# Hibernate Framework

🡺Hibernate is a java framework that simplifies the development of java application to interact with the database.

🡺Hibernate is ORM(Object Relational Mapping)

🡺Hibernate is an Object Source lightweight.

🡺Hibernate is a non-invasive framework , means it won’t forces the programmers to extend/implements any class interface.

🡺It is invented by Gavin King in 2001.

🡺Any type of application can build with hibernate framework.

**Working of Hibernate:-**

In case of JDBC , we need to write all the query to store the data inside the table. First we create the object of the class and feed all the element inside the object and then we fetch the elements from the object in DAO layer of the application to feed the data inside the table.

But in case of hibernate , we create the object and feed all the element inside the object and then we call the hibernate , what hibernate will do is it will automatically feed the data from the object to the database we don’t need to write the query for it . It automatically takes the object and feed the data in the database.

Now the question arises how hibernate will knows in which table it need to feed the data, actually what we do we map the each class with the table and map each field with the particular column using annotation. Hibernate will know through the maping that in which column, the data of the field will be feeded so whenever we create the object of the class it automatically feed the data using maping.

Behind the scene hibernate work on JDBC only it will create query behind the scene we only need to map the class with table and field with the column.

**Connecting hibernate with the database:-**

**🡺**First we need to create the xml file with file name **hibernate.cfg.xml** and we need to add DTD(Document Type of Definition) of hibernate in the xml. We need to download the DTD for the particular version in order to tell the xml file for which purpose we are using the xml. Simply go the Google and find the DTD of particular version and add it into the xml file.

**<!DOCTYPE hibernate-configuration PUBLIC**

**"-//Hibernate/Hibernate Configuration DTD 3.0//EN"**

**"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">**

**🡺** Now we need to add **<hibernate-configuration> </hibernate-configuration>** inside this we need to write **<session-factory></session-factory>** tag where we need to add property tag. Inside property tag we need to define the driver name , url, username,password to connect the hibernate with java file.

**<property name="hdm2ddl.auto">update</property>**

This property is used to create table automatically if table is not available , if table is available then it only update the table.

As we know that hibernate will fire the query automatically and if want to know the query then we will use this property to know which query is fired from the hibernate.

**<property name="show\_sql">true</property>**

**XML file will be like this:-**

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

**<!DOCTYPE hibernate-configuration SYSTEM "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">**

**<hibernate-configuration>**

**<session-factory>**

**<property name="connection.driver\_class">**

**com.mysql.jdbc.Driver**

**</property>**

**<property name="connection.url">jdbc:mysql://localhost:3306/myhiber</property>**

**<property name="connection.username">root</property>**

**<property name="connection.password">Gaurav45</property>**

**<property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>**

**<property name="hbm2ddl.auto">update</property>**

**<property name="show\_sql">true</property>**

**</session-factory>**

**</hibernate-configuration>**

Now let’s see how we are going to create a connection in hibernate :- First of all we will create the variable of **SessionFactory** Which is same like **Connection in JDBC** which provides us Prepare statement:-

🡺In Case of **SessionFactory** It will give us session in which we will create the table, insert data and all other thing. It is threadSafe interface which mean there is only one **SessionFactory**  in the entire project.

1st way

**SessionFactory factory = new Configuration().configure().buildSessionFactory();**

**2nd Way**

**Configuration cfg = new Configuration();**

**cfg.configure("hibernate.cfg.xml");**

**SessionFactory factory = cfg.buildSessionFactory();**

These are two way in which we can create the sessionFactory object. If Hibernate is not able to find the xml file at that we need to use 2nd way where we have specify the xml file name to give information about the xml file.

**What is an Entity in Hibernate?**

In the context of Hibernate, an Entity is an object that represents data stored in a database. Entities are a crucial element in modelling relational data in Java-based applications. Each Hibernate entity corresponds to a specific table in the database, and instances of these entities are mapped to records in that table.

