# **Guided LAB 305.5.1A - Demonstration to ManyToOne and OneToMany Relationships and Mapping**

# 

## **Lab Overview:**

## In this lab, we will explore and demonstrate the **OnetoMany** relationship and the **ManytoOne** relationship in Hibernate. This demonstration consists of the following files:

## **Model classes: Department.java and Teacher.java**

## **Hibernate XML configuration file: hibernate.cfg.xml**

## **Run program: App.java**

## **Maven project: pom.xml**

## **Learning Objectives:**

By the end of this lab, learners will be able to:

* Describe the One-to-Many relationship in Hibernate.
* Describe the Many-to-One relationship in Hibernate.
* Utilize the One-to-Many and Many-to-One in the Java applications.

## **Example: @ManyToOne Relationship and Mapping**

**Scenario:**

Let us consider an example of a relationship between a **Teacher** anda **department** entity. A Department consists of multiple Teachers, but each Teacher belongs to only one Department. That is a typical example of a **many-to-one** relationship or association. We will model this in a database where we need to store the [primary key](https://thorben-janssen.com/jpa-generate-primary-keys/) of the ***Department*** record as a foreign key in the ***Teacher*** table.

### **Step 1: Setup Java Maven Project and Add Jar Dependencies**

* For **Hibernate** and **MySQL Database:** Add the following **jar dependencies** in the **pom.xml** file under the **</dependencies> tag** of your maven project.

| ***<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->***  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **<version>8.0.25</version>**  **</dependency>**  **<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core --> <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>5.5.7.Final</version> </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-annotations</artifactId>  <version>3.5.5-Final</version> </dependency>** |
| --- |

### **Step 2: Create the Persistence class (Model class or Pojo).**

* Create a package **“com.perscholas.model”** under  ***src →. Main → java.***
* Create an entity class named **“Department.java”** under the above package.
  + ***src\main\java\com\perscholas\model\Department.java***
* Following is the initial code of the **Department.java** class:

| import java.io.Serializable; import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; import javax.persistence.Table; @Entity @Table public class Department implements Serializable {  private static final long serialVersionUID = 1L;  @Id  @GeneratedValue( strategy=GenerationType.IDENTITY )  private int did;  private String dname;    public Department(int did, String dname) {  super();  this.did = did;  this.dname = dname;  }  public Department()  {   }  public int getDid() {  return did;  }  public void setDid(int did) {  this.did = did;  }  public String getDname() {  return dname;  }  public void setDname(String dname) {  this.dname = dname;  } } |
| --- |

* Create a second entity class named **“Teacher.java”** under the above package.
  + ***src\main\java\com\perscholas\model\Teacher.java***
* Here is the initial code of the **Teacher.java** class:

In the **Teacher** class, we specified a ***ManyToOne*** relationship between the **Department** and **Teacher** entities, as one Department can have more than one Teacher.

| import java.io.Serializable; import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; import javax.persistence.ManyToOne; import javax.persistence.Table; @Entity @Table public class Teacher implements Serializable {  private static final long serialVersionUID = 1L;  @Id  @GeneratedValue( strategy=GenerationType.IDENTITY )  private int tit;  private String salary;  private String Teachername;    **@ManyToOne**  private Department department;  public Teacher(int tit, String salary, String teachername) {  super();  this.tit = tit;  this.salary = salary;  Teachername = teachername; }  public Teacher()  {}    public Department getDep() {  return department; }  public void setDep(Department department) {  this.department = department;  }  public int getTit() {  return tit;  }  public void setTit(int tit) {  this.tit = tit;  }  public String getSalary() {  return salary;  }  public void setSalary(String salary) {  this.salary = salary;  }  public String getTeachername() {  return Teachername;  }  public void setTeachername(String teachername) {  Teachername = teachername; }  } |
| --- |

### **Explanation:**

### The **Teacher** entity represents the many sides of the relationship, and the **Teacher** table contains the **foreign key** of the record in the **Department**table. As you can see in the above code snippet, we can model this association with an attribute/variable of type **department** and a **@ManyToOne** annotation.

