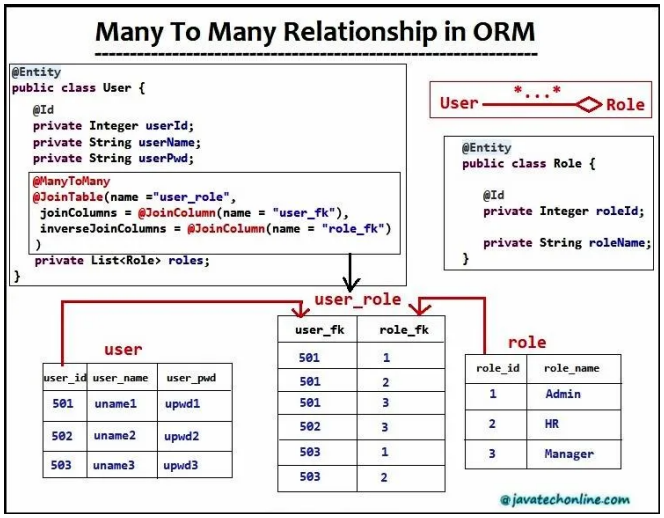
# Entity Relationship In JPA/Hibernate/ORM

java Entity Relationship Hibernate Spring Data JPA by devs5003 - October 28, 2021 0




Needless to say, ORM (Object Relational Mapping) concept has made the developers life easier. Developers don't need to do much work in order to map two tables in the database. If we need to maintain a relationship between two tables, utilization of an annotation in our entity/class is more than sufficient. Further, we don't need to design the DB tables. However, developers need to have a better understanding of the usage of the annotations that create a relationship between two tables. Here in this article 'Entity Relationship in JPA/Hibernate/ORM', we will discuss about the different types of relations between tables that are relevant in ORM concept.

Let's start discussing our topic 'Entity Relationship in JPA/Hibernate/ORM' and its related concepts.

Some of the below FAQs are related to the terminologies used in out article header 'Entity Relationship in JPA/Hibernate/ORM' one by one.

Table of Contents (Click on links below to navigate) [hide]

## What is ORM in Java?

**ORM** stands for **Object-Relational Mapping**. The ORM is a technique for converting data between Java objects and relational databases. It converts data between two incompatible type systems (such as Java and MySQL). After conversion, each model/entity class becomes a table in our database and each instance becomes a row of the table. Additionally, each property/variable of the entity/java class becomes a column in the table.

In order to make ORM work in our Java applications we need an API or specification. In Java, this API is nothing, but JPA (Java Persistence API). Almost all ORM frameworks in Java are built on top of the JPA specification.

## What are the popular ORM tools/frameworks in Java?

The most popular ORM tools in Java are:

1) Hibernate – Open Source. One of the most used framework.
2) Top Link – an Oracle product.
3) Eclipse Link – Eclipse Persistence Platform.
4) Open JPA – an Apache product.
5) MyBatis (Formerly known as iBATIS) – Open Source.

After discussing the article's header 'Entity Relationship in JPA/Hibernate/ORM', let's emphasize more on the terminologies of the entity relationships.

## What is unidirectional and bidirectional relationship?

Using ORM technique, we can configure a relationship as either one way or two way. A unidirectional relationship means that the flow of data is just in one direction. Needless to say, a bidirectional relationship means that the flow of data is in both the directions. Furthermore, a bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

Now let's understand it programmatically. In a unidirectional relationship, only one entity has a relationship field or property that refers to the other. In a bidirectional relationship, each entity has a relationship field or property that refers to the other entity.

## What is owning side & inverse/referencing side in an ORM?

In fact, owning side and inverse side are the technical terminologies of the ORM technology. They are not the concepts of participating entities. In fact, they are the two sides of a bidirectional relationship. The owning side initiates the creation of the relationship to the database. Generally, this is the side where the foreign key resides.

However, from the database point-of-view owning side entity (owner entity) will have a foreign key column. Obviously, the other remaining side is the inverse side or referencing side.The inverse side maps the owning side using mappedBy attribute. Moreover, people in the industry also call them roles of the entity. In every relation there are two entities that are related to one another, each entity play a role which is either Owning Entity or Non-Owning Entity.

## What are the must follow rules for bidirectional relationship?

Bidirectional relationships must follow these rules as mentioned below.

