# Many-To-Many Relationship in JPA

Last modified: May 12, 2022

by Attila Fejér (https://www.baeldung.com/author/attila-fejer/)

**Persistence (https://www.baeldung.com/category/persistence/)**

**JPA (https://www.baeldung.com/tag/jpa/)**

## 1. Overview

In this tutorial, we'll see multiple ways to **deal with many-to-many relationships using JPA.**

We'll use a model of students, courses, and various relationships between them.

For the sake of simplicity, in the code examples, we'll only show the attributes and JPA configuration that's related to the many-to-many relationships.

### Further reading:

#### Mapping Entity Class Names to SQL Table Names with JPA (/jpa-entity-table-names)

Learn how table names are generated by default and how to override that behavior.

**Read more (/jpa-entity-table-names)** →

### Overview of JPA/Hibernate Cascade Types (/jpa-cascade-types)

A quick and practical overview of JPA/Hibernate Cascade Types.

**Read more (/jpa-cascade-types)** →

## 2. Basic Many-to-Many

### 2.1. Modeling a Many-to-Many Relationship

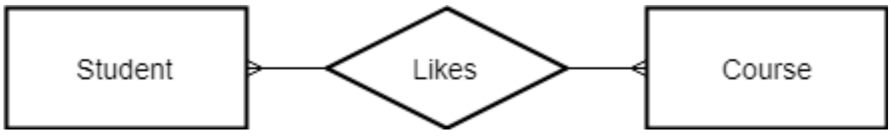
**A relationship is a connection between two types of entities. In the case of a many-to-many relationship, both sides can relate to multiple instances of the other side.**

Note that it's possible for entity types to be in a relationship with themselves. Think about the example of modeling family trees: Every node is a person, so if we talk about the parent-child relationship, both participants will be a person.

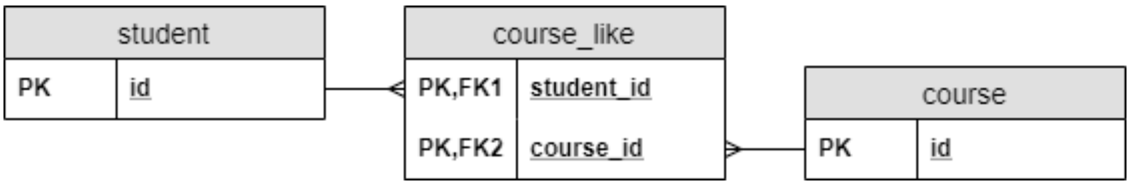
However, it doesn't make such a difference whether we talk about a relationship between single or multiple entity types. Since it's easier to think about relationships between two different entity types, we'll use that to illustrate our cases.

Let's take the example of students marking the courses they like.

A student can like **many** courses, and **many** students can like the same course:



(/wp-content /uploads/2018/11/simple-er.png)
As we know, in RDBMSs we can create relationships with foreign keys. Since both sides should be able to reference the other, **we need to create a separate table to hold the foreign keys**:



(/wp-content/uploads/2018/11/simple-model-updated.png)
Such a table is called a **join table.** In a join table, the combination of the foreign keys will be its composite primary key.

## 2.2. Implementation in JPA

**Modeling a many-to-many relationship with POJOs** is easy. We should **include a *Collection* in both classes**, which contains the elements of the others.

After that, we need to mark the class with *@Entity* and the primary key with *@Id* to make them proper JPA entities.

Also, we should configure the relationship type. So, **we mark the collections with *@ManyToMany*** annotations:

```
@Entity
class Student {

    @Id
    Long id;

    @ManyToMany
    Set<Course> likedCourses;

    // additional properties
    // standard constructors, getters, and setters
}

@Entity
class Course {

    @Id
    Long id;

    @ManyToMany
    Set<Student> likes;

    // additional properties
    // standard constructors, getters, and setters
}
```

Additionally, we have to configure how to model the relationship in the RDBMS.