### @ManyToOne private Department department;

### The “***Department department”*** attribute/variable models the association, and The annotation tells Hibernate how to map it to the database.

### That is all you need to do to model this association. By default, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute. In this example, Hibernate would use a column named ***department\_did*** to store the foreign key to the ***Teacher*** entity.

### If you want to use a different column, you need to define the foreign key column name with a **@JoinColumn** annotation. The example in the following code snippet tells Hibernate to use the column fk\_order to store the foreign key.

| @Entity  @Table  public class Teacher{  @ManyToOne  @JoinColumn(name = “fk\_dep”)  private Department department;  } |
| --- |

### **Step 3: Create the Hibernate Configuration File (hibernate.cfg.xml)**

* **For Eclipse IDE**: Create the configuration file. To do so, right-click on ***src/main/java*** *→* ***New → Other - search files from search panel → Click on File → specify the file nam****e “hibernate.cfg.xml” →* ***Finish****.*
* **For the IntelliJ IDE:** Create a configuration file named *hibernate.cfg.xml* under the resources folder and write the following code in it.
* Open the ***hibernate.cfg.xml*** file and paste the following XML code for configuration

*Note: In this lab, we will use the* ***“usersdb”*** *database, but if you want to use another database for that, you have to change the database name in the below code.*

| <?xml version="1.0" encoding="UTF-8"?>  **<!DOCTYPE hibernate-configuration PUBLIC**  **"-//Hibernate/Hibernate Configuration DTD 3.0//EN"**  **"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"*>***  <hibernate-configuration>   <session-factory>  <property name="hibernate.hbm2ddl.auto"> update </property>  <property name="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  <property name="connection.url">jdbc:mysql://localhost:3306/usersdb</property> <property name="connection.username">root</property> <property name="connection.password">password</property>  <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>  <property name="hibernate.show\_sql" >true </property> <property name="hibernate.format\_sql" >true </property>   <!-- Mapping entity file -->  <mapping class="com.perscholas.model.Teacher"/>   <mapping class="com.perscholas.model.Department"/>   </session-factory>  </hibernate-configuration> |
| --- |

The **“tit**” value is auto generated, so the value need not be set here. The session.persist() method is used to persist the value in the database, and once the value is saved, the id value (primary key) is returned. Once the object is saved, the transaction is committed. If an exception occurs, the transaction is rolled back. The transaction ends either through a commit or rollback action. Once the transaction ends, the session is closed.

**Step 4: Create App.java(main class)**

Note: If you are using the **IntelliJ IDE,** you have to create an “**App.java”** class. However, if you are using the **Eclipse IDE, the** “**App.java”** class is created automatically.

Add the following code to the App.java class:

**App.java**

| import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration; public class App {  public static void main(String[] args) {  // TODO Auto-generated method stub  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction t = session.beginTransaction();  //creating departments  Department dep = new Department();  dep.setDname("IT");    Department dep2= new Department();  dep2.setDname("HR");  //creating teacher  Teacher t1 = new Teacher();  t1.setDep(dep);  t1.setSalary("1000");  t1.setTeachername("MHaseeb");    Teacher t2 = new Teacher();  t2.setDep(dep);  t2.setSalary("2220");  t2.setTeachername("Shahparan");    Teacher t3 = new Teacher();  t3.setDep(dep);  t3.setSalary("30000");  t3.setTeachername("James");    Teacher t4 = new Teacher();  t4.setDep(dep2);  t4.setSalary("40000");  t4.setTeachername("Joseph");  //Storing Departments in database  session.persist(dep);  session.persist(dep2);  //Storing teachers in database  session.persist(t1);  session.persist(t2);  session.persist(t3);  session.persist(t4);  t.commit(); } } |
| --- |

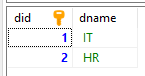
### **Step 4: Run an Application**

Now, let us execute the code we created above. Let’s run the App class.