1) The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy attribute provides the value of property or field in the entity that is the owner of the relationship.

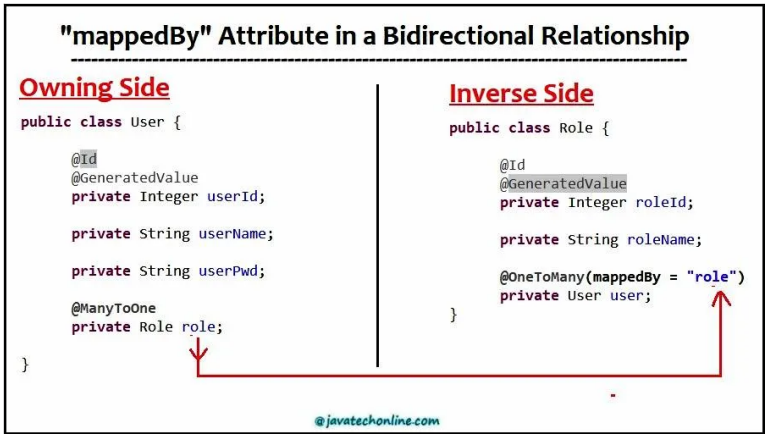
2) The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship. In fact, ManyToOne is always the owning side of a bidirectional association. Similarly, OneToMany is always the inverse side of a bidirectional association.

3) For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.

4) For many-to-many bidirectional relationships, either side may be the owning side. Hence, we can choose the owning side of a many-to-many association ourself.

## Why do we use mappedBy attribute?

The mappedBy attribute contains the name of the association-field on the owning side. We apply mappedBy attribute at the non-owning/inverse side of the relationship. The attribute instructs hibernate/JPA that don't create extra column for this field in the table as it is already created at the owning side. In other words, when we use mappedBy attribute, it means that the relation between entities has already been mapped at owning side, so don't do that twice. Additionally, mappedBy indicates that the relationship between entities is bidirectional. If we don't use mappedBy attribute, it represents two unidirectional relationships. Further, there will be an additional mapping column in each direction. For example, below screen shot represents the use of mappedBy attribute.

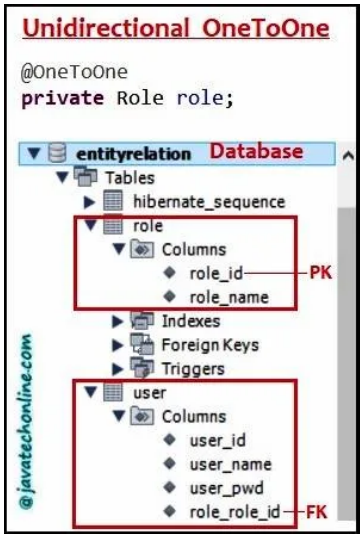



# One To One

**Use-case:** Let's assume that we have to maintain a relation between User & Role table. In order to satisfy One to One relationship between the User and the Role table, one User will have only one Role.

## One To One: Unidirectional

We can obtain a unidirectional relationship between User & Role entity by applying @OneToOne on the relational field at any side. For example, if we want to have a one to one relationship from User to Role, we need to add a field with type Role in the User entity. It means one user will have one role. Hence, we need to apply @OneToOne on the field with a type Role in the User entity.

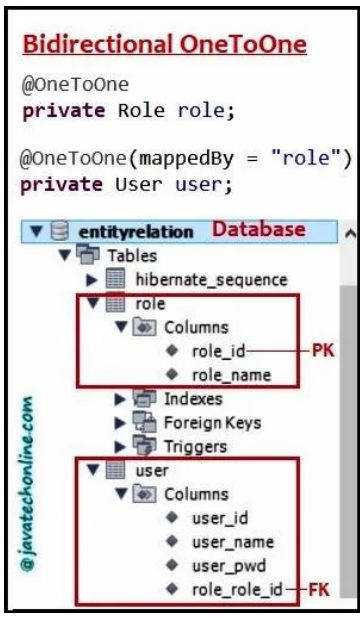



```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class User {
    @Id
    @GeneratedValue
    private Integer userId;
```

```
    private String userName;
    private String userPwd;

    @OneToOne
    private Role role;
}
```