The owner side is where we configure the relationship. We'll use the *Student* class.

**We can do this with the *@JoinTable* annotation in the *Student* class.** We provide the name of the join table (*course_like*) as well as the foreign keys with the *@JoinColumn* annotations. The *joinColumn* attribute will connect to the owner side of the relationship, and the *inverseJoinColumn* to the other side:

```
@ManyToMany
@JoinTable(
  name = "course_like",
  joinColumns = @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id"))
Set<Course> likedCourses;
```

Note that using *@JoinTable* or even *@JoinColumn* isn't required. JPA will generate the table and column names for us. However, the strategy JPA uses won't always match the naming conventions we use. So, we need the possibility to configure table and column names.

**On the target side, we only have to provide the name of the field, which maps the relationship.**

Therefore, we set the *mappedBy* attribute of the *@ManyToMany* annotation in the *Course* class:

```
@ManyToMany(mappedBy = "likedCourses")
Set<Student> likes;
```

Keep in mind that since **a many-to-many relationship doesn't have an owner side in the database**, we could configure the join table in the *Course* class and reference it from the *Student* class.

## 3. Many-to-Many Using a Composite Key

### 3.1. Modeling Relationship Attributes

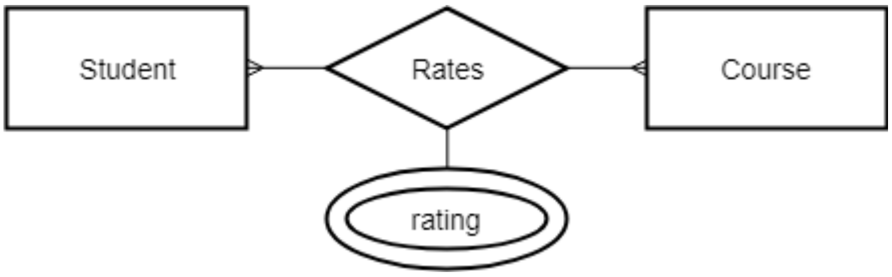
Let's say we want to let students rate the courses. A student can rate any number of courses, and any number of students can rate the same course. Therefore, it's also a many-to-many relationship.

What makes this example a bit more complicated is that **there is more to the rating relationship than the fact that it exists. We need to store the rating score the student gave on the course.**

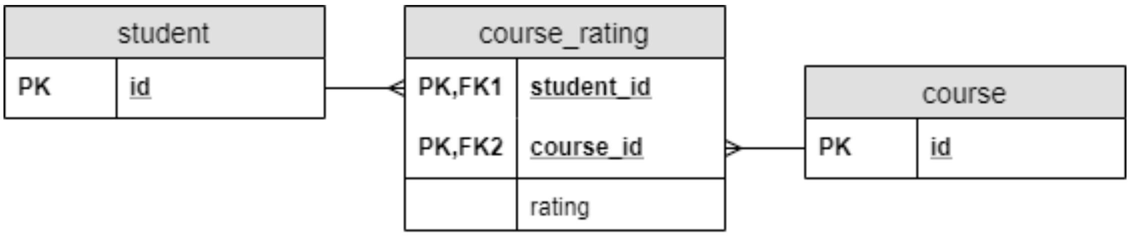
Where can we store this information? We can't put it in the *Student* entity since a student can give different ratings to different courses. Similarly, storing it in the *Course* entity wouldn't be a good solution either.