# Creating an Entity Class

To make a class an entity in Hibernate, it must be annotated with @Entity. Additionally, it must have at least one field annotated with @Id, representing a unique identifier for the entity. Below is a simple example of an entity class in Hibernate:

**package com.tut;**

**import javax.persistence.Column;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.GenerationType;**

**import javax.persistence.Id;**

**import javax.persistence.Table;**

**@Entity**

**@Table(name="Student\_details")**

**public class Student {**

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

**@Column(name = "student\_id")**

**private Integer id;**

**@Override**

**public String toString() {**

**return "Student [id=" + id + ", name=" + name + ", city=" + city + "]";**

**}**

**private String name;**

**private String city;**

**public Student(Integer id, String name, String city) {**

**super();**

**this.id = id;**

**this.name = name;**

**this.city = city;**

**}**

**public Student() {**

**super();**

**}**

**public Integer getId() {**

**return id;**

**}**

**public void setId(Integer id) {**

**this.id = id;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public String getCity() {**

**return city;**

**}**

**public void setCity(String city) {**

**this.city = city;**

**}**

**}**

**Column Mapping**

The @Column annotation is used to map class fields to columns in the database table. It allows the customization of column names, data types, and other properties. In the above we have use column to change the name of the id with Student id.

Types of Identifier Generation Strategies

Hibernate offers several strategies for generating identifiers (primary keys) for entities. The choice of the appropriate strategy depends on the developer’s preferences and the characteristics of the database. Here’s a detailed description of some commonly used identifier generation strategies in Hibernate.

**1. GenerationType.IDENTITY**

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

**@Column(name = "student\_id")**

**private Integer id;**

The GenerationType.IDENTITY strategy relies on auto-increment mechanisms provided by the database. It means that the database automatically increments the primary key value with each new row. This strategy is often used for databases supporting auto-increment, such as MySQL or PostgreSQL.

**2.GeneratedType.Auto**

**@Id**

**@GeneratedValue(strategy = GenerationType.Auto)**

**@Column(name = "student\_id")**

**private Integer id;**

The GenerationType.AUTO strategy leaves the choice of identifier generation to Hibernate. Hibernate decides whether to use auto-increment, sequences or another method available in the specific database. This strategy provides the most flexibility but may lead to different behaviours on different database systems.

**Session Management in Hibernate**

Hibernate introduces the concept of a session, which represents a connection to the database. The session is essential for performing operations on data, such as saving, retrieving, updating, or deleting entities.

It’s important to note that changes made to entities, such as saving, updating, or deleting, are managed within the Hibernate session. These modifications are persisted in the database only upon the successful completion and commit of the transaction. The session serves as a unit of work, allowing developers to perform operations on entities in a controlled and effective manner, ensuring that changes are applied to the database in a coherent and transactional fashion.

Here, we’ll discuss how session management works in Hibernate and demonstrate some sample operations.

**Opening a Session**

To interact with the database, we need to open a session. The session is typically opened through a SessionFactory, which is responsible for creating sessions. Here's an example of how to open a session:-

**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 )**

**{**

**System.out.println( "projected started" );**

**Configuration cfg = new Configuration();**

**cfg.configure("hibernate.cfg.xml");**

**SessionFactory factory = cfg.buildSessionFactory();**

**//creating Student**

**Student st= new Student();**

**st.setCity("Ranchi");**

**st.setName("Gautam");**

**System.out.println(st);**

**Session session=factory.openSession();**

**Transaction tx=session.beginTransaction();**

**session.save(st);**

**tx.commit();**

**session.close();**

**}**

**}**

We need to map the class name with the configuration file so that hibernate will mark it as a table name:-

**<mapping class="com.tut.Student"/>**

**Commonly Used Hibernate Annotation:-**

**🡺@Entity-** Use to mark any class as Entity

🡺**@Table:** It specifies the table in the database with which the entity is mapped. Attributes like name, schema, and catalog can be used to customize the mapping.

🡺**@Id:** Marks a field as the primary key of the entity.

🡺**@GeneratedValue:** Specifies how the primary key should be generated, with strategies like AUTO, IDENTITY, SEQUENCE, or TABLE.

🡺**@Column:** Defines the column mapping for a field. Attributes such as name, length, nullable, and unique can be used for further customization.

🡺**@Transient:** Indicates that a field is not to be persisted in the database. This tell hibernate not to save this field.