## One To One: Bidirectional

In order to satisfy the bidirectional relationship, we need to apply @OneToOne on both the sides ie. on the field with type Role in User entity and also on the field with type User in the Role entity. Additionally, we need to have mappedBy attribute into any side of @OneToOne to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @OneToOne.




```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;
    private String userName;
    private String userPwd;

    @OneToOne
    private Role role;
}
```

And,

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Role {
```

```
    @Id
    @GeneratedValue
    private Integer roleId;
    private String roleName;

    @OneToOne(mappedBy = "role")
    private User user;
}
```

Here, we completed the first relationship of our article 'Entity Relationship in JPA/Hibernate/ORM'.

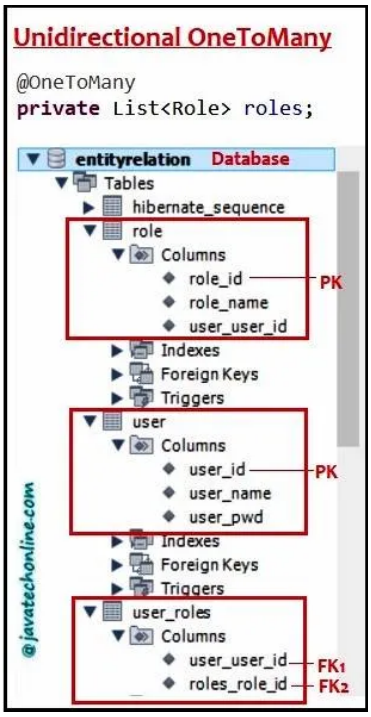
♥ **Note:** To represent the many side of the relationship, here we have used a collection of type List. We can use any other collection as per our requirement.

# One To Many

**Use-case:** Let's assume that we have to maintain a relation of one to many between User & Role table. In order to satisfy One to Many relationship between the User and the Role table, One user will have multiple roles.

## One To Many: Unidirectional

We can obtain a unidirectional relationship between User & Role entity by applying @OneToMany on the relational field at any side. For example, if we want to have a one to many relationship from User to Role, we need to add a field with type List<Role> in the User entity. It means one user will have many roles. Hence, we need to apply @OneToMany on the field with a type List<Role> in the User entity. For example, below code demonstrates the concept.

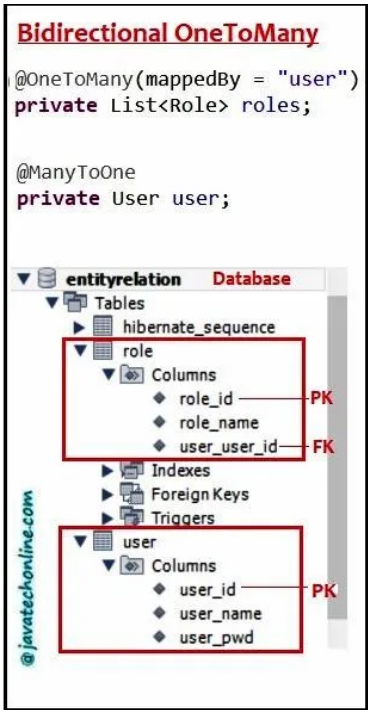



```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;
```

```
    private String userName;
    private String userPwd;

    @OneToMany
    private List<Role> roles;
}
```

## One To Many: Bidirectional

In order to satisfy the bidirectional relationship, we need to apply @OneToMany at one sides ie. on the field with type List<Role> in User entity and @ManyToOne at other side also ie. on the field with type User in the Role entity. Additionally, we need to have mappedBy attribute in @OneToMany to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @ManyToOne and @OneToMany




```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;

    private String userName;
    private String userPwd;

    @OneToMany(mappedBy = "user")
    private List<Role> roles;
}
```

And,

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Role {

@Id
@GeneratedValue
private Integer roleId;

private String roleName;

@ManyToOne
private User user;
}
```

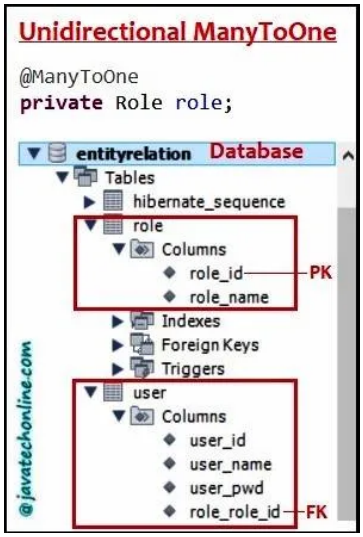
Here, we completed the second relationship of our article 'Entity Relationship in JPA/Hibernate/ORM'.

