## Leave Management System using Hibernate Persistence in Eclipse IDE

.

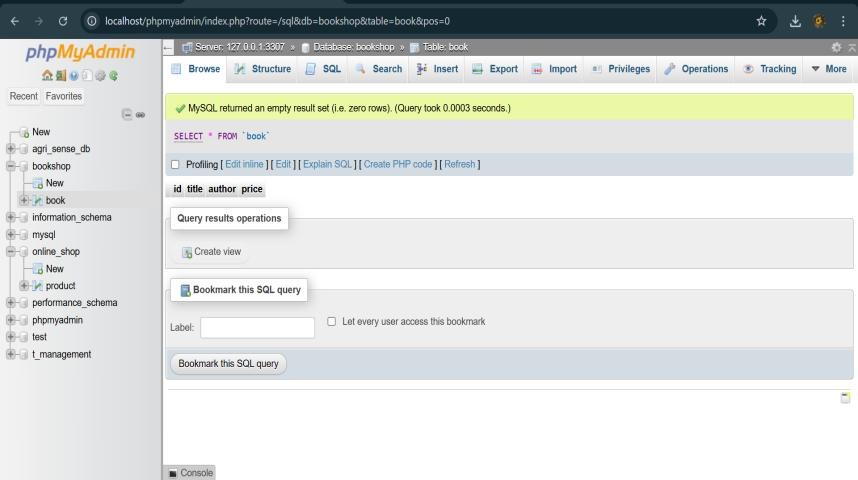
* Create and configure a MySQL database table tailored for contact information.
* Set up the Hibernate configuration file to establish database connectivity.
* Develop entity classes using JPA annotations to map Java objects to database tables.
* Implement a Data Access Object (DAO) class for essential CRUD (Create, Read, Update, Delete) operations.
* Test your contact management system with a main class demonstrating these functions.

## Step 1: Creating the MySQL Table for Contacts

The first step in building the Bus Reservation System is to create the database table that will store contact information. We will create a table named contacts in your MySQL database with the following fields:

* **id:** An INT field, set as the primary key with AUTO\_INCREMENT to uniquely identify each contact.
* **name:** A VARCHAR(255) field to store the contact's full name.
* **email:** A VARCHAR(255) field for the contact’s email address.
* **phone:** A VARCHAR(20) field to save the contact’s phone number.
* **address:** A VARCHAR(255) field to hold the contact’s physical address.

