Information Systems Design & Development

Relational Object Mapping (JPA - Hibernate)

Objectives

- Know and understand the technique of Relational Object Mapping
- Learn how to use a specific implementation of Relational Object Mapping (ORM) software
- Learn how to implement CRUD operations in Java using an ORM tool

Mapeo Objeto Relacional. Conceptos y definiciones

- An ORM is a programming model that allows you to "map" the structures of a relational database (SQL Server, Oracle, MySQL, etc.) on a logical structure of entities to simplify and accelerate the development of our applications
- The structures of the relational database are linked to the logical entities or virtual database defined in the ORM, so that the CRUD actions (Create, Read, Update, Delete) are performed, indirectly, through the ORM

- ORM tools, in addition to generating code automatically, use their own language to perform queries and manage data persistence
- The objects or entities of the virtual database created by the ORM can be managed with general purpose languages
- The interaction with the DBMS will be carried out using the ORM's own update methods
- Interacting directly with virtual database entities without generating SQL code can lead to significant benefits in accelerating application development or deployment

Advantages and disadvantages of ORMs

Advantage

- You don't need to write SQL code. Many programmers don't master it and sometimes it can be complex and error-prone.
- Increase code reuse and improve code maintenance
- Reduces development time
- Greater security, avoiding possible SQL injection attacks and the like
- They are well optimized for insert, delete, and update CRUD operations, but for recovery (select) it is better to use native SQ

Inconvenience

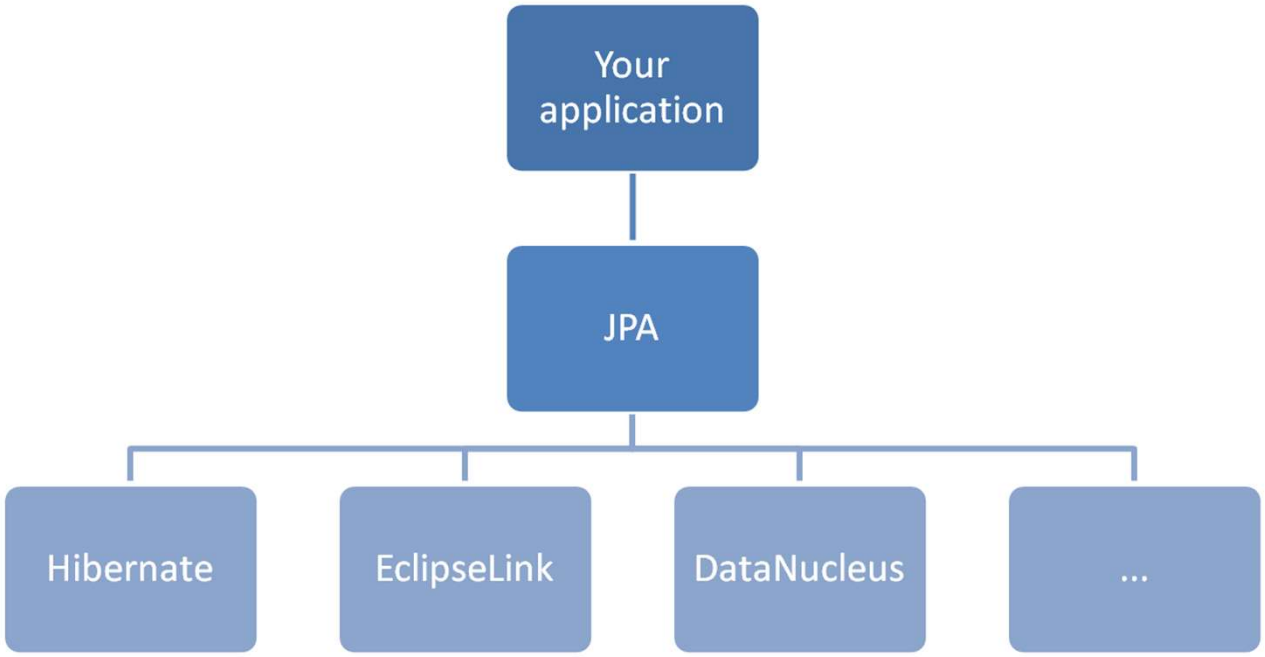
In high-load environments it can reduce performance as an extra layer is being added to the system

Their learning can become very complex

If the queries are complex, the ORM does not guarantee good optimization, as a developer would do manually. Some ORMs offer extensions for writing queries in native SQL.

What is JPA? What is Hibernate?

- JPA (Java Persistence API) is a specification that defines how object persistence works in Java
- Hibernate is one of the frameworks that implements the JPA specification. There are others like EclipseLink, DataNucleus, TopLink, etc.



JDBC and Hibernate

```
... Trainer
      ps = connectionDB.getConexion().prepareStatement("INSERT INTO trainer VALUES (?, ?, ?, ?, ?, ?, ?)");
      ps.setString(1, trainer.getCodMonitor());
      ps.setString(2, trainer.getName());
      ps.setString(3, trainer.getID());
      ps.setString(4, trainer.getPhone());
      ps.setString(5, trainer.getEmail());
      ps.setString(6, trainer.getDate());
      ps.setString(7, trainer.getNick());

      ps.executeUpdate();
      ps.close();
   ...
```

