1. persistence layer is a group of files which is used to communicate between the application and DB.
2. The **persistence layer** deals with persisting (storing and retrieving) data from a data store (such as a database, for example).
3. so basically hibernate sessions are the persistence layer which used to copy the values of the data from the database to the java entity class, or mapping the java classes back to the database.

EntityManagerFactory

EntityManager

EntityTransaction

getEntityManager()

em.persist()

et.begin()

et.commit()

@Entity

@Table

@Id

@GeneratedValue

@Column

@Transient

# [**Why does JPA have a @Transient annotation?**](https://stackoverflow.com/questions/2154622/why-does-jpa-have-a-transient-annotation)

Java's transient keyword is used to denote that a field is not to be serialized, whereas JPA's @Transient annotation is used to indicate that a field is not to be persisted in the database, i.e. their semantics are different.

As others have said, @Transient is used to mark fields which shouldn't be persisted. Consider this short example:

public enum Gender { MALE, FEMALE, UNKNOWN }

@Entity

public Person {

private Gender g;

private long id;

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

public long getId() { return id; }

public void setId(long id) { this.id = id; }

public Gender getGender() { return g; }

public void setGender(Gender g) { this.g = g; }

@Transient

public boolean isMale() {

return Gender.MALE.equals(g);

}

@Transient

public boolean isFemale() {

return Gender.FEMALE.equals(g);

}

}

When this class is fed to the JPA, it persists the gender and id but doesn't try to persist the helper boolean methods - without @Transient the underlying system would complain that the Entity class Person is missing setMale() and setFemale() methods and thus wouldn't persist Person at all.

**Hibernate JPA Annotations - Contents:**

|  |  |
| --- | --- |
| **Annotation** | **Package Detail/Import statement** |
| [@Entity](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Entity) | import javax.persistence.Entity; |
| [@Table](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Table) | import javax.persistence.Table; |
| [@Column](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Column) | import javax.persistence.Column; |
| [@Id](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Id) | import javax.persistence.Id; |
| [@GeneratedValue](http://www.techferry.com/articles/hibernate-jpa-annotations.html#GeneratedValue) | import javax.persistence.GeneratedValue; |
| [@Version](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Version) | import javax.persistence.Version; |
| [@OrderBy](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OrderBy) | import javax.persistence.OrderBy; |
| [@Transient](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Transient) | import javax.persistence.Transient; |
| [@Lob](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Lob) | import javax.persistence.Lob; |
| [Hibernate Association Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateAssociations) | |
| [@OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToOne) | import javax.persistence.OneToOne; |
| [@ManyToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#ManyToOne) | import javax.persistence.ManyToOne; |
| [@OneToMany](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToMany) | import javax.persistence.OneToMany; |
| [@ManyToMany](http://www.techferry.com/articles/hibernate-jpa-annotations.html#ManyToMany) | import javax.persistence.ManyToMany; |
| [@PrimaryKeyJoinColumn](http://www.techferry.com/articles/hibernate-jpa-annotations.html#PrimaryKeyJoinColumn) | import javax.persistence.PrimaryKeyJoinColumn; |
| [@JoinColumn](http://www.techferry.com/articles/hibernate-jpa-annotations.html#JoinColumn) | import javax.persistence.JoinColumn; |
| [@JoinTable](http://www.techferry.com/articles/hibernate-jpa-annotations.html#JoinTable) | import javax.persistence.JoinTable; |
| [@MapsId](http://www.techferry.com/articles/hibernate-jpa-annotations.html#MapsId) | import javax.persistence.MapsId; |
| [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) | |
| [@Inheritance](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Inheritance) | import javax.persistence.Inheritance; |
| [@DiscriminatorColumn](http://www.techferry.com/articles/hibernate-jpa-annotations.html#DiscriminatorColumn) | import javax.persistence.DiscriminatorColumn; |
| [@DiscriminatorValue](http://www.techferry.com/articles/hibernate-jpa-annotations.html#DiscriminatorValue) | import javax.persistence.DiscriminatorValue; |

**@Entity**

Annotate all your entity beans with @Entity. 

|  |  |
| --- | --- |
| 1  2  3  4 | @Entity  public class Company implements Serializable {  ...  } |

**@Table**

Specify the database table this Entity maps to using the name attribute of @Table annotation. In the example below, the data will be stored in 'company' table in the database. 

