1. a. Develop a Dynamic web project, in that create an index.jsp that takes input from the user(Login Page), create a Bean class LoginBean.java with Setters, Getters and boolean validate().It should validate name and password with values Admin and Admin@123 Creme a controller Servlet that should get the values of index.jsp. Based on the validation forward the request to success.jsp else forward the request to error.jsp.

b. Demonstrate the steps to create Maven Hibernate Dynamic web project.

**Part 1(a): Creating a Dynamic Web Project with JSP, Servlet, and Bean**

**Steps to Develop a Dynamic Web Project**

1. **Set up the Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it something relevant, like LoginApp.
   * Configure the project to use a server (like Apache Tomcat) and Java version 17 as per your environment.
2. **Create**index.jsp**(Login Page):**
   * This page will collect username and password inputs from the user.
   * Use basic HTML form elements inside JSP to gather input.

<!-- index.jsp -->

<html>

<head><title>Login Page</title></head>

<body>

<form action="LoginController" method="post">

<label>Username:</label>

<input type="text" name="username" required /><br>

<label>Password:</label>

<input type="password" name="password" required /><br>

<input type="submit" value="Login" />

</form>

</body>

</html>

**Create**LoginBean.java**:**

* This Bean class will contain private fields for username and password, with their corresponding getter and setter methods.
* It will also contain a boolean validate() method to validate the input.

// LoginBean.java

public class LoginBean {

private String username;

private String password;

// Getters and Setters

public String getUsername() { return username; }

public void setUsername(String username) { this.username = username; }

public String getPassword() { return password; }

public void setPassword(String password) { this.password = password; }

// Validate method

public boolean validate() {

return "Admin".equals(username) && "Admin@123".equals(password);

}

}

**Create the Servlet Controller (**LoginController.java**):**

* This servlet handles requests from index.jsp, instantiates LoginBean, and validates the credentials.
* If credentials are correct, forward the user to success.jsp; otherwise, to error.jsp.

// LoginController.java

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/LoginController")

public class LoginController extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String username = request.getParameter("username");

String password = request.getParameter("password");

// Create and populate LoginBean

LoginBean loginBean = new LoginBean();

loginBean.setUsername(username);

loginBean.setPassword(password);

// Validate login

if (loginBean.validate()) {

request.getRequestDispatcher("success.jsp").forward(request, response);

} else {

request.getRequestDispatcher("error.jsp").forward(request, response);

}

}

}

**Create**success.jsp**and**error.jsp**:**

* These pages display feedback to the user based on login success or failure.

<!-- success.jsp -->

<html>

<head><title>Login Success</title></head>

<body>

<h2>Welcome, Admin!</h2>

</body>

</html>

<!-- error.jsp -->

<html>

<head><title>Login Error</title></head>

<body>

<h2>Invalid credentials, please try again.</h2>

</body>

</html>

**Run and Test the Application:**

* Deploy the project on your server and navigate to index.jsp.

**Part 1(b): Steps to Create a Maven Hibernate Dynamic Web Project**

**To integrate Hibernate with a Maven-based dynamic web project, follow these steps:**

1. **Create a Maven Project in Eclipse/IDE:**
   * **Open Eclipse and choose File > New > Maven Project.**
   * **Select an appropriate archetype, such as maven-archetype-webapp, and name your project, e.g., HibernateApp.**
2. **Set Up the pom.xml File:**
   * **In the Maven project, open pom.xml and add dependencies for Hibernate, MySQL connector (or your chosen database), and optionally, the Servlet API.**

**<!-- pom.xml -->**

**<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>com.example</groupId>**

**<artifactId>HibernateApp</artifactId>**

**<version>1.0-SNAPSHOT</version>**

**<packaging>war</packaging>**

**<dependencies>**

**<!-- Hibernate Dependency -->**

**<dependency>**

**<groupId>org.hibernate</groupId>**

**<artifactId>hibernate-core</artifactId>**

**<version>5.6.10.Final</version>**

**</dependency>**

**<!-- MySQL Connector Dependency -->**

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**<version>8.0.30</version>**

**</dependency>**

**<!-- Servlet API Dependency -->**

**<dependency>**

**<groupId>javax.servlet</groupId>**

**<artifactId>javax.servlet-api</artifactId>**

**<version>4.0.1</version>**

**<scope>provided</scope>**

**</dependency>**

**</dependencies>**

**</project>**

**Configure hibernate.cfg.xml for Database Connection:**

* **In src/main/resources, create hibernate.cfg.xml to set up the database connection properties, dialect, and entity mappings.**

**<!-- hibernate.cfg.xml -->**

**<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"**

**"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">**

**<hibernate-configuration>**

**<session-factory>**

**<!-- JDBC Database Connection Settings -->**

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

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

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

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

**<!-- Hibernate Settings -->**

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

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

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

**</session-factory>**

**</hibernate-configuration>**

**Create an Entity Class:**

* **Define a simple entity, such as User.java, to map to a database table.**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**@Entity**

**public class User {**

**@Id**

**private int id;**

**private String username;**

**private String password;**

**// Getters and Setters**

**}**

**Create a Hibernate Utility Class:**

* **Implement HibernateUtil.java to manage session factory creation.**

**import org.hibernate.SessionFactory;**

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

**public class HibernateUtil {**

**private static final SessionFactory sessionFactory = buildSessionFactory();**

**private static SessionFactory buildSessionFactory() {**

**try {**

**return new Configuration().configure().buildSessionFactory();**

**} catch (Throwable ex) {**

**throw new ExceptionInInitializerError(ex);**

**}**

**}**

**public static SessionFactory getSessionFactory() {**

**return sessionFactory;**

**}**

**}**

**Part 2(a): Servlet Application to Display Server Information**

This servlet application will retrieve and display details about the server (server name, port number, protocol, and scheme) in the response.

**Steps to Create the Servlet Application**

1. **Create a New Dynamic Web Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it something like ServerInfoApp.
   * Make sure the project is configured with your server, such as Apache Tomcat, and Java version 17.
2. **Create ServerInfoServlet.java:**
   * This servlet will handle requests and respond with server information.
   * Use the HttpServletRequest object to retrieve details about the server and protocol.

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/ServerInfoServlet")

public class ServerInfoServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Get server information

String serverName = request.getServerName();

int serverPort = request.getServerPort();

String protocol = request.getProtocol();

String scheme = request.getScheme();

// Prepare the response

response.setContentType("text/html");

response.getWriter().append("<html><body>");

response.getWriter().append("<h1>Server Information</h1>");

response.getWriter().append("<p>Server Name: " + serverName + "</p>");

response.getWriter().append("<p>Port Number: " + serverPort + "</p>");

response.getWriter().append("<p>Protocol: " + protocol + "</p>");

response.getWriter().append("<p>Scheme: " + scheme + "</p>");

response.getWriter().append("</body></html>");

}

}

**Configure web.xml (Optional):**

* If your IDE doesn’t automatically detect annotations, configure the servlet in web.xml.

 <!-- web.xml -->

<servlet>

<servlet-name>ServerInfoServlet</servlet-name>

<servlet-class>ServerInfoServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ServerInfoServlet</servlet-name>

<url-pattern>/ServerInfoServlet</url-pattern>

</servlet-mapping>

 **Run and Test the Servlet:**

* Deploy the application on your server, navigate to http://localhost:8080/ServerInfoApp/ServerInfoServlet, and it should display server information like server name, port, protocol, and scheme.