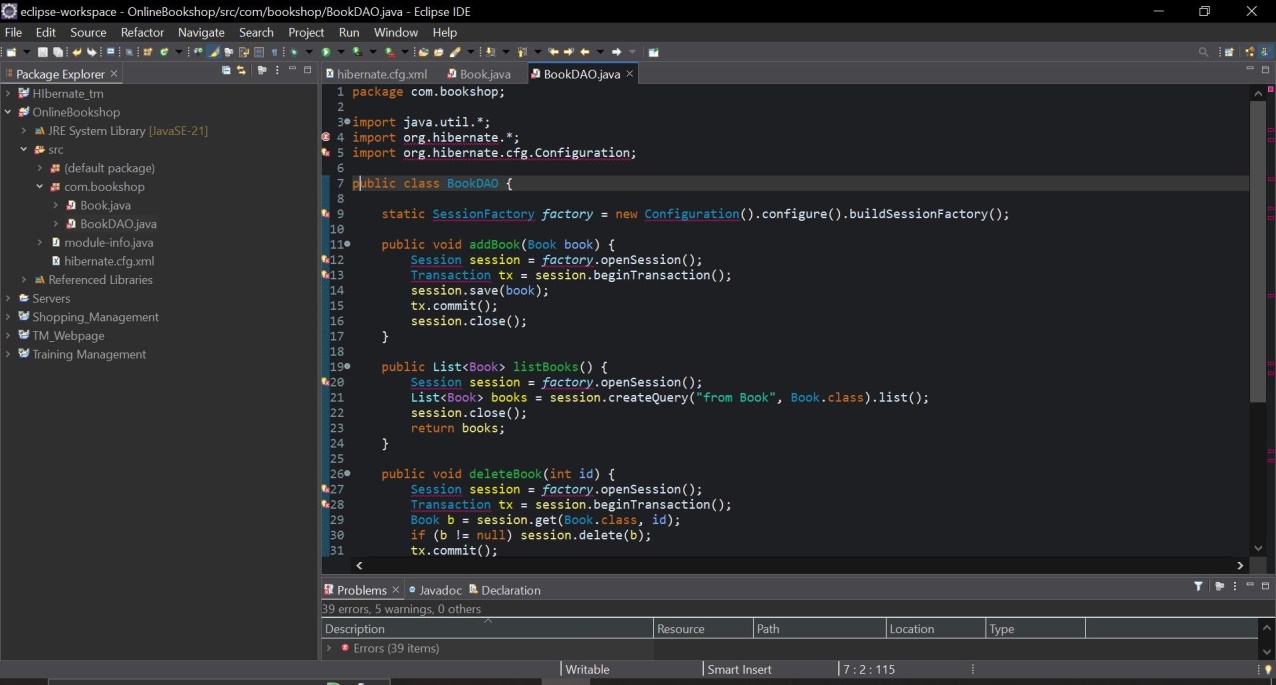
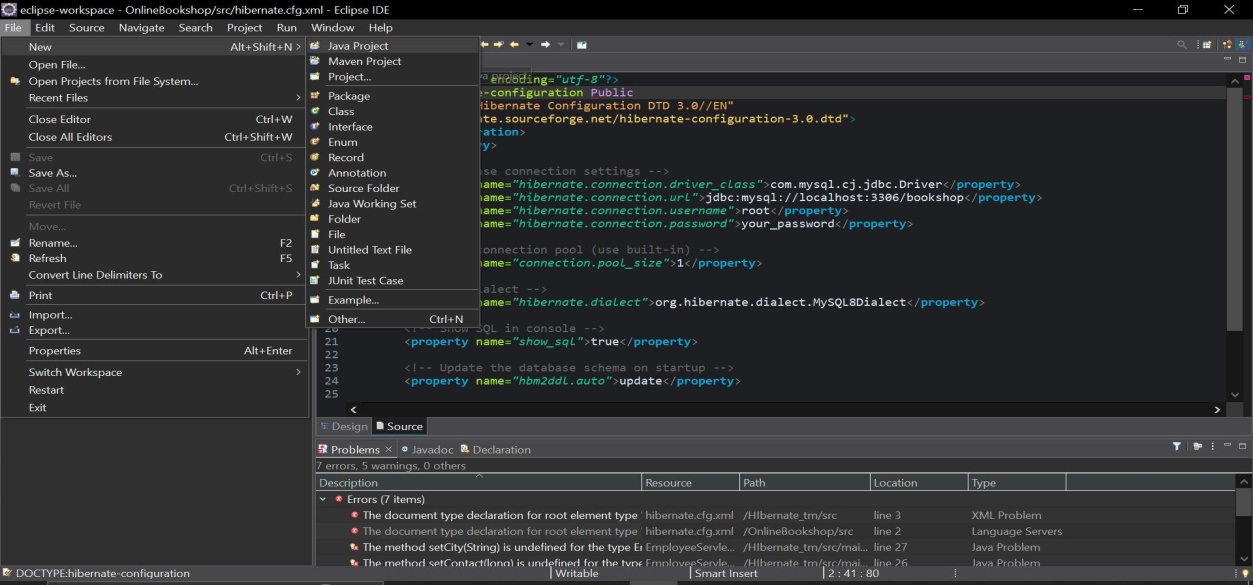
To create this table,