🡺**@Temporal:** Used to fine-tune the mapping of java.util.Date and java.util.Calendar to SQL [types. @Temporal](mailto:types.@Temporal) over a date field tells hibernate the format in which the date needs to be saved.

🡺**@Lob:** Marks a field as a Large Object, mapping it to a BLOB or CLOB SQL type.

🡺**@OrderBy:** Specifies the ordering of elements in a collection.

**Let’s see how all the annotation work inside by code:-**

**package com.tut;**

**import java.util.Date;**

**import java.util.Arrays;**

**import javax.persistence.Column;**

**import javax.persistence.Entity;**

**import javax.persistence.GeneratedValue;**

**import javax.persistence.GenerationType;**

**import javax.persistence.Id;**

**import javax.persistence.Lob;**

**import javax.persistence.Table;**

**import javax.persistence.Temporal;**

**import javax.persistence.TemporalType;**

**import javax.persistence.Transient;**

**@Entity**

**@Table(name="student\_address")**

**public class Address {**

**@Id**

**@GeneratedValue(strategy=GenerationType.IDENTITY)**

**@Column(name="address\_id")**

**private Integer addressId;**

**@Column(length=50,name="STREET")**

**private String street;**

**@Column(length=100, name="CITY")**

**private String city;**

**@Column(name="is\_open")**

**private boolean isOpen;**

**@Transient**

**private double x;**

**@Column(name="added\_date")**

**@Temporal(TemporalType.DATE)**

**private Date addedDate;**

**@Lob**

**private byte[] image;**

**public Address() {**

**super();**

**// TODO Auto-generated constructor stub**

**}**

**public Address(Integer addressId, String street, String city, boolean isOpen, double x, Date addedDate,**

**byte[] image) {**

**super();**

**this.addressId = addressId;**

**this.street = street;**

**this.city = city;**

**this.isOpen = isOpen;**

**this.x = x;**

**this.addedDate = addedDate;**

**this.image = image;**

**}**

**@Override**

**public String toString() {**

**return "Address [addressId=" + addressId + ", street=" + street + ", city=" + city + ", isOpen=" + isOpen**

**+ ", x=" + x + ", addedDate=" + addedDate + ", image=" + Arrays.toString(image) + "]";**

**}**

**public Integer getAddressId() {**

**return addressId;**

**}**

**public void setAddressId(Integer addressId) {**

**this.addressId = addressId;**

**}**

**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 boolean isOpen() {**

**return isOpen;**

**}**

**public void setOpen(boolean isOpen) {**

**this.isOpen = isOpen;**

**}**

**public double getX() {**

**return x;**

**}**

**public void setX(double x) {**

**this.x = x;**

**}**

**public Date getAddedDate() {**

**return addedDate;**

**}**

**public void setAddedDate(Date addedDate) {**

**this.addedDate = addedDate;**

**}**

**public byte[] getImage() {**

**return image;**

**}**

**public void setImage(byte[] image) {**

**this.image = image;**

**}**

**}**

**Object creation for the address entity class:-**

**package com.tut;**

**import java.io.FileInputStream;**

**import java.io.IOException;**

**import java.util.Date;**

**import org.hibernate.Session;**

**import org.hibernate.SessionFactory;**

**import org.hibernate.Transaction;**

**import org.hibernate.cfg.Configuration;**

**public class AddressApp {**

**public static void main(String[] args) throws IOException {**

**Configuration cfg = new Configuration();**

**cfg.configure("hibernate.cfg.xml");**

**SessionFactory factory = cfg.buildSessionFactory();**

**// creating the object of address**

**Address ad = new Address();**

**ad.setStreet("ParkStreet");**

**ad.setCity("kolkata");**

**ad.setOpen(true);**

**ad.setAddedDate(new Date());**

**ad.setX(122.555);**

**FileInputStream fis = new FileInputStream("E:\\workspaces\\hibernate workspace\\ProjectWIthMaven\\src\\main\\java\\sytem.png");**