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Entity  @Table(name = "company")  public class Company implements Serializable {  ...  } |

**@Column**

Specify the column mapping using @Column annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @Column(name = "name")    private String name;    ...  } |

**@Id**

Annotate the id column using @Id.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @Id    @Column(name = "id")    private int id;    ...  } |

**@GeneratedValue**

Let database generate (auto-increment) the id column.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @Id    @Column(name = "id")    @GeneratedValue    private int id;    ...  } |

**@Version**

Control versioning or concurrency using @Version annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @Version    @Column(name = "version")    private Date version;    ...  } |

**@OrderBy**

Sort your data using @OrderBy annotation. In example below, it will sort all contacts in a company by their firstname in ascending order.

|  |  |
| --- | --- |
| 1  2 | @OrderBy("firstName asc")  private Set contacts; |

**@Transient**

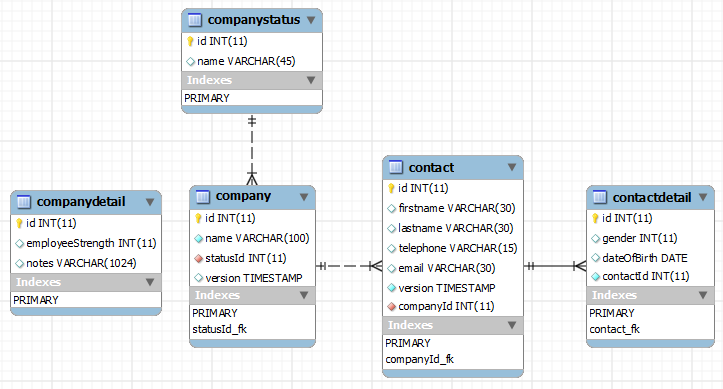
Annotate your transient properties with @Transient.

**@Lob**

Annotate large objects with @Lob.

**Hibernate Association Mapping Annotations**

**Example App DB Schema**

   
The database for this tutorial is designed to illustrate various association mapping concepts.   
In RDBMS implementations, entities are joined using the following ways:

* Shared Primary Key
* Foreign Key
* Association Table

In our example app,

* Tables company and companyDetail have shared values for primary key. It is a one-to-one assoication.
* Tables contact and contactDetail are linked through a foreign key. It is also a one to one association.
* Tables contact and company are linked through a foriegn key in many-to-one association with contact being the owner.
* Tables company and companyStatus are linked through a foreign key in many-to-one association with company being the owner.

**@OneToOne**

|  |  |
| --- | --- |
| Hibernate Annotation Tip | * Use @PrimaryKeyJoinColumn for associated entities sharing the same primary key. * Use @JoinColumn & @OneToOne mappedBy attribute when foreign key is held by one of the entities. * Use @JoinTable and mappedBy entities linked through an association table. * Persist two entities with shared key using @MapsId |

For entities Company and CompanyDetail sharing the same primary key, we can associate them using @OneToOne and @PrimaryKeyJoinColumn as shown in the example below.   
  
Notice that the id property of CompanyDetail is NOT annotated with @GeneratedValue. It will be populated by id value of Company.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @Id    @Column(name = "id")    @GeneratedValue    private int id;      @OneToOne(cascade = CascadeType.MERGE)    @PrimaryKeyJoinColumn    private CompanyDetail companyDetail;      ...  }    @Entity  @Table(name = "companyDetail")  public class CompanyDetail implements Serializable {      @Id    @Column(name = "id")    private int id;      ...  } |

For entities Contact and ContactDetail linked through a foriegn key, we can use @OneToOne and @JoinColumn annotations. In example below, the id genereated for Contact will be mapped to 'contact\_id' column of ContactDetail table. Please note the usage of @MapsId for the same.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | @Entity  @Table(name = "contactDetail")  public class ContactDetail implements Serializable {      @Id    @Column(name = "id")    @GeneratedValue    private int id;      @OneToOne    @MapsId    @JoinColumn(name = "contactId")    private Contact contact;      ...  }    @Entity  @Table(name = "contact")  public class Contact implements Serializable {      @Id    @Column(name = "ID")    @GeneratedValue    private Integer id;      @OneToOne(mappedBy = "contact", cascade = CascadeType.ALL)    private ContactDetail contactDetail;      ....  } |

Also note that the relationship between Company and CompanyDetail is uni-directional. On the other hand, the relationship between Contact and Contact Detail is bi-directional and that can be achieved using 'mappedBy' attribute.   
  
The rationale to have one relationship as uni-directional and other as bi-directional in this tutorial is to illustrate both concepts and their usage. You can opt for uni-directional or bi-directional relationships to suit your needs.