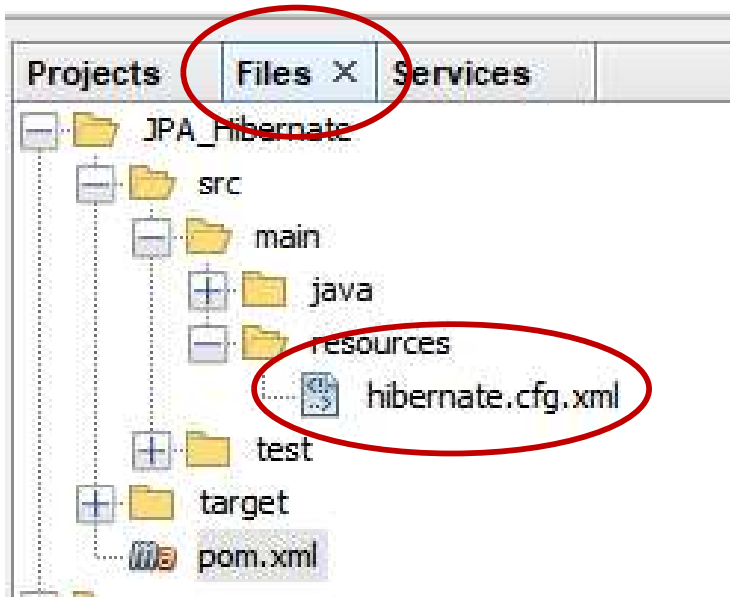
```
...
            sesion = HibernateUtil.getCurrentSession();
            sesion.beginTransaction();
            sesion.save(trainer);
            sesion.getTransaction().commit();
            sesion.close();
...
```

Adding MAVEN dependency

```
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-tools-maven-plugin</artifactId>
        <version>5.6.0.Final</version>
        <type>maven-plugin</type>
</dependency>
```

Hibernate configuration file

- Hibernate uses a configuration file, called hibernate.cfg.xml
- This file must be placed in the root folder of the project. Specifically in src/main/resources



- if the resources folder does not exist it must be created
- Once in the folder, with the right button select "New" + "Xml file"

Hibernate configuration file

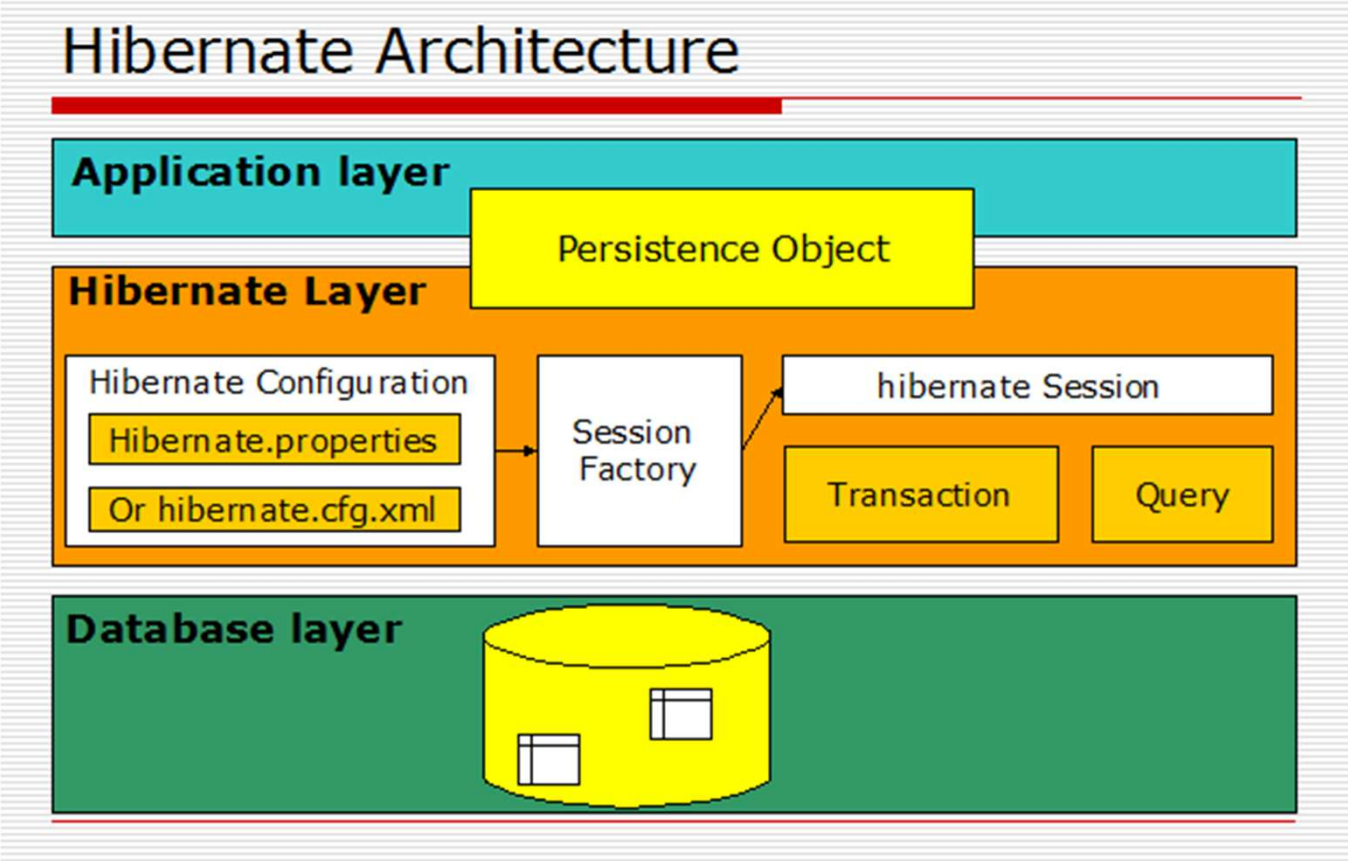
- Example

```
<?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">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver</property>
      <property name="hibernate.connection.url">jdbc:oracle:thin:@172.17.20.75:1521:rabida</property>
      <property name="hibernate.connection.username">user</property>
      <property name="hibernate.connection.password">password</property>
      <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
      <property name="hibernate.show_sql">true</property>
      <mapping class="Model.Trainer"/>
      <mapping class="Model.Activity"/>
      <mapping class="Model.Member"/>
   </session-factory>
</hibernate-configuration>
```