**byte[] data = new byte[fis.available()];**

**fis.read(data);**

**ad.setImage(data);**

**System.out.println(ad);**

**Session session = factory.openSession();**

**Transaction tx = session.beginTransaction();**

**session.save(ad);**

**tx.commit();**

**session.close();**

**System.out.println("Done....");**

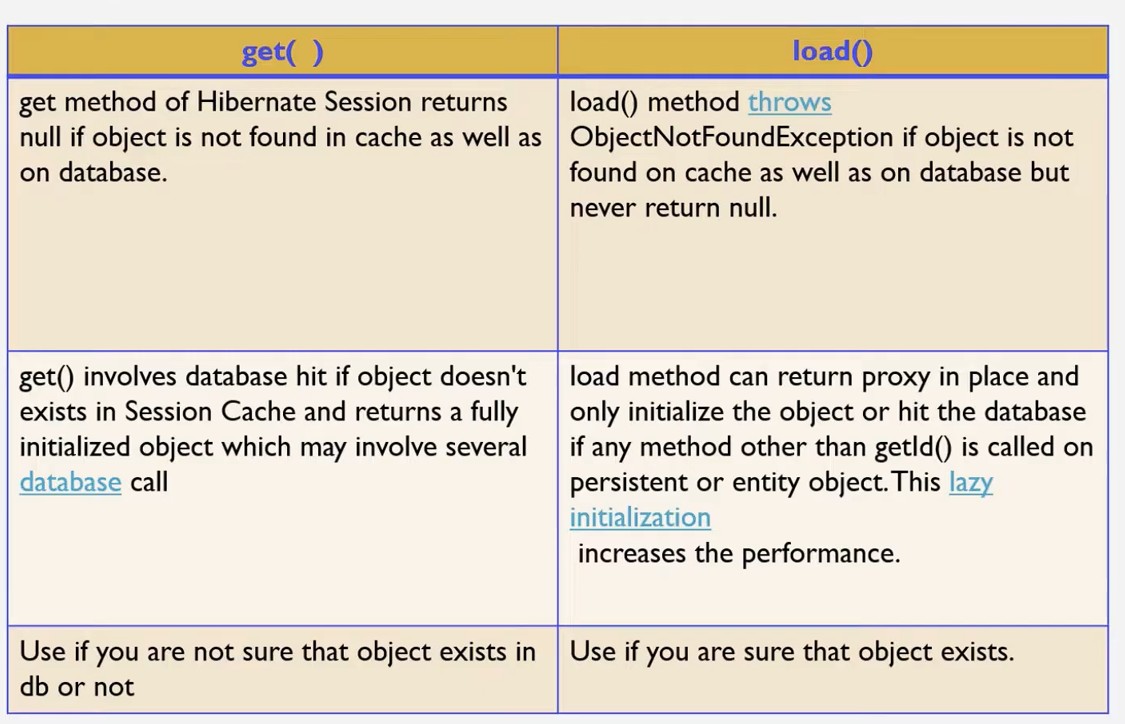
**}**

**}**

**To Fetch the data in Hibernate, Session has two function**

**1.get()**

**2.load()**



To Fetch the data from database we don’t need transaction object in the hibernate code:-

Let’ s see how we will code the in hibernate to fetch the data from database using hibernate.

public class FetchDemo {

public static void main(String[] args) {

**//get ,load**

System.out.println("projected started");

Configuration cfg = new Configuration();

cfg.configure("hibernate.cfg.xml");

SessionFactory factory = cfg.buildSessionFactory();

Session session= factory.openSession();

Student student=(Student)session.load(Student.class, 1);

System.out.println(student);

Address ad =(Address)session.get(Address.class, 1);

System.out.println(ad);

session.close();

factory.close();

}

}

To fetch the data in hibernate we use two methods of hibernate get and load. Where we pass entity class name and primary key

Get(Entityclass name, primary key)

(Student)session.load(Student.class, 1);

session.get(Address.class, 1);

**@Embeddable Annotation**

The @Embeddable and @Embedded annotations in Hibernate are used to map an object’s properties to columns in a database table. These annotations are used in combination to allow the properties of one class to be included as a value type in another class and then be persisted in the database as part of the containing class.