**Part 2(b): Hibernate Maven Project to Persist Course Objects**

This Hibernate project will manage Course objects, persisting them in the database and displaying the persisted records in the console.

**Steps to Create the Hibernate Maven Project**

1. **Create a Maven Project:**
   * Open your IDE and choose **File > New > Maven Project**.
   * Select the maven-archetype-quickstart archetype for a basic project structure.
   * Name the project CourseApp.
2. **Configure pom.xml with Hibernate and MySQL Dependencies:**
   * Add dependencies for Hibernate and MySQL.

 <!-- pom.xml -->

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>CourseApp</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- Hibernate Dependency -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.10.Final</version>

</dependency>

<!-- MySQL Connector Dependency -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.30</version>

</dependency>

</dependencies>

</project>

 **Configure Hibernate with hibernate.cfg.xml:**

* In src/main/resources, create hibernate.cfg.xml with your database connection details.

 <!-- hibernate.cfg.xml -->

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

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

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

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

<property name="hibernate.connection.password">password</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

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

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

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

</session-factory>

</hibernate-configuration>

 **Create the Course Entity Class:**

* Define a Course class to represent the course table in the database.

 // Course.java

package com.example;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

private int duration;

private double fee;

// Constructors, Getters, and Setters

public Course() {}

public Course(String name, int duration, double fee) {

this.name = name;

this.duration = duration;

this.fee = fee;

}

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public int getDuration() { return duration; }

public void setDuration(int duration) { this.duration = duration; }

public double getFee() { return fee; }

public void setFee(double fee) { this.fee = fee; }

@Override

public String toString() {

return "Course [id=" + id + ", name=" + name + ", duration=" + duration + " months, fee=" + fee + "]";

}

}

 **Create HibernateUtil.java for Managing SessionFactory:**

* A utility class to create and manage the Hibernate SessionFactory.

 // HibernateUtil.java

package com.example;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class HibernateUtil {

private static final SessionFactory sessionFactory = buildSessionFactory();

private static SessionFactory buildSessionFactory() {

try {

return new Configuration().configure().buildSessionFactory();

} catch (Throwable ex) {

throw new ExceptionInInitializerError(ex);

}

}

public static SessionFactory getSessionFactory() {

return sessionFactory;

}

}

 **Create Main.java to Persist and Retrieve Courses:**

* This main class will create and save Course objects, then display them.

 // Main.java

package com.example;

import org.hibernate.Session;

import org.hibernate.Transaction;

public class Main {

public static void main(String[] args) {

// Get SessionFactory and open a session

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = session.beginTransaction();

// Persist Course objects

Course course1 = new Course("Java Basics", 3, 1500.0);

Course course2 = new Course("Spring Boot", 2, 2000.0);

session.save(course1);

session.save(course2);

// Commit the transaction

transaction.commit();

// Retrieve and display all courses

System.out.println("List of Courses:");

for (Course course : session.createQuery("from Course", Course.class).list()) {

System.out.println(course);

}

// Close session and SessionFactory

session.close();

HibernateUtil.getSessionFactory().close();

}

}

 **Run the Application:**

* Run Main.java to insert Course records into the database and display them in the console.

**Part 3(a): Servlet Application to Retrieve HTTP Request Header Information**

This servlet application will retrieve and display the HTTP request headers such as header names and their corresponding values.

**Steps to Create the Servlet Application**

1. **Create a New Dynamic Web Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it something like HeaderInfoApp.
   * Ensure that the project is configured with your server (Apache Tomcat, for example).
2. **Create HeaderInfoServlet.java:**
   * This servlet will handle requests and extract headers from the HTTP request.

 import java.io.IOException;

import java.util.Enumeration;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/HeaderInfoServlet")

public class HeaderInfoServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Retrieve all header names from the request

Enumeration<String> headerNames = request.getHeaderNames();

// Set content type for response

response.setContentType("text/html");

// Write response in HTML format

response.getWriter().append("<html><body>");

response.getWriter().append("<h1>HTTP Request Headers</h1>");

// Loop through header names and display header values

while (headerNames.hasMoreElements()) {

String headerName = headerNames.nextElement();

String headerValue = request.getHeader(headerName);

response.getWriter().append("<p>" + headerName + ": " + headerValue + "</p>");

}

response.getWriter().append("</body></html>");

}

}

 **Configure web.xml (Optional):**

* If needed, configure the servlet in web.xml (if annotations are not enabled).

1. <!-- web.xml -->
2. <servlet>
3. <servlet-name>HeaderInfoServlet</servlet-name>
4. <servlet-class>HeaderInfoServlet</servlet-class>
5. </servlet>
6. <servlet-mapping>
7. <servlet-name>HeaderInfoServlet</servlet-name>
8. <url-pattern>/HeaderInfoServlet</url-pattern>
9. </servlet-mapping>
10. **Run and Test the Servlet:**
    * Deploy the application on your server, navigate to http://localhost:8080/HeaderInfoApp/HeaderInfoServlet, and it should display HTTP header names and values.

**Part 3(b): Hibernate Maven Dynamic Web Project to Persist Faculty Objects and Display Them in a Table**

This Hibernate project will manage Faculty objects, persisting them in a database and displaying them in a web-based table.

**Steps to Create the Hibernate Maven Project**

1. **Create a Maven Project:**
   * Open your IDE and choose **File > New > Maven Project**.
   * Select the maven-archetype-webapp archetype for a web project structure.
   * Name the project FacultyApp.
2. **Configure pom.xml with Hibernate and MySQL Dependencies:**
   * Add dependencies for Hibernate, MySQL, and JSTL (for displaying tables in JSP).

 <!-- pom.xml -->

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>FacultyApp</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<!-- Hibernate Core Dependency -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.10.Final</version>

</dependency>

<!-- MySQL Connector Dependency -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.30</version>

</dependency>

<!-- JSTL Dependency for JSP -->

<dependency>

<groupId>javax.servlet.jsp.jstl</groupId>

<artifactId>javax.servlet.jsp.jstl-api</artifactId>

<version>1.2.1</version>

</dependency>

</dependencies>

</project>

 **Configure Hibernate with hibernate.cfg.xml:**

* In src/main/resources, create hibernate.cfg.xml with the database connection details.

 <!-- hibernate.cfg.xml -->

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

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

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

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

<property name="hibernate.connection.password">password</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

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

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

<mapping class="com.example.Faculty" />

</session-factory>

</hibernate-configuration>

 **Create the Faculty Entity Class:**

* Define a Faculty class with id, name, department, and email properties.

 // Faculty.java

package com.example;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Faculty {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

private String department;

private String email;

// Constructors, Getters, and Setters

public Faculty() {}

public Faculty(String name, String department, String email) {

this.name = name;

this.department = department;

this.email = email;

}

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

@Override

public String toString() {

return "Faculty [id=" + id + ", name=" + name + ", department=" + department + ", email=" + email + "]";

}

}

 **Create FacultyUtil.java for Managing Hibernate Sessions:**

* A utility class to create and manage the Hibernate SessionFactory.