**@ManyToOne**

|  |  |
| --- | --- |
| Hibernate Annotation Tip | * Use @JoinColumn when foreign key is held by one of the entities. * Use @JoinTable for entities linked through an association table. |

The two examples below illustrate many-to-one relationships. Contact to Company and Company to CompanyStatus. Many contacts can belong to a company. Similary many companies can share the same status (Lead, Prospect, Customer) - there will be many companies that are currently leads.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | @Entity  @Table(name = "contact")  public class Contact implements Serializable {      @ManyToOne    @JoinColumn(name = "companyId")    private Company company;      ...     }    @Entity  @Table(name = "company")  public class Company implements Serializable {      @ManyToOne    @JoinColumn(name = "statusId")    private CompanyStatus status;      ...     } |

**@OneToMany**

|  |  |
| --- | --- |
| Hibernate Annotation Tip | * Use mappedBy attribute for bi-directional associations with ManyToOne being the owner. * OneToMany being the owner or unidirectional with foreign key - try to avoid such associations but can be achieved with @JoinColumn * @JoinTable for Unidirectional with association table |

Please see the many-to-one relationship between Contact and Company above. Company to Contact will be a one-to-many relationship. The owner of this relationship is Contact and hence we will use 'mappedBy' attribute in Company to make it bi-directional relationship.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @OneToMany(mappedBy = "company", fetch = FetchType.EAGER)    @OrderBy("firstName asc")    private Set contacts;      ...     } |

Again, for this tutorial, we have kept Company to CompanyStatus relationship as uni-directional.

**@ManyToMany**

|  |  |
| --- | --- |
| Hibernate Annotation Tip | * Use @JoinTable for entities linked through an association table. * Use mappedBy attribute for bi-directional association. |

**@PrimaryKeyJoinColumn**

@PrimaryKeyJoinColumn annotation is used for associated entities sharing the same primary key. See [OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html" \l "OneToOne" \o "Hibernate JPA Annotation @OneToOne) section for details.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Entity  @Table(name = "company")  public class Company implements Serializable {      @Id    @Column(name = "id")    @GeneratedValue    private int id;      @OneToOne(cascade = CascadeType.MERGE)    @PrimaryKeyJoinColumn    private CompanyDetail companyDetail;      ...  } |

**@JoinColumn**

Use @JoinColumn annotation for one-to-one or many-to-one associations when foreign key is held by one of the entities. We can use @OneToOne or @ManyToOne mappedBy attribute for bi-directional relations. Also see [OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html" \l "OneToOne" \o "Hibernate JPA Annotation @OneToOne) and [ManyToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html" \l "ManyToOne" \o "Hibernate JPA Annotation @ManyToOne) sections for more details.

|  |  |
| --- | --- |
| 1  2  3 | @ManyToOne  @JoinColumn(name = "statusId")  private CompanyStatus status; |

**@JoinTable**

Use @JoinTable and mappedBy for entities linked through an association table.

**@MapsId**

Persist two entities with shared key (when one entity holds a foreign key to the other) using @MapsId annotation. See [OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html" \l "OneToOne" \o "Hibernate JPA Annotation @OneToOne) section for details.

|  |  |
| --- | --- |
| 1  2  3  4 | @OneToOne  @MapsId  @JoinColumn(name = "contactId")  private Contact contact; |

**Hibernate Inheritance Mapping Annotations**

To understand Inheritance Mapping annotations, you must first understand [Inheritance Mapping in Hiberate](http://docs.jboss.org/hibernate/core/3.5/reference/en/html/inheritance.html) in detail. Once you understand Inheritance mapping concepts, please review below for annotations to be used.

* table per class hierarchy - single table per Class Hierarchy Strategy: the <subclass> element in Hibernate

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING )    @DiscriminatorValue("Plane")  public class Plane { ... }    @Entity  @DiscriminatorValue("A320")  public class A320 extends Plane { ... } |

* table per class/subclass - joined subclass Strategy: the <joined-subclass> element in Hibernate

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Entity  @Inheritance(strategy=InheritanceType.JOINED)  public class Boat implements Serializable { ... }    @Entity  @PrimaryKeyJoinColumn  public class Ferry extends Boat { ... } |

* table per concrete class - table per Class Strategy: the <union-class> element in Hibernate

|  |  |  |
| --- | --- | --- |
| 1  2  3 | @Entity  @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  public class Flight implements Serializable { ... } | |
| Hibernate Annotation Tip | Note: This strategy does not support the IDENTITY generator strategy: the id has to be shared across several tables. Consequently, when using this strategy, you should not use AUTO nor IDENTITY. |

**@Inheritance**

See [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) section for details.

|  |  |
| --- | --- |
| 1  2 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE) |

**@DiscriminatorColumn**

See [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) section for details.

|  |  |
| --- | --- |
| 1  2  3 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING ) |

**@DiscriminatorValue**

See [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) section for details.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING )    @DiscriminatorValue("Plane")  public class Plane { ... }    @Entity  @DiscriminatorValue("A320")  public class A320 extends Plane { ... } |