- <mapping class> tags define the classes that are mapped with the database tables
- Tables that arise from "many to many" relationships are not mapped in the application (their operation will be seen later)

Connection and communication between the application and the database

- Sessionfactory and Session establish the connection and communication between the database and the application
- The SessionFactory class, along with its main methods, is usually created in the HibernateUtil class, which we can put anywhere in the project

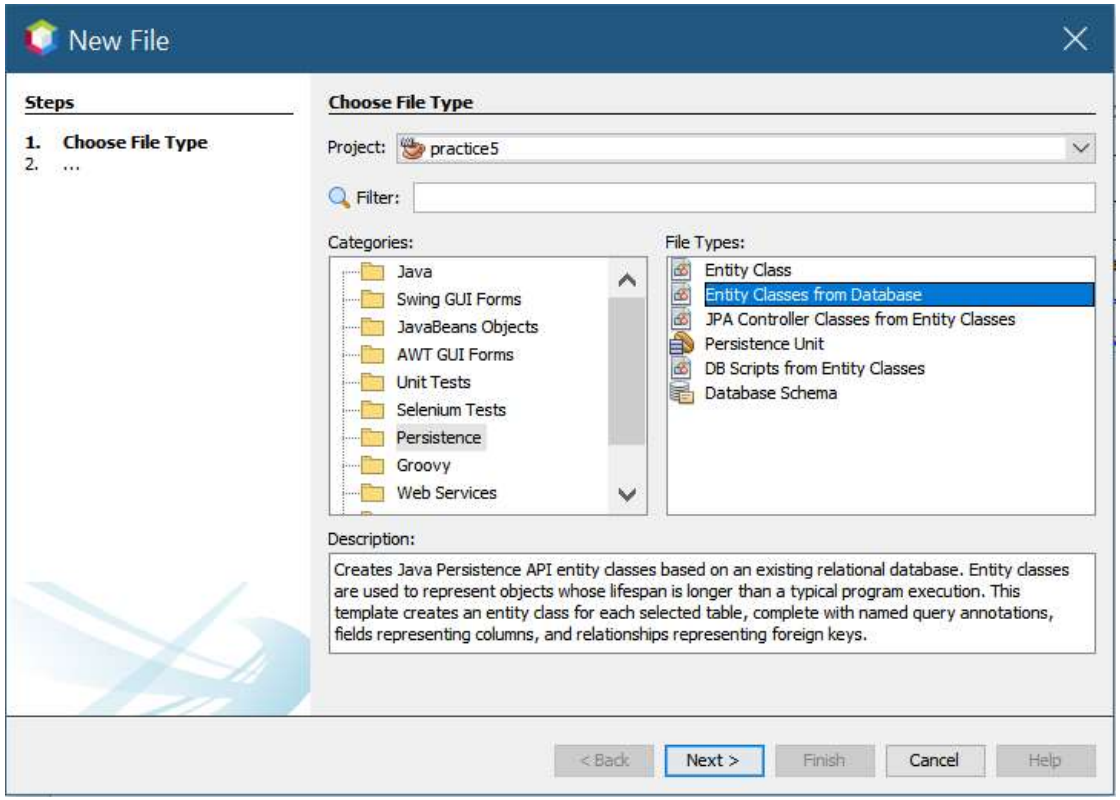


HibernateUtil example

```
public class HibernateUtil {

   private static final SessionFactory sessionFactory = buildSessionFactory();

   private static SessionFactory buildSessionFactory() {
      try {
         ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
                .configure("hibernate.cfg.xml").build();
         Metadata metadata = new MetadataSources(serviceRegistry).getMetadataBuilder().build();
         return metadata.getSessionFactoryBuilder().build();

      } catch (Throwable ex) {
         System.err.println("Build SeesionFactory failed :" + ex);
         throw new ExceptionInInitializerError(ex);
      }
   }

   public static SessionFactory getSessionFactory() {  return sessionFactory;   }

   public static void close() {
      if ((sessionFactory!=null) && (sessionFactory.isClosed()==false)) {
                sessionFactory.close();
                sessionFactory.close();
      }
   }
}
```

