HIBERNATE

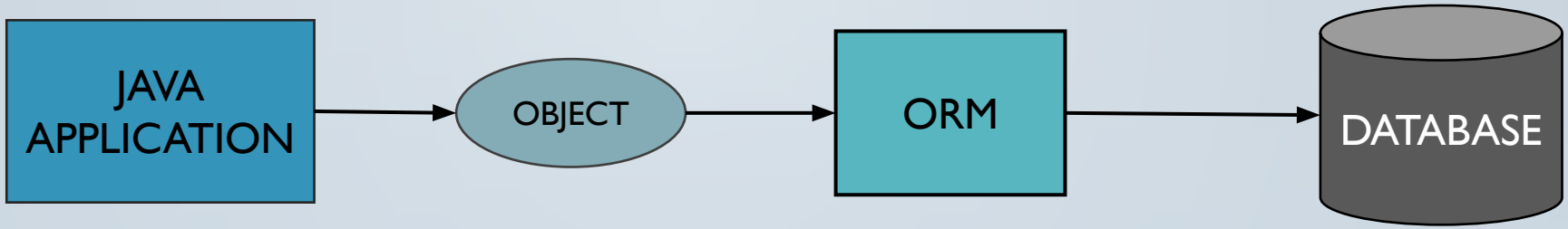
WITH JPA

ADVANTAGES OF HIBERNATE

- Open Source and Lightweight
- Fast Performance
- Database Independent Query
- Automatic Table Creation
- Simplifies Complex Join
- Provides Query Statistics and Database Status

ORM TOOL

- An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object as one row in the table in the database.



DIFFERENT ORM TOOLS

- Hibernate.
- TopLink.
- EclipseLink.
- OpenJPA.
- MyBatis (formally known as iBatis).

HIBERNATE

- Hibernate is a Java framework that simplifies the development of Java applicati
- on to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

JPA

- Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

ENTITY MANAGER FACTORY

- EntityManagerFactory provides instances of EntityManager for connecting to same database. All the instances are configured to use the same setting as defined by the default implementation. Several entity manager factories can be prepared for connecting to different data base.
- The **EntityManagerFactory** interface present in **java.persistence** package is used to provide an entity manager.
- **Persistence -** The Persistence is a bootstrap class which is used to obtain an EntityManagerFactory interface.

Syntax :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(“String”);//string = persistence unit name.
```

ENTITY MANAGER

- JPA EntityManager is used to access a database in a particular application. It is used to manage persistent entity instances, to find entities by their primary key identity, and to query over all entities.
- IMPORTANT METHODS :
- persist – Make an instance managed and persistent.
- merge – Merge the state of the given entity into the current persistence context.
- remove – Remove the entity instance.
- find – Find by primary key. Search for an entity of the specified class and primary key. If the entity instance is contained in the persistence context, it is returned from there.

Syntax :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(“String”);//string = persistence unit name

EntityManager em1 = EntityManagerFactory.createEntityManager();

EntityManager em2 = EntityManagerFactory.createEntityManager();

EntityManager em3 = EntityManagerFactory.createEntityManager();
```

NOTE : we can create multiple entity managers for one entity manager factory.

ENTITY TRANSACTION

- Interface used to control transactions on resource-local entity managers.
- IMPORTANT METHODS :
- **begin() method -** This method is used to start the transaction.
- **commit() method -** This method is used to commit the transaction.

