**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(strategy=GenerationTypes.IDENTITY)

@Column(name=”product\_id”)

private Long id;

@Column(name=”product\_name”)

private String name;

@Column(name=”price”)

private double price;

// getters and setters

}

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

Product product = new Product();

product.setName("Sample Product");

product.setPrice(100.00);

Session session = sessionFactory.openSession();

session.beginTransaction();

session.merge(product);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Product product = session.get(Product.class, 1L);

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Product product = session.get(Product.class, 1L);

product.setName("Updated Product");

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Product product = session.get(Product.class, 1L);

session.delete(product);

session.getTransaction().commit();

session.close();

**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.**

Address address = new Address();

address.setCity("New York");

address.setZipCode("10001");

Employee employee = new Employee();

employee.setName("John Doe");

employee.setAddress(address);

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(employee);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Employee employee = session.get(Employee.class, 1L);

Address address = employee.getAddress();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Employee employee = session.get(Employee.class, 1L);

Address address = employee.getAddress();

address.setCity("New City");

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Employee employee = session.get(Employee.class, 1L);

session.delete(employee);

session.getTransaction().commit();

session.close();

**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.**

Department department = new Department();

department.setName("HR");

Employee employee1 = new Employee();

employee1.setName("John");

employee1.setDepartment(department);

Employee employee2 = new Employee();

employee2.setName("Jane");

employee2.setDepartment(department);

department.setEmployees(Arrays.asList(employee1, employee2));

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(department);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Department department = session.get(Department.class, 1L);

List<Employee> employees = department.getEmployees();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Employee employee = session.get(Employee.class, 1L);

employee.setName("Updated Name");

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Department department = session.get(Department.class, 1L);

List<Employee> employees = department.getEmployees();

Employee employeeToRemove = employees.get(0);

employees.remove(employeeToRemove);

session.delete(employeeToRemove);

session.getTransaction().commit();

session.close();

**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.**

Student student1 = new Student();

student1.setName("John");

Student student2 = new Student();

student2.setName("Jane");

Course course1 = new Course();

course1.setName("Math");

Course course2 = new Course();

course2.setName("Science");

student1.setCourses(Arrays.asList(course1, course2));

student2.setCourses(Arrays.asList(course1));

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(student1);

session.save(student2);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Student student = session.get(Student.class, 1L);

List<Course> courses = student.getCourses();

session.close();

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

Session session = sessionFactory.openSession();

Course course = session.get(Course.class, 1L);

List<Student> students = course.getStudents();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Student student = session.get(Student.class, 1L);

Course course = session.get(Course.class, 1L);

student.getCourses().remove(course);

session.getTransaction().commit();

session.close();

**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.**

Session session = sessionFactory.openSession();

String hql = "FROM Book b WHERE b.price > :price";

List<Book> books = session.createQuery(hql)

.setParameter("price", 50.00)

.getResultList();

session.close();

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

Session session = sessionFactory.openSession();

String hql = "FROM Book b WHERE b.author = :author";

List<Book> books = session.createQuery(hql)

.setParameter("author", "John Doe")

.getResultList();

session.close();

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

Session session = sessionFactory.openSession();

String hql = "SELECT b.title FROM Book b ORDER BY b.title";

List<String> bookTitles = session.createQuery(hql).getResultList();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

String hql = "UPDATE Book b SET b.price = :newPrice WHERE b.author = :author";

int updatedCount = session.createQuery(hql)

.setParameter("newPrice", 60.00)

.setParameter("author", "John Doe")

.executeUpdate();

session.getTransaction().commit();

session.close();

**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.**

Session session = sessionFactory.openSession();

String sql = "SELECT \* FROM product WHERE price < :price";

List<Product> products = session.createNativeQuery(sql, Product.class)

.setParameter("price", 100.00)

.getResultList();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

String sql = "UPDATE product SET price = :newPrice WHERE id = :id";

int updatedCount = session.createNativeQuery(sql)

.setParameter("newPrice", 120.00)

.setParameter("id", 1L)

.executeUpdate();

session.getTransaction().commit();

session.close();

**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.**

Session session = sessionFactory.openSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<Customer> query = builder.createQuery(Customer.class);

Root<Customer> root = query.from(Customer.class);

query.select(root).where(builder.gt(root.get("age"), 25));

List<Customer> customers = session.createQuery(query).getResultList();

session.close();

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

Session session = sessionFactory.openSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<String> query = builder.createQuery(String.class);

Root<Customer> root = query.from(Customer.class);

query.select(root.get("name")).orderBy(builder.desc(root.get("age")));

List<String> customerNames = session.createQuery(query).getResultList();

session.close();

**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.**

Session session = sessionFactory.openSession();

List<Employee> employees = session.createNamedQuery("Employee.findAll", Employee.class)

.getResultList();

session.close();