This is a situation when **the relationship itself has an attribute.**

Using this example, attaching an attribute to a relation looks like this in an ER diagram:



(/wp-content
/uploads/2018/11/relation-attribute-er.png)
**We can model it almost the same way as the simple many-to-many relationship. The only difference is that we attach a new attribute to the join table:**



(/wp-
content/uploads/2018/11/relation-attribute-model-updated.png)

## 3.2. Creating a Composite Key in JPA

The implementation of a simple many-to-many relationship was rather straightforward. The only problem is that we cannot add a property to a relationship that way because we connected the entities directly. Therefore, **we had no way to add a property to the relationship itself.**

Since we map DB attributes to class fields in JPA, **we need to create a new entity class for the relationship.**

Of course, every JPA entity needs a primary key. **Because our primary key is a composite key, we have to create a new class that will hold the different parts of the key**:

```
@Embeddable
class CourseRatingKey implements Serializable {

    @Column(name = "student_id")
    Long studentId;

    @Column(name = "course_id")
    Long courseId;

    // standard constructors, getters, and setters
    // hashcode and equals implementation
}
```

Note that a composite key class has to fulfill some **key requirements**:

- We have to mark it with *@Embeddable*.
- It has to implement *java.io.Serializable*.
- We need to provide an implementation of the *hashcode()* and *equals()* methods.
- None of the fields can be an entity themselves.

## 3.3. Using a Composite Key in JPA

Using this composite key class, we can create the entity class, which models the join table:

```
@Entity
class CourseRating {

    @EmbeddedId
    CourseRatingKey id;

    @ManyToOne
    @MapsId("studentId")
    @JoinColumn(name = "student_id")
    Student student;

    @ManyToOne
    @MapsId("courseId")
    @JoinColumn(name = "course_id")
    Course course;

    int rating;

    // standard constructors, getters, and setters
}
```

This code is very similar to a regular entity implementation. However, we have some key differences:

- We used ***@EmbeddedId* to mark the primary key**, which is an instance of the *CourseRatingKey* class.
- **We marked the *student* and *course* fields with *@MapsId*.**

*@MapsId* means that we tie those fields to a part of the key, and they're the foreign keys of a many-to-one relationship. We need it because, as we mentioned, we can't have entities in the composite key.

After this, we can configure the inverse references in the *Student* and *Course* entities as before:

```
class Student {

    // ...

    @OneToMany(mappedBy = "student")
    Set<CourseRating> ratings;

    // ...
}

class Course {

    // ...

    @OneToMany(mappedBy = "course")
    Set<CourseRating> ratings;

    // ...
}
```

Note that there's an alternative way to use composite keys: the *@IdClass* (/hibernate-identifiers) annotation.

## 3.4. Further Characteristics

**We configured the relationships to the *Student* and *Course* classes as *@ManyToOne*. We could do this because with the new entity we structurally decomposed the many-to-many relationship to two many-to-one relationships.**

Why were we able to do this? If we inspect the tables closely in the previous case, we can see that it contained two many-to-one relationships. **In other words, there isn't any many-to-many relationship in an RDBMS. We call the structures we create with join tables many-to-many relationships because that's what we model.**

Besides, it's more clear if we talk about many-to-many relationships because that's our intention. Meanwhile, a join table is just an implementation detail; we don't really care about it.

Moreover, this solution has an additional feature we haven't mentioned yet. The simple many-to-many solution creates a relationship between two entities. Therefore, we cannot expand the relationship to more entities. But we don't have this limit in this solution: **we can model relationships between any number of entity types.**

For example, when multiple teachers can teach a course, students can rate how a specific teacher teaches a specific course. **That way, a rating would be a relationship between three entities: a student, a course and a teacher.**

## 4. Many-to-Many With a New Entity

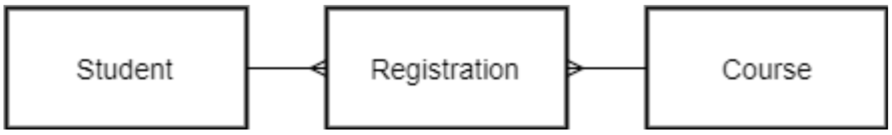
### 4.1. Modeling Relationship Attributes

Let's say we want to let students register for courses. Also, **we need to store the point when a student registered for a specific course.** On top of that, we want to store what grade she received in the course.