 // FacultyUtil.java

package com.example;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class FacultyUtil {

private static final SessionFactory sessionFactory = buildSessionFactory();

private static SessionFactory buildSessionFactory() {

try {

return new Configuration().configure().addAnnotatedClass(Faculty.class).buildSessionFactory();

} catch (Throwable ex) {

throw new ExceptionInInitializerError(ex);

}

}

public static SessionFactory getSessionFactory() {

return sessionFactory;

}

}

 **Create FacultyServlet.java to Handle CRUD Operations:**

* This servlet will handle the display and insertion of Faculty records.

 // FacultyServlet.java

package com.example;

import org.hibernate.Session;

import org.hibernate.Transaction;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.util.List;

@WebServlet("/FacultyServlet")

public class FacultyServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Get the current session and start a transaction

Session session = FacultyUtil.getSessionFactory().openSession();

Transaction transaction = session.beginTransaction();

// Retrieve all faculty records

List<Faculty> faculties = session.createQuery("from Faculty", Faculty.class).list();

// Close the session

session.close();

// Set the faculty list as a request attribute

request.setAttribute("faculties", faculties);

// Forward the request to the JSP page

request.getRequestDispatcher("/facultyList.jsp").forward(request, response);

}

}

 **Create facultyList.jsp to Display the Faculty Records in a Table:**

* This JSP page will display the faculty records in an HTML table.

 <!-- facultyList.jsp -->

<html>

<body>

<h2>Faculty List</h2>

<table border="1">

<tr>

<th>ID</th>

<th>Name</th>

<th>Department</th>

<th>Email</th>

</tr>

<c:forEach var="faculty" items="${faculties}">

<tr>

<td>${faculty.id}</td>

<td>${faculty.name}</td>

<td>${faculty.department}</td>

<td>${faculty.email}</td>

</tr>

</c:forEach>

</table>

</body>

</html>

 **Run the Application:**

* Deploy the application and access the servlet at http://localhost:8080/FacultyApp/FacultyServlet to view the table of faculty records.

4. a. Develop a Servlet application which will keep on updating the time in every second. b. Develop a Spring Boot application to persist the Faculty objects and test the APIs save and fetch in Postman App

**ChatGPT said:**

ChatGPT

Here’s a detailed solution for question 4, covering both parts a and b:

**Part 4(a): Servlet Application to Update Time Every Second**

This Servlet application will keep updating the time every second and display it on the web page.

**Steps to Create the Servlet Application**

1. **Create a New Dynamic Web Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it TimeUpdateApp.
   * Make sure to add a servlet container like **Apache Tomcat**.
2. **Create TimeUpdateServlet.java:**
   * This servlet will handle displaying the current time and keep refreshing every second.

 import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.text.SimpleDateFormat;

import java.util.Date;

@WebServlet("/TimeUpdateServlet")

public class TimeUpdateServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

// Get current time

SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss");

String currentTime = formatter.format(new Date());

// Write the current time to the response

response.getWriter().append("<html><body>");

response.getWriter().append("<h1>Current Time: " + currentTime + "</h1>");

// Add a meta-refresh tag to update the time every second

response.getWriter().append("<meta http-equiv='refresh' content='1'>");

response.getWriter().append("</body></html>");

}

}

 **Configure web.xml (Optional):**

* If needed, configure the servlet in web.xml (if annotations are not enabled).

1. <servlet>
2. <servlet-name>TimeUpdateServlet</servlet-name>
3. <servlet-class>TimeUpdateServlet</servlet-class>
4. </servlet>
5. <servlet-mapping>
6. <servlet-name>TimeUpdateServlet</servlet-name>
7. <url-pattern>/TimeUpdateServlet</url-pattern>
8. </servlet-mapping>
9. **Run and Test the Servlet:**
   * Deploy the application on your server, navigate to http://localhost:8080/TimeUpdateApp/TimeUpdateServlet, and the page should display the current time, refreshing every second.

**Part 4(b): Spring Boot Application to Persist Faculty Objects and Test APIs in Postman**

In this part, you will create a Spring Boot application to persist Faculty objects and test the save and fetch APIs using Postman.

**Steps to Create the Spring Boot Application**

1. **Create a New Spring Boot Project:**
   * You can create a Spring Boot project using Spring Initializr (<https://start.spring.io/>) or directly from your IDE.
   * Add dependencies for **Spring Web**, **Spring Data JPA**, and **H2 Database** (or **MySQL** if you prefer).

 <!-- pom.xml -->

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

</dependencies>

 **Create the Faculty Entity:**

* Define a Faculty class with properties such as id, name, department, and email.

 // Faculty.java

package com.example.facultyapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Faculty {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String department;

private String email;

public Faculty() {}

public Faculty(String name, String department, String email) {

this.name = name;

this.department = department;

this.email = email;

}

// Getters and Setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

 **Create a FacultyRepository Interface:**

* This interface will extend JpaRepository for CRUD operations.

 // FacultyRepository.java

package com.example.facultyapp.repository;

import com.example.facultyapp.model.Faculty;

import org.springframework.data.jpa.repository.JpaRepository;

public interface FacultyRepository extends JpaRepository<Faculty, Long> {}

 **Create a FacultyController to Handle API Requests:**

* This controller will handle the save and fetch operations.

 // FacultyController.java

package com.example.facultyapp.controller;

import com.example.facultyapp.model.Faculty;

import com.example.facultyapp.repository.FacultyRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/faculty")

public class FacultyController {

@Autowired

private FacultyRepository facultyRepository;

@PostMapping("/save")

public Faculty saveFaculty(@RequestBody Faculty faculty) {

return facultyRepository.save(faculty);

}

@GetMapping("/all")

public List<Faculty> getAllFaculties() {

return facultyRepository.findAll();

}

}

 **Configure application.properties:**

* Set up database configuration (Here, we use H2 for simplicity, but you can use MySQL by changing the URL, username, and password).

 # application.properties

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true

 **Run the Application:**

* Run your Spring Boot application using the main method in your Application.java class.

 // Application.java

package com.example.facultyapp;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

 **Test the APIs Using Postman:**

* **Save a Faculty Object** (POST Request):
  + Method: POST
  + URL: http://localhost:8080/api/faculty/save
  + Body (JSON):
    - {
    - "name": "Dr. John Doe",
    - "department": "Computer Science",
    - "email": "johndoe@example.com"
    - }
    - The response should be the saved Faculty object with a generated ID.
  + **Fetch All Faculty Objects** (GET Request):
    - Method: GET
    - URL: http://localhost:8080/api/faculty/all
    - The response should be a list of all Faculty objects.

This solution provides a detailed Spring Boot application for persisting Faculty objects and testing the save and fetch APIs in Postman.

**You said:**

5. a. Develop a Servlet application to retrieve init parameters values which we have given in the web.xml file. b. Develop a Spring Boot application to persist the Course objects and test the API fetch using Thymeleaf template engine.

**ChatGPT said:**

ChatGPT

**Part 5(a): Servlet Application to Retrieve Init Parameters from web.xml**

In this part, we will create a servlet that retrieves initialization parameters that are defined in the web.xml file. These parameters can be used to configure the servlet during deployment.

**Steps to Create the Servlet Application**

1. **Create a New Dynamic Web Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it InitParameterApp.
   * Add the necessary libraries for the servlet (e.g., Apache Tomcat).
2. **Define Init Parameters in web.xml:**
   * You can define init parameters for your servlet inside the web.xml file.