# Many To One

**Use-case:** Let's assume that we have to maintain a relation between User & Role table. In order to satisfy Many to One relationship between the User and the Role table, multiple Users will have only one Role.

## Many To One: Unidirectional

We can obtain a unidirectional relationship between User & Role entity by applying @ManyToOne on the relational field at any side. For example, if we want to have a many to one relationship from User to Role, we need to add a field with type Role in the User entity. It means many users will have one role. Hence, we need to apply @ManyToOne on the field with a type Role in the User entity. For example, below code demonstrates the concept.

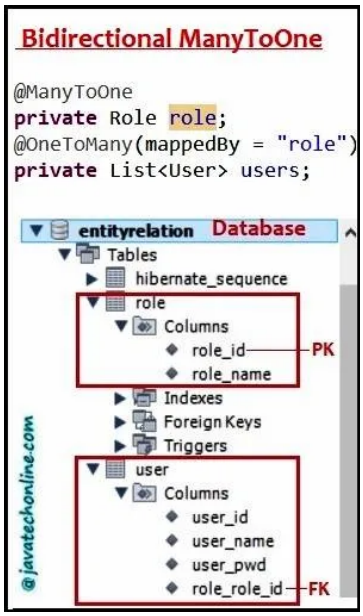



```
import javax.persistence.Entity;
Import Java.persistence.GeneratedValue;
import javax.persistence.Id;
Import Java.persistence.ManyToOne;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;
```

```
    private String userName;
    private String userPwd;

    @ManyToOne
    private Role role;
}
```

## Many To One: Bidirectional

In order to satisfy the bidirectional relationship, we need to apply @ManyToOne at one sides ie. on the field with type Role in User entity and @OneToMany at other side also ie. on the field with type List<User> in the Role entity. Additionally, we need to have mappedBy attribute in @OneToMany to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @ManyToOne and @OneToMany.




```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;

    private String userName;
    private String userPwd;

    @ManyToOne
    private Role role;
}
```

And,

```
import java.util.List;
import javax.persistence.CascadeType;
```

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Role {

    @Id
    @GeneratedValue
    private Integer roleId;

    private String roleName;

    @OneToMany(mappedBy = "role", cascade = CascadeType.ALL)
    private List<User> users;
}
```

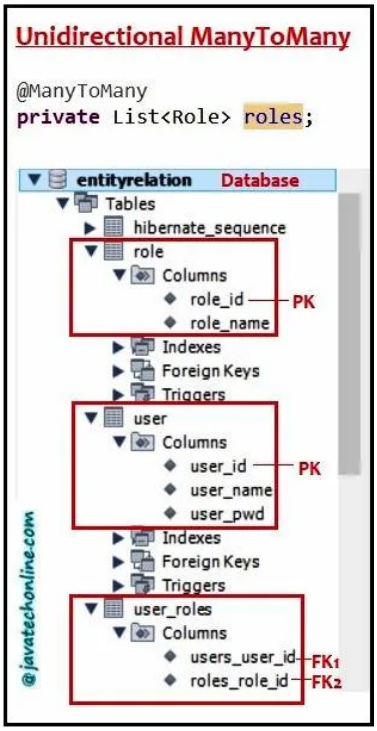
Here, we completed the third relationship of our article 'Entity Relationship in JPA/Hibernate/ORM'.

# Many To Many

**Use-case:** Let's assume that we have to maintain a relation of many to many between User & Role table. In order to satisfy many to many relationship between the User and the Role table, multiple Users will have multiple roles.

## Many To Many: Unidirectional

We can obtain a unidirectional relationship between User & Role entity by applying @ManyToMany on the relational field at any side. For example, if we want to have a many to many relationship from User to Role, we need to add a field with type List<Role> in the User entity. It means many users will have many roles. Hence, we need to apply @ManyToMany on the field with a type List<Role> in the User entity. For example, below code demonstrates the concept.

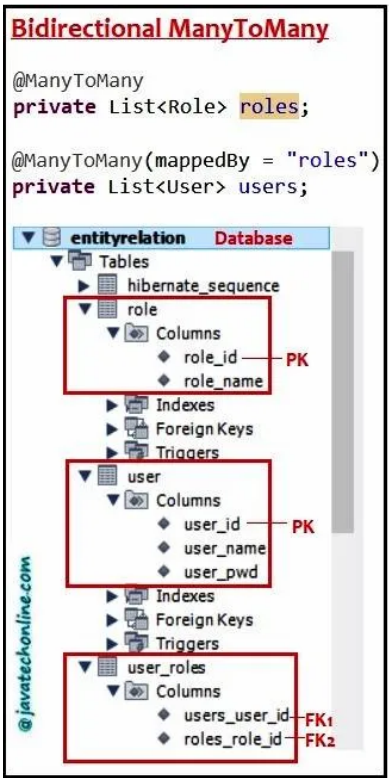