# [**What are the possible values of the #Hibernate hbm2ddl.auto configuration and what do they do**](https://stackoverflow.com/questions/438146/what-are-the-possible-values-of-the-hibernate-hbm2ddl-auto-configuration-and-wh)

hibernate.hbm2ddl.auto Automatically validates or exports schema DDL to the database when the SessionFactory is created. With create-drop, the database schema will be dropped when the SessionFactory is closed explicitly.

e.g. validate | update | create | create-drop

So the list of possible options are,

* validate: validate the schema, makes no changes to the database.
* update: update the schema.
* create: creates the schema, destroying previous data.
* create-drop: drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped.

# How does @OrderBy work?

It is not working here in the following code:

**Employee.java**

package com.semanticbits.pojo;

import java.util.List;

import javax.persistence.CascadeType;

import javax.persistence.Embedded;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.OneToMany;

import javax.persistence.OrderBy;

@Entity

public class Employee {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int employeeId;

private String name;

private double salary;

@OneToMany(cascade=CascadeType.ALL)

@JoinColumn(name="EMP\_ID")

@OrderBy("city DESC")

private List<Address> address;

//setters and getters

public int getEmployeeId() {

return employeeId;

}

public void setEmployeeId(int employeeId) {

this.employeeId = employeeId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public List<Address> getAddress() {

return address;

}

public void setAddress(List<Address> address) {

this.address = address;

}

}

**Address.java**

package com.semanticbits.pojo;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Address {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int addressId;

private String street;

private String city;

private String state;

private int zipCode;

public String getStreet() {

return street;

}

public void setStreet(String street) {

this.street = street;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getState() {

return state;

}

public void setState(String state) {

this.state = state;

}

public int getZipCode() {

return zipCode;

}

public void setZipCode(int zipCode) {

this.zipCode = zipCode;

}

}

**persistence.xml**

<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd" version="1.0">

<persistence-unit name="orderbyannotationdemo" transaction-type="RESOURCE\_LOCAL">

<provider></provider>

<class>com.semanticbits.pojo.Employee</class>

<class>com.semanticbits.pojo.Address</class>

<properties>

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/shoaib"/>

<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>

<property name="javax.persistence.jdbc.user" value="root"/>

<property name="javax.persistence.jdbc.password" value="root"/>

<property name="eclipselink.logging.level" value="FINE"/>

<property name="eclipselink.ddl-generation" value="create-tables"/>

</properties>

</persistence-unit>

</persistence>

This is the test class......check out the city name and it is not storing address values in order in descending order in the ADDRESS table

**JPAOrderByAnnotationTest**

package com.semanticbits.test;

import java.util.ArrayList;

import java.util.List;

import javax.persistence.EntityManager;

import javax.persistence.EntityManagerFactory;

import javax.persistence.Persistence;

import com.semanticbits.pojo.Address;

import com.semanticbits.pojo.Employee;

public class JPAOrderByAnnotationTest {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

EntityManagerFactory factory=Persistence.createEntityManagerFactory("orderbyannotationdemo");

EntityManager manager=factory.createEntityManager();

Employee employee=new Employee();

employee.setName("Shoaib");

employee.setSalary(1452365);

Address addressOffice=new Address();

addressOffice.setCity("Hyderabad");

addressOffice.setStreet("Gachibowli");

addressOffice.setState("AP");

addressOffice.setZipCode(500016);

Address addressHome=new Address();

addressHome.setCity("Noida");

addressHome.setStreet("Chandai Chowk");

addressHome.setState("UP");

addressHome.setZipCode(415608);

Address addressCollege=new Address();

addressCollege.setCity("Antartica");

addressCollege.setState("Canada");

addressCollege.setStreet("New York");

addressCollege.setZipCode(402103);

List<Address> addresses=new ArrayList<Address>();

addresses.add(addressHome);

addresses.add(addressOffice);

addresses.add(addressCollege);

employee.setAddress(addresses);

manager.getTransaction().begin();

manager.persist(employee);

manager.getTransaction().commit();

manager.close();

}

}

I think you're misunderstanding what the @Orderby annotation actually does. According to the [javadoc](http://docs.oracle.com/javaee/7/api/javax/persistence/OrderBy.html):

Specifies the ordering of the elements of a collection valued association or element collection at the point when the association or collection is **retrieved**.

[emphasis added] The annotation does not dictate insertion order. Continuing with your example, if you were to fetch an Employee:

Employee employee = manager.find(Employee.class, employeeId);

List<Address> addresses = employee.getAddress();

Then addresses would be sorted by city in descending order.