In an ideal world, we could solve this with the previous solution, where we had an entity with a composite key. However, the world is far from ideal, and students don't always accomplish a course on the first try.

In this case, there are **multiple connections between the same student-course pairs**, or multiple rows, with the same *student_id-course_id* pairs. We can't model it using any of the previous solutions because all primary keys must be unique. So, we need to use a separate primary key.

Therefore, **we can introduce an entity**, which will hold the attributes of the registration:



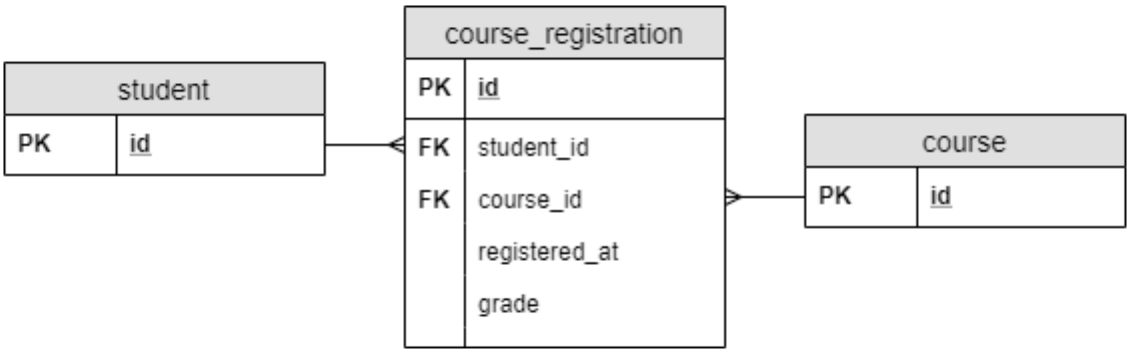
(/wp-content

/uploads/2018/11/relation-entity-er-updated.png)
In this case, **the Registration entity represents the relationship** between the other two entities.

Since it's an entity, it'll have its own primary key.

In the previous solution, remember that we had a composite primary key we created from the two foreign keys.

Now the two foreign keys won't be part of the primary key:



(/wp-content/uploads/2018/11/relation-entity-model-updated.png)

## 4.2. Implementation in JPA

Since the *course_registration* became a regular table, we can create a plain old JPA entity modeling it:

```
@Entity
class CourseRegistration {

    @Id
    Long id;

    @ManyToOne
    @JoinColumn(name = "student_id")
    Student student;

    @ManyToOne
    @JoinColumn(name = "course_id")
    Course course;

    LocalDateTime registeredAt;

    int grade;

    // additional properties
    // standard constructors, getters, and setters
}
```

We also need to configure the relationships in the *Student* and *Course* classes:

```
class Student {

    // ...

    @OneToMany(mappedBy = "student")
    Set<CourseRegistration> registrations;

    // ...
}

class Course {

    // ...

    @OneToMany(mappedBy = "course")
    Set<CourseRegistration> registrations;

    // ...
}
```

Again, we configured the relationship earlier, so we only need to tell JPA where can it find that configuration.

We could also use this solution to address the previous problem of students rating courses. However, it feels weird to create a dedicated primary key unless we have to.

Moreover, from an RDBMS perspective, it doesn't make much sense since combining the two foreign keys made a perfect composite key. Besides, that **composite key had a clear meaning: which entities we connect in the relationship.**

Otherwise, the choice between these two implementations is often simply personal preference.

## 5. Conclusion

In this article, we saw what a many-to-many relationship is and how can we model it in an RDBMS using JPA.

We saw three ways to model it in JPA. All three have different advantages and disadvantages when it comes to these aspects:

- code clarity
- DB clarity

- ability to assign attributes to the relationship
- how many entities we can link with the relationship
- support for multiple connections between the same entities

As usual, the examples are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-jpa-2).

Comments are closed on this article!