**You can use the following xml file for all the exercises**

**Hibernate.cfg.xml file**

**<?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>**

**<!-- Database connection properties -->**

**<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>**

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

**<property name="hibernate.connection.username">your\_username</property>**

**<property name="hibernate.connection.password">your\_password</property>**

**<!-- Dialect for the database -->**

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

**<!-- Show SQL statements in the console -->**

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

**<property name="hibernate.format\_sql">true</property>**

**<!-- Auto-create tables -->**

**<property name="hibernate.hbm2ddl.auto">create</property>**

**<!-- Entity classes -->**

**<mapping class="com.example.Product" />**

**<mapping class="com.example.Employee" />**

**<mapping class="com.example.Address" />**

**<mapping class="com.example.Customer" />**

**<mapping class="com.example.Department" />**

**<mapping class="com.example.Book" />**

**<mapping class="com.example.Author" />**

**<mapping class="com.example.Course" />**

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

**<mapping class="com.example.Vehicle" />**

**<mapping class="com.example.Car" />**

**<mapping class="com.example.Motorcycle" />**

**<mapping class="com.example.Animal" />**

**<mapping class="com.example.Dog" />**

**<mapping class="com.example.Cat" />**

**<mapping class="com.example.Contact" />**

**<mapping class="com.example.ContactInfo" />**

**</session-factory>**

**</hibernate-configuration>**

**Exercise 1: Basic Entity Mapping**

@Entity

@Table(name=”products”)

public class Product {

@Id

@GeneratedValue

private Long id;

private String name;

private double price;

// getters and setters

}

**Create a new Product and save it to the database.**

**Fetch the product with id = 1 from the database**.

**Update the name of the product with id = 1 to "Updated Product".**

**Delete the product with id = 1 from the database.**

**Exercise 2: One-to-One Association**

@Entity

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

@OneToOne(cascade = CascadeType.ALL)

private Address address;

// getters and setters

}

@Entity

public class Address {

@Id

@GeneratedValue

private Long id;

private String city;

private String zipCode;

// getters and setters

}

**Create a new Employee with an associated Address and save both to the database.**

**Fetch an Employee and retrieve its associated Address.**

**Update the city of the Address associated with Employee id = 1 to "New City".**

**Delete an Employee and its associated Address from the database.**

**Exercise 3: One-to-Many Association**

@Entity

public class Department {

@Id

@GeneratedValue

private Long id;

private String name;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "department")

private List<Employee> employees;

// getters and setters

}

@Entity

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

@ManyToOne

private Department department;

// getters and setters

}

**Create a new Department with multiple Employees and save them to the database.**

**Fetch a Department and retrieve its list of Employees.**

**Update the name of an Employee and save the changes.**

**Remove an Employee from the Department and delete it from the database.**

**Exercise 4: Many-to-Many Association**

@Entity

public class Student {

@Id

@GeneratedValue

private Long id;

private String name;

@ManyToMany(cascade = CascadeType.ALL)

@JoinTable(name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id"))

private List<Course> courses;

// getters and setters

}

@Entity

public class Course {

@Id

@GeneratedValue

private Long id;

private String name;

@ManyToMany(mappedBy = "courses")

private List<Student> students;

// getters and setters

}

**Create multiple Students and multiple Courses and enroll students in courses.**

**Fetch a Student and retrieve the list of Courses they are enrolled in.**

**Fetch a Course and retrieve the list of Students enrolled in it.**

**Remove a Student from a Course and save the changes.**

**Exercise 5: Hibernate Query Language (HQL)**

@Entity

public class Book {

@Id

@GeneratedValue

private Long id;

private String title;

private String author;

private double price;

// getters and setters

}

**Write an HQL query to fetch all books with a price greater than 50.00.**

**Write an HQL query to fetch all books written by a specific author.**

**Write an HQL query to fetch the titles of all books sorted alphabetically.**

**Write an HQL query to update the price of all books written by a specific author.**

**Exercise 6: Native SQL Queries**

@Entity

public class Product {

@Id

@GeneratedValue

private Long id;

private String name;

private double price;

// getters and setters

}

**Write a native SQL query to fetch all products with a price less than 100.00.**

**Write a native SQL query to update the price of a specific product by id.**

**Exercise 7: Criteria API**

@Entity

public class Customer {

@Id

@GeneratedValue

private Long id;

private String name;

private int age;

// getters and setters

}

**Use Criteria API to fetch all customers whose age is greater than 25.**

**Use Criteria API to fetch the names of all customers sorted in descending order of their age.**

**Exercise 8: Named Queries**

@Entity

@NamedQuery(name = "Employee.findAll", query = "SELECT e FROM Employee e")

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private double salary;

// getters and setters

}

**Use the named query "Employee.findAll" to fetch all employees from the database.**

**Exercise 9: Fetch Strategies**

@Entity

public class Order {

@Id

@GeneratedValue

private Long id;

private String orderNumber;

@ManyToOne(fetch = FetchType.LAZY)

private Customer customer;

// getters and setters

}

@Entity

public class Customer {

@Id

@GeneratedValue

private Long id;

private String name;

// getters and setters

}

**Fetch an Order and retrieve its associated Customer using lazy loading.**

**Change the fetch strategy to eager loading and fetch an Order again.**

@Entity

public class Order {

// ...

