**Java Spring & AWS**
Sep 7, 2021 - Oct 8, 2021
Monday to Friday
9:30 AM ET - 4:30 PM ET


# Agenda: Day -12

- Database access using spring
  - Spring Data JPA

**@Entity signifies that the entity object has significance all by itself it doesn't require any further association with any other object. Where as @Embeddable object doesn't carry any significance all by itself, it needs association with some other object.**



# Annotations

you need to inform Spring Data JPA on what queries you need to execute. You do it using the @Query annotation

# Annotations

```
import java.util.List;

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

import com.bezkoder.spring.datajpa.model.Tutorial;

public interface TutorialRepository extends JpaRepository<Tutorial, Long> {
  List<Tutorial> findByPublished(boolean published);
  List<Tutorial> findByTitleContaining(String title);
}
```

**Now we can use JpaRepository's methods: save(), findOne(), findById(), findAll(), count(), delete(), deleteById()… without implementing these methods.**

**We also define custom finder methods:**
**– findByPublished(): returns all Tutorials with published having value as input published.**
**– findByTitleContaining(): returns all Tutorials which title contains input title.**



# Annotations

```
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
@Entity(name = "Book")
@Builder
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Book {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int id;
  @Column(unique=true)
  private String isbn;
  private String title;
  private String author;
  private boolean status;
}
```

# Simple @Query

The repository interface extends CrudRepository (https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html). Here, I will use the @Query annotation to create a custom query to find all books.

The code of the BookRepository is this.

```
import guru.springframework.customquery.domain.Book;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
@Repository
public interface BookRepository extends CrudRepository<Book, Integer> {
  @Query("SELECT b FROM Book b")
List<Book> findAllBooks();
}
```

In the preceding code, the findAllBooks() method is annotated with the @Query annotation. This annotation takes a custom query as a string. In this example, the custom query returns all books.



# JPQL Select @Query with Index Parameters

One way to pass method parameters to a query is through an index.

Let's define a custom query using Java Persistence Query Language (JPQL) to find a book for a given title and author.

The code for querying a book with index parameters using JPQL is this.

```
@Query("SELECT b FROM Book b WHERE b.title = ?1 and b.author = ?2")
Book findBookByTitleAndAuthorIndexJpql(String title, String authorName);
```

In the preceding code, the title method parameter will be assigned to the query parameter with index 1. Similarly, authorName will be assigned to the query parameter with index 2.

It is important to note that the order of the query parameter indexes and the method parameters must be the same.

# JPQL vs Native query

JPQL is used to write database independent queries. You can define these queries in the entity class..

If you want to write database specific queries, you can define native queries in the entity.

```
@Query("delete from Student where firstName = :firstName")
void deleteStudentsByFirstName(@Param("firstName") String firstName);

@Query(value = "select * from student", nativeQuery = true)
List<Student> findAllStudentNQ();
```