@MAppedBy

in a bidirectional relationship, one of the sides (and only one) has to be the owner. The owner is responsible for the association column(s) update. To declare a side as not responsible for the relationship, the attribute ***[mappedBy](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html" \l "mappedBy%28%29" \o "mappedBy)*** is used. ‘mappedBy’ refers to the property name of the association on the owner side.

# **@JoinColumn Annotation Explained**

Last modified: October 28, 2018

by [baeldung](https://www.baeldung.com/author/baeldung/" \o "Posts by baeldung)

* [**Persistence**](https://www.baeldung.com/category/persistence/)
* [**Hibernate**](https://www.baeldung.com/tag/hibernate/)
* [**JPA**](https://www.baeldung.com/tag/jpa/)

### I just announced the new Spring Boot 2 material, coming in REST With Spring:

[**>> CHECK OUT THE COURSE**](https://www.baeldung.com/rws-course-start)

## ****1. Introduction****

The annotation javax.persistence.JoinColumn marks a column for as a join column for an entity association or an element collection.

In this quick tutorial, we’ll show some examples of basic @JoinCloumn usage.

## ****2.****@OneToOne****Mapping Example****

The @JoinColumn annotation combined with a @OneToOne mapping indicates that a given column in the owner entity refers to a primary key in the reference entity:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity  public class Office {      @OneToOne(fetch = FetchType.LAZY)      @JoinColumn(name = "addressId")      private Address address;  } |

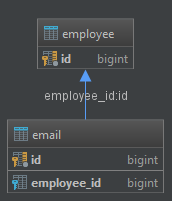
The above code example will create a foreign key linking the Office entity with the primary key from the Address entity. The name of the foreign key column in the Office entity is specified by name property.

## ****3.****@OneToMany****Mapping Example****

When using a @OneToMany mapping we can use the mappedBy parameter to indicate that the given column is owned by another entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Entity  public class Employee {        @Id      private Long id;        @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")      private List<Email> emails;  }    @Entity  public class Email {        @ManyToOne(fetch = FetchType.LAZY)      @JoinColumn(name = "employee\_id")      private Employee employee;  } |

In the above example, Email (the owner entity) has a join column employee\_id that stores the id value and has a foreign key to the Employee entity.

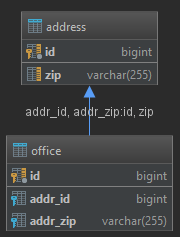


## ****4.****@JoinColumns

In situations when we want to create multiple join columns we can use the @JoinColumnsannotation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  public class Office {      @ManyToOne(fetch = FetchType.LAZY)      @JoinColumns({          @JoinColumn(name="ADDR\_ID", referencedColumnName="ID"),          @JoinColumn(name="ADDR\_ZIP", referencedColumnName="ZIP")      })      private Address address;  } |

The above example will create two foreign keys pointing to ID and ZIP columns in the Address entity:



## ****5. Conclusion****

In this article, we’ve learned how to use the @JoinColumn annotation. We’ve shown examples how to create both single entity association and element collection.

# [**JPA “@JoinTable” annotation**](https://stackoverflow.com/questions/5478328/jpa-jointable-annotation)

## 4 Answers

[active](https://stackoverflow.com/questions/5478328/jpa-jointable-annotation?answertab=active#tab-top)[oldest](https://stackoverflow.com/questions/5478328/jpa-jointable-annotation?answertab=oldest#tab-top)[votes](https://stackoverflow.com/questions/5478328/jpa-jointable-annotation?answertab=votes#tab-top)

289

**EDIT 2017-04-29**: As pointed to by some of the commenters, the JoinTable example does not need the mappedBy annotation attribute. In fact, recent versions of Hibernate refuse to start up by printing the following error:

org.hibernate.AnnotationException:

Associations marked as mappedBy must not define database mappings

like @JoinTable or @JoinColumn

Let's pretend that you have an entity named Project and another entity named Task and each project can have many tasks.

You can design the database schema for this scenario in two ways.