CREATE TABLE contacts (  
 id INT NOT NULL AUTO\_INCREMENT,  
 name VARCHAR(255) NOT NULL,  
 email VARCHAR(255),  
 phone VARCHAR(20),  
 address VARCHAR(255),  
 PRIMARY KEY (id)  
);  


## Step 2: Setting up the Eclipse Project with Required Libraries

1. **Create a new Java Project:**
   * Open Eclipse IDE.
   * Go to *File > New > Java Project*.
   * Enter ContactManager as the project name.
   * Keep the default JRE configuration and click *Finish*.
2. **Create the package structure:**
   * Right-click on src folder.
   * Select *New > Package*.
   * Name the package com.contactmanager and click *Finish*.
3. **Add Hibernate and MySQL libraries:**

* To enable Hibernate ORM functionality and MySQL database connectivity, you must include the necessary JAR files in your project build path. It is good practice to organize these JARs inside a lib folder within your project:
  + Create a new folder named lib at the root of the project.
  + Download and place the following JAR files into the lib folder:
    - **Hibernate Core:** hibernate-core-x.x.x.Final.jar
    - **Hibernate Commons Logging:** hibernate-commons-annotations-x.x.x.jar
    - **Hibernate JPA API:** javax.persistence-api-x.x.jar
    - **MySQL JDBC Driver:** mysql-connector-java-x.x.x.jar
  + In Eclipse, right-click your project and select *Build Path > Configure Build Path*.
  + Under the *Libraries* tab, click *Add JARs...*, navigate to the lib folder, select all JAR files, and click *OK*.



**Project structure overview:** After completing these steps, your project's basic structure should look like this:

* ContactManager/ (root project folder)
* src/ – Source folder containing your Java packages, such as com.contactmanager.
* lib/ – Folder housing all required JAR libraries.
* hibernate.cfg.xml – Hibernate configuration file (to be created in subsequent steps).

This organized layout ensures your source code, configuration files, and external libraries remain cleanly separated, simplifying maintenance and development as you build your Bus Reservation System.

## Step 3: Creating Hibernate Configuration File (hibernate.cfg.xml)

The hibernate.cfg.xml file is a crucial configuration file that instructs Hibernate how to connect to your MySQL database and manage sessions. This XML file should be placed inside your project's src folder or inside a resources folder if you have one, so that it is available in the classpath at runtime. Hibernate automatically looks for this file when initializing.

Below is a complete example of the hibernate.cfg.xml file configured for the Bus Reservation System:

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE hibernate-configuration PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
 <session-factory>  
  
 <!-- Database connection settings -->  
 <property name="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  
 <property name="connection.url">jdbc:mysql://localhost:3306/contactdb?useSSL=false&serverTimezone=UTC</property>  
 <property name="connection.username">your\_username</property>  
 <property name="connection.password">your\_password</property>  
  
 <!-- SQL dialect for MySQL -->  
 <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>  
  
 <!-- Show executed SQL statements in the console (useful for debugging) -->  
 <property name="show\_sql">true</property>  
  
 <!-- Format the SQL displayed for readability -->  
 <property name="format\_sql">true</property>  
  
 <!-- Automatically validate or update the schema (optional) -->  
 <property name="hbm2ddl.auto">update</property>  
  
 <!-- Enable second level cache (optional) -->  
 <!-- <property name="cache.provider\_class">org.hibernate.cache.NoCacheProvider</property> -->  
  
 <!-- Specify annotated entity classes -->  
 <mapping class="com.contactmanager.Contact"/>  
  
 </session-factory>  
</hibernate-configuration>

## Step 4: Creating the Contact Entity Class (Contact.java) with JPA Annotations.

### Annotations Overview

* **@Entity**: Marks the class as a persistent Java class, meaning it should be mapped to a database table by Hibernate.
* **@Table(name = "contacts")**: Specifies the exact name of the table in the database that this entity maps to, ensuring the mapping matches the table created in Step 1.
* **@Id**: Defines the primary key field of the entity. This uniquely identifies each contact record.
* **@GeneratedValue(strategy = GenerationType.IDENTITY)**: Specifies that the primary key id is auto-incremented by the database (matching MySQL’s AUTO\_INCREMENT behavior).
* **@Column**: Maps each field to a table column with optional attributes such as nullable and length. This maintains consistency and allows control over the database schema.