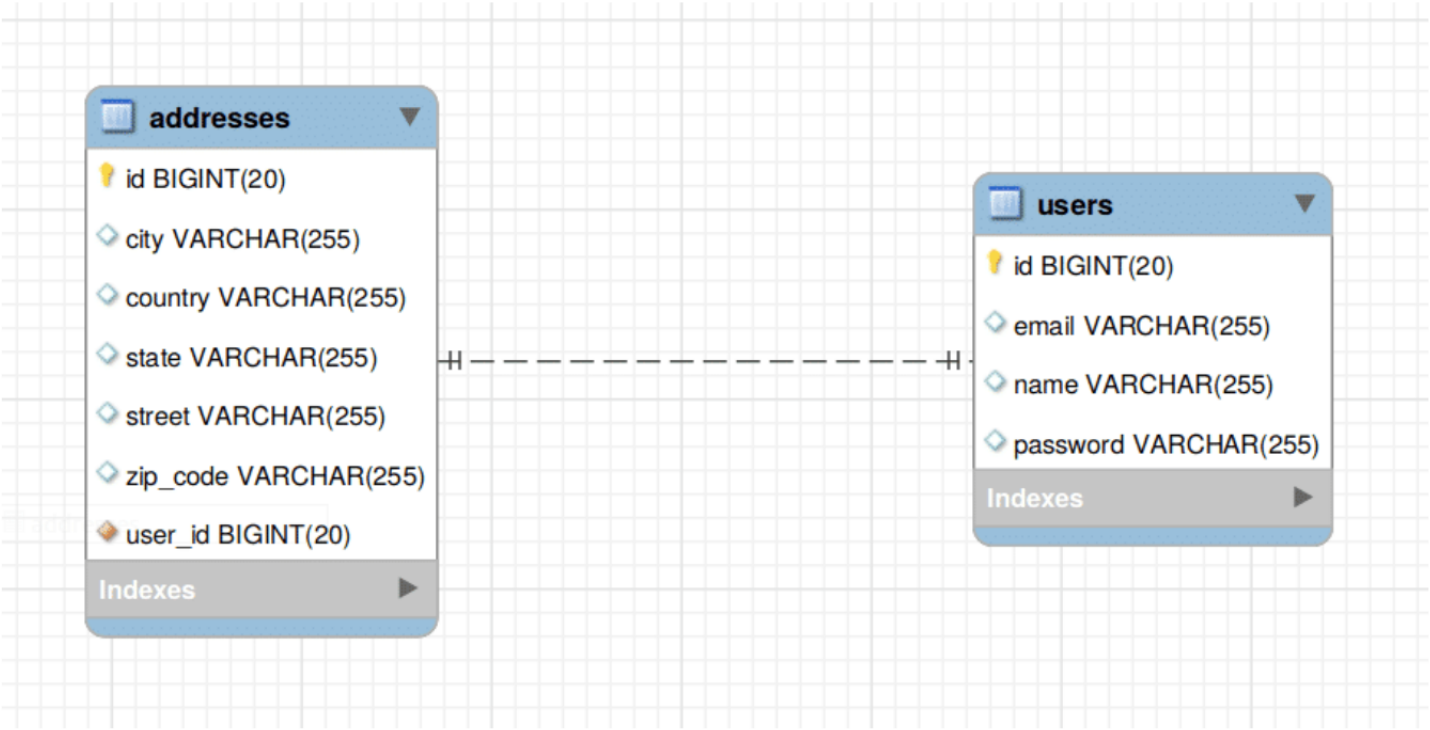
# JPA associations

## One-To-One Relationship

A one-to-one relationship refers to the relationship between two entities/database tables A and B in which **only one element/row of A may only be linked to one element/row of B**, and vice versa.

# JPA associations

## One-To-One Relationship




The one-to-one relationship is defined by using a foreign key called user_id in the addresses table.



# JPA associations

## One-To-One Relationship

```
@Entity
@Table(name = "users")
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    private String password;

    @OneToOne(mappedBy = "user", fetch = FetchType.LAZY,
          cascade = CascadeType.ALL)
    private Address address;

    public User() {
    }
```

```
@Entity
@Table(name = "addresses")
public class Address implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String street;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    @OneToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "user_id", nullable = false)
    private User user;
```

@ Copyright 2020, Summitworks Technologies Inc.

## @OneToOne Annotation

In Spring Data JPA, a one-to-one relationship between two entities is declared by using the **@OneToOne** annotation. It accepts the following parameters:

**fetch** — Defines a strategy for fetching data from the database. By default, it is EAGER which means that the data must be eagerly fetched. We have set it to LAZY to fetch the entities lazily from the database.
**cascade** — Defines a set of cascadable operations that are applied to the associated entity. CascadeType.ALL means to apply all cascading operations to the related entity. Cascading operations are applied when you delete or update the parent entity.
**mappedBy** — Defines the entity that owns the relationship which is the Address entity in our case.
**optional** — Defines whether the relationship is optional. If set to false then a non-null relationship must always exist.



JPA associations

## @OneToOne Annotation

In a bidirectional relationship, we have to specify the @OneToOne annotation in both entities. But only one entity is the owner of the association. Usually, the child entity is one that owns the relationship and the parent entity is the inverse side of the relationship

@ Copyright 2020, Summitworks Technologies Inc.

## @JoinColumn Annotation

The @JoinColumn annotation is used to specify the foreign key column in the owner of the relationship. The inverse-side of the relationship sets the @OneToOne's mappedBy parameter to indicate that the relationship is mapped by the other entity.

The @JoinColumn accepts the following two important parameters, among others:

name — Defines the name of the foreign key column.
nullable — Defines whether the foreign key column is nullable. By default, it is true.