Overview

🡺The @Embeddable annotation is used to mark a class as being embeddable, meaning its properties can be included in another class as a value type. The class marked with @Embeddable is called the embeddable class.

🡺The @Embedded annotation is used to mark a field in a class as being an embeddable object, and it is used in the class that contains the embeddable object.

By using these annotations, Hibernate can automatically persist the properties of the embeddable class within the containing class to the database table, without the need to create a separate table for the embeddable class.

🡺Using @Embeddable and @Embedded annotations in Hibernate allows for better data modeling, code reusability, normalization, and better performance. The annotations also allow to encapsulation of the business logic within the embeddable class.

Let’s understand with the code:-

In the below entitiy class we have added one of the reference of embeddable class which need to create the column…

package com.tut;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="Student\_details")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "student\_id")

private Integer id;

private String name;

private String city;

private Certificate certi; //embeddable reference for the table.

@Override

public String toString() {

return "Student [id=" + id + ", name=" + name + ", city=" + city + "]";

}

public Student(Integer id, String name, String city) {

super();

this.id = id;

this.name = name;

this.city = city;

}

public Student() {

super();

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public Certificate getCerti() {

return certi;

}

public void setCerti(Certificate certi) {

this.certi = certi;

}

}

Let’s see the embeddable class which need to be embedded with other entity class to make any class embeddable we need to use an annotation @embeddable. Let’s see the code here.

package com.tut;

import javax.persistence.Embeddable;

@Embeddable //annotation which is used to make the embeddable

public class Certificate {

private String course;

private String duration;

public String getCourse() {

return course;

}

public void setCourse(String course) {

this.course = course;

}

public String getDuration() {

return duration;

}

public void setDuration(String duration) {

this.duration = duration;

}

public Certificate() {

super();

// TODO Auto-generated constructor stub

}

public Certificate(String course, String duration) {

super();

this.course = course;

this.duration = duration;

}

@Override

public String toString() {

return "Certificate [course=" + course + ", duration=" + duration + "]";

}

}

Let’s see the main class which will call the entity class:-

package com.tut;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class EmDemo {

public static void main(String[] args) {

System.out.println("projected started");

Configuration cfg = new Configuration();

cfg.configure("hibernate.cfg.xml");

SessionFactory factory = cfg.buildSessionFactory();

Student student1 = new Student();

student1.setId(14134);

student1.setName("Gautam");

student1.setCity("ranchi");

Certificate certi =new Certificate();

certi.setCourse("Android");

certi.setDuration("2month");

student1.setCerti(certi);

Student student2 = new Student();

student2.setId(14135);

student2.setName("Ramu");

student2.setCity("Ramgarh");

Certificate certi1 =new Certificate();

certi1.setCourse("web");

certi1.setDuration("3month");

student2.setCerti(certi1);

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

//objects save

session.save(student1);

session.save(student2);

tx.commit();

session.close();

factory.close();

}

}

**One to one Mapping**

In RDBMS for one to one mapping we use the concept of foreign key , but in case of hibernate let’s see how we are going to deal with it suppose we have two tables Table 1 is question and table 2 is answer and both the table is related with one other let’s see how we are to connect it.



Simply what we have done is we added the reference of the answer with the question class and added one annotation **@onetoone** to describe it is one to one relation.

Question class

**package com.map;**

**import javax.persistence.Column;**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**import javax.persistence.JoinColumn;**

**import javax.persistence.OneToOne;**

**@Entity**

**public class Question {**

**@Id**

**@Column(name = "question\_id")**

**private int questionId;**

**private String question;**

**@OneToOne**

**@JoinColumn(name = "answer\_answerId", referencedColumnName = "answer\_id")**

**private Answer answer;**

**// getters, setters, toString**

**public int getQuestionId() {**

**return questionId;**

**}**

**public void setQuestionId(int questionId) {**

**this.questionId = questionId;**

**}**

**public String getQuestion() {**

**return question;**

**}**

**public void setQuestion(String question) {**

**this.question = question;**

**}**

**public Answer getAnswer() {**

**return answer;**

**}**

**public void setAnswer(Answer answer) {**

**this.answer = answer;**

**}**

**@Override**

**public String toString() {**

**return "Question [questionId=" + questionId + ", question=" + question + ", answer=" + answer + "]";**

**}**

**}**

 @JoinColumn(name = "answer\_answerId") specifies the foreign key column in the question table.

 referencedColumnName = "answer\_id" tells Hibernate to map it to the answer\_id column in the answer table.