Syntax :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("String");//string = persistence unit name

EntityManager em1 = EntityManagerFactory.createEntityManager();

EntityTransaction et = EntityManager.getTransaction();
```

DIFFERENCE BETWEEN PERSIST AND MERGE

- Persist should be called only on new entities.
- It will persist the entity object in database.
- If you pass the object with duplicate primary key it will throw exception.

- merge is meant to reattach detached entities.
- It will update the object in the database for duplicate key.
- If the primary key is not matched it will insert the object as new record in table.

MAPPING IN HIBERNATE

- The ***mapping*** of associations between entity classes and the relationships between tables is the soul of ORM.
- hibernate mappings are one of the key features of hibernate. they establish the relationship between two database tables as attributes in your model. that allows you to easily navigate the associations in your model and criteria queries.
- you can establish either unidirectional or bidirectional i.e you can either model them as an attribute on only one of the associated entities or on both. it will not impact your database mapping tables, but it defines in which direction you can use the relationship in your model and criteria queries.

DIFFERENT MAPPING ANNOTATIONS ARE :

- @OneToOne(uni-direction)
- @OneToMany(uni-direction)
- @ManyToOne(uni-direction)
- @ManyToMany(uni-direction)
- @OneToOne(bi-direction)
- @OneToMany(bi-direction)
- @ManyToMany(bi-direction)

ONE-TO-ONE MAPPING(UNI-DIRECTION)

- One to one represents that a single entity is associated with a single instance of the other entity. An instance of a source entity can be at most mapped to one instance of the target entity.
- In this type of mapping one entity has a property or a column that references to a property or a column in the target entity.
- Here you can retrive the data in one direction only i.e if you have person id you can get pan details also but you cannot get person details with pan id.

Ex : One person has one pan, a pan is associated with a single person

```java
@Entity
public class Pan {

@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String address;
private String panNumber;
```

```java
@Entity
public class Person {

@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String email;
private long phone;
@OneToOne
private Pan pan;
```

ONE-TO-ONE MAPPING(BI-DIRECTION)

- Bidirectional association allows us to fetch details of dependent object from both side. In such case, we have the reference of two classes in each other. Let's take an example of same Person and Pan, now Person class has-a reference of Pan and Pan has a reference of Person.
- Here you can retrive the data in bi-direction .If you have person id you can get pan details and also you can get person details with pan id.

Ex : One person has one pan, a pan is associated with a single person.

```java
@Entity
public class Pan {

@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String address;
private String panNumber;
@OneToOne
private Person person;
```

```java
@Entity
public class Person {

@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String email;
private long phone;
@OneToOne
private Pan pan;
```

ONE-TO-MANY MAPPING(UNI-DIRECTION)

- In one-to-many mapping one entity is associated with many instances of other entity. for example one person has many bank accounts.
- Here you can retrive the data in uni-direction .If you have person id you can get all the account details and you cannot get person details with account id.

Ex : One person has many accounts, many accounts are associated with a single person.

```
@Entity
public class account{

@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private long accNo;
private String ifscCode;

private Person person;
```

```
@Entity
public class Person {

@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String email;
private long phone;
@OneToMany
private List<Account> accounts;
```

MANY-TO-ONE MAPPING(UNI-DIRECTION)

- In many-to-one mapping many instances of entity is associated with one instance of other entity. for example many branches have one hospital.
- Here you can retrive the data in uni-direction .If you have branch id you can get the hospital details and you cannot get branch details with hospital id.

Ex : Many branches are associated with one hospital.

```
@Entity
public class Branch {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String state;
private String country;
private long phone;
@ManyToOne
private Hospital hospital;
```

```
@Entity
public class Hospital {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String web;
```

MANY-TO-ONE OR ONE-TO-MANY(BI-DIRECTION)

- In bi-direction, in many-to-one mapping many instances of entity is associated with one instance of other entity. for example many branches have one hospital . In one-to-many mapping one instance of entity is associated with many instance of other entity. for example one hospital have many branches.
- Here you can retrive the data in bi-direction .If you have branch id you can get the hospital details and you can get branch details with hospital id.

Ex : Many branches are associated with one hospital.

```
@Entity
public class Branch {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String state;
private String country;
private long phone;
@ManyToOne
private Hospital hospital;
```

```
@Entity
public class Hospital {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String web;
@OneToMany
private List<Branch> branches;
```

MANY-TO-MANY MAPPING (UNI-DIRECTION)

- In many-to-many mapping many instances of entity is associated with many instances of other entity. for example many students have many courses.
- Here you can retrive the data in uni-direction .If you have student id you can get the all the details of courses but you cannot get student details with course id.

Ex : Many students are associated with many courses.