```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
```

```
@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;

    private String userName;
    private String userPwd;

     @ManyToMany
    private List<Role> roles;
}
```

## Many To Many: Bidirectional

In order to satisfy the bidirectional relationship, we need to apply @ManyToMany at both the sides ie. on the field with type List<Role> in User entity and @ManyToMany at other side also ie. on the field with type List<User> in the Role entity. Additionally, we need to have mappedBy attribute in any side to tell JPA/Hibernate that the mapping is already done by other side and don't create additional column. For Example, below code demonstrates how we will create this relation between two tables using annotation @ManyToMany.




```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;

    private String userName;
```

```
    private String userPwd;

    @ManyToMany
    private List<Role> roles;
}
```

And,

```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Role {

    @Id
    @GeneratedValue
    private Integer roleId;

    private String roleName;

    @ManyToMany(mappedBy = "roles")
    private List<User> users;
}
```

Here, we completed the fourth and last relationship of our article 'Entity Relationship in JPA/Hibernate/ORM'.

# What is fetch type in entity relationship?

Fetch Type represents the strategy for fetching data from the database. We have two types of strategies; LAZY & EAGER.

**FetchType.*LAZY*** indicates that the data should be fetched lazily when it called for the first time. Persistence Provider (JPA/Hibernate etc.) retrieves parent entity data first then retrieves child entity data on demand only.

**FetchType.*EAGER*** indicates that the data should be fetched eagerly when it called for the first time. Persistence Provider (JPA/Hibernate etc.) retrieves parent & child entity data together at a time.

## What is Lazy & Eager Loading?

Let's assume that we have two entities and there's a relationship between them. For example, we might have an entity as Teacher and another entity as Student and a Teacher might have many students. The Teacher entity might have some basic properties such as id, name, address, etc. as well as a collection property as students that returns the list of students for a given teacher. Now when we load a Teacher from the database, JPA loads its id, name, and address fields for us. But we have two options to load students.

1. To load it together with the rest of the fields (i.e. eagerly), or
2. To load it on-demand (i.e. lazily) when you call the teacher's getStudents( ) method.

When a teacher has many students it is not efficient to load all of its students together with it, especially when they are not needed and in such cases we can declare that we want students to be loaded when they are actually needed. This is called lazy loading.

# What is Cascade Operation in entity relationship?

Whenever we manipulate (insert, delete, update) rows in the parent table, the respective rows of the child table with a matching key column will also be manipulated. In database terminology, we call it Cascading effect. For example, a role is part of a user. If we try to delete the user, the role also should be deleted. This is called a cascade delete relationship.

The javax.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations.

| Cascade Operation | Description |
|---|---|

| Cascade Operation | Description |
|---|---|
| **CascadeType.*ALL*** | cascade.*ALL* is equivalent to `cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}`<br><br>It means, we are applying all cascade operations to the related entity of the parent entity. For example:<br><br>**@OneToMany(cascade = CascadeType.**<br><br>***ALL*)** |
| `CascadeType.DETACH` | If we detach the parent entity from the persistence context, the related entity will also be detached. |
| `CascadeType.MERGE` | If we merge parent entity into the persistence context, the related entity will also gets merged. |
| `CascadeType.PERSIST` | If we persist the parent entity into the persistence context, the related entity will also be persisted. |
| `CascadeType.REFRESH` | If we refresh the parent entity in the current persistence context, the related entity will also be refreshed. |
| `CascadeType.REMOVE` | If we delete the parent entity from the current persistence context, the related entity will also be deleted. |

## What is @JoinColumn and When to use it?

The @JoinColumn annotation offers us to specify the column that we will use for joining an entity association or element collection. Moreover, we can use this annotation in any entity, either Parent or Child.

## Is @JoinColumn mandatory to use?

It is not mandatory to provide @JoinColumn while defining relationship. If we don't provide it in our code, then the persistence provider (like Hibernate) will automatically generate one for you i.e. default name for your column. By default, it uses the names of entity and variable and includes underscore between two words. However, if we want to assign column names of our own choice, we can do it using @JoinColumn  annotation.

## What is @JoinTable?

Typically, when a mapping creates a separate table to accommodate two foreign keys, we call it Join Table. In this table one column is called Join column and another one is called Inverse Join Column. A Join table is generally created when we use either @ManyToMany or @OneToMany unidirectional. We can specify this table using annotation @JoinTable. If don't specify this annotation, the persistence provider will create it using the default name.

## What is Inverse Join Column?

As aforementioned, the foreign key columns of the join table which reference the primary table of the entity that does not own the relation. That is the inverse side of the relation.

Below is an example of using @JoinTable, @JoinColumn & InverseJoinColumn.