Answer class

**package com.map;**

**import javax.persistence.Column;**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**@Entity**

**public class Answer {**

**@Id**

**@Column(name = "answer\_id")**

**private int answerId;**

**private String answer;**

**@Override**

**public String toString() {**

**return "Answer [answerId=" + answerId + ", answer=" + answer + "]";**

**}**

**public int getAnswerId() {**

**return answerId;**

**}**

**public void setAnswerId(int answerId) {**

**this.answerId = answerId;**

**}**

**public String getAnswer() {**

**return answer;**

**}**

**public void setAnswer(String answer) {**

**this.answer = answer;**

**}**

**// getters, setters, toString**

**}**

MapDemo

**package com.map;**

**import org.hibernate.Session;**

**import org.hibernate.SessionFactory;**

**import org.hibernate.Transaction;**

**import org.hibernate.cfg.Configuration;**

**public class MapDemo {**

**public static void main(String[] args) {**

**Configuration cfg = new Configuration();**

**cfg.configure("hibernate.cfg.xml");**

**SessionFactory factory = cfg.buildSessionFactory();**

**Session session = factory.openSession();**

**Transaction tx = session.beginTransaction();**

**// Creating Answer first**

**Answer answer1 = new Answer();**

**answer1.setAnswerId(343);**

**answer1.setAnswer("Java is a programming language");**

**session.save(answer1); // ✅ Save first**

**Question question1 = new Question();**

**question1.setQuestionId(1212);**

**question1.setQuestion("What is Java?");**

**question1.setAnswer(answer1);**

**session.save(question1); // ✅ Then save question**

**// Second pair**

**Answer answer2 = new Answer();**

**answer2.setAnswerId(344);**

**answer2.setAnswer("API to work with data structures in Java");**

**session.save(answer2); // ✅ Save first**

**Question question2 = new Question();**

**question2.setQuestionId(242);**

**question2.setQuestion("What is Collection Framework?");**

**question2.setAnswer(answer2);**

**session.save(question2); // ✅ Then save question**

**tx.commit();**

**session.close();**

**factory.close();**

**}**

**}**

**One to Many technique**

**Question.java**

**package com.map;**

**import java.util.List;**

**import javax.persistence.Column;**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**import javax.persistence.JoinColumn;**

**import javax.persistence.OneToMany;**

**import javax.persistence.OneToOne;**

**@Entity**

**public class Question {**

**@Id**

**@Column(name = "question\_id")**

**private int questionId;**

**private String question;**

**@OneToMany(mappedBy="question")**

**private List<Answer>answers;**

**// getters, setters, toString**

**public int getQuestionId() {**

**return questionId;**

**}**

**public void setQuestionId(int questionId) {**

**this.questionId = questionId;**

**}**

**public String getQuestion() {**

**return question;**

**}**

**public void setQuestion(String question) {**

**this.question = question;**

**}**

**public List<Answer> getAnswers() {**

**return answers;**

**}**

**public void setAnswers(List<Answer> answers) {**

**this.answers = answers;**

**}**

**@Override**

**public String toString() {**

**return "Question [questionId=" + questionId + ", question=" + question + ", answers=" + answers + "]";**

**}**

**}**

**Answer java**

**package com.map;**

**import javax.persistence.Column;**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**import javax.persistence.ManyToOne;**