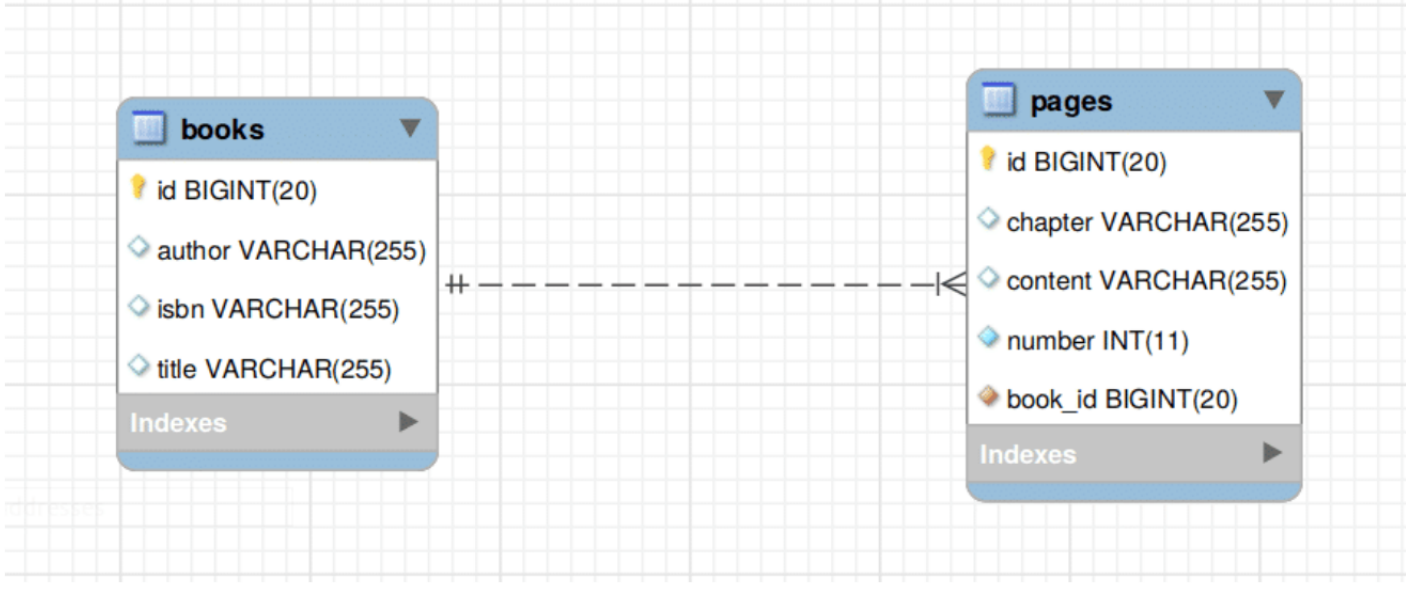


## One-To-Many Relationship

A one-to-many relationship refers to the relationship between two entities/tables A and B in which **one element/row of A may only be linked to many elements/rows of B, but a member of B is linked to only one element/row of A**.

For instance, think of A as a book, and B as pages. A book can have many pages but a page can only exist in one book, forming a one-to-many relationship. The **opposite of one-to-many is many-to-one** relationship.
Let us model the above relationship in the database by creating two tables, one for the books and another for the pages

## One-To-Many Relationship




The one-to-many relationship is defined by the foreign key book_id in the pages table.



JPA associations

## One-To-Many Relationship

```
@Entity
@Table(name = "books")
public class Book implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String author;
    @Column(unique = true)
    private String isbn;

    @OneToMany(mappedBy = "book", fetch = FetchType.LAZY,
            cascade = CascadeType.ALL)
    private Set<Page> pages;

    public Book() {
    }
```

```
@Entity
@Table(name = "pages")
public class Page implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private int number;
    private String content;
    private String chapter;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "book_id", nullable = false)
    private Book book;

    public Page() {
    }
```

@ Copyright 2020, Summitworks Technologies Inc.

## One-To-Many Relationship

**@OneToMany Annotation:**
A one-to-many relationship between two entities is defined by using the @OneToMany annotation in Spring Data JPA. It declares the mappedBy element to indicate the entity that owns the bidirectional relationship. Usually, the child entity is one that owns the relationship and the parent entity contains the @OneToMany annotation.

**@ManyToOne Annotation:**
The @ManyToOne annotation is used to define a many-to-one relationship between two entities in Spring Data JPA. The child entity, that has the join column, is called the owner of the relationship defined using the @ManyToOne annotation.