Creating classes derived from database tables

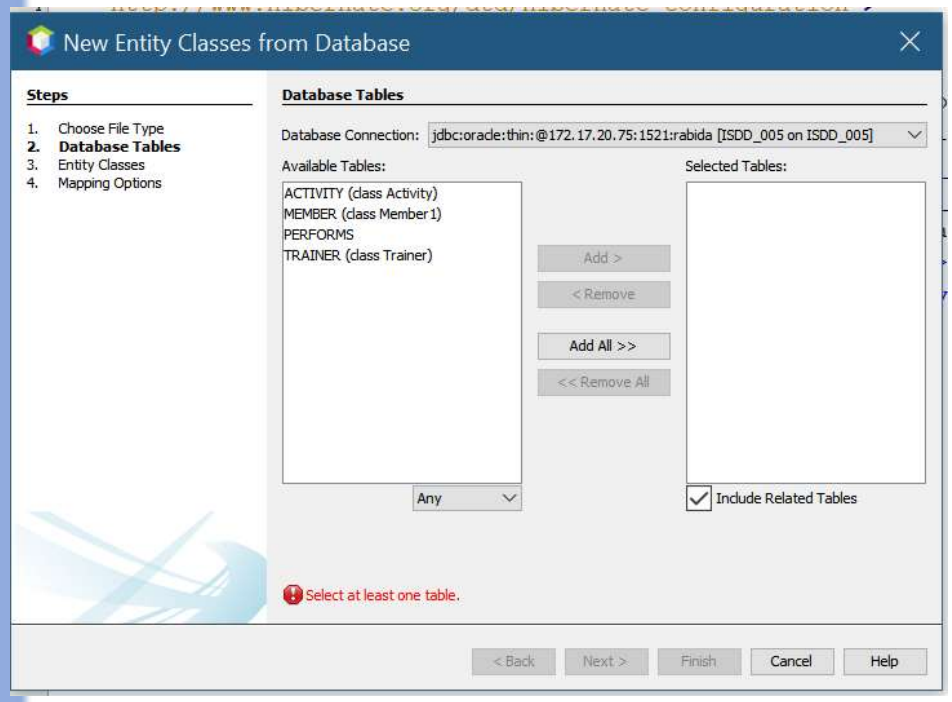
- The next step is creating the classes on which we will map the database tables
- As always, it can be done manually or with a tool. In this case we will rely on a NetBeans functionality
- The functionality is called "Entity classes from databases". Simply choose the database tables that need to be mapped
- as a result, as many classes as tables have been selected will be created

