**1. Introduction**

JPA Relationships can be either unidirectional or bidirectional. It simply means we can model them as an attribute on exactly one of the associated entities or both.

Defining the direction of the relationship between entities has no impact on the database mapping. It only defines the directions in which we use that relationship in our domain model.

For a bidirectional relationship, we usually define:

* the owning side
* inverse or the referencing side

The [*@JoinColumn*](https://www.baeldung.com/jpa-join-column) annotation helps us specify the column we’ll use for joining an entity association or element collection. On the other hand, the *mappedBy* attribute is used to define the referencing side (non-owning side) of the relationship.

In this quick tutorial, we’ll look at **the difference between *@JoinColumn* and *mappedBy*** **in JPA. We’ll also present how to use them in a one-to-many association.**

**2. Initial Setup**

To follow along with this tutorial, let’s say we have two entities: *Employee* and *Email.*

Clearly, an employee can have multiple email addresses. However, a given email address can belong exactly to a single employee.

It means they share a one-to-many association:

[](https://www.baeldung.com/wp-content/uploads/2018/11/12345789.png)

Also in our RDBMS model, we’ll have a foreign key *employee\_id* in our *Email* entity referring to the *id*attribute of an *Employee*.

**3. *@JoinColumn* Annotation**

In a One-to-Many/Many-to-One relationship, **the owning side is usually defined on the ‘*many’* side of the relationship.** It’s usually the side which owns the foreign key.

The *@JoinColumn* annotation defines that actual physical mapping on the owning side:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Entity  public class Email {        @Id      @GeneratedValue(strategy = GenerationType.AUTO)      private Long id;        @ManyToOne(fetch = FetchType.LAZY)      @JoinColumn(name = "employee\_id")      private Employee employee;        // ...    } |

It simply means that our *Email* entity will have a foreign key column named *employee\_id* referring to the primary attribute *id* of our *Employee*entity.

**4. *mappedBy* Attribute**

Once we have defined the owning side of the relationship, Hibernate already has all the information it needs to map that relationship in our database. To make this association bidirectional, all we’ll have to do is to define the referencing side. The inverse or the referencing side simply maps to the owning side.

We can easily use the *mappedBy* attribute of *@OneToMany* annotation to do so. So, let’s define our *Employee* entity:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Entity  public class Employee {        @Id      @GeneratedValue(strategy = GenerationType.AUTO)      private Long id;        @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")      private List<Email> emails;        // ...  } |

Here,**the value of *mappedBy* is the name of the association-mapping attribute on the owning side.**With this, we have now established a bidirectional association between our *Employee* and *Email* entities.

**mappedBy :**

By specifying the @JoinColumn on both models you don't have a two way relationship. You have two one way relationships, and a very confusing mapping of it at that. You're telling both models that they "own" the IDAIRLINE column. Really only one of them actually should! The 'normal' thing is to take the @JoinColumn off of the @OneToMany side entirely, and instead add mappedBy to the @OneToMany.

@OneToMany(cascade = CascadeType.ALL, mappedBy="airline")

public Set<AirlineFlight> getAirlineFlights() {

return airlineFlights;

}

That tells Hibernate "Go look over on the bean property named 'airline' on the thing I have a collection of to find the configuration."