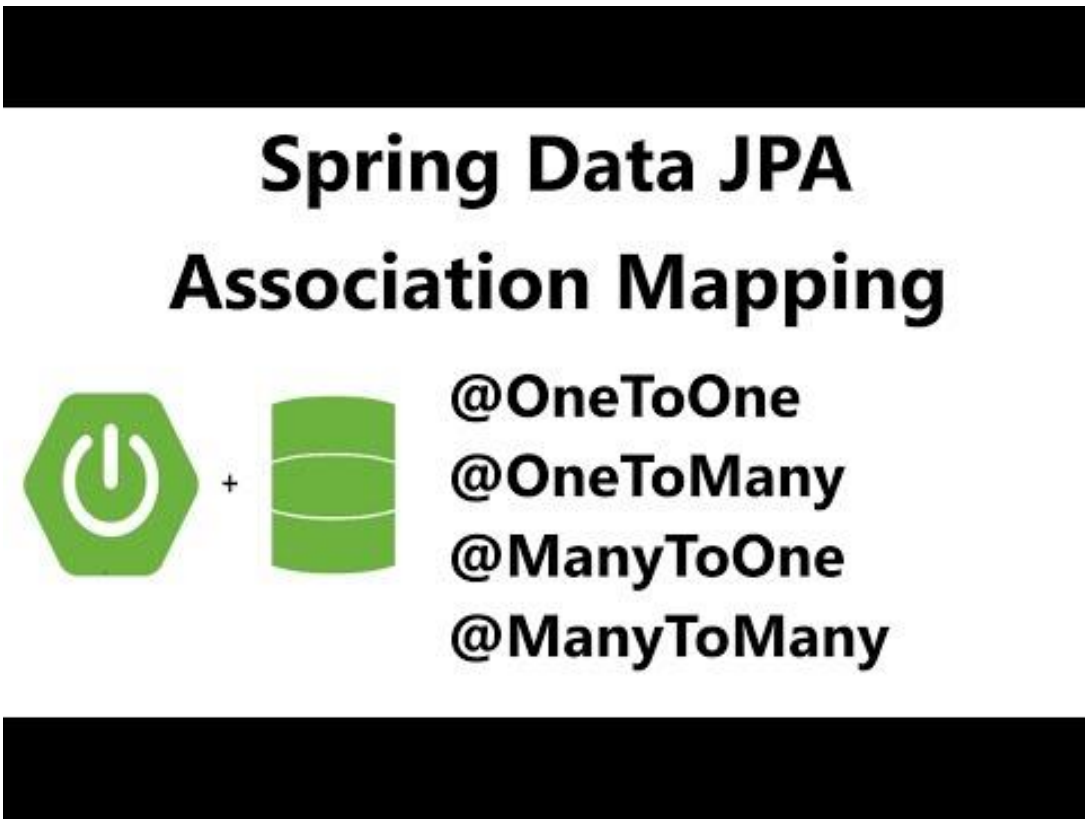


Key Points to understand Associations in JPA and Hibernate

1. **Entity Role.**
2. **Cascade.**
3. **Fetch Type.**
4. **Direction.**
5. **mappedBy Attribute.**
6. **Join Column.**
7. **Join Tables.**
8. **Inverse Join Column.**

1. Entity Role :

- In every association there are two entities that are related to one another, each entity play a role which is either Owning Entity or Non-Owning Entity.
- Assume that there are two tables USER and USER_CREDENTIALS and tables associated by USER_CREDENTAILS having foreign key referenced by USER table. Then mapping is like following example.

```java
@Entity(name="USER_CREDENTIALS")
public class Credentials {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "CREDS_ID")
    private Long credentialId;

    @Column(name = "USERNAME")
    private String userName;

    @Column(name = "PASSWORD")
    private String password;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="USER_ID")
    private User user;//Non-Owning Entity

        //Setters and Getters
```

The owning side of the entity was determined by referencing both entities in the data model and identifying the entity containing the **foreign key**.

So the role of **Credentials entity is Owing Entity** and other entity **User role is Non-Owning** Entity or inverse side of entity

2. Cascade :

- Whenever rows in the parent table manipulated (inserted, updated, deleted) the respective rows of the child table with a matching key column will be manipulated as well. This is called Cascade in Database.
- JPA translates entity state transitions to database DML statements.

- JPA allows cascadable operations (SELECT, INSERT, UPDATE, DELETE) to propagate entity state changes from owning to non-owning entities.
- JPA cascade types are PERSIST, MERGE, REFRESH, REMOVE, DETACH, ALL.

