",  
 referencedColumnName = "id"  
  
 )  
)  
private List<Student> students;