 <!-- web.xml -->

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

<servlet>

<servlet-name>InitParamServlet</servlet-name>

<servlet-class>InitParamServlet</servlet-class>

<init-param>

<param-name>username</param-name>

<param-value>admin</param-value>

</init-param>

<init-param>

<param-name>password</param-name>

<param-value>admin123</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>InitParamServlet</servlet-name>

<url-pattern>/InitParamServlet</url-pattern>

</servlet-mapping>

</web-app>

 **Create the InitParamServlet.java:**

* This servlet will retrieve the init parameters from the web.xml file and display them on the web page.

1. import javax.servlet.ServletException;
2. import javax.servlet.annotation.WebServlet;
3. import javax.servlet.http.HttpServlet;
4. import javax.servlet.http.HttpServletRequest;
5. import javax.servlet.http.HttpServletResponse;
6. import java.io.IOException;
7. @WebServlet("/InitParamServlet")
8. public class InitParamServlet extends HttpServlet {
9. @Override
10. public void init() throws ServletException {
11. // Get init parameters from the web.xml file
12. String username = getServletConfig().getInitParameter("username");
13. String password = getServletConfig().getInitParameter("password");
14. getServletContext().setAttribute("username", username);
15. getServletContext().setAttribute("password", password);
16. }
17. @Override
18. protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
19. // Retrieve init parameters and display them
20. String username = (String) getServletContext().getAttribute("username");
21. String password = (String) getServletContext().getAttribute("password");
22. // Set content type and print the parameters on the web page
23. response.setContentType("text/html");
24. response.getWriter().println("<html><body>");
25. response.getWriter().println("<h3>Retrieved Init Parameters:</h3>");
26. response.getWriter().println("<p>Username: " + username + "</p>");
27. response.getWriter().println("<p>Password: " + password + "</p>");
28. response.getWriter().println("</body></html>");
29. }
30. }
31. **Run the Application:**
    * Deploy the application on your server (e.g., Tomcat), and access the servlet at http://localhost:8080/InitParameterApp/InitParamServlet.
    * You should see the retrieved init parameters (username and password) displayed on the web page.

**Part 5(b): Spring Boot Application to Persist Course Objects and Test the Fetch API Using Thymeleaf Template Engine**

In this part, we will develop a Spring Boot application to persist Course objects and use Thymeleaf to display them in the UI. We will also test the fetch API using a Thymeleaf template.

**Steps to Create the Spring Boot Application**

1. **Create a New Spring Boot Project:**
   * You can create a Spring Boot project using Spring Initializr (<https://start.spring.io/>) or directly from your IDE.
   * Add dependencies for **Spring Web**, **Spring Data JPA**, **Thymeleaf**, and **H2 Database** (or **MySQL** if you prefer).

 <!-- pom.xml -->

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

 **Create the Course Entity:**

* Define a Course class to represent the course entity, with properties like id, name, and duration.

 // Course.java

package com.example.courseapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String duration;

public Course() {}

public Course(String name, String duration) {

this.name = name;

this.duration = duration;

}

// Getters and Setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDuration() { return duration; }

public void setDuration(String duration) { this.duration = duration; }

}

 **Create a CourseRepository Interface:**

* This interface will handle CRUD operations for Course objects.

 // CourseRepository.java

package com.example.courseapp.repository;

import com.example.courseapp.model.Course;

import org.springframework.data.jpa.repository.JpaRepository;

public interface CourseRepository extends JpaRepository<Course, Long> {}

 **Create a CourseController to Handle API and Thymeleaf Views:**

* The controller will handle requests for both saving and fetching courses, as well as rendering views with Thymeleaf.

 // CourseController.java

package com.example.courseapp.controller;

import com.example.courseapp.model.Course;

import com.example.courseapp.repository.CourseRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.ui.Model;

@Controller

public class CourseController {

@Autowired

private CourseRepository courseRepository;

// Display the list of courses

@GetMapping("/courses")

public String getCourses(Model model) {

model.addAttribute("courses", courseRepository.findAll());

return "courseList";

}

// Add a new course

@PostMapping("/addCourse")

public String addCourse(@RequestParam String name, @RequestParam String duration) {

Course course = new Course(name, duration);

courseRepository.save(course);

return "redirect:/courses";

}

}

 **Create the Thymeleaf Template:**

* Create a courseList.html file in the src/main/resources/templates directory to display the courses.

 <!-- courseList.html -->

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Course List</title>

</head>

<body>

<h1>Course List</h1>

<table border="1">

<tr>

<th>ID</th>

<th>Name</th>

<th>Duration</th>

</tr>

<tr th:each="course : ${courses}">

<td th:text="${course.id}"></td>

<td th:text="${course.name}"></td>

<td th:text="${course.duration}"></td>

</tr>

</table>

<h2>Add New Course</h2>

<form action="/addCourse" method="post">

<label for="name">Course Name:</label>

<input type="text" name="name" required><br>

<label for="duration">Duration:</label>

<input type="text" name="duration" required><br>

<button type="submit">Add Course</button>

</form>

</body>

</html>

 **Configure application.properties:**

* Set up database configuration (H2 database for simplicity in this case).

1. # application.properties
2. spring.datasource.url=jdbc:h2:mem:testdb
3. spring.datasource.driverClassName=org.h2.Driver
4. spring.datasource.username=sa
5. spring.datasource.password=password
6. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
7. spring.h2.console.enabled=true
8. **Run the Application:**
   * Launch the Spring Boot application, and visit http://localhost:8080/courses.
   * You should see the list of courses, and you can add new courses using the form.

This will create a simple Spring Boot application that displays courses using Thymeleaf and allows for the addition of new courses, saving them to the database.

**You said:**

6. a. Develop a Servlet application to find whether the session is pre-existing or not. b. Develop a Spring Boot application to persist the Faculty objects and test the fetch using Thymeleaf template engine.

**ChatGPT said:**

ChatGPT

**Part 6(a): Servlet Application to Find Whether the Session is Pre-existing or Not**

In this part, we will develop a servlet application to check if a session already exists or not. If a session exists, the servlet will retrieve and display a message from the session. If no session exists, it will create a new session.

**Steps to Develop the Servlet Application**

1. **Create a New Dynamic Web Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it SessionCheckApp.
   * Add the necessary libraries for the servlet (e.g., Apache Tomcat).
2. **Create the SessionCheckServlet.java:**
   * This servlet will check if the session exists and either retrieve the session attributes or create a new session.
3. import javax.servlet.ServletException;
4. import javax.servlet.annotation.WebServlet;
5. import javax.servlet.http.HttpServlet;
6. import javax.servlet.http.HttpServletRequest;
7. import javax.servlet.http.HttpServletResponse;
8. import javax.servlet.http.HttpSession;
9. import java.io.IOException;
10. @WebServlet("/SessionCheckServlet")
11. public class SessionCheckServlet extends HttpServlet {
12. @Override
13. protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
14. // Get the current session or create a new one if it doesn't exist
15. HttpSession session = request.getSession(false);
16. if (session == null) {
17. // No session exists, create a new session and set an attribute
18. session = request.getSession(true);
19. session.setAttribute("message", "This is a new session.");
20. response.getWriter().println("No existing session. A new session has been created.");
21. } else {
22. // Session exists, retrieve the attribute
23. String message = (String) session.getAttribute("message");
24. response.getWriter().println("Existing session. Message: " + message);
25. }
26. }
27. }
28. **Run the Application:**
    * Deploy the application on your server (e.g., Tomcat), and access the servlet at http://localhost:8080/SessionCheckApp/SessionCheckServlet.
    * On the first request, you should see a message saying "No existing session. A new session has been created."
    * Refresh the page, and you should see "Existing session. Message: This is a new session."