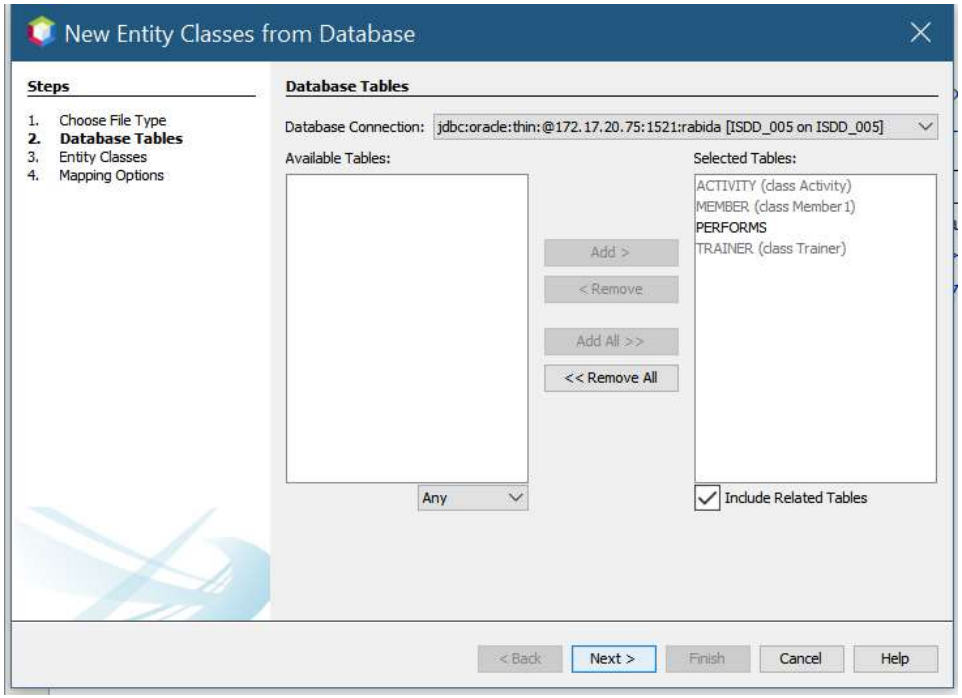


Creating classes derived from database tables

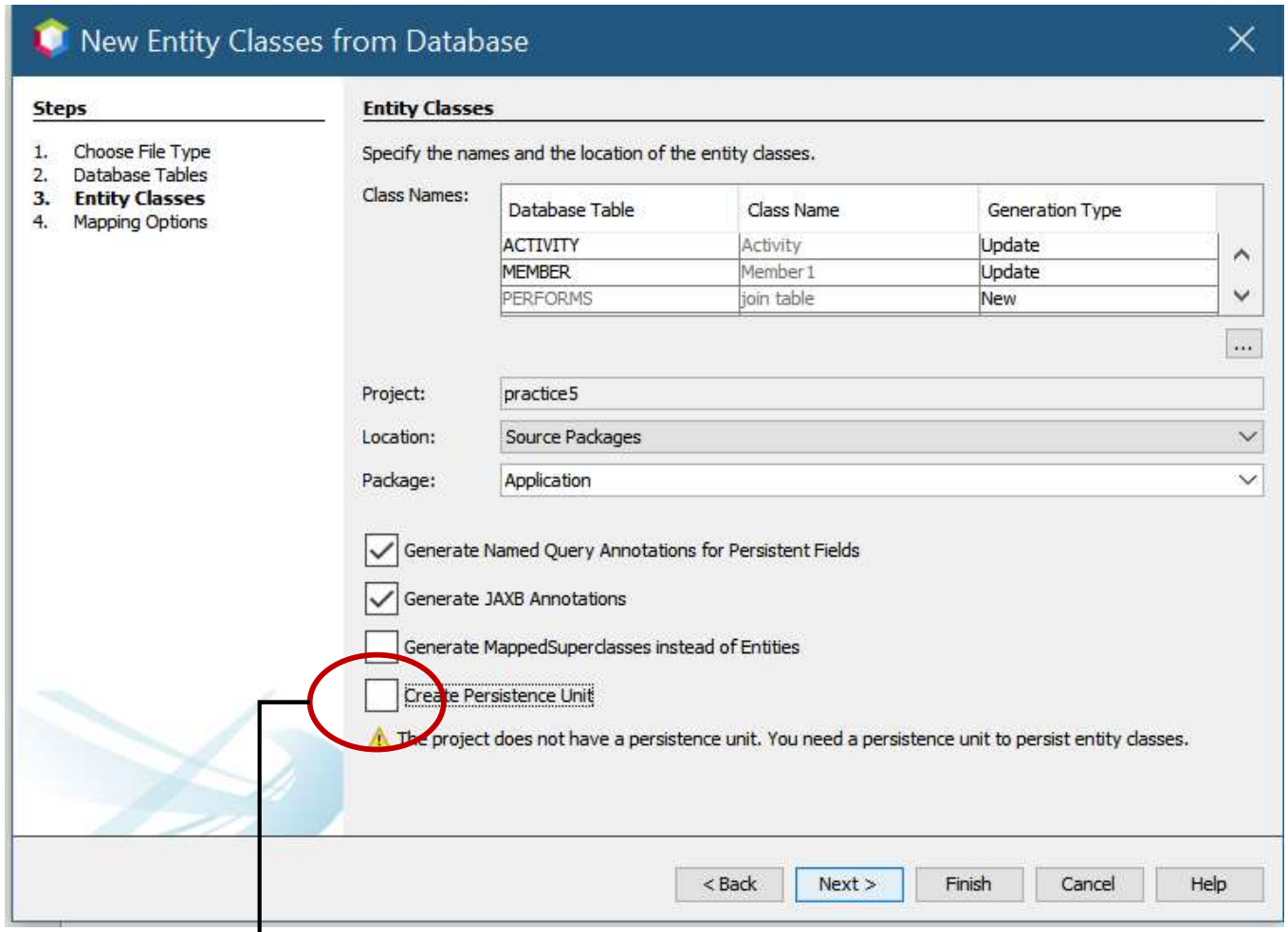
IMPORTANT

- JPA/Hibernate does not map the "intermediate" tables that arise from "many to many" relationships



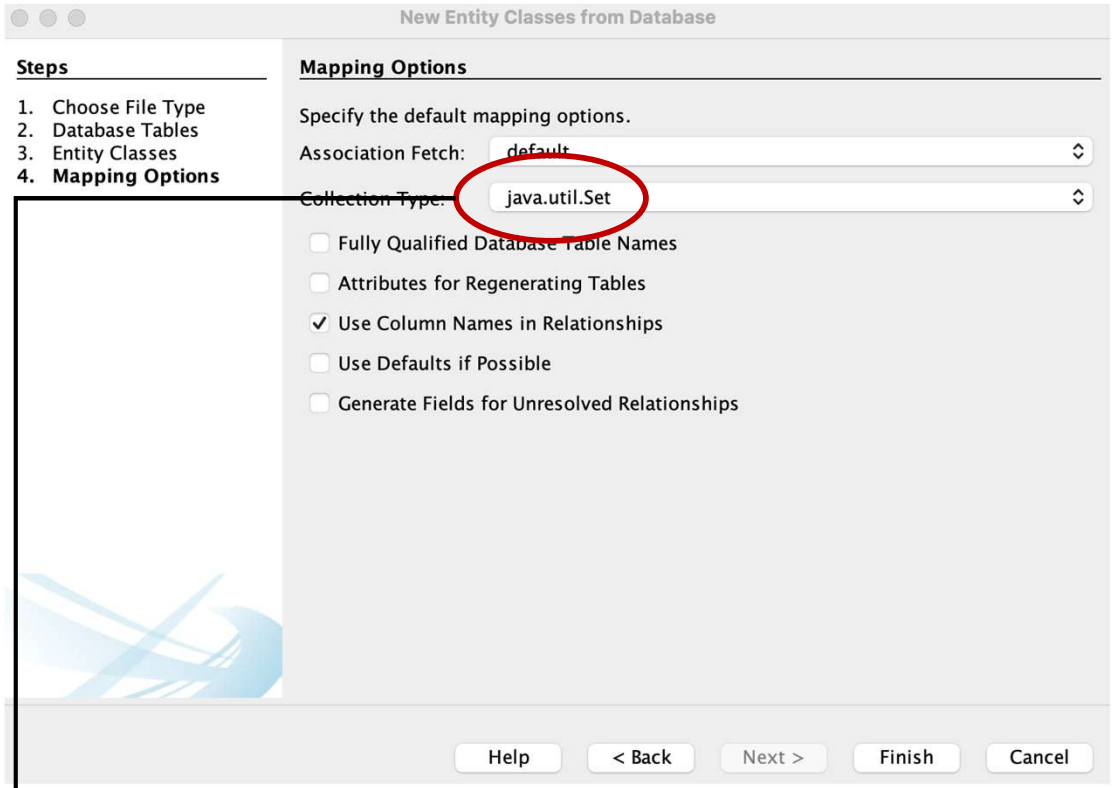


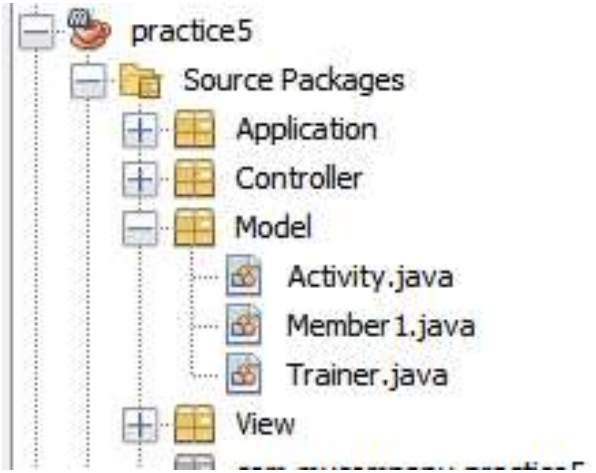
Creating classes derived from database tables