The first solution is to create a table named Project and another table named Task and add a foreign key column to the task table named project\_id:

Project Task

------- ----

id id

name name

project\_id

This way, it will be possible to determine the project for each row in the task table. If you use this approach, in your entity classes you won't need a join table:

@Entity

public class Project {

@OneToMany(mappedBy = "project")

private Collection<Task> tasks;

}

@Entity

public class Task {

@ManyToOne

private Project project;

}

The other solution is to use a third table, e.g. Project\_Tasks, and store the relationship between projects and tasks in that table:

Project Task Project\_Tasks

------- ---- -------------

id id project\_id

name name task\_id

The Project\_Tasks table is called a "Join Table". To implement this second solution in JPA you need to use the @JoinTable annotation. For example, in order to implement a uni-directional one-to-many association, we can define our entities as such:

**Project entity:**

@Entity

public class Project {

@Id

@GeneratedValue

private Long pid;

private String name;

@JoinTable

@OneToMany

private List<Task> tasks;

public Long getPid() {

return pid;

}

public void setPid(Long pid) {

this.pid = pid;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public List<Task> getTasks() {

return tasks;

}

public void setTasks(List<Task> tasks) {

this.tasks = tasks;

}

}

**Task entity:**

@Entity

public class Task {

@Id

@GeneratedValue

private Long tid;

private String name;

public Long getTid() {

return tid;

}

public void setTid(Long tid) {

this.tid = tid;

}

public String getName() {

return name;

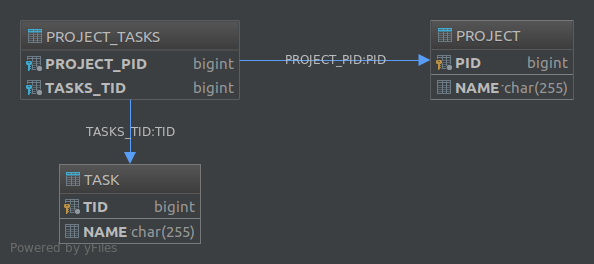
}

public void setName(String name) {

this.name = name;

}

}

This will create the following database structure: 

The @JoinTable annotation also lets you customize various aspects of the join table. For example, had we annotated the tasks property like this:

@JoinTable(

name = "MY\_JT",

joinColumns = @JoinColumn(

name = "PROJ\_ID",

referencedColumnName = "PID"

),

inverseJoinColumns = @JoinColumn(

name = "TASK\_ID",

referencedColumnName = "TID"

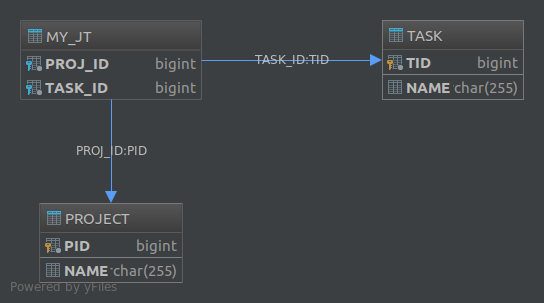
)

)

@OneToMany

private List<Task> tasks;

The resulting database would have become:



Finally, if you want to create a schema for a many-to-many association, using a join table is the only available solution.

-----

It's the only solution to map a ManyToMany association : you need a join table between the to entities tables to map the association.

It's also used for OneToMany (usually unidirectional) associations, when you don't want to add a foreign key in the table of the many side, and thus keep it independant of the one side.

Search for @JoinTable in the [hibernate documentation](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/) for explanations and examples.

It lets you handle Many to Many relationship. Example:

Table 1: post

post has following columns

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| ID | DATE |

|\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_|

| | |

|\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_|

Table 2: user

user has the following columns:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| ID |NAME |

|\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_|

| | |

|\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_|

Join Table lets you create a mapping using:

@JoinTable(

name="USER\_POST",

joinColumns=@JoinColumn(name="USER\_ID", referencedColumnName="ID"),

inverseJoinColumns=@JoinColumn(name="POST\_ID", referencedColumnName="ID"))

will create a table:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| USER\_ID| POST\_ID |

