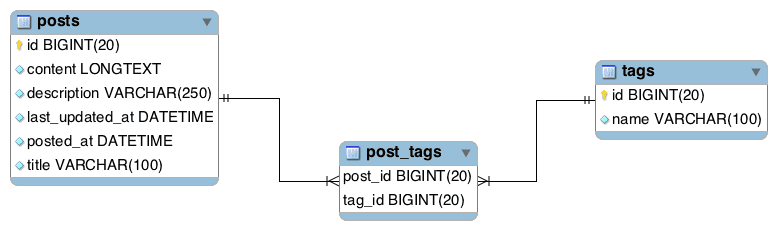
You’ll learn how to map a many-to-many database relationship at the object level in your application using JPA and Hibernate.

Consider the following tables where posts and tags exhibit a many-to-many relationship between each other -

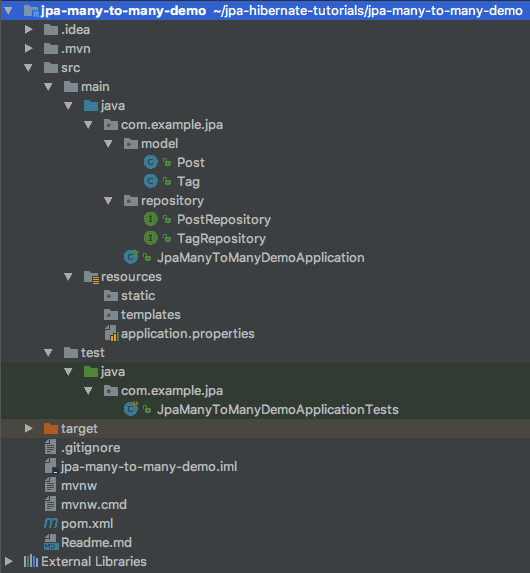
[](https://www.callicoder.com/static/e152cb51ecd5e08c75161a930c3e2f65/108f8/hibernate-spring-boot-jpa-many-to-many-mapping-example-table-structure.png)

The many-to-many relationship is implemented using a third table called post\_tags which contains the details of posts and their associated tags.

Let’s now create a project from scratch and learn how to go about implementing such many-to-many relationship using JPA and Hibernate.

**Creating the Project**

Following is the directory structure of the project for your reference -

[](https://www.callicoder.com/static/72bba7e5a43bd97ed72bf5b5ade578ed/b6a9b/hibernate-spring-boot-jpa-many-to-many-mapping-example-directory-structure.png)

Your bootstrapped project won’t have model and repository packages and all the classes inside these packages at this point. We’ll create them shortly.

**Configuring the Database and Hibernate Log levels**

We’ll need to configure MySQL database URL, username, and password so that Spring can establish a connection with the database on startup.

Open src/main/resources/application.properties and add the following properties to it -

# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)

spring.datasource.url=jdbc:mysql://localhost:3306/jpa\_many\_to\_many\_demo?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false

spring.datasource.username=root

spring.datasource.password=root

# Hibernate

# The SQL dialect makes Hibernate generate better SQL for the chosen database

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = create

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type=TRACE

Be sure to change the spring.datasource.username and spring.datasource.password properties as per your MySQL installation. Also, create a database named jpa\_many\_to\_many\_demo.

The spring.jpa.hibernate.ddl-auto = create property makes sure that the database tables and the domain models in your application are in sync. Whenever you change the domain model, hibernate will automatically update the mapped table in the database when you restart the application.

I have also specified the log levels for hibernate so that we can debug the SQL queries executed by hibernate.

**Defining the Domain models**

Let’s define the domain models which will be mapped to the tables we saw earlier. First, Create a package named model inside com.example.jpa, and then add the following classes inside the model package -

**1. Post model**

package com.example.jpa.model;

import javax.persistence.\*;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import java.util.Date;

import java.util.HashSet;

import java.util.Set;

@Entity

@Table(name = "posts")

public class Post {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotNull

@Size(max = 100)

@Column(unique = true)

private String title;

@NotNull

@Size(max = 250)

private String description;

@NotNull

@Lob

private String content;

@NotNull

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "posted\_at")

private Date postedAt = new Date();

@NotNull

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "last\_updated\_at")

private Date lastUpdatedAt = new Date();

@ManyToMany(fetch = FetchType.LAZY,

cascade = {

CascadeType.PERSIST,

CascadeType.MERGE

})