```
@Entity
public class Student {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private int age;
@ManyToMany
private List<Course> courses;
```

```
@Entity
public class Course {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String duration;
```

MANY-TO-MANY MAPPING (BI-DIRECTION)

- In many-to-many mapping many instances of entity is associated with many instances of other entity. for example many students have many courses and many courses have many students.
- Here you can retrive the data in bi-direction .If you have student id you can get the all the details of courses and you can get all student details with course id.

Ex : Many students are associated with many courses.

```
@Entity
public class Student {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private int age;
@ManyToMany
private List<Course> courses;
```

```
@Entity
public class Course {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private String duration;
@ManyToMany
private List<student> students;
```

JPQL

- *Java Persistence Query Language (JPQL)* is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification. JPQL is used to make queries against entities stored in a relational database. JPQL is inspired by SQL.
- JPQL is object-oriented. In JPQL we work with entities and collection of entities, while in SQL we work with columns and rows.

TYPES OF QUERY PARAMETERS

- Similar to JDBC prepared statement parameters, JPA specifies two different ways to write parameterized queries by using:
  - Positional parameters
  - Named parameters

MAPPEDBY AND @JOINCOLUMN

- JPA Relationships can be either unidirectional or bidirectional. It simply means we can model them as an attribute on exactly one of the associated entities or both.
- Defining the direction of the relationship between entities has no impact on the database mapping. It only defines the directions in which we use that relationship in our domain model.
- For a bidirectional relationship, we usually define:
  - the owning side
  - inverse or the referencing side
- The *@JoinColumn* annotation helps us specify the column we'll use for joining an entity association or element collection. On the other hand, the *mappedBy* attribute is used to define the referencing side (non-owning side) of the relationship.

BEFORE USING @JOINCOLUMN AND MAPPEDBY

```java
@Entity
public class Charcy {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String cNo;
private String type;
@OneToOne
private Vehicle vehicle;
```

```java
@Entity
public class Vehicle {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private double cost;
@OneToOne
private Charcy charcy;
```

TABLES IN DATABASE

charcy

id	cNo	type	Vehicle_id

vehicle

id	name	cost	charcy_id

AFTER USING @JOINCOLUMN AND MAPPEDBY

```java
@Entity
public class Charcy {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String cNo;
private String type;
@OneToOne(mappedBy="charcy")
private Vehicle vehicle;
```

```java
@Entity
public class Vehicle {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
private double cost;
@OneToOne
@JoinColumn
private Charcy charcy;
```

TABLES IN DATABASE

charcy

id	cNo	type

vehicle

id	name	cost	charcy_id

@JOINTABLE ANNOTATION

- When we use mapping in hibernate to build relationship between two entites , sometimes duplicate tables get created.
- To avoid this duplication in tables we use @JoinTable.

Ex :

```
@ManyToMany @JoinTable(joinColumns = @JoinColumn, inverseJoinColumns = @JoinColumn)
```

BEFORE USING @JOINTABLE

```java
@Entity
public class Cab {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String driverName;
private double cost;
@ManyToMany
private List<Person> persons;
```

```java
@Entity
public class Person {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
@ManyToMany
private List<Cab> cabs;
```

TABLES IN DATABASE

cab

id	cost	driverName

person

id	name

cab_person

cab_id	person_id

person_cab

person_id	cab_id

AFTER USING @JOINTABLE

```java
@Entity
public class Cab {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String driverName;
private double cost;
@ManyToMany(mappedBy = "cabs")
private List<Person> persons;
```