**@Entity**

**public class Answer {**

**@Id**

**@Column(name = "answer\_id")**

**private int answerId;**

**private String answer;**

**@ManyToOne**

**private Question question;**

**public Question getQuestion() {**

**return question;**

**}**

**public void setQuestion(Question question) {**

**this.question = question;**

**}**

**@Override**

**public String toString() {**

**return "Answer [answerId=" + answerId + ", answer=" + answer + "]";**

**}**

**public int getAnswerId() {**

**return answerId;**

**}**

**public void setAnswerId(int answerId) {**

**this.answerId = answerId;**

**}**

**public String getAnswer() {**

**return answer;**

**}**

**public void setAnswer(String answer) {**

**this.answer = answer;**

**}**

**// getters, setters, toString**

**}**

**MapDemo**

**package com.map;**

**import java.util.ArrayList;**

**import java.util.List;**

**import org.hibernate.Session;**

**import org.hibernate.SessionFactory;**

**import org.hibernate.Transaction;**

**import org.hibernate.cfg.Configuration;**

**public class MapDemo {**

**public static void main(String[] args) {**

**Configuration cfg = new Configuration();**

**cfg.configure("hibernate.cfg.xml");**

**SessionFactory factory = cfg.buildSessionFactory();**

**Session session = factory.openSession();**

**Transaction tx = session.beginTransaction();**

**Question question1 = new Question();**

**question1.setQuestionId(1212);**

**question1.setQuestion("What is Java?");**

**Answer answer1 = new Answer();**

**answer1.setAnswerId(343);**

**answer1.setAnswer("Java is a programming language");**

**answer1.setQuestion(question1);**

**Answer answer2 = new Answer();**

**answer2.setAnswerId(344);**

**answer2.setAnswer("with the help of java we can build software");**

**answer2.setQuestion(question1);**

**Answer answer3 = new Answer();**

**answer3.setAnswerId(363);**

**answer3.setAnswer("java has different type of framework");**

**answer3.setQuestion(question1);**

**session.save(answer3);**

**session.save(answer2);**

**session.save(answer1);**

**List<Answer> list = new ArrayList<Answer>();**

**list.add(answer1);**

**list.add(answer3);**

**list.add(answer2);**

**question1.setAnswers(list);**

**session.save(question1**

**tx.commit();**

**session.close();**

**factory.close();**

**}**

**}**

**Fetch Type of Hibernate:-**

**🡺Lazy🡪**In lazy loading, associated data loads only when we explicitly call getter or size method

**🡺Eager🡪**It is design pattern in which data loading occurs on the spot.

@OneToMany(mappedBy="question",fetch=FetchType.EAGER)

private List<Answer>answers;

To make it lazy or eager

**Hibernate/Persistence Lifecycle states**

Great question! In Hibernate, every object/entity goes through a **lifecycle** of states during its journey with the Session. These states define how Hibernate treats and tracks the entity.

Here are the **four main lifecycle states** in Hibernate:

### 🔵 1. **Transient**

* **Definition:** The object is created using new but is **not associated with any Hibernate session** and **not saved in the database**.
* **Hibernate doesn't track it.**
* **Example:**

Student s = new Student(); // Transient state

s.setId(1);

s.setName("John");

### 🟢 2. **Persistent**

* **Definition:** The object is **associated with a Hibernate session**, and any changes made to it will be automatically **tracked and saved to the database** (when commit() happens).
* **Managed by Hibernate.**
* **Example:**

Session session = factory.openSession();

Transaction tx = session.beginTransaction();

Student s = new Student();

s.setId(1);

s.setName("John");

session.save(s); // Now s is in Persistent state

tx.commit();

session.close();

### 🔴 3. **Detached**

* **Definition:** The object was once in Persistent state, but the **session is now closed**. So it's no longer being tracked.
* **You can still modify it, but Hibernate won’t care unless you reattach it.**
* **Example:**

session.close(); // Now s becomes Detached

s.setName("John Updated"); // Changes won't be saved

### ⚫ 4. **Removed (or Deleted)**

* **Definition:** The object is **marked for deletion** using session.delete(). It is still in persistent state until the transaction is committed.
* **Once committed, the record is deleted from the DB.**

session.delete(s); // Now s is marked as Removed

### 🔁 Quick Summary Table:

| **State** | **Managed by Hibernate?** | **In Database?** | **Notes** |
| --- | --- | --- | --- |
| Transient | ❌ No | ❌ No | Just a Java object |
| Persistent | ✅ Yes | ✅ Yes | Synced with DB; tracked by Hibernate |
| Detached | ❌ No | ✅ Yes | Was persistent, but session is closed |
| Removed | ✅ (until commit) | ❌ (after commit) | Marked for deletion |

**HQL(Hibernate Query language)**

### 🔷 **HQL (Hibernate Query Language)**

**HQL** is an object-oriented query language **used to interact with the database** in Hibernate. It is similar to SQL but **works with Java classes and objects**, **not table names or columns directly**.