@JoinTable(name = "post\_tags",

joinColumns = { @JoinColumn(name = "post\_id") },

inverseJoinColumns = { @JoinColumn(name = "tag\_id") })

private Set<Tag> tags = new HashSet<>();

public Post() {

}

public Post(String title, String description, String content) {

this.title = title;

this.description = description;

this.content = content;

}

// Getters and Setters (Omitted for brevity)

}

**2. Tag model**

package com.example.jpa.model;

import org.hibernate.annotations.NaturalId;

import javax.persistence.\*;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import java.util.HashSet;

import java.util.Set;

@Entity

@Table(name = "tags")

public class Tag {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotNull

@Size(max = 100)

@NaturalId

private String name;

@ManyToMany(fetch = FetchType.LAZY,

cascade = {

CascadeType.PERSIST,

CascadeType.MERGE

},

mappedBy = "tags")

private Set<Post> posts = new HashSet<>();

public Tag() {

}

public Tag(String name) {

this.name = name;

}

// Getters and Setters (Omitted for brevity)

}

We use @ManyToMany annotation to create a many-to-many relationship between two entities. In a bi-directional association, the @ManyToMany annotation is used on both the entities but only one entity can be the owner of the relationship.

The entity that specifies the @JoinTable is the owning side of the relationship and the entity that specifies the mappedBy attribute is the inverse side.

In the above example, Post entity is the owner of the relationship and Tag entity is the inverse side.

**Defining the Repositories**

Let’s define the repositories for accessing the Post and Tag data from the database.

First, Create a new package called repository inside com.example.jpa package, then create the following repositories inside the repository package -

**1. PostRepository**

package com.example.jpa.repository;

import com.example.jpa.model.Post;

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

import org.springframework.stereotype.Repository;

@Repository

public interface PostRepository extends JpaRepository<Post, Long> {

}

**2. TagRepository**

package com.example.jpa.repository;

import com.example.jpa.model.Tag;

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

import org.springframework.stereotype.Repository;

@Repository

public interface TagRepository extends JpaRepository<Tag, Long> {

}

**Writing code to test the Many-to-Many relationship**

Time to write some code to test our many-to-many association setup. Open the main class JpaManyToManyDemoApplication.java and replace it with the following code -

package com.example.jpa;

import com.example.jpa.model.Post;

import com.example.jpa.model.Tag;

import com.example.jpa.repository.PostRepository;

import com.example.jpa.repository.TagRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class JpaManyToManyDemoApplication implements CommandLineRunner {

@Autowired

private TagRepository tagRepository;

@Autowired

private PostRepository postRepository;

public static void main(String[] args) {

SpringApplication.run(JpaManyToManyDemoApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

// Cleanup the tables

postRepository.deleteAllInBatch();

tagRepository.deleteAllInBatch();

// =======================================

// Create a Post

Post post = new Post("Hibernate Many to Many Example with Spring Boot",

"Learn how to map a many to many relationship using hibernate",

"Entire Post content with Sample code");

// Create two tags

Tag tag1 = new Tag("Spring Boot");

Tag tag2 = new Tag("Hibernate");

// Add tag references in the post

post.getTags().add(tag1);

post.getTags().add(tag2);

// Add post reference in the tags

tag1.getPosts().add(post);

tag2.getPosts().add(post);

postRepository.save(post);

// =======================================

}

}

The run() method in the above class is called when the application is started. In the run() method, we first clean up the tables, then create a Post, two Tags and then add the tags in the post. Finally, we save the post reference using PostRepository::save() method which saves the Tags as well.

**Running the Application**

You can run the application by typing the following command in the terminal -

mvn spring-boot:run

Once the application starts, check the console for hibernate specific logs to see what SQL statements are executed by hibernate for the operations that we have performed in the run() method -

org.hibernate.SQL : delete from post\_tags where (post\_id) in (select id from posts)

org.hibernate.SQL : delete from posts

org.hibernate.SQL : delete from post\_tags where (tag\_id) in (select id from tags)

org.hibernate.SQL : delete from tags

org.hibernate.SQL : insert into posts (content, description, last\_updated\_at, posted\_at, title) values (?, ?, ?, ?, ?)

org.hibernate.SQL : insert into tags (name) values (?)

org.hibernate.SQL : insert into tags (name) values (?)

org.hibernate.SQL : insert into post\_tags (post\_id, tag\_id) values (?, ?)

org.hibernate.SQL : insert into post\_tags (post\_id, tag\_id) values (?, ?)