```java
@Entity
public class Person {
@Id
@GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;
private String name;
@ManyToMany
@JoinTable(joinColumns = @JoinColumn
, inverseJoinColumns = @JoinColumn)
private List<Cab> cabs;
```

TABLES IN DATABASE

cab

id	cost	driverName

person

id	name

person_cab

persons_id	cabs_id

CASCADING IN HIBERNATE

- Cascading is a feature in Hibernate, which is **used to manage the state of the mapped entity whenever the state of its relationship owner (superclass) affected**. When the relationship owner (superclass) is saved/ deleted, then the mapped entity associated with it should also be saved/ deleted automatically.
- **When we perform some action on the target entity, the same action will be applied to the associated entity.**

TYPES OF CASCADING

- All JPA-specific cascade operations are represented by the javax.persistence.CascadeType enum containing entries:
  - ALL
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
- DETACH

FETCHTYPE

- When working with an ORM, data fetching/loading can be classified into two types: eager and lazy.
- In this quick tutorial, we are going to point out differences and show how we can use these in Hibernate.
- Two types :
- 1.Lazy
- 2.Eager

DIFFERENT FETCHTYPES

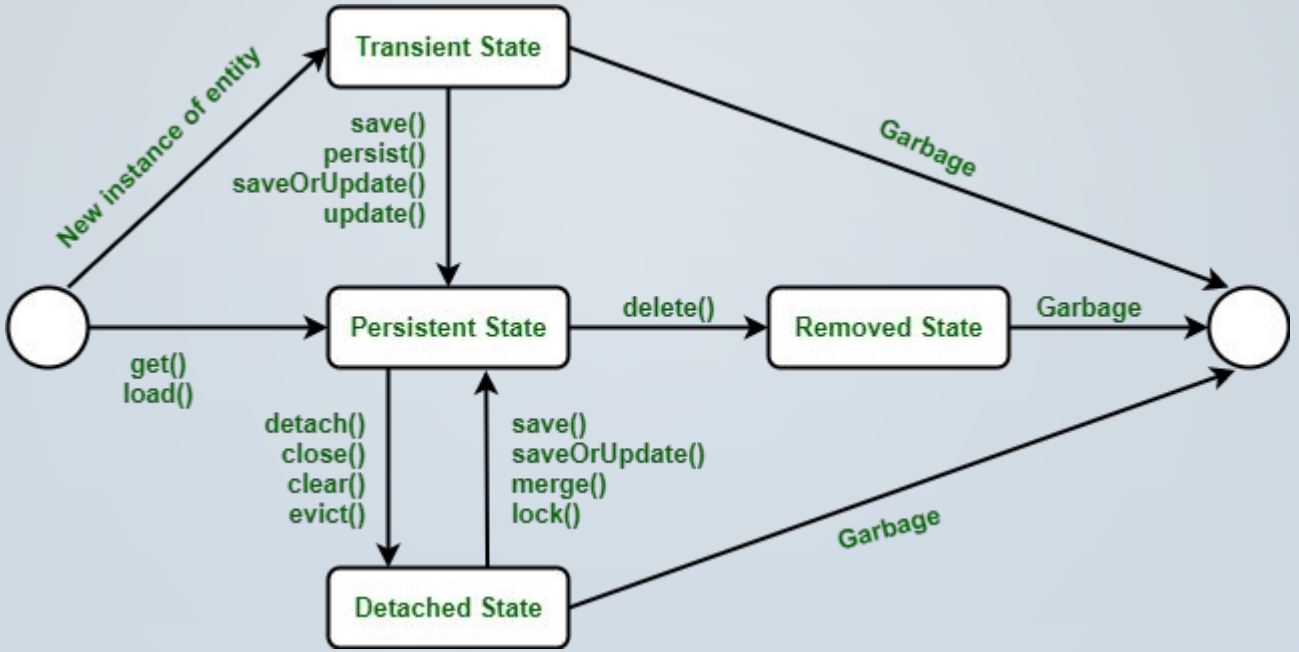
- **LAZY :**
- The *FetchType.LAZY* tells Hibernate to only fetch the related entities from the database when you use the relationship. This is a good idea in general because there's no reason to select entities you don't need for your uses case.

- **EAGER :**
- The *FetchType.EAGER* tells Hibernate to get all elements of a relationship when selecting the root entity.

DEFAULT FETCH TYPES FOR MAPPING ANNOTATIONS

- @OneToOne – EAGER
- @ManyToOne – EAGER
- @OneToMany – LAZY
- @ManyToMany – LAZY

ENTITY LIFE CYCLE IN HIBERNATE



TRANSIENT STATE

- The transient state is the initial state of an object.
- Once we create an instance of POJO class, then the object entered in the transient state.
- Here, an object is not associated with the Session. So, the transient state is not related to any database.
- Hence, modifications in the data don't affect any changes in the database.
- The transient objects exist in the heap memory. They are independent of Hibernate.