This will demonstrate how a session is created and retrieved in a servlet.

**Part 6(b): Spring Boot Application to Persist Faculty Objects and Test the Fetch Using Thymeleaf Template Engine**

In this part, we will develop a Spring Boot application to persist Faculty objects and use Thymeleaf to display them in a template. We will also test the fetch API to display the faculty details.

**Steps to Develop the Spring Boot Application**

1. **Create a New Spring Boot Project:**
   * You can create a Spring Boot project using Spring Initializr (<https://start.spring.io/>) or directly from your IDE.
   * Add dependencies for **Spring Web**, **Spring Data JPA**, **Thymeleaf**, and **H2 Database** (or **MySQL** if you prefer).

 <!-- pom.xml -->

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

 **Create the Faculty Entity:**

* Define a Faculty class to represent the faculty entity, with properties like id, name, department, and email.

 // Faculty.java

package com.example.facultyapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Faculty {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String department;

private String email;

public Faculty() {}

public Faculty(String name, String department, String email) {

this.name = name;

this.department = department;

this.email = email;

}

// Getters and Setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

 **Create a FacultyRepository Interface:**

* This interface will handle CRUD operations for Faculty objects.

 // FacultyRepository.java

package com.example.facultyapp.repository;

import com.example.facultyapp.model.Faculty;

import org.springframework.data.jpa.repository.JpaRepository;

public interface FacultyRepository extends JpaRepository<Faculty, Long> {}

 **Create a FacultyController to Handle API and Thymeleaf Views:**

* The controller will handle requests for both saving and fetching faculty details and rendering views with Thymeleaf.

 // FacultyController.java

package com.example.facultyapp.controller;

import com.example.facultyapp.model.Faculty;

import com.example.facultyapp.repository.FacultyRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.ui.Model;

@Controller

public class FacultyController {

@Autowired

private FacultyRepository facultyRepository;

// Display the list of faculty

@GetMapping("/faculties")

public String getFaculties(Model model) {

model.addAttribute("faculties", facultyRepository.findAll());

return "facultyList";

}

// Add a new faculty

@PostMapping("/addFaculty")

public String addFaculty(@RequestParam String name, @RequestParam String department, @RequestParam String email) {

Faculty faculty = new Faculty(name, department, email);

facultyRepository.save(faculty);

return "redirect:/faculties";

}

}

 **Create the Thymeleaf Template:**

* Create a facultyList.html file in the src/main/resources/templates directory to display the list of faculties.

 <!-- facultyList.html -->

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Faculty List</title>

</head>

<body>

<h1>Faculty List</h1>

<table border="1">

<tr>

<th>ID</th>

<th>Name</th>

<th>Department</th>

<th>Email</th>

</tr>

<tr th:each="faculty : ${faculties}">

<td th:text="${faculty.id}"></td>

<td th:text="${faculty.name}"></td>

<td th:text="${faculty.department}"></td>

<td th:text="${faculty.email}"></td>

</tr>

</table>

<h2>Add New Faculty</h2>

<form action="/addFaculty" method="post">

<label for="name">Faculty Name:</label>

<input type="text" name="name" required><br>

<label for="department">Department:</label>

<input type="text" name="department" required><br>

<label for="email">Email:</label>

<input type="text" name="email" required><br>

<button type="submit">Add Faculty</button>

</form>

</body>

</html>

 **Configure application.properties:**

* Set up database configuration (H2 database for simplicity in this case).

1. # application.properties
2. spring.datasource.url=jdbc:h2:mem:testdb
3. spring.datasource.driverClassName=org.h2.Driver
4. spring.datasource.username=sa
5. spring.datasource.password=password
6. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
7. spring.h2.console.enabled=true
8. **Run the Application:**
   * Launch the Spring Boot application, and visit http://localhost:8080/faculties.
   * You should see the list of faculties, and you can add new faculties using the form.

This will create a simple Spring Boot application to persist faculty data and display it using Thymeleaf.

**You said:**

7. a. Develop a Servlet application to validate user details with cookie. b. Develop a Hibernate Maven Dynamic web project to persist Book objects and display them in a table.

**ChatGPT said:**

ChatGPT

**Part 7(a): Servlet Application to Validate User Details with Cookie**

In this part, we will develop a servlet application to validate user login details using cookies. If the user successfully logs in, we will store a cookie with the username; otherwise, we will display an error message.

**Steps to Develop the Servlet Application**

1. **Create a New Dynamic Web Project:**
   * Open your IDE (e.g., Eclipse) and create a new **Dynamic Web Project**. Name it CookieValidationApp.
   * Add the necessary libraries for the servlet (e.g., Apache Tomcat).
2. **Create the LoginServlet.java:**
   * This servlet will validate user login and store the username in a cookie if the login is successful.

 import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@WebServlet("/LoginServlet")

public class LoginServlet extends HttpServlet {

@Override

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String username = request.getParameter("username");

String password = request.getParameter("password");

// Validate user details (for example, username: admin, password: admin@123)

if ("admin".equals(username) && "admin@123".equals(password)) {

// Create a cookie to store the username

Cookie userCookie = new Cookie("username", username);

userCookie.setMaxAge(60 \* 60); // 1 hour

response.addCookie(userCookie);

// Redirect to success page

response.sendRedirect("success.jsp");

} else {

// If credentials are invalid, redirect to error page

response.sendRedirect("error.jsp");

}

}

}

 **Create index.jsp:**

* The login page where the user will enter their username and password.

 <!-- index.jsp -->

<html>

<body>

<h2>Login</h2>

<form action="LoginServlet" method="post">

Username: <input type="text" name="username"><br>

Password: <input type="password" name="password"><br>

<input type="submit" value="Login">

</form>

</body>

</html>

 **Create success.jsp:**

* The page to display when the user logs in successfully.

 <!-- success.jsp -->

<html>

<body>

<h2>Welcome, you have logged in successfully!</h2>

</body>

</html>

 **Create error.jsp:**

* The page to display when the user provides incorrect credentials.

1. <!-- error.jsp -->
2. <html>
3. <body>
4. <h2>Invalid login credentials. Please try again.</h2>
5. </body>
6. </html>
7. **Run the Application:**
   * Deploy the application on your server (e.g., Tomcat) and access the login page at http://localhost:8080/CookieValidationApp/index.jsp.
   * Enter the username "admin" and password "admin@123". If successful, a cookie with the username will be created, and the user will be redirected to the success page. If invalid, they will be redirected to the error page.

**Part 7(b): Hibernate Maven Dynamic Web Project to Persist Book Objects and Display Them in a Table**

In this part, we will develop a Hibernate Maven Dynamic Web project to persist Book objects and display them in a table. The project will use Hibernate for data persistence and JSP/Servlet for displaying the data.

**Steps to Develop the Hibernate Maven Dynamic Web Project**

1. **Create a Maven Project:**
   * Create a Maven project in your IDE (e.g., Eclipse) and add the required dependencies to the pom.xml file for Hibernate, MySQL, and Maven Web.