1. **CascadeType.PERSIST** : We have to persist the *owning* entity, and the associated *non-owning* entity is persisted as well.
2. **CascadeType.MERGE** : We have to merge the *owning* entity, and the associated *non-owning* entity is merged as well.
3. **CascadeType.REFRESH** : We have to refresh the *owning* entity, and the associated *non-owning* entity is refreshed implicitly.
4. **CascadeType.REMOVE** : *Removes all related associated non-owning entities if owning* entity removed.
5. ***CascadeType.DETACH*** *: detaches all related non–owning entities if a owning* entity detached.
6. ***CascadeType.ALL : cascade = ALL means { PERSIST, MERGE, REMOVE, REFRESH, DETACH }***

Usage :

```
1.  @Entity(name="USER_CREDENTIALS")
2.  public class Credentials {
3.
4.      @Id
5.      @GeneratedValue(strategy = GenerationType.AUTO)
6.      @Column(name = "CREDS_ID")
7.      private Long credentialId;
8.
9.      @Column(name = "USERNAME")
10.     private String userName;
11.
12.     @Column(name = "PASSWORD")
13.     private String password;
14.
15.     @OneToOne(cascade=CascadeType.ALL)//CascadeType
16.     @JoinColumn(name="USER_ID")
17.     private User user;
18.
19.         //Setters and Getters
```

In Credentials entity used CascadeType.ALL, it means if EntityManager manipulate (persist, merge, refresh, remove , detach) on Credentials entity will also be affected same cascade operation on User entity.

3. Fetch Type :

FetchType defines strategies for fetching data from the database. There are 2 strategies **EAGER and LAZY**.

FetchType.EAGER : EAGER strategy is a requirement on the persistence provider runtime that data must be eagerly fetched (fetch in one query) .

If EAGER strategy used EntityManager fetch results in one query (parent and childs).

FetchType.LAZY : The LAZY strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed(fetch when needed as sub-queries). EntityManager retrieves parent entity data first then retrieves child entity data on demand.

Lazy Fetching :

```
1.  @Entity(name="USER_CREDENTIALS")
2.  public class Credentials {
3.
4.      @Id
5.      @GeneratedValue(strategy = GenerationType.AUTO)
6.      @Column(name = "CREDS_ID")
7.      private Long credentialId;
8.
9.      @Column(name = "USERNAME")
10.     private String userName;
11.
12.     @Column(name = "PASSWORD")
13.     private String password;
14.
15.     @OneToOne(cascade=CascadeType.ALL, fetch=FetchType.LAZY)//Lazy Loading
16.     @JoinColumn(name="USER_ID")
17.     private User user;
18. //Setters and Getters
```

From the above association Fetch Type is LAZY. To test Lazy loading – retrieve Credentials results using below code. then see console.

Credentials credential= entityManager.find(Credentials.class, new Long(1));
In console you will see similar to below query.

```
select
   credential0_.CREDS_ID as CREDS_ID1_1_0_,
   credential0_.PASSWORD as PASSWORD2_1_0_,
   credential0_.USER_ID as USER_ID4_1_0_,
   credential0_.USERNAME as USERNAME3_1_0_
   from USER_CREDENTIALS credential0_ where credential0_.CREDS_ID=1;
```

It Means if FetchType is Lazy, EntityManager loads only Credential entity data on retrieval, later whenever we call **credential.getUser()** then it loads User entity data. This is called Lazy Loading or retrieve data on demand.

Eager Fetching :

If we use Fetch strategy as **EAGER** then you will see similar query like below one.

```
select
   credential0_.CREDS_ID as CREDS_ID1_1_1_,
   credential0_.PASSWORD as PASSWORD2_1_1_,
   credential0_.USER_ID as USER_ID4_1_1_,
   credential0_.USERNAME as USERNAME3_1_1_,
   user1_.ID as ID1_0_0_,
   user1_.CREATED_TIME as CREATED_2_0_0_,
   user1_.DOB as DOB3_0_0_,
   user1_.FIRST_NAME as FIRST_NA4_0_0_,
   user1_.LAST_NAME as LAST_NAM5_0_0_,
   user1_.UPDATED_TIME as UPDATED_6_0_0_,
   user1_.USER_TYPE as USER_TYP7_0_0_
   from USER_CREDENTIALS credential0_ left outer join USER user1_ on credential0_.USER_ID=use
```

It means if **FetchType is EAGER,** EntityManager loads all data in single query. This is called Eager fetching.

4. Direction :

- Relationships can be **unidirectional** or **bidirectional**. **Unidirectional** is a relation where one side does not know about the relation. In a **Bidirectional** relation both sides know about the other side.
- **Bidirectiona**l relationship provides navigational access in both directions, so that you can access the other side without explicit queries.

5. mappedBy :

1. The **mappedBy** element defines a ***bidirectional relationship***. This attribute allows you to refer the associated entities from both sides.

```
1.      //Other fields mapping
2.
3.      @OneToOne(mappedBy="user")//--> user is User entity reference defined in Crede
4.      private Credentials credentials;
5.
6.      //Setter and Getters
```

6. Join Column:

- **@JoinColumn** Specifies a column for joining an entity association or element collection. The annotation **@JoinColumn** indicates that this entity is the *owner* of the relationship.
- That is the corresponding table has a column with a foreign key to the referenced table.