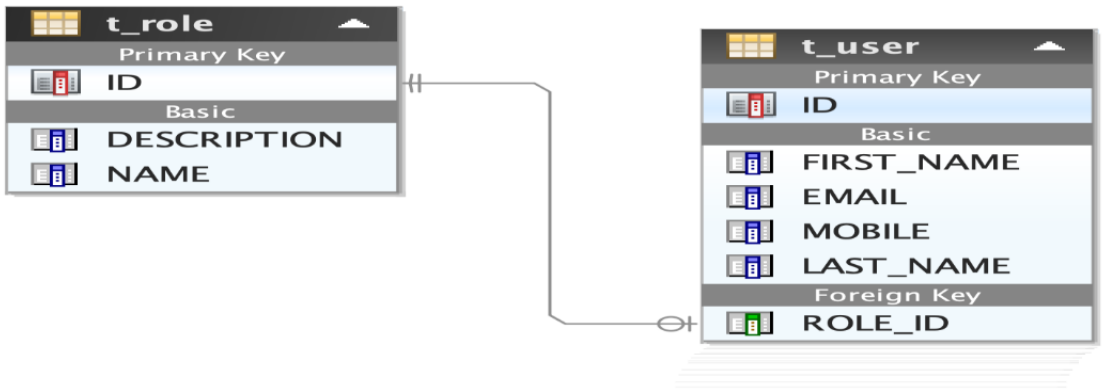
**@JoinColumn Annotation:**
The @JoinColumn annotation is used to specify the foreign key column in the owner of the relationship. The inverse-side of the relationship sets the mappedBy attribute to indicate that the relationship is owned by the other entity.



JPA associations

### One-To-Many Relationship:

If its bidirectional, like below




@ Copyright 2020, Summitworks Technologies Inc.

## One-To-Many Relationship:

If its bidirectional, like below

```
@Entity
@Table(name = "t_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String mobile;
    @Column(unique = true)
    private String email;
    @ManyToOne
    private Role role;
    public Long getId() {
        return id;
    }
```

```
@Entity
@Table(name = "t_role")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String description;
    @OneToMany(targetEntity = User.class)
    private List<User> users;
    public Long getId() {
        return this.id;
    }
    public void setId(Long id) {
        this.id = id;
    }
```