Unclick "Create Persistence Util"

Creating classes derived from database tables





Select Set as the data type to store attribute collections

```
@Entity
@Table(name = "TRAINER")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Trainer.findAll", query = "SELECT t FROM Trainer t"),
    @NamedQuery(name = "Trainer.findByTCod", query = "SELECT t FROM Trainer t WHERE t.tCod = :tCod"),
    @NamedQuery(name = "Trainer.findByTName", query = "SELECT t FROM Trainer t WHERE t.tName = :tName"),
    @NamedQuery(name = "Trainer.findByTSurname1", query = "SELECT t FROM Trainer t WHERE t.tSurname1 = :tSurname1"),
    @NamedQuery(name = "Trainer.findByTSurname2", query = "SELECT t FROM Trainer t WHERE t.tSurname2 = :tSurname2"),
    @NamedQuery(name = "Trainer.findByTIdnumber", query = "SELECT t FROM Trainer t WHERE t.tIdnumber = :tIdnumber"),
    @NamedQuery(name = "Trainer.findByTPhonenumber", query = "SELECT t FROM Trainer t WHERE t.tPhonenumber = :tPhonenumbe
    @NamedQuery(name = "Trainer.findByTEmail", query = "SELECT t FROM Trainer t WHERE t.tEmail = :tEmail"),
    @NamedQuery(name = "Trainer.findByTDate", query = "SELECT t FROM Trainer t WHERE t.tDate = :tDate"),
    @NamedQuery(name = "Trainer.findByTNick", query = "SELECT t FROM Trainer t WHERE t.tNick = :tNick")})
public class Trainer implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "T_COD")
    private String tCod;
    @Basic(optional = false)
    @Column(name = "T_NAME")
    private String tName;
    @Basic(optional = false)
    @Column(name = "T_SURNAME1")
    private String tSurname1;
    @Column(name = "T_SURNAME2")
    private String tSurname2;
    @Basic(optional = false)
    @Column(name = "T_IDNUMBER")
    private String tIdnumber;
    @Column(name = "T_PHONENUMBER")
    private String tPhonenumber;
    @Column(name = "T_EMAIL")
    private String tEmail;
    @Column(name = "T_DATE")
    @Temporal(TemporalType.TIMESTAMP)
    private Date tDate;
    @Column(name = "T_NICK")
    private String tNick;
    @OneToMany(mappedBy = "aTrainerincharge")
    private Set<Activity> activitySet = new HashSet<Activity>();
;
```

It is necessary to initialize this structure and, maybe - for clarity, give it a more significant name

```
@OneToMany(mappedBy = " aTrainerincharge)
private Set<Activity> activitySet = new HashSet<Activity>();
```

```
public Trainer() { }
    public Trainer(String tCod) {  this.tCod = tCod;  }

    public Trainer(String tCod, String tName, String tSurname1, String tIdnumber) {
        this.tCod = tCod;
        this.tName = tName;
        this.tSurname1 = tSurname1;
        this.tIdnumber = tIdnumber;
    }

    public String getTCod() {     return tCod;   }
```

```
    public Set<Activity> getActivitySet() {
        return activitySet;
    }

    public void setActivitySet(Set<Activity> activitySet) {
        this.activitySet = activitySet;
    }
```

```
    @Override
    // other methods
}
```

Change this attributes names in case if you did it before

You can verify that the tool has automatically generated 3 constructors for the Trainer class:

public Trainer ()
public Trainer (String tCod)
public Trainer (String tCod, String tName, String tSurname1, String tIdnumber)

However, it has not generated the constructor with all the attributes (since some are not mandatory), so we will have to add it manually

- Class Activity (1/2)

```
@Entity
@Table(name = "ACTIVITY")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Activity.findAll", query = "SELECT a FROM Activity a"),
    @NamedQuery(name = "Activity.findByAId", query = "SELECT a FROM Activity a WHERE a.aId = :aId"),
    @NamedQuery(name = "Activity.findByAName", query = "SELECT a FROM Activity a WHERE a.aName = :aName"),
    @NamedQuery(name = "Activity.findByADescription", query = "SELECT a FROM Activity a WHERE a.aDescription = :aDescription"),
    @NamedQuery(name = "Activity.findByAPrize", query = "SELECT a FROM Activity a WHERE a.aPrize = :aPrize")})
public class Activity implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "A_ID")
    private String aId;
    @Basic(optional = false)
    @Column(name = "A_NAME")
    private String aName;
    @Column(name = "A_DESCRIPTION")
    private String aDescription;
    @Column(name = "A_PRIZE")
    private Short aPrize;
    @JoinTable(name = "PERFORMS", joinColumns = {
        @JoinColumn(name = "P_ID", referencedColumnName = "A_ID")}, inverseJoinColumns = {
        @JoinColumn(name = "P_NUM", referencedColumnName = "M_NUM")})
    @ManyToMany
    private Set<Member1> member1Set = new HashSet<Member1>();
    @JoinColumn(name = "A_TRAINERINCHARGE", referencedColumnName = "T_COD")
    @ManyToOne
    private Trainer aTrainerincharge;
```