 <dependencies>

<!-- Hibernate Dependencies -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.4.30.Final</version>

</dependency>

<!-- MySQL Connector -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.23</version>

</dependency>

<!-- Servlet API -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

<!-- JSP and Servlet -->

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-jsp-api</artifactId>

<version>9.0.41</version>

<scope>provided</scope>

</dependency>

</dependencies>

 **Create the Hibernate Configuration File (hibernate.cfg.xml):**

* In the src/main/resources directory, create a hibernate.cfg.xml file to configure Hibernate.

 <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- JDBC Database connection settings -->

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

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

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your\_database</property>

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

<property name="hibernate.connection.password">password</property>

<!-- JDBC connection pool settings -->

<property name="hibernate.c3p0.min\_size">5</property>

<property name="hibernate.c3p0.max\_size">20</property>

<property name="hibernate.c3p0.timeout">300</property>

<!-- Specify dialect -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Enable Hibernate's automatic session context management -->

<property name="hibernate.current\_session\_context\_class">thread</property>

<!-- Echo all executed SQL to stdout -->

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

<!-- Drop and re-create the database schema on startup -->

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

<!-- Disable the second-level cache -->

<property name="hibernate.cache.provider\_class">org.hibernate.cache.NoCacheProvider</property>

</session-factory>

</hibernate-configuration>

 **Create the Book Entity Class:**

 // Book.java

package com.example.bookapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

private double price;

public Book() {}

public Book(String title, String author, double price) {

this.title = title;

this.author = author;

this.price = price;

}

// Getters and setters

public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getAuthor() { return author; }

public void setAuthor(String author) { this.author = author; }

public double getPrice() { return price; }

public void setPrice(double price) { this.price = price; }

}

 **Create a BookDAO Class to Handle Database Operations:**

 // BookDAO.java

package com.example.bookapp.dao;

import com.example.bookapp.model.Book;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import java.util.List;

public class BookDAO {

private SessionFactory factory;

public BookDAO() {

factory = new Configuration()

.configure("hibernate.cfg.xml")

.addAnnotatedClass(Book.class)

.buildSessionFactory();

}

public void saveBook(Book book) {

Session session = factory.getCurrentSession();

session.beginTransaction();

session.save(book);

session.getTransaction().commit();

}

public List<Book> getBooks() {

Session session = factory.getCurrentSession();

session.beginTransaction();

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

session.getTransaction().commit();

return books;

}

}

 **Create the BookServlet to Handle Requests:**

 // BookServlet.java

package com.example.bookapp.servlet;

import com.example.bookapp.dao.BookDAO;

import com.example.bookapp.model.Book;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.util.List;

@WebServlet("/BookServlet")

public class BookServlet extends HttpServlet {

private BookDAO bookDAO;

@Override

public void init() {

bookDAO = new BookDAO();

}

@Override

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

List<Book> books = bookDAO.getBooks();

request.setAttribute("books", books);

request.getRequestDispatcher("/booklist.jsp").forward(request, response);

}

}

 **Create booklist.jsp to Display Books in a Table:**

1. <!-- booklist.jsp -->
2. <html>
3. <body>
4. <h2>Book List</h2>
5. <table border="1">
6. <tr>
7. <th>Title</th>
8. <th>Author</th>
9. <th>Price</th>
10. </tr>
11. <c:forEach var="book" items="${books}">
12. <tr>
13. <td>${book.title}</td>
14. <td>${book.author}</td>
15. <td>${book.price}</td>
16. </tr>
17. </c:forEach>
18. </table>
19. </body>
20. </html>
21. **Run the Application:**
    * Deploy the application on your server and navigate to http://localhost:8080/BookApp/BookServlet to see the list of books stored in your database displayed in a table.

This completes both parts of the servlet and Hibernate application!

**You said:**

8. a. Develop a Dynamic web project, in that create model class Course which consists of three instance variables cid, ename, cdomain with setters, getters and parameterized constructor, create another model class CourseAllocation which consists of a method optional<Course> getCid(courseid), create a Servlet CourseSelection. If cid exists it should display the details cid, cname, cdomain in success.jsp else it should display error.jsp with a message invalid id and again it should include index.jsp. b . Demonstrate the steps to create SpringBoot application.

**ChatGPT said:**

ChatGPT

**Part 8(a): Dynamic Web Project to Handle Course Selection**

In this part, we will create a dynamic web project that contains two model classes (Course and CourseAllocation) and a servlet (CourseSelection) to handle course selection based on a given course ID. If the course ID exists, the details will be displayed; otherwise, an error message will be shown.

**Steps to Implement:**

1. **Create a Dynamic Web Project:**
   * Create a new **Dynamic Web Project** in your IDE (e.g., Eclipse) and name it CourseSelectionApp.
2. **Create the Course Model Class:**

The Course class will represent the course details and contain instance variables cid, ename, and cdomain, along with their setters, getters, and a parameterized constructor.

 // Course.java

package com.example.model;

public class Course {

private String cid;

private String ename;

private String cdomain;

public Course(String cid, String ename, String cdomain) {

this.cid = cid;

this.ename = ename;

this.cdomain = cdomain;

}

// Getters and Setters

public String getCid() { return cid; }

public void setCid(String cid) { this.cid = cid; }

public String getEname() { return ename; }

public void setEname(String ename) { this.ename = ename; }

public String getCdomain() { return cdomain; }

public void setCdomain(String cdomain) { this.cdomain = cdomain; }

}

 **Create the CourseAllocation Model Class:**

The CourseAllocation class will contain a method getCid(courseid) that returns an Optional<Course> to check if the course exists or not.

 // CourseAllocation.java

package com.example.model;

import java.util.HashMap;

import java.util.Map;

import java.util.Optional;

public class CourseAllocation {

// Map to store courses with course id as key

private Map<String, Course> courses;

public CourseAllocation() {

courses = new HashMap<>();

// Adding some sample courses

courses.put("CSE101", new Course("CSE101", "Computer Science", "Engineering"));

courses.put("ECE102", new Course("ECE102", "Electrical Engineering", "Engineering"));

courses.put("IT103", new Course("IT103", "Information Technology", "Engineering"));

}

// Method to fetch course by ID

public Optional<Course> getCid(String courseid) {

return Optional.ofNullable(courses.get(courseid));

}

}

 **Create the CourseSelectionServlet to Handle Course Selection:**

The servlet will take the courseid as input from the user, validate it using the CourseAllocation class, and forward the request to the appropriate JSP based on the result.

 // CourseSelectionServlet.java

package com.example.servlet;

import com.example.model.Course;

import com.example.model.CourseAllocation;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@WebServlet("/CourseSelection")

public class CourseSelectionServlet extends HttpServlet {

private CourseAllocation courseAllocation;

@Override

public void init() {

courseAllocation = new CourseAllocation();

}

@Override

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String courseid = request.getParameter("courseid");

Optional<Course> course = courseAllocation.getCid(courseid);

if (course.isPresent()) {

// Forward to success.jsp with course details

request.setAttribute("course", course.get());

request.getRequestDispatcher("success.jsp").forward(request, response);

} else {

// Forward to error.jsp with invalid ID message

request.setAttribute("errorMessage", "Invalid Course ID");

request.getRequestDispatcher("error.jsp").forward(request, response);

}

}

}

 **Create index.jsp:**

The index.jsp page will allow the user to input a courseid and submit it to the servlet.