PERSISTENT STATE

- soon as the object associated with the Session, it entered in the persistent state.
- Hence, we can say that an object is in the persistence state when we save or persist it.
- Here, each object represents the row of the database table.
- So, modifications in the data make changes in the database.

DETACHED STATE

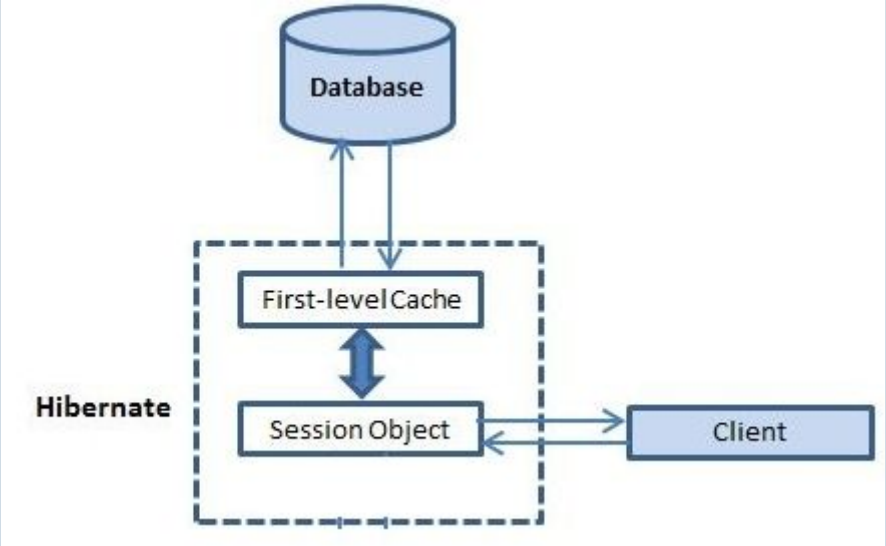
- Once we either close the session or clear its cache, then the object entered into the detached state.
- As an object is no more associated with the Session, modifications in the data don't affect any changes in the database.
- However, the detached object still has a representation in the database.
- If we want to persist the changes made to a detached object, it is required to reattach the application to a valid Hibernate session.
- To associate the detached object with the new hibernate session, use any of these methods - load(), merge(), refresh(), update() or save() on a new session with the reference of the detached object.

CACHING IN HIBERNATE

- Hibernate caching improves the performance of the application by pooling the object in the cache. It is useful when we have to fetch the same data multiple times.
- There are mainly two types of caching:
  - First Level Cache, and
  - Second Level Cache

FIRST-LEVEL CACHE

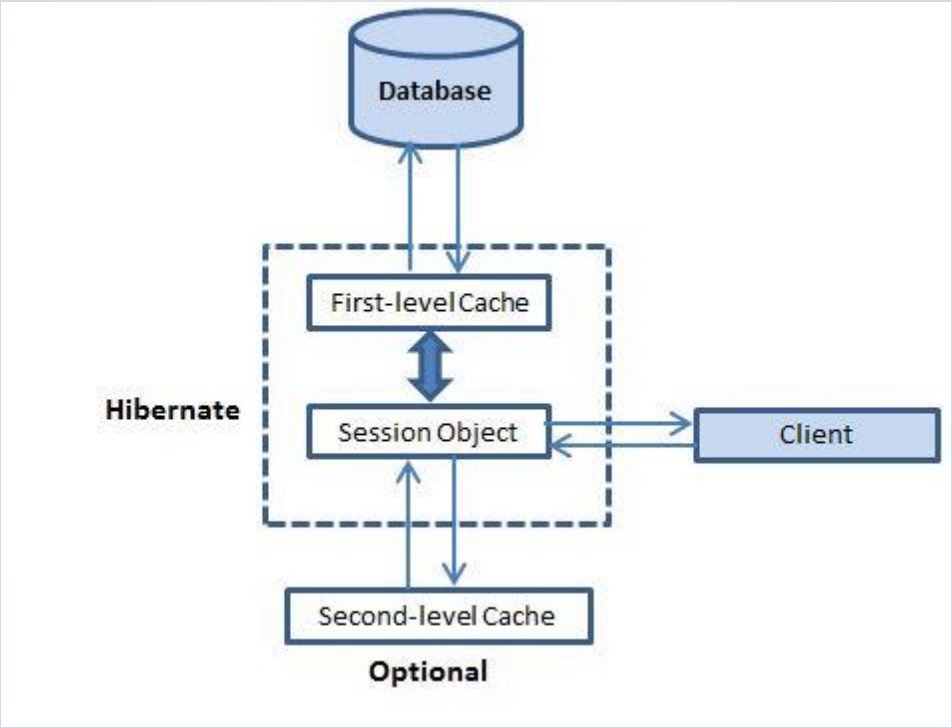
- Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.



SECOND-LEVEL CACHE

- SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitely.
- Three steps to enable second level cache :
- 1.Add hibernate-echache dependency . (version should be same as hibernate version)
- 2.Add @cacheble annotation to Entity class.
- 3.Enable second-level cache by configuring in persistence.xml file.

SECOND-LEVEL CACHE



COMPOSITE KEY IN HIBERNATE

- A composite primary key, also called a composite key, is a combination of two or more columns to form a primary key for a table.
- In JPA, we can define the composite keys by : the @Embaddable and @EmbeddedId annotations.
- **In order to define the composite primary keys, we should follow some rules:**
  - The composite primary key class must be public.
  - It must have a no-arg constructor.
  - It must define the equals() and hashCode() methods.
  - It must be Serializable.