### 🧠 Key Features of HQL:

* **Object-oriented**: You use class names and property names, not table names and column names.
* **Database-independent**: Since it uses Java entity names, it's more portable across DBs.
* **Supports joins, grouping, sorting**, etc.
* Can perform **select, update, delete** operations.

### ✅ Example: Basic HQL Query

#### Suppose you have an entity Student:

@Entity

public class Student {

@Id

private int id;

private String name;

private int marks;

// getters/setters

}

#### ➤ Select all students:

String query = "from Student";

Query q = session.createQuery(query);

List<Student> list = q.list();

#### ➤ Select with condition:

String query = "from Student where marks > 80";

#### ➤ Named parameters:

String query = "from Student where name = :name";

Query q = session.createQuery(query);

q.setParameter("name", "John");

List<Student> list = q.list();

#### ➤ Update using HQL:

String query = "update Student set marks = 90 where name = :name";

Query q = session.createQuery(query);

q.setParameter("name", "John");

int count = q.executeUpdate();

#### ➤ Delete using HQL:

String query = "delete from Student where id = :id";

Query q = session.createQuery(query);

q.setParameter("id", 101);

q.executeUpdate();

### 🚫 Difference from SQL:

| **SQL** | **HQL** |
| --- | --- |
| Uses table names | Uses entity (class) names |
| Uses column names | Uses property (field) names |
| Not DB independent | DB independent |
| No object navigation | Supports object navigation (e.g. student.address.city) |

### 🔄 Bonus: Native SQL in Hibernate (Just FYI)

If needed, you can also run raw SQL:

SQLQuery query = session.createSQLQuery("SELECT \* FROM student");

But HQL is preferred for **ORM-style, clean, DB-independent code**.

**Hibernate cascading**

Great topic! Let's break down **Hibernate Cascading** in a clear and simple way:

### 🔄 **What is Cascading in Hibernate?**

**Cascading** in Hibernate is a way to tell Hibernate **what to do with the associated (related) entities** when a certain operation is performed on the parent entity.

➡️ In short: If you **save/update/delete** an object, do the **same** for the objects it references.

### 🔥 Example Use Case

Say you have two entities: Question and Answer, where one Question has one Answer.

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "answer\_id")

private Answer answer;

Now, if you **save Question**, Hibernate will **automatically save Answer** too—no need to call session.save(answer) separately.

### 🧩 Common Cascade Types

| **Cascade Type** | **What it does** |
| --- | --- |
| CascadeType.ALL | Applies all cascade operations (save, delete, update, etc.) |
| CascadeType.PERSIST | Save the child entity when the parent is saved |
| CascadeType.MERGE | Update the child when the parent is updated |
| CascadeType.REMOVE | Delete the child when the parent is deleted |
| CascadeType.DETACH | Detach child from session if parent is detached |
| CascadeType.REFRESH | Refresh child from DB when parent is refreshed |

### ✅ Example in Code

@Entity

public class Question {

@Id

private int questionId;

private String question;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "answer\_id")

private Answer answer;

// getters/setters

}

Now, this works:

Question q = new Question();

q.setQuestion("What is Hibernate?");

Answer a = new Answer();

a.setAnswer("It is an ORM framework");

q.setAnswer(a);

session.save(q); // Automatically saves Answer too 🎉

Without cascade, you’d need to do:

session.save(a);

session.save(q);

### 🚫 When not to use cascade?

* If the child is shared between multiple parents (e.g. multiple Questions sharing the same Answer)
* If you want to manage the lifecycle of each entity manually

### 🧠 Summary

* Cascading automates child entity persistence operations
* Saves you from writing repetitive code
* Needs careful use—especially with shared relationships