**Right click on App.java → Run As → Java Application.**

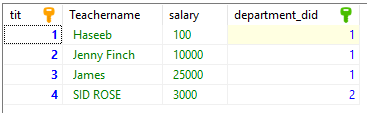
Once you execute your **App.java** class, the database schema will be created, and you will see the **Department table** and **Teacher table** created, along with the records in the ***usersdb*** database.

**Department Table**



* + DID in the Primary key in a Department Table

**Teacher Table**



* **DEPARTMENT\_DID** in the Teacher Table is the foreign key (reference field) from the Department table, and tit is the Primary key.

## **Example: @OneToMany Relationship and Mapping Example**

In a **one-to-many** relationship, each row of one entity is referenced by many child records in another entity. The important thing is that children's records cannot have multiple parents. In a one-to-many relationship between Table A and Table B, each row in Table A is linked to 0, 1, or many rows in Table B.

Let us consider the above example. If **Teacher** and **Department** are in a reverse unidirectional manner, the relation is a many-to-one relation.

In this example, we will modify the above example, but you can also set up a new Hibernate project, and then apply a one-to-manyrelationship there.

### **Step 1: Teacher.java Class**

Below are the modifications we will perform in the **Teacher()** class:

* Remove the Department variable/attribute highlighted in red.
* Remove the @manytoone annotation highlighted in red.
* Remove getDep() and setDep() methods.

Below is the **Teacher** class with modifications:

| package com.perscholas.model; import java.io.Serializable; import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; import javax.persistence.JoinColumn; import javax.persistence.ManyToOne; import javax.persistence.Table; @Entity @Table public class Teacher implements Serializable {  private static final long serialVersionUID = 1L;  @Id  @GeneratedValue( strategy=GenerationType.IDENTITY )  private int tit;  private String salary;  private String Teachername;  **~~@ManyToOne~~**  **~~private Department department;~~**  public Teacher(int tit, String salary, String teachername) {  super();  this.tit = tit;  this.salary = salary;  Teachername = teachername;  }  public Teacher()  {}    public int getTit() {  return tit;  }  public void setTit(int tit) {  this.tit = tit;  }  public String getSalary() {  return salary;  }  public void setSalary(String salary) {  this.salary = salary;  }  public String getTeachername() {  return Teachername;  }  public void setTeachername(String teachername) {  Teachername = teachername;  }  } |
| --- |

### **Step 2: Department.java Class**

One-to-many relationship mapping is not very common. In the following example, it only models the association on the **Department** entity and not on the **Teacher** entity. The basic mapping definition is very similar to the one-to-many association. It consists of the “**List teacherList”** attribute, which stores the associated entities, and a @OneToMany association, as shown in the following source code:

| package com.perscholas.model; import java.io.Serializable; import java.util.List; import javax.persistence.CascadeType; import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; import javax.persistence.OneToMany; import javax.persistence.Table; @Entity @Table public class Department implements Serializable {  private static final long serialVersionUID = 1L;  @Id  @GeneratedValue( strategy=GenerationType.IDENTITY )  private int did;  private String dname;   **@OneToMany(targetEntity=Teacher.class, cascade = {CascadeType.ALL})**  private List teacherList;  public Department(int did, String dname) {  super();  this.did = did;  this.dname = dname;  }  public Department()  {    }  public List getTeacherList() {  return teacherList;  }   public void setTeacherList(List teacherList) {  this.teacherList = teacherList;  }  public int getDid() {  return did;  }  public void setDid(int did) {  this.did = did;  }    public String getDname() {  return dname;  }  public void setDname(String dname) {  this.dname = dname;  } } |
| --- |

**Explanation**:

We used the **‘List<E>’** datatype for the **Teachers** class. You can also use the **“Set<E>”** in the **Teacher** class so that every record is unique.