```java
import java.io.Serializable;

import javax.persistence.Embeddable;

@Embeddable
public class UserId implements
Serializable {
private String email;
private long phone;
```

```java
import
javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class User {

@EmbeddedId
private UserId userId;
private String name;
private int age;
private String gender;
private String password;
```

PERSISTENT UNIT IN HIBERNATE

- A persistence unit defines a set of all entity classes that are managed by entity manager instance in an application.
- It contains configurations file which is required for connection.
- It consists of url , username , password.
- It is used to perform actions on database.

```
<persistence-unit name="uday">
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
<properties>
<property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3307/many_to_many_bi" />
<property name="javax.persistence.jdbc.user"
value="root" />
<property name="javax.persistence.jdbc.password"
value="root" />
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.hbm2ddl.auto" value="update" />
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
</properties>
</persistence-unit>
```

PERSISTENCE.XML FILE

- It is a standard configuration file.
- It has configurations for the Object-Relation mapping.
- It can have multiple persistence units to connect to different database.
- Persistence unit name must be unique.
- This persistence.xml file should be created in src/main/resources . create folder with name META-INF and create persistence.xml file in it.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
version="2.1">
<persistence-unit name="vikas">
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
<properties>
<property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3307/many_to_many_bi" />
<property name="javax.persistence.jdbc.user"
value="root" />
<property name="javax.persistence.jdbc.password"
value="root" />
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.hbm2ddl.auto" value="update" />
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
</properties>
</persistence-unit>
</persistence>
```