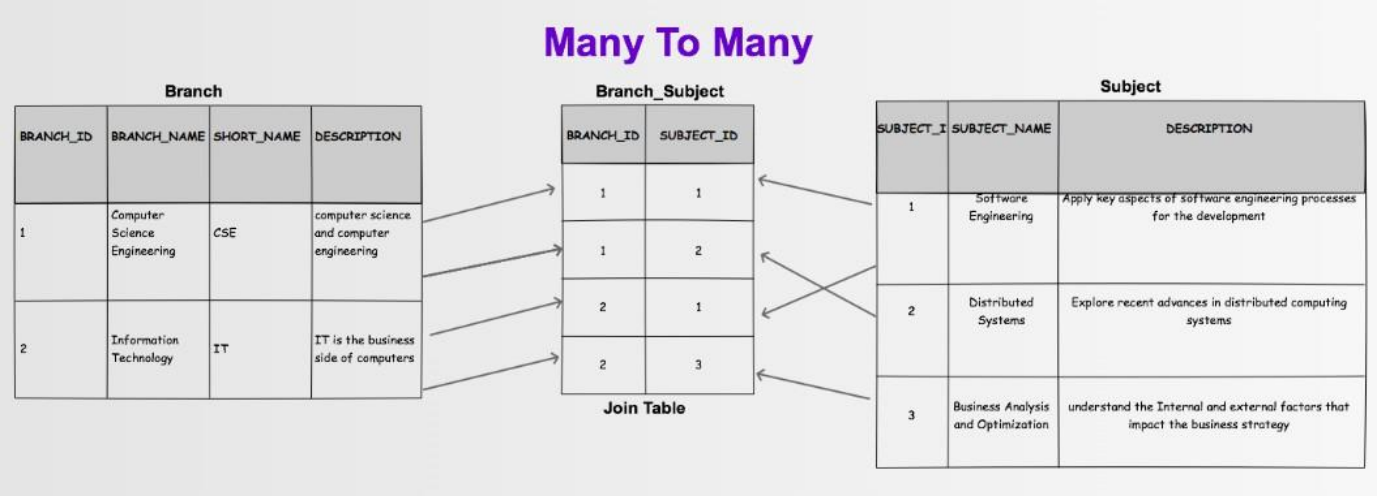
- In the below example, the owner Entity Credentials, has a Join Column named USER_ID that has a foreign key to the non-owner User entity.
- The owning entity is always has the joincolumn mapping.

```
1.  //Other fields mapping
2.
3.
4.      @OneToOne(cascade=CascadeType.ALL, fetch=FetchType.LAZY)
5.      @JoinColumn(name="USER_ID")
6.      private User user;
7.
8.      //Setter and Getters
```

7. Join Table:

- **@JoinTable** Used in the mapping of associations. It is specified on the owning side of an association.
- **@JoinTable** can be used with embeddable types as well.
- When a join table is used in mapping a relationship with an embeddable class on the owning side of the relationship, the containing entity rather than the embeddable class is considered the owner of the relationship.
- To understand little more about join table see below image. This image describes association between Engineering Branch and Subjects having each branch for all semesters. There is many to many relation between BRANCH and SUBJECT tables.
- If we take one side relation Each Branch has multiple subjects – in the mapping, Branch entity having collection of Subjects, In other side inverse side also same. BRANCH_SUBJECT table having the foreign keys of BRANCH and SUBJECT tables.

- BRANCH_SUBJECT joins the multiple references of BRANCH and SUBJECT table, this is called join table.



8. Inverse Join Columns:

- The foreign key columns of the join table which reference the primary table of the entity that does not own the association. That is the inverse side of the association
  - From the above many-to-many relation if take one scenario that each branch has many subjects, in mapping Branch entity becomes owning entity and Subject entity becomes non-owning entity. So we have to map BRANCH_ID with @JoinColumn and the other side column will become inverse join column to associate inverse side non-owning entity.
  - For usage see below mapping.

```java
 @Entity
public class Branch implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="BRANCH_ID")
    private int branchId;

    @Column(name="BRANCH_NAME")
    private String branchName;

    @Column(name="BRANCH_SHORT_NAME")
    private String branchShortName;

    private String description;

    //Uni-directional many-to-many association to Subject
    @ManyToMany(cascade={CascadeType.ALL})
    @JoinTable(name="BRANCH_SUBJECT", joinColumns=@JoinColumn(name="BRANCH_ID"),
    inverseJoinColumns=@JoinColumn(name="SUBJECT_ID"))
    private List subjects;

    //Setter and Getters
```