**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.**

Session session = sessionFactory.openSession();

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

Customer customer = order.getCustomer(); // Lazy loading occurs here

session.close();

**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.**

Account account = new Account();

account.setAccountNumber("12345");

account.setBalance(1000.00);

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

session.save(account);

transaction.commit();

session.close();

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

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Account account = session.get(Account.class, 1L);

account.setBalance(1500.00);

transaction.commit();

session.close();

**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.**

Author author = new Author();

author.setName("John Doe");

Book book1 = new Book();

book1.setTitle("Book 1");

book1.setAuthor(author);

Book book2 = new Book();

book2.setTitle("Book 2");

book2.setAuthor(author);

author.setBooks(Arrays.asList(book1, book2));

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(author);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

session.beginTransaction();

Author author = session.get(Author.class, 1L);

session.delete(author);

session.getTransaction().commit();

session.close();

**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.**

Session session = sessionFactory.openSession();

Category category = session.get(Category.class, 1L);

List<Product> products = category.getProducts(); // Lazy loading occurs here

session.close();

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

@Entity

public class Category {

// ...

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

private List<Product> products;

// ...

}

Session session = sessionFactory.openSession();

Category category = session.get(Category.class, 1L);

List<Product> products = category.getProducts(); // No lazy loading; Products are fetched eagerly

session.close();

**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.**

Customer customer = new Customer();

customer.setName("John Doe");

customer.setBalance(1000.00);

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

session.save(customer);

transaction.commit();

session.close();

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

Customer customer = new Customer();

customer.setName("Jane Doe");

customer.setBalance(-500.00);

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

try {

session.save(customer);

transaction.commit();

} catch (Exception e) {

transaction.rollback();

e.printStackTrace();

}

session.close();

**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.**

Car car = new Car();

car.setManufacturer("Toyota");

car.setNumberOfDoors(4);

Motorcycle motorcycle = new Motorcycle();

motorcycle.setManufacturer("Harley-Davidson");

motorcycle.setBikeType("Cruiser");

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(car);

session.save(motorcycle);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Vehicle vehicle = session.get(Vehicle.class, 1L);

if (vehicle instanceof Car) {

System.out.println("Car");

} else if (vehicle instanceof Motorcycle) {

System.out.println("Motorcycle");

}

session.close();

**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.**

Dog dog = new Dog();

dog.setName("Fido");

dog.setBreed("Labrador");

Cat cat = new Cat();

cat.setName("Whiskers");

cat.setLives(9);

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(dog);

session.save(cat);

session.getTransaction().commit();

session.close();

**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.**

Session session = sessionFactory.openSession();

List<Employee> employees = session.createNamedQuery("Employee.findAll", Employee.class)

.getResultList();

session.close();

**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.**

Session session = sessionFactory.openSession();

String hql = "FROM Book b JOIN FETCH b.author";

List<Book> books = session.createQuery(hql, Book.class).getResultList();

session.close();

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

Session session = sessionFactory.openSession();

String hql = "SELECT a, COUNT(b) FROM Author a JOIN a.books b GROUP BY a";

List<Object[]> results = session.createQuery(hql).getResultList();

for (Object[] result : results) {

Author author = (Author) result[0];

Long bookCount = (Long) result[1];

System.out.println("Author: " + author.getName() + ", Books: " + bookCount);

}

session.close();

**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.**

Session session = sessionFactory.openSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<String> query = builder.createQuery(String.class);

Root<Product> root = query.from(Product.class);

query.select(root.get("name"));

List<String> productNames = session.createQuery(query).getResultList();

session.close();

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

Session session = sessionFactory.openSession();

CriteriaBuilder builder = session.getCriteriaBuilder();

CriteriaQuery<Object[]> query = builder.createQuery(Object[].class);

Root<Product> root = query.from(Product.class);

query.multiselect(root.get("name"), root.get("price")).where(builder.gt(root.get("price"), 50.00));

List<Object[]> results = session.createQuery(query).getResultList();

for (Object[] result : results) {

String name = (String) result[0];

double price = (double) result[1];

System.out.println("Product: " + name + ", Price: " + price);

}

session.close();

**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.**

Employee employee = new Employee();

employee.setName("John Doe");

employee.setDepartment(Department.IT);

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(employee);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Employee employee = session.get(Employee.class, 1L);

Department department = employee.getDepartment();

session.close();

**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.**

ContactInfo contactInfo = new ContactInfo();

contactInfo.setEmail("john@example.com");

contactInfo.setPhone("123-456-7890");

Contact contact = new Contact();

contact.setName("John Doe");

contact.setContactInfo(contactInfo);

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(contact);

session.getTransaction().commit();

session.close();

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

Session session = sessionFactory.openSession();

Contact contact = session.get(Contact.class, 1L);

ContactInfo contactInfo = contact.getContactInfo();

session.close();