Below is the full annotated Contact.java class with getter and setter methods for each field:

package com.contactmanager;  
  
import javax.persistence.Entity;  
import javax.persistence.Table;  
import javax.persistence.Id;  
import javax.persistence.Column;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
  
@Entity  
@Table(name = "contacts")  
public class Contact {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "id")  
 private int id;  
  
 @Column(name = "name", nullable = false, length = 255)  
 private String name;  
  
 @Column(name = "email", length = 255)  
 private String email;  
  
 @Column(name = "phone", length = 20)  
 private String phone;  
  
 @Column(name = "address", length = 255)  
 private String address;  
  
 // Default constructor (required by Hibernate)  
 public Contact() {  
 }  
  
 // Getter and setter for id  
 public int getId() {  
 return id;  
 }  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 // Getter and setter for name  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 // Getter and setter for email  
 public String getEmail() {  
 return email;  
 }  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 // Getter and setter for phone  
 public String getPhone() {  
 return phone;  
 }  
 public void setPhone(String phone) {  
 this.phone = phone;  
 }  
  
 // Getter and setter for address  
 public String getAddress() {  
 return address;  
 }  
 public void setAddress(String address) {  
 this.address = address;  
 }  
}  
 .

## Step 5: Creating the ContactDAO Class (ContactDAO.java) for CRUD Operations

### Core DAO Methods Overview

* **addContact(Contact contact):** Saves a new contact record to the database.
* **listContacts():** Retrieves all contact records as a list.
* **updateContact(Contact contact):** Updates an existing contact’s information.
* **deleteContact(int contactId):** Deletes a contact by its ID.

### Full Source Code for ContactDAO.java

package com.contactmanager;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.Transaction;  
import org.hibernate.cfg.Configuration;  
  
import java.util.List;  
  
public class ContactDAO {  
  
 private static SessionFactory sessionFactory;  
  
 static {  
 try {  
 // Build SessionFactory from hibernate.cfg.xml  
 sessionFactory = new Configuration().configure().buildSessionFactory();  
 } catch (Throwable ex) {  
 System.err.println("Failed to create SessionFactory object." + ex);  
 throw new ExceptionInInitializerError(ex);  
 }  
 }  
  
 /\*\*  
 \* Adds a new contact to the database.  
 \* @param contact the Contact object to be saved  
 \*/  
 public void addContact(Contact contact) {  
 Transaction transaction = null;  
 try (Session session = sessionFactory.openSession()) {  
 transaction = session.beginTransaction();  
 session.save(contact); // Persist the contact entity  
 transaction.commit();  
 System.out.println("Contact saved successfully: " + contact.getName());  
 } catch (Exception e) {  
 if (transaction != null) transaction.rollback();  
 e.printStackTrace();  
 }  
 }  
  
 /\*\*  
 \* Retrieves a list of all contacts from the database.  
 \* @return List of Contact objects  
 \*/  
 public List<Contact> listContacts() {  
 try (Session session = sessionFactory.openSession()) {  
 return session.createQuery("FROM Contact", Contact.class).list();  
 } catch (Exception e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
  
 /\*\*  
 \* Updates an existing contact's details in the database.  
 \* @param contact the Contact object containing updated information  
 \*/  
 public void updateContact(Contact contact) {  
 Transaction transaction = null;  
 try (Session session = sessionFactory.openSession()) {  
 transaction = session.beginTransaction();  
 session.update(contact); // Update the contact entity  
 transaction.commit();  
 System.out.println("Contact updated successfully: " + contact.getName());  
 } catch (Exception e) {  
 if (transaction != null) transaction.rollback();  
 e.printStackTrace();  
 }  
 }  
  