@ManyToOne(fetch = FetchType.EAGER)

private Customer customer;

// ...

}

Session session = sessionFactory.openSession();

Order order = session.get(Order.class, 1L);

Customer customer = order.getCustomer(); // No lazy loading; Customer is fetched eagerly

session.close();

**Exercise 10: Transaction Management**

@Entity

public class Account {

@Id

@GeneratedValue

private Long id;

private String accountNumber;

private double balance;

// getters and setters

}

**Create a new Account and save it to the database within a transaction.**

**Fetch an Account and update its balance within a transaction.**

**Exercise 11: Cascade Operations**

@Entity

public class Author {

@Id

@GeneratedValue

private Long id;

private String name;

@OneToMany(mappedBy = "author", cascade = CascadeType.ALL)

private List<Book> books;

// getters and setters

}

@Entity

public class Book {

@Id

@GeneratedValue

private Long id;

private String title;

@ManyToOne

private Author author;

// getters and setters

}

**Create a new Author along with multiple Books and save them to the database using cascading.**

**Fetch an Author and delete it along with all associated Books using cascading.**

**Exercise 12: Lazy Loading and Fetch Types**

@Entity

public class Category {

@Id

@GeneratedValue

private Long id;

private String name;

@OneToMany(mappedBy = "category", fetch = FetchType.LAZY)

private List<Product> products;

// getters and setters

}

@Entity

public class Product {

@Id

@GeneratedValue

private Long id;

private String name;

@ManyToOne

private Category category;

// getters and setters

}

**Fetch a Category and retrieve its associated Products using lazy loading.**

**Change the fetch type to eager loading and fetch a Category along with its Products.**

**Exercise 13: Transactions and Rollback**

@Entity

public class Customer {

@Id

@GeneratedValue

private Long id;

private String name;

private double balance;

// getters and setters

}

**Create a new Customer and save it to the database within a transaction.**

**Attempt to save a Customer with a negative balance. Ensure the transaction rolls back.**

**Exercise 14: Inheritance Mapping - Single Table Strategy**

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "vehicle\_type", discriminatorType = DiscriminatorType.STRING)

public abstract class Vehicle {

@Id

@GeneratedValue

private Long id;

private String manufacturer;

// getters and setters

}

@Entity

@DiscriminatorValue("car")

public class Car extends Vehicle {

private int numberOfDoors;

// getters and setters

}

@Entity

@DiscriminatorValue("motorcycle")

public class Motorcycle extends Vehicle {

private String bikeType;

// getters and setters

}

**Create a new Car and Motorcycle and save them to the database.**

**Fetch a Vehicle and verify its actual type.**

**Exercise 15: Inheritance Mapping - Joined Strategy**

@Entity

@Inheritance(strategy = InheritanceType.JOINED)

public abstract class Animal {

@Id

@GeneratedValue

private Long id;

private String name;

// getters and setters

}

@Entity

public class Dog extends Animal {

private String breed;

// getters and setters

}

@Entity

public class Cat extends Animal {

private int lives;

// getters and setters

}

**Create a new Dog and Cat and save them to the database.**

**Fetch a Dog and retrieve its specific attributes.**

Session session = sessionFactory.openSession();

Dog dog = session.get(Dog.class, 1L);

System.out.println("Dog Name: " + dog.getName());

System.out.println("Dog Breed: " + dog.getBreed());

session.close();

**Exercise 16: Named Native Queries**

@Entity

@NamedNativeQuery(name = "Employee.findAll", query = "SELECT \* FROM employee", resultClass = Employee.class)

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private double salary;

// getters and setters

}

**Use the named native query "Employee.findAll" to fetch all employees from the database.**

**Exercise 17: HQL Joins**

@Entity

public class Author {

@Id

@GeneratedValue

private Long id;

private String name;

// getters and setters

}

@Entity

public class Book {

@Id

@GeneratedValue

private Long id;

private String title;

@ManyToOne

private Author author;

// getters and setters

}

**Write an HQL query to fetch all books along with their authors.**

**Write an HQL query to fetch all authors and their total number of books.**

**Exercise 18: Criteria API Projections**

@Entity

public class Product {

@Id

@GeneratedValue

private Long id;

private String name;

private double price;

// getters and setters

}

**Use Criteria API to fetch only the names of all products.**

**Use Criteria API to fetch the names and prices of products having a price greater than 50.00.**

**Exercise 19: Mapping Enums**

@Entity

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

@Enumerated(EnumType.STRING)

private Department department;

// getters and setters

}

public enum Department {

IT, HR, SALES, MARKETING

}

**Create a new Employee with a specified Department and save it to the database.**

**Fetch an Employee and verify their Department.**

**Exercise 20: Embeddable Objects**

@Embeddable

public class ContactInfo {

private String email;

private String phone;

// getters and setters

}

@Entity

public class Contact {

@Id

@GeneratedValue

private Long id;

private String name;

@Embedded

private ContactInfo contactInfo;

// getters and setters

}

**Create a new Contact with a ContactInfo and save it to the database.**

**Fetch a Contact and retrieve their ContactInfo.**