 <!-- index.jsp -->

<html>

<body>

<h2>Course Selection</h2>

<form action="CourseSelection" method="post">

Course ID: <input type="text" name="courseid"><br>

<input type="submit" value="Select Course">

</form>

</body>

</html>

 **Create success.jsp:**

The success.jsp page will display the details of the selected course.

 <!-- success.jsp -->

<html>

<body>

<h2>Course Details</h2>

<p>Course ID: ${course.cid}</p>

<p>Course Name: ${course.ename}</p>

<p>Course Domain: ${course.cdomain}</p>

</body>

</html>

 **Create error.jsp:**

The error.jsp page will display an error message when the user enters an invalid course ID.

1. <!-- error.jsp -->
2. <html>
3. <body>
4. <h2>${errorMessage}</h2>
5. <a href="index.jsp">Try Again</a>
6. </body>
7. </html>
8. **Run the Application:**
   * Deploy the application on your server (e.g., Tomcat) and access the page at http://localhost:8080/CourseSelectionApp/index.jsp.
   * If you enter a valid course ID (like "CSE101"), the details of the course will be displayed in success.jsp. If the course ID is invalid, you will see the error message in error.jsp and be able to return to index.jsp.

**Part 8(b): Demonstrating the Steps to Create a Spring Boot Application**

In this part, we will go through the steps to create a Spring Boot application, which is an easy and quick way to set up a Java application with minimal configuration.

**Steps to Create a Spring Boot Application:**

1. **Create a New Spring Boot Project:**
   * Go to [Spring Initializr](https://start.spring.io/).
   * Select the following:
     + Project: Maven
     + Language: Java
     + Spring Boot: Choose the latest version
     + Group: com.example
     + Artifact: SpringBootDemo
     + Dependencies: Add **Spring Web** and **Spring Boot DevTools** for quick reloading.
   * Click on **Generate**, and it will download a .zip file. Extract it to your workspace.
2. **Import the Project into IDE:**
   * Open your IDE (e.g., IntelliJ IDEA or Eclipse) and import the downloaded project.
3. **Create the Main Application Class:**

 // SpringBootDemoApplication.java

package com.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringBootDemoApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootDemoApplication.class, args);

}

}

The @SpringBootApplication annotation is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.

 **Create a Controller Class:**

1. // HelloController.java
2. package com.example.controller;
3. import org.springframework.web.bind.annotation.GetMapping;
4. import org.springframework.web.bind.annotation.RestController;
5. @RestController
6. public class HelloController {
7. @GetMapping("/hello")
8. public String hello() {
9. return "Hello, Spring Boot!";
10. }
11. }
12. The @RestController annotation indicates that the class will handle HTTP requests and return data (usually JSON or plain text).
13. **Run the Spring Boot Application:**
    * Right-click on the SpringBootDemoApplication.java class and choose **Run**.
    * Spring Boot will start an embedded Tomcat server on port 8080 by default.
    * Open a browser and go to http://localhost:8080/hello. You should see the message: Hello, Spring Boot!.
14. **Test the Application:**
    * You can also test the application using Postman or any other API testing tool by sending a GET request to http://localhost:8080/hello.

These are the steps to create a dynamic web project with servlet handling course selection and to demonstrate creating a Spring Boot application.

**You said:**

9. a. Develop a Servlet application to print current date & time. b. Develop a Java Hibernate project to persist Student objects and display them in the console.

**ChatGPT said:**

ChatGPT

**Part 9(a): Servlet Application to Print Current Date & Time**

In this part, we'll develop a simple servlet application that prints the current date and time.

**Steps to Implement:**

1. **Create a Dynamic Web Project:**
   * Create a new **Dynamic Web Project** (e.g., DateTimeServletApp) in your IDE (e.g., Eclipse or IntelliJ IDEA).
2. **Create the DateTimeServlet Servlet Class:** This servlet will display the current date and time.
3. // DateTimeServlet.java
4. package com.example.servlet;
5. import javax.servlet.ServletException;
6. import javax.servlet.annotation.WebServlet;
7. import javax.servlet.http.HttpServlet;
8. import javax.servlet.http.HttpServletRequest;
9. import javax.servlet.http.HttpServletResponse;
10. import java.io.IOException;
11. import java.text.SimpleDateFormat;
12. import java.util.Date;
13. @WebServlet("/DateTimeServlet")
14. public class DateTimeServlet extends HttpServlet {
15. @Override
16. protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
17. response.setContentType("text/html");
18. // Get current date and time
19. Date currentDate = new Date();
20. SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
21. // Format the date
22. String formattedDate = dateFormat.format(currentDate);
23. // Write the response
24. response.getWriter().write("<html><body>");
25. response.getWriter().write("<h2>Current Date & Time:</h2>");
26. response.getWriter().write("<p>" + formattedDate + "</p>");
27. response.getWriter().write("</body></html>");
28. }
29. }
    * The DateTimeServlet class uses SimpleDateFormat to format the current date and time. When a GET request is made, it responds with an HTML page displaying the formatted date and time.
30. **Deploy and Test the Application:**
    * Deploy this servlet on your server (e.g., Tomcat).
    * Navigate to http://localhost:8080/DateTimeServletApp/DateTimeServlet to view the current date and time.

**Part 9(b): Hibernate Project to Persist Student Objects and Display Them in the Console**

In this part, we will create a simple Hibernate project to persist Student objects and display the saved data in the console.

**Steps to Implement:**

1. **Create a Maven Project:**
   * Create a new Maven project in your IDE and add the necessary Hibernate and MySQL dependencies in the pom.xml file.

 <dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.0.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.26</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.32</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.32</version>

</dependency>

</dependencies>

 **Create the Student Model Class:** This class represents the Student entity that will be persisted in the database.

 // Student.java

package com.example.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

@Entity

public class Student {

@Id

@GeneratedValue

private int id;

private String name;

private int age;

// Constructors, Getters and Setters

public Student() {}

public Student(String name, int age) {

this.name = name;

this.age = age;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

 **Configure Hibernate (hibernate.cfg.xml):** Configure the Hibernate session factory by creating a hibernate.cfg.xml file in the src/main/resources directory.

 <?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>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

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

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

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

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

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

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

<property name="hibernate.connection.password">password</property>

</session-factory>

</hibernate-configuration>

* Make sure you have created a MySQL database hibernate\_db and replace username and password with your actual database credentials.

 **Create a Hibernate Utility Class:** This class will help to get the Hibernate session factory.

 // HibernateUtil.java

package com.example.util;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class HibernateUtil {

private static final SessionFactory sessionFactory;

static {

try {

// Create SessionFactory from hibernate.cfg.xml

sessionFactory = new Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Student.class).buildSessionFactory();

} catch (Exception e) {

e.printStackTrace();

throw new ExceptionInInitializerError("Initial SessionFactory creation failed" + e);

}

}

public static SessionFactory getSessionFactory() {

return sessionFactory;

}

}

 **Create the StudentDAO Class:** This class will be responsible for saving and retrieving Student objects from the database.