|\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_|

| | |

|\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_|

[share](https://stackoverflow.com/a/44332640)[improve this answer](https://stackoverflow.com/posts/44332640/edit)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| JPA - @JoinTable Examples | Top of Form   |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | |  |   Bottom of Form |
| [JPA](https://www.logicbig.com/how-to/jpa.html) [JAVA EE](https://www.logicbig.com/how-to/java-ee.html)  @JoinTable can be used to map following associations to database table: bidirectional many-to-one/one-to-many, unidirectional many-to-one, and one-to-one (both bidirectional and unidirectional) associations.  import javax.persistence.\*; import java.util.List;  @Entity public class EntityA {  @Id  @GeneratedValue  private int myIdA;  @OneToMany  @JoinTable(name = "MY\_JOIN\_TABLE",  joinColumns = {@JoinColumn(name = "MY\_ENTITY\_A\_FK")},  inverseJoinColumns = {@JoinColumn(name = "MY\_ENTITY\_B\_FK")}  )  private List<EntityB> entityBList;   public List<EntityB> getEntityBList() {  return entityBList;  }   public void setEntityBList(List<EntityB> entityBList) {  this.entityBList = entityBList;  }   @Override  public String toString() {  return "EntityA{" +  "myIdA=" + myIdA +  ", entityBList=" + entityBList +  '}';  } }  [Original Post](https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/one-to-many-with-join-table.html)    @Entity public class EntityA {  @Id  @GeneratedValue  private int myIdA;  private String strA;  @OneToMany  @JoinTable(name = "MY\_JOIN\_TABLE",  joinColumns = {@JoinColumn(name = "MY\_ENTITY\_A\_FK", referencedColumnName = "myIdA")},  inverseJoinColumns = {@JoinColumn(name = "MY\_ENTITY\_B\_FK", referencedColumnName = "myIdB")}  )  private List<EntityB> entityBList;   public String getStrA() {  return strA;  }   public void setStrA(String strA) {  this.strA = strA;  }   public List<EntityB> getEntityBList() {  return entityBList;  }   public void setEntityBList(List<EntityB> entityBList) {  this.entityBList = entityBList;  }   @Override  public String toString() {  return "EntityA{" +  "myIdA=" + myIdA +  ", strA='" + strA + '\'' +  '}';  } }  @Entity public class EntityB {  @Id  @GeneratedValue  private int myIdB;  private String strB;  @ManyToOne  @JoinTable(name = "MY\_JOIN\_TABLE",  joinColumns = {@JoinColumn(name = "MY\_ENTITY\_B\_FK", insertable = false,  updatable = false, referencedColumnName = "myIdB")},  inverseJoinColumns = {@JoinColumn(name = "MY\_ENTITY\_A\_FK", insertable = false,  updatable = false, referencedColumnName = "myIdA")}  )  private EntityA refEntityA;   public String getStrB() {  return strB;  }   public void setStrB(String strB) {  this.strB = strB;  }   public EntityA getRefEntityA() {  return refEntityA;  }   public void setRefEntityA(EntityA refEntityA) {  this.refEntityA = refEntityA;  }   @Override  public String toString() {  return "EntityB{" +  "myIdB=" + myIdB +  ", strB='" + strB + '\'' +  '}';  } }  [Original Post](https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/one-to-many-bidirectional-join-table.html)    @OneToOne unidirectional with @JoinTable:  @Entity public class EntityA {  @Id  @GeneratedValue  private int myIdA;  private String stringA;  @OneToOne  @JoinTable(name = "MY\_JOIN\_TABLE",  joinColumns = {  @JoinColumn(name = "ENTITYA\_FK", referencedColumnName = "myIdA")  },  inverseJoinColumns = {  @JoinColumn(name = "ENTITYB\_FK", referencedColumnName = "myIdB", unique = true)  }  )  private EntityB entityB;  ............. }  @Entity public class EntityB {  @Id  @GeneratedValue  private int myIdB;  private String stringB;  ............. }  [Original Post](https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/one-to-one-join-table.html)    @OneToOne bidirectional with @JoinTable:  @Entity public class EntityA {  @Id  @GeneratedValue  private int myIdA;  private String stringA;  @OneToOne  @JoinTable(name = "MY\_JOIN\_TABLE",  joinColumns = {  @JoinColumn(name = "ENTITYA\_FK", referencedColumnName = "myIdA")  },  inverseJoinColumns = {  @JoinColumn(name = "ENTITYB\_FK", referencedColumnName = "myIdB", unique = true)  }  )  private EntityB entityB;  ............. }  @Entity public class EntityB {  @Id  @GeneratedValue  private int myIdB;  private String stringB;   @OneToOne(mappedBy = "entityB")  private EntityA entityA;  ............. }  [Original Post](https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/one-to-one-bidirectional-join-table.html)  @Entity public class Employee {  @Id  @GeneratedValue  private long id;  private String name;  @OneToMany  @JoinTable(name = "ASSIGNED\_TASKS",  joinColumns = {@JoinColumn(name = "EMPLOYEE\_FK")},  inverseJoinColumns = {@JoinColumn(name = "TASK\_FK")})  @MapKeyColumn(name = "TASK\_DATE")  private Map<Date, Task> tasks;  ............. }  @Entity public class Task {  @Id  @GeneratedValue  private long id;  private String name;  private String description;  ............. } |  |

## Chapter 24. Best Practices

**Write fine-grained classes and map them using <component>:**

Use an Address class to encapsulate street, suburb, state, postcode. This encourages code reuse and simplifies refactoring.

**Declare identifier properties on persistent classes:**

Hibernate makes identifier properties optional. There are a range of reasons why you should use them. We recommend that identifiers be 'synthetic', that is, generated with no business meaning.

**Identify natural keys:**

Identify natural keys for all entities, and map them using <natural-id>. Implement equals() andhashCode() to compare the properties that make up the natural key.