```
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

@Entity
public class User {

    @Id
    @GeneratedValue
    private Integer userId;

    private String userName;
    private String userPwd;

    @ManyToMany
    @JoinTable(name ="user_role",
      joinColumns = @JoinColumn(name = "user_fk"),
```

```
        inverseJoinColumns = @JoinColumn(name = "role_fk")
    )
    private List<Role> roles;
}
```

## Conclusion

In this article we have covered all the theoretical and example part of 'Entity Relationship in JPA/Hibernate/ORM', finally, you should be able to implement all kinds of entity relationships.. Similarly, we expect from you to further extend these examples, as per your requirement. Additionally, if you want to learn the usage of Spring Data JPA in real projects, kindly visit our Spring Boot Tutorials. Also, try to implement them in your project accordingly. Moreover, Feel free to provide your comments in the comments section below.

MCQ on Spring and Hibernate
March 6, 2022
In "Hibernate"

Spring Boot Batch Example CSV to MySQL Using JPA
June 30, 2022
In "java"

How to Become a Good Java Developer?
May 7, 2020
In "java"

Tagged @ManyToMany @ManyToOne @OneToMany @OneToOne Entity Relationship in JPA/Hibernate/ORM Entity Relationships hibernate orm hibernate relationships hibernate relationships with examples Is @JoinColumn mandatory to use? jpa with hibernate Many To Many many to many relationship in hibernate Many To Many: Bidirectional Many To Many: Unidirectional Many To One Many To One: Bidirectional Many To One: Unidirectional manytoone hibernate mappedby in hibernate One To Many one to many hibernate one to many in hibernate One To Many: Bidirectional One To Many: Unidirectional One To One One To One: Bidirectional One To One: Unidirectional onetomany hibernate orm in hibernate orm in java What are the must follow rules for bidirectional relationship? What are the popular ORM tools/frameworks in Java? What is @JoinColumn and When to use it? What is @JoinTable? What is Cascade Operation in entity relationship? What is fetch type in entity relationship? What is Inverse Join Column? What is Lazy & Eager Loading? What is ORM in Java? What is owning side & inverse/referencing side in an ORM? What is unidirectional and bidirectional relationship? Why do we use mappedBy attribute?

‹ **Previous article**
How to export data into Excel in a Spring Boot MVC Application?

**Next article** ›
How to export data into PDF in a Spring Boot MVC Application?

### Leave a Reply

Name *

Email Address *

Website

Comment *

Yes, add me to your mailing list

Post Comment

### FOLLOW US

### RECENTLY PUBLISHED POSTS

- **Spring Transaction Annotations With Examples**
- **Spring Scheduling Annotations With Examples**
- **Spring Security Annotations With Examples**
- **Spring Boot Interview Questions & Answers**

- **Java Interface**
- **Spring Security Without WebSecurityConfigurerAdapter**
- **Spring Boot Batch Example CSV to MySQL Using JPA**
- **Spring Batch Tutorial**
- **Java 14 Features**
- **Java Features After Java 8**

**DO YOU HAVE A QUERY ?**

Submit your Query

Email *

**Subscribe to receive Updates !**