- Class Activity (2/2)

```
public Activity() {
}

public Activity(String aId) {
    this.aId = aId;
}

public Activity(String aId, String aName) {
    this.aId = aId;
    this.aName = aName;
}

public String getAId() {
    return aId;
}

public void setAId(String aId) {
    this.aId = aId;
}

public Trainer getATrainerincharge() {
    return aTrainerincharge;
}

public void setATrainerincharge(Trainer aTrainerincharge) {
    this.aTrainerincharge = aTrainerincharge;
}


public Set<Member1> getMember1Set() {
    return member1Set;
}

public void setMember1Set(Set<Member1> member1Set) {
    this.member1Set = member1Set;
}
```

Migrating your project from JDBC to Hibernate

Login Controller

- Must have a session class attribute which will be for connecting to the database through hibernate
- The connect() method need to be changed adding:

```
Session session = HibernateUtil.getSessionFactory().openSession();
```

- so our connection will be done by Hibernate using username, password and other parameters stablished in the configuration file.

```
public class Controller implements ActionListener {
    private Session s = null;
    private MessageView vMessage = null;
    private MainWindow vMainView = null;
    private ViewMember vMember = null;
    private Logger logger = null;
    private UtilTables utilTables = null;

    public Controller() {
```

Migrating your project from JDBC to Hibernate

Login Controller

- if the connection is successful an object of type controller is created and the connection is passed by parameter (now it is session)

```
@Override
  public void actionPerformed(ActionEvent ae) {
     switch (ae.getActionCommand()){
        case "Connect" :

                    session = connect();
                    if (connectionOK) {

                       vLogin.dispose();
                       Controller controller = new Controller(sesion);

                    }
                    break;
         case "Exit" :
                    vLogin.dispose();
                    System.exit(0);
                    break;
     }

  }
```

CRUD operations with Hibernate

Queries

- To perform queries you can use the HQL language (Hibernate Query Language) or perform the query directly in SQL language ("native")
- One of the advantages of using native queries is that migrating between JDBC and JPA/Hibernate is easier
- as far as possible, we will maintain the syntax of the method specification in the DAO and only modify the implementation

CRUD operations with Hibernate

Queries

- To perform queries you can use the HQL language (Hibernate Query Language) or perform the query directly in SQL language ("native")
- One of the advantages of using native queries is that migrating between JDBC and JPA/Hibernate is easier

Before

```
public ArrayList<Member> listAllMembers() throws SQLException {
    ArrayList listMembers = new ArrayList();

    String sql = "SELECT * FROM MEMBER";
    ps = connectionDB.getConnection().prepareStatement(sql);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        Member member = new Member(rs.getString(1), rs.getString(2),
                rs.getString(3), rs.getString(4), rs.getString(5),
                rs.getString(6), rs.getString(7),rs.getString(8));

        listMembers.add(member);
    }

    return listMembers;
}
```

Now

```
public ArrayList<Member1> listAllMembers() throws Exception {
    Transaction transaction = session.beginTransaction();

    ArrayList<Member1> lMembers = (ArrayList<Member1>) session.createQuery("FROM Member1 T").list();
    //ArrayList<Member1> lMembers = (ArrayList<Member1>)session.createNativeQuery ("SELECT * FROM Member M", Member1.class).list();
    transaction.commit();
    return lMembers;
}
```

Operaciones CRUD con Hibernate

Queries

- Queries with parameters that involve JOIN or return specific fields, we will use native queries

```
public ArrayList<Object[]> listIDandNamesTrainers() throws Exception {
  Transaction transaction = session.beginTransaction();
  Query query = session.createNativeQuery("SELECT m_num, m_name FROM Member M");
  ArrayList<Object[]> members = (ArrayList<Object[]>) query.list();

  transaction.commit();
  return members;
}
public ArrayList<String> listNamesTrainers() throws Exception {
  Transaction transaction = session.beginTransaction();
  Query query = session.createNativeQuery("SELECT m_name FROM Member M");
  ArrayList<String> members = (ArrayList<String>) query.list();

  transaction.commit();
  return members;
}
```

Query to return two fields from MEMBER

Query that returns a single column from MEMBER

Operaciones CRUD con Hibernate

Queries

```
public ArrayList<Member1> listTrainersBy(String letter) throws Exception {
  Transaction transaction = session.beginTransaction();
  letter = letter + "%";
  Query query = session.createNativeQuery("SELECT * FROM "
          + "MEMBER M "
           + "WHERE M_name LIKE :letter", Member1.class).setParameter("letter", letter);
  ArrayList<Member1> members = (ArrayList<Member1>) query.list();

  transaction.commit();
  return members;
}
```

CRUD operations with Hibernate

Insert

- To insert a new row it is necessary to create the class beforehand. We will use the method save()

```
public void insertMember(Member1 member) throws Exception {
    Transaction transaction = session.beginTransaction();
    session.save(member);
    transaction.commit();
}
```

Delete

- To delete a tuple, the delete() method is used. the get() method retrieves an object using its primary key

```
public void deletetMember(String codMember) throws Exception {
    Transaction transaction = session.beginTransaction();
    Member1 member = session.get(Member1.class, codMember);
    session.delete(member);
    transaction.commit();

}
```

Operaciones CRUD con Hibernate

Update

- Updating is done in the same way as inserting a new one. Hibernate decides what to do (insert or update) based on the value of the primary key of the object being sent

```
public void actualizaMonitor(Monitor monitor) throws Exception {
      Transaction transaction = sesion.beginTransaction();
      sesion.save(monitor);
      transaction.commit();
   }
```