**Place each class mapping in its own file:**

Do not use a single monolithic mapping document. Map com.eg.Foo in the file com/eg/Foo.hbm.xml. This makes sense, particularly in a team environment.

**Load mappings as resources:**

Deploy the mappings along with the classes they map.

**Consider externalizing query strings:**

This is recommended if your queries call non-ANSI-standard SQL functions. Externalizing the query strings to mapping files will make the application more portable.

**Use bind variables.**

As in JDBC, always replace non-constant values by "?". Do not use string manipulation to bind a non-constant value in a query. You should also consider using named parameters in queries.

**Do not manage your own JDBC connections:**

Hibernate allows the application to manage JDBC connections, but his approach should be considered a last-resort. If you cannot use the built-in connection providers, consider providing your own implementation of org.hibernate.connection.ConnectionProvider.

**Consider using a custom type:**

Suppose you have a Java type from a library that needs to be persisted but does not provide the accessors needed to map it as a component. You should consider implementing org.hibernate.UserType. This approach frees the application code from implementing transformations to/from a Hibernate type.

**Use hand-coded JDBC in bottlenecks:**

In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. Do not assume, however, that JDBC is necessarily faster. Please wait until you know something is a bottleneck. If you need to use direct JDBC, you can open a Hibernate Session and usingfile:///usr/share/doc/HTML/en-US/index.html that JDBC connection. This way you can still use the same transaction strategy and underlying connection provider.

**Understand Session flushing:**

Sometimes the Session synchronizes its persistent state with the database. Performance will be affected if this process occurs too often. You can sometimes minimize unnecessary flushing by disabling automatic flushing, or even by changing the order of queries and other operations within a particular transaction.

**In a three tiered architecture, consider using detached objects:**

When using a servlet/session bean architecture, you can pass persistent objects loaded in the session bean to and from the servlet/JSP layer. Use a new session to service each request. Use Session.merge()or Session.saveOrUpdate() to synchronize objects with the database.

**In a two tiered architecture, consider using long persistence contexts:**

Database Transactions have to be as short as possible for best scalability. However, it is often necessary to implement long running application transactions, a single unit-of-work from the point of view of a user. An application transaction might span several client request/response cycles. It is common to use detached objects to implement application transactions. An appropriate alternative in a two tiered architecture, is to maintain a single open persistence contact session for the whole life cycle of the application transaction. Then simply disconnect from the JDBC connection at the end of each request and reconnect at the beginning of the subsequent request. Never share a single session across more than one application transaction or you will be working with stale data.

**Do not treat exceptions as recoverable:**

This is more of a necessary practice than a "best" practice. When an exception occurs, roll back the Transaction and close the Session. If you do not do this, Hibernate cannot guarantee that in-memory state accurately represents the persistent state. For example, do not use Session.load() to determine if an instance with the given identifier exists on the database; use Session.get() or a query instead.

**Prefer lazy fetching for associations:**

Use eager fetching sparingly. Use proxies and lazy collections for most associations to classes that are not likely to be completely held in the second-level cache. For associations to cached classes, where there is an a extremely high probability of a cache hit, explicitly disable eager fetching using lazy="false". When join fetching is appropriate to a particular use case, use a query with a left join fetch.

**Use the open session in view pattern, or a disciplined assembly phase to avoid problems with unfetched data:**

Hibernate frees the developer from writing tedious Data Transfer Objects (DTO). In a traditional EJB architecture, DTOs serve dual purposes: first, they work around the problem that entity beans are not serializable; second, they implicitly define an assembly phase where all data to be used by the view is fetched and marshalled into the DTOs before returning control to the presentation tier. Hibernate eliminates the first purpose. Unless you are prepared to hold the persistence context (the session) open across the view rendering process, you will still need an assembly phase. Think of your business methods as having a strict contract with the presentation tier about what data is available in the detached objects. This is not a limitation of Hibernate. It is a fundamental requirement of safe transactional data access.

**Consider abstracting your business logic from Hibernate:**

Hide Hibernate data-access code behind an interface. Combine the DAO and Thread Local Sessionpatterns. You can even have some classes persisted by handcoded JDBC associated to Hibernate via a UserType. This advice is, however, intended for "sufficiently large" applications. It is not appropriate for an application with five tables.

**Do not use exotic association mappings:**

Practical test cases for real many-to-many associations are rare. Most of the time you need additional information stored in the "link table". In this case, it is much better to use two one-to-many associations to an intermediate link class. In fact, most associations are one-to-many and many-to-one. For this reason, you should proceed cautiously when using any other association style.

**Prefer bidirectional associations:**

Unidirectional associations are more difficult to query. In a large application, almost all associations must be navigable in both directions in queries