 /\*\*  
 \* Deletes the contact with the specified ID from the database.  
 \* @param contactId the id of the contact to delete  
 \*/  
 public void deleteContact(int contactId) {  
 Transaction transaction = null;  
 try (Session session = sessionFactory.openSession()) {  
 transaction = session.beginTransaction();  
 Contact contact = session.get(Contact.class, contactId);  
 if (contact != null) {  
 session.delete(contact); // Remove the contact entity  
 System.out.println("Contact deleted successfully: " + contact.getName());  
 } else {  
 System.out.println("Contact with ID " + contactId + " not found.");  
 }  
 transaction.commit();  
 } catch (Exception e) {  
 if (transaction != null) transaction.rollback();  
 e.printStackTrace();  
 }  
 }  
  
 /\*\*  
 \* Closes the SessionFactory, releasing all resources.  
 \* Call this method when the application is shutting down.  
 \*/  
 public static void shutdown() {  
 if (sessionFactory != null) {  
 sessionFactory.close();  
 }  
 }  
}

## Step 6: Creating the Main Class (Main.java) to Test CRUD Operations

### Creating the Main.java Class

Inside the src/com/contactmanager directory, create a new Java class named Main and add the following sample code:

package com.contactmanager;  
  
import java.util.List;  
  
public class Main {  
  
 public static void main(String[] args) {  
 ContactDAO contactDAO = new ContactDAO();  
  
 // 1. Add sample contacts  
 System.out.println("Adding contacts...");  
 Contact contact1 = new Contact();  
 contact1.setName("Alice Johnson");  
 contact1.setEmail("alice.johnson@example.com");  
 contact1.setPhone("123-456-7890");  
 contact1.setAddress("123 Maple Street, Springfield");  
 contactDAO.addContact(contact1);  
  
 Contact contact2 = new Contact();  
 contact2.setName("Bob Smith");  
 contact2.setEmail("bob.smith@example.com");  
 contact2.setPhone("555-987-6543");  
 contact2.setAddress("456 Oak Avenue, Metropolis");  
 contactDAO.addContact(contact2);  
  
 // 2. List all contacts  
 System.out.println("\nListing all contacts:");  
 List<Contact> contacts = contactDAO.listContacts();  
 if (contacts != null) {  
 for (Contact c : contacts) {  
 System.out.println(  
 "ID: " + c.getId() +  
 ", Name: " + c.getName() +  
 ", Email: " + c.getEmail() +  
 ", Phone: " + c.getPhone() +  
 ", Address: " + c.getAddress()  
 );  
 }  
 }  
  
 // 3. Update a contact (change Bob's phone number)  
 System.out.println("\nUpdating contact with ID 2...");  
 Contact contactToUpdate = contacts.stream()  
 .filter(c -> c.getId() == 2)  
 .findFirst()  
 .orElse(null);  
  
 if (contactToUpdate != null) {  
 contactToUpdate.setPhone("555-444-3333");  
 contactDAO.updateContact(contactToUpdate);  
 } else {  
 System.out.println("Contact with ID 2 not found for update.");  
 }  
  
 // 4. Delete a contact by ID (delete contact with ID 1)  
 System.out.println("\nDeleting contact with ID 1...");  
 contactDAO.deleteContact(1);  
  
 // 5. List contacts again to verify changes  
 System.out.println("\nListing all contacts after update and delete:");  
 List<Contact> updatedContacts = contactDAO.listContacts();  
 if (updatedContacts != null) {  
 for (Contact c : updatedContacts) {  
 System.out.println(  
 "ID: " + c.getId() +  
 ", Name: " + c.getName() +  
 ", Email: " + c.getEmail() +  
 ", Phone: " + c.getPhone() +  
 ", Address: " + c.getAddress()  
 );  
 }  
 }  
  
 // 6. Shutdown Hibernate SessionFactory to release resources  
 ContactDAO.shutdown();  
 }  
}