We wanted to save the mapped entity whenever the relationship owner entity is saved. To enable this, we had to use the **“CascadeType”** attribute.

Look at the bold line in the above source code for department.java. We used the @***onetoman***y annotation, along with two additional attributes:

‘targetEntity=Teacher.class' and “cascade=CascadeType.ALL”

Cascade=CascadeType.ALL: Any change that happens to the **teacher** must cascade to the **department** as well. If you save a **Teacher**, all associated accounts will also be saved into the database. If you delete a **Teacher**, all accounts associated with that **Teacher** will also be deleted.

targetEntity=Teacher.class: It is used to determine the entity class that is the target of the association.

targetEntity: Specify the entity class that is the target of the association. This is optional only if the collection property is defined using Java generics; It must be specified otherwise.

### **Step 3: App.java class**

| package com.perscholas.model; import java.util.ArrayList; import java.util.List; import javax.persistence.Query; import org.hibernate.Session; import org.hibernate.SessionFactory; import org.hibernate.Transaction; import org.hibernate.cfg.Configuration; public class App  {  public static void main( String[] args )  {  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction t = session.beginTransaction(); Teacher t1 = new Teacher();  t1.setTeachername("Haseeb");  t1.setSalary("100");   Teacher t2 = new Teacher();  t2.setTeachername("Jenny Finch");  t2.setSalary("10000");    Teacher t3 = new Teacher();  t3.setTeachername("James");  t3.setSalary("25000");    Teacher t4 = new Teacher();  t4.setTeachername("SID ROSE");  t4.setSalary("3000");    Teacher t5 = new Teacher();  t5.setSalary("200");  t5.setTeachername("Ali");    //Add teacher entity object to the list  List<Teacher> teachlist = new ArrayList();  teachlist.add(t1);  teachlist.add(t2);  teachlist.add(t3);  teachlist.add(t4);  teachlist.add(t5);  session.persist(t1);  session.persist(t2);  session.persist(t3);  session.persist(t4);  session.persist(t5);  //Creating Department   Department department = new Department();  department.setDname("Development");  department.setTeacherList(teachlist);   //Storing Department  session.persist(department);  t.commit(); } } |
| --- |

### **Step 4: Run an Application**

Now let us execute the code we created above. Let’s run the App class.

**Right-click on App.java → Run As → Java Application.**

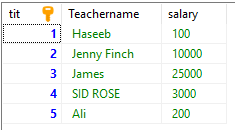
Note: If you are using **the IntelliJ IDE,** you have to create an App.java class; however, if you are using the **Eclipse IDE,** the “App.java” class is created automatically.

**Explanation**: In the above code, we have created one department object and several teacher objects. We added all teacher objects to one department. As a result, all teachers will belong to one department.

The **“tit**” value is auto generated; therefore, the value need not be set here. The **session.persist()** method is used to persist the value in the database, and once the value is saved, the id value (Primary Key) is returned. Once the object is saved, the transaction is committed. If an exception occurs, the transaction is rolled back. The transaction ends either through a commit or rollback action. Once the transaction ends, the session is closed.

**Result**: At the start of each thread, a database schema will be created and you will see the following result in ***userdb*** Database

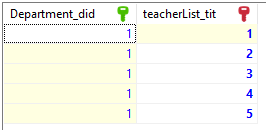
Teacher screenshot



Department screenshot



Department\_teacher table:



As you can see in the above screenshot, the joined table has been created. The join table (Department\_teacher) will have two foreign key columns (i.e., it will have both primary key columns from the two entity tables from the Department and Teacher tables [**Department\_did** and **techerlist\_tit]**, respectively). One of the foreign keys will serve as the primary key for the join table. A unique constraint will be applied to the remaining foreign key column; hence, both foreign key columns will not have duplicate values.

**Submission Instructions:**

Include the following deliverables in your submission:

* Submit your source code or screenshot using the Start Assignment button in the top-right corner of the assignment page in Canvas.

## 