JPA associations

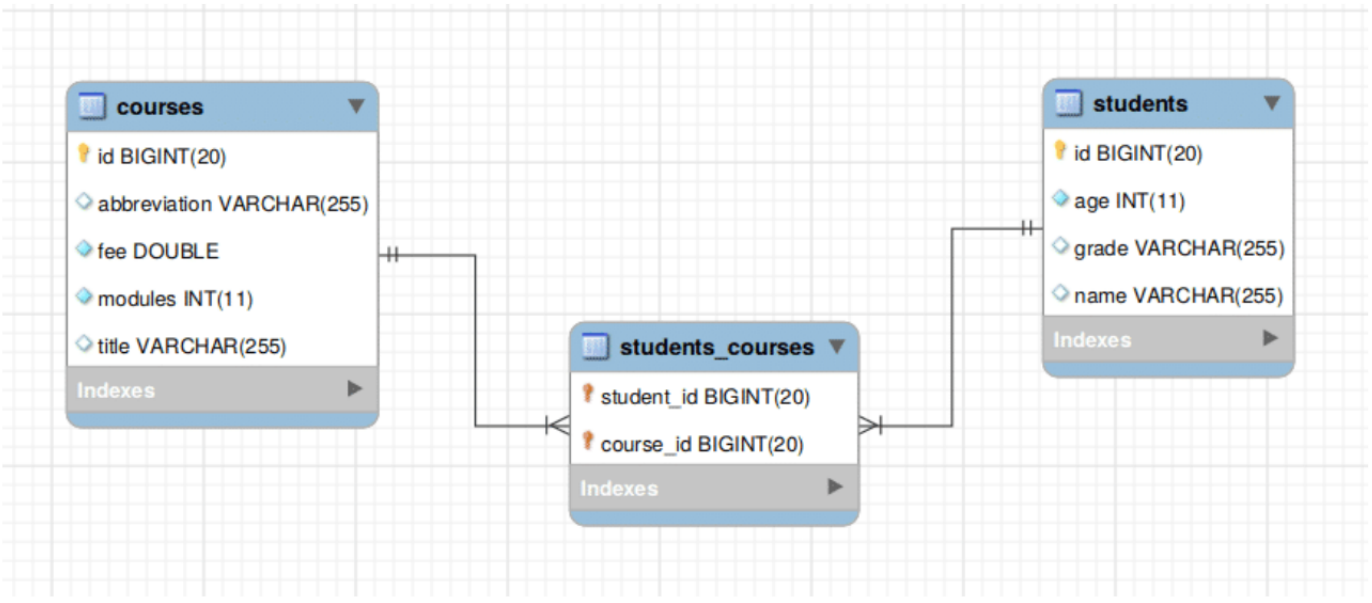
**Many-To-Many Relationship**

A **many-to-many** relationship refers to the relationship between two entities/tables A and B in which one element/row of A may only be associated with many elements/rows of B and vice versa.

A typical example of such a many-to-many relationship is the relationship between students and courses. A student can enroll in multiple courses and a course can also have multiple students, thus forming a many-to-many relationship.

To model the above relationship in the database, you need to create three tables, one each for both students and courses, and another one for holding relationship keys

**Many-To-Many Relationship**




students_courses is a join table that contains two foreign keys, student_id and course_id, to reference both students and courses database tables. Both these foreign keys also act as a composite primary key for the students_courses table.



# JPA associations : Many-To-Many Relationship

```
@Entity
public class Student implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private int age;
    private String grade;

    @ManyToMany(fetch = FetchType.LAZY, cascade =
CascadeType.PERSIST)
    @JoinTable(name = "students_courses",
          joinColumns = {
                @JoinColumn(name = "student_id", referencedColumnName
= "id",
                      nullable = false, updatable = false)},
          inverseJoinColumns = {
                @JoinColumn(name = "course_id", referencedColumnName
= "id",
                      nullable = false, updatable = false)})
    private Set<Course> courses = new HashSet<>();

    public Student() {
    }
```

```
@Entity
@Table(name = "courses")
public class Course implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String abbreviation;
    private int modules;
    private double fee;

    @ManyToMany(mappedBy = "courses", fetch = FetchType.LAZY)
    private Set<Student> students = new HashSet<>();

    public Course() {
    }
```

# JPA associations : Many-To-Many Relationship

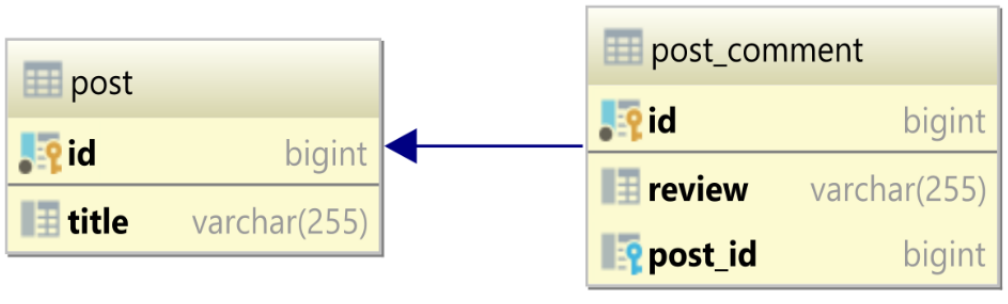
**@ManyToMany Annotation**
A many-to-many relationship between two entities is defined by using the @ManyToMany annotation in Spring Data JPA. It uses the mappedBy attribute to indicate the entity that owns the bidirectional relationship. In a bidirectional relationship, the @ManyToMany annotation is defined in both entities but only one entity can own the relationship. We've picked the Student class as an owner of the relationship in the above example.

**@JoinTable Annotation**
The @JoinTable annotation defines the join table between two entities on the owner's side of the relationship. We have used this annotation to define the students_courses table. If the @JoinTable annotation is left out, the default values of the annotation elements apply. The name of the join table is supposed to be the table names of the associated primary tables concatenated together (owning side first) using an underscore.



## JPA associations

**Many to one Relationship:**




The post_comment table has a post_id column that has a Foreign Key relationship with the id column in the parent post table. The post_id Foreign Key column drives the one-to-many table relationship

# JPA associations

## Many to one Relationship:

```
@Table(name = "post_comment")
public class PostComment {

    @Id
    @GeneratedValue
    private Long id;

    private String review;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    private Post post;

    //Getters and setters omitted for brevity
}
```



## Any queries?

@ Copyright 2020, Summitworks Technologies Inc.