 // StudentDAO.java

package com.example.dao;

import com.example.model.Student;

import com.example.util.HibernateUtil;

import org.hibernate.Session;

import org.hibernate.Transaction;

public class StudentDAO {

public void saveStudent(Student student) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

session.save(student);

transaction.commit();

} catch (Exception e) {

if (transaction != null) {

transaction.rollback();

}

e.printStackTrace();

} finally {

session.close();

}

}

public void getStudents() {

Session session = HibernateUtil.getSessionFactory().openSession();

try {

List<Student> students = session.createQuery("from Student", Student.class).getResultList();

for (Student student : students) {

System.out.println(student.getId() + ": " + student.getName() + ", " + student.getAge());

}

} catch (Exception e) {

e.printStackTrace();

} finally {

session.close();

}

}

}

 **Main Class to Test Saving and Fetching Students:**

1. // Main.java
2. package com.example;
3. import com.example.dao.StudentDAO;
4. import com.example.model.Student;
5. public class Main {
6. public static void main(String[] args) {
7. StudentDAO studentDAO = new StudentDAO();
8. // Save a new student
9. Student student = new Student("John Doe", 22);
10. studentDAO.saveStudent(student);
11. // Fetch all students and print to the console
12. studentDAO.getStudents();
13. }
14. }
15. **Run the Project:**
    * Run the Main.java class to save a Student object and fetch all students from the database.
    * In the console, you should see the saved student's details.

**Conclusion:**

* **Part 9(a):** The servlet application prints the current date and time in an HTML page.
* **Part 9(b):** The Hibernate application persists Student objects into a MySQL database and displays them in the console.

Make sure you have the necessary MySQL database (hibernate\_db) set up and configured correctly in the hibernate.cfg.xml file.

**You said:**

10. a. Develop a Java application to create new database "CSE", create a table "Student" under the database and insert the details into student table. b. Develop a Spring Boot application to persist the Book objects and test the APIs save and fetchById in Postman App.

**ChatGPT said:**

ChatGPT

**Part 10(a): Java Application to Create a New Database, Table, and Insert Data**

In this part, we'll develop a Java application to:

1. Create a new database CSE.
2. Create a Student table under the CSE database.
3. Insert data into the Student table.

**Steps to Implement:**

1. **Set Up MySQL:** Ensure that you have MySQL running on your machine and have access to it.
2. **Create a Java Application to Connect to MySQL:**

We'll use JDBC to connect to the MySQL server and create the database, table, and insert records.

**Database Setup in MySQL:**

* + Database: CSE
  + Table: Student
  + Columns: id, name, age

1. **JDBC Code to Create Database, Table, and Insert Data:**
2. import java.sql.Connection;
3. import java.sql.DriverManager;
4. import java.sql.Statement;
5. import java.sql.ResultSet;
6. public class DatabaseApp {
7. public static void main(String[] args) {
8. // MySQL JDBC connection details
9. String url = "jdbc:mysql://localhost:3306/";
10. String username = "root";
11. String password = "password"; // Replace with your MySQL password
12. Connection connection = null;
13. Statement statement = null;
14. try {
15. // Load MySQL JDBC Driver
16. Class.forName("com.mysql.cj.jdbc.Driver");
17. // Establish connection to MySQL server
18. connection = DriverManager.getConnection(url, username, password);
19. statement = connection.createStatement();
20. // Step 1: Create the "CSE" database
21. String createDatabase = "CREATE DATABASE IF NOT EXISTS CSE";
22. statement.executeUpdate(createDatabase);
23. System.out.println("Database 'CSE' created successfully!");
24. // Step 2: Use the "CSE" database
25. String useDatabase = "USE CSE";
26. statement.executeUpdate(useDatabase);
27. // Step 3: Create the "Student" table
28. String createTable = "CREATE TABLE IF NOT EXISTS Student (" +
29. "id INT AUTO\_INCREMENT PRIMARY KEY, " +
30. "name VARCHAR(100), " +
31. "age INT)";
32. statement.executeUpdate(createTable);
33. System.out.println("Table 'Student' created successfully!");
34. // Step 4: Insert data into the "Student" table
35. String insertData = "INSERT INTO Student (name, age) VALUES ('John Doe', 22), ('Jane Smith', 21)";
36. statement.executeUpdate(insertData);
37. System.out.println("Data inserted successfully into 'Student' table!");
38. // Step 5: Query the table to display the inserted records
39. String selectData = "SELECT \* FROM Student";
40. ResultSet resultSet = statement.executeQuery(selectData);
41. while (resultSet.next()) {
42. System.out.println("ID: " + resultSet.getInt("id") + ", Name: " + resultSet.getString("name") + ", Age: " + resultSet.getInt("age"));
43. }
44. } catch (Exception e) {
45. e.printStackTrace();
46. } finally {
47. try {
48. if (statement != null) statement.close();
49. if (connection != null) connection.close();
50. } catch (Exception e) {
51. e.printStackTrace();
52. }
53. }
54. }
55. }

**Explanation:**

1. **JDBC Connection**: The connection is established using DriverManager.getConnection by providing the URL, username, and password.
2. **Database and Table Creation**: The CREATE DATABASE and CREATE TABLE SQL queries are executed to create the database CSE and the Student table.
3. **Inserting Data**: The INSERT INTO query is used to insert sample student data.
4. **Fetching Data**: A SELECT query retrieves and displays all records from the Student table.

**Output:**

* The program creates the CSE database, creates the Student table, inserts data, and retrieves the records in the console.

**Part 10(b): Spring Boot Application to Persist Book Objects and Test APIs Using Postman**

In this part, we will develop a Spring Boot application to persist Book objects and test the APIs (save and fetchById) using Postman.

**Steps to Implement:**

1. **Set Up Spring Boot Project:** Create a Spring Boot application using Spring Initializr or manually using your IDE.
2. **Add Dependencies to pom.xml:** Add the necessary dependencies for Spring Boot, Spring Data JPA, and MySQL.

 <dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.26</version>

</dependency>

</dependencies>

 **Configure application.properties:** Configure MySQL database connection in the src/main/resources/application.properties.

 spring.datasource.url=jdbc:mysql://localhost:3306/library\_db

spring.datasource.username=root

spring.datasource.password=password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

 **Create the Book Model Class:**

 // Book.java

package com.example.library.model;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class Book {

@Id

private Long id;

private String title;

private String author;

// Constructors, Getters, and Setters

public Book() {}

public Book(Long id, String title, String author) {

this.id = id;

this.title = title;

this.author = author;

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

}

 **Create the BookRepository Interface:**

 // BookRepository.java

package com.example.library.repository;

import com.example.library.model.Book;

import org.springframework.data.jpa.repository.JpaRepository;

public interface BookRepository extends JpaRepository<Book, Long> {

}

 **Create the BookController Class:**

 // BookController.java

package com.example.library.controller;

import com.example.library.model.Book;

import com.example.library.repository.BookRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/books")

public class BookController {

@Autowired

private BookRepository bookRepository;

// Save a book

@PostMapping

public Book saveBook(@RequestBody Book book) {

return bookRepository.save(book);

}

// Fetch book by ID

@GetMapping("/{id}")

public Book getBookById(@PathVariable Long id) {

return bookRepository.findById(id).orElse(null);

}

}

 **Run the Application:**

* Start your Spring Boot application by running the Application.java class, which is generated automatically.
* Ensure MySQL is running and the database library\_db is created.

 **Test the APIs using Postman:**

* **POST /books**: Add a new book.
  + URL: http://localhost:8080/books
  + Method: POST
  + Body (JSON):

{

"id": 1,

"title": "Spring Boot Guide",

"author": "John Doe"

}