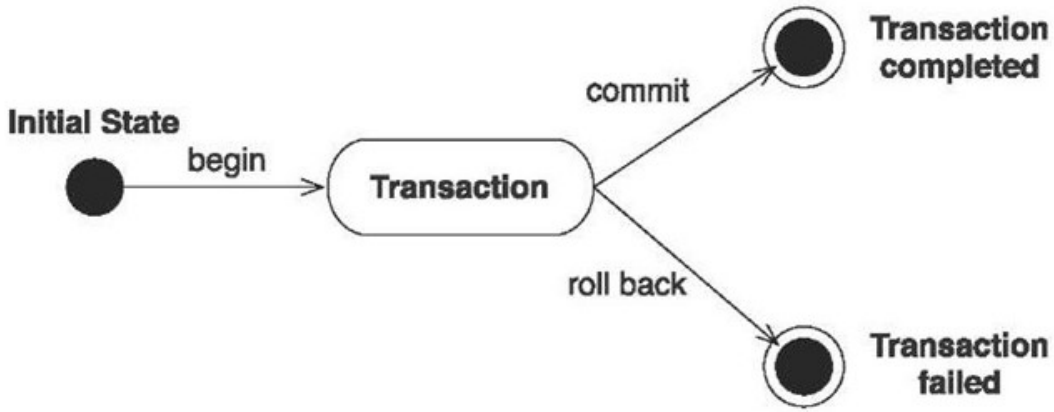
The only difference with the insertion is that now the object monitor that is passed through the parameter has been obtained, previously, with the get() method of hibernate.

Hibernate Transactions Interface

A **Transaction** is a sequence of operation which works as an atomic unit. A transaction only completes if all the operations completed successfully. A transaction has the Atomicity, Consistency, Isolation, and Durability properties (ACID).

So, a transaction is a unit of work in which all the operations must be executed or none of them.

- **Atomicity**: Is defined as either all operations can be done or all operation can be undone
- **Consistency**: After a transaction is completed successfully, the data in the datastore should be a reliable data. This reliable data is also called as consistent data
- **Isolation**: If two transactions are going on the same data then one transaction will not disturb the other transaction
- **Durability**: After a transaction is completed, the data in the datastore will be permanent until another transaction is going to be performed on that data



Hibernate Transactions Interface

In Hibernate framework, we have Transaction interface that defines the unit of work. It maintains the abstraction from the transaction implementation (JTA, JDBC).

A Transaction is associated with Hibernate Session and instantiated by calling the sessionObj.beginTransaction(). The methods of Transaction interface are as follows:

Name	Description	Syntax
begin()	It starts a new transaction.	`public void begin() throws HibernateException`
commit()	It ends the transaction and flushes the associated session.	`public void rollback() throws HibernateException`
rollback()	It rolls back the current transaction.	`public void rollback()throws HibernateException`
setTimeout(int seconds)	It set the transaction timeout for any transaction started by a subsequent call to begin() on this instance.	`public void setTimeout(int seconds) throws HibernateException`
isActive()	It checks if this transaction is still active or not.	`public boolean isActive()throws HibernateException`
wasRolledBack()	It checks if this transaction roll backed successfully or not.	`public boolean wasRolledBack()throws HibernateException`
wasCommitted()	It checks if this transaction committed successfully or not.	`public boolean wasCommitted()throws HibernateException`
registerSynchronization(Synchronization synchronization)	It registers a user synchronization callback for this transaction.	`public boolean registerSynchronization(Synchronization synchronization)throws HibernateException`

Hibernate Transaction Management Basic Structure

This is the basic structure that Hibernate programs should have, concerning Transaction handling

```
01  Transaction transObj = null;
02  Session sessionObj = null;
03  try {
04      sessionObj = HibernateUtil.buildSessionFactory().openSession();
05      transObj = sessionObj.beginTransaction();
06
07      //Perform Some Operation Here
08      transObj.commit();
09  } catch (HibernateException exObj) {
10      if(transObj!=null){
11          transObj.rollback();
12      }
13      exObj.printStackTrace();
14  } finally {
15      sessionObj.close();
16  }
```

Transactions and Concurrency

To use transactions in Hibernate, you can configure Hibernate transactions as JDBCTransaction or JTATransaction. The lifecycle of these two transactions is different. You can specify which transaction is used **in hibernate.cfg.xml**. (If not configured, Hibernate will also use JDBC transactions by default).

```
<session-factory>
……
<property name="hibernate.transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory
</property>
……
</session-factory>
```

Concurrency Problems

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in database environment. The five concurrency problems that can occur in database are:

(i). Temporary Update Problem
(ii). Incorrect Summary Problem
(iii). Lost Update Problem
(iv). Unrepeatable Read Problem
(v). Phantom Read Problem

Transaction isolation level:

To resolve issues caused by multiple transactions at the same time. The database system provides four levels of transaction isolation for users to choose from.

READ UNCOMMITTED
READ COMMITTED (protecting against dirty reads)
REPEATABLE READ (protecting against dirty and non-repeatable reads)
SERIALIZABLE (protecting against dirty, non-repeatable reads and phantom reads)

ISOLATION LEVEL

Each level of isolation corresponds to a positive integer.

Read Uncommitted: 1
Read Committed: 2
Repeatable Read: 4
Serializable: 8

```
<property name="hibernate.transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
<! - Isolation level ->
<property name="hibernate.connection.isolation">2</property>
```