Web Technology

Problem Statement No.1
Develop a simple online bookstore's inventory management system using JSP and JSTL. The bookstore has various genres of books, and you need to provide an interface for both users and administrators.
Requirements:

1. **User Interface:** Create a JSP page that displays a list of books available in the bookstore. Each book should have the following details:
   - Title
   - Author
   - Genre
   - Price
   - Availability (In stock/Out of stock)
2. **Filtering:** Implement filtering options using JSTL to allow users to filter books by genre. The genres should be displayed as checkboxes on the page.
3. **Admin Functionality:** Create an admin JSP page where administrators can add new books to the inventory. Include a form that collects the necessary information about the book.
4. **Data Source:** Use an in-memory list or a mock database (like an array or a list) to manage the book data, with sample data provided.

ANSWER-

Online Bookstore Inventory Management System

---

1. Project Setup

1. **Create a Maven-based Java Web Application project.**
2. **Add dependencies for JSP and JSTL in the `pom.xml` file:**

```
<dependencies>
    <!-- Servlet API -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>
```

---

2. Directory Structure

```
src
│
├── main
│   ├── java
│   │   ├── com.example.bookstore
│   │   │   ├── Book.java
│   │   │   ├── BookService.java
│   │   │   ├── BookController.java
│   │   │   └── BookInventoryApp.java
│   ├── webapp
│   │   ├── WEB-INF
│   │   │   └── web.xml
│   │   ├── index.jsp
│   │   ├── admin.jsp
│   │   └── result.jsp
│
```

---

3. Backend Implementation

Book Entity (`Book.java`)

```
package com.example.bookstore;

public class Book {
    private String title;
    private String author;
    private String genre;
    private double price;
    private boolean isInStock;

    public Book(String title, String author, String genre, double price,
boolean isInStock) {
        this.title = title;
        this.author = author;
        this.genre = genre;
        this.price = price;
        this.isInStock = isInStock;
    }

    // Getters and Setters
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public String getGenre() { return genre; }
    public double getPrice() { return price; }
    public boolean isInStock() { return isInStock; }
}
```

Book Service (`BookService.java`)

```
package com.example.bookstore;

import java.util.ArrayList;
import java.util.List;

public class BookService {
    private List<Book> books = new ArrayList<>();

    public BookService() {
        // Sample data
        books.add(new Book("The Great Gatsby", "F. Scott Fitzgerald",
"Fiction", 10.99, true));
        books.add(new Book("1984", "George Orwell", "Dystopia", 8.99,
true));
        books.add(new Book("Moby Dick", "Herman Melville", "Classic",
12.99, false));
        books.add(new Book("To Kill a Mockingbird", "Harper Lee",
"Fiction", 9.99, true));
    }

    // Get all books
    public List<Book> getBooks() {
        return books;
    }

    // Add new book
    public void addBook(Book book) {
        books.add(book);
    }

    // Filter books by genre
    public List<Book> getBooksByGenre(String genre) {
        List<Book> filteredBooks = new ArrayList<>();
        for (Book book : books) {
            if (book.getGenre().equals(genre)) {
                filteredBooks.add(book);
            }
        }
        return filteredBooks;
    }
}
```

Book Controller (`BookController.java`)

```
package com.example.bookstore;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

public class BookController extends HttpServlet {
    private BookService bookService;

    @Override
    public void init() throws ServletException {
        bookService = new BookService();
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String genre = request.getParameter("genre");

        if (genre != null) {
            request.setAttribute("books",
bookService.getBooksByGenre(genre));
        } else {
            request.setAttribute("books", bookService.getBooks());
        }

        request.getRequestDispatcher("index.jsp").forward(request,
response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String title = request.getParameter("title");
        String author = request.getParameter("author");
        String genre = request.getParameter("genre");
        double price = Double.parseDouble(request.getParameter("price"));
        boolean isInStock =
Boolean.parseBoolean(request.getParameter("isInStock"));

        bookService.addBook(new Book(title, author, genre, price,
isInStock));

        response.sendRedirect("admin.jsp");
    }
}
```

---

4. Frontend (JSP Pages)

Book List Page (`index.jsp`)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Bookstore</title>
</head>
<body>
    <h1>Welcome to the Bookstore</h1>

    <!-- Genre Filter -->
    <h3>Filter by Genre</h3>
    <form action="BookController" method="get">
        <label><input type="checkbox" name="genre" value="Fiction">
Fiction</label><br>
        <label><input type="checkbox" name="genre" value="Dystopia">
Dystopia</label><br>
        <label><input type="checkbox" name="genre" value="Classic">
Classic</label><br>
        <button type="submit">Filter</button>
    </form>

    <h2>Available Books</h2>
    <table>
        <tr>
            <th>Title</th>
            <th>Author</th>
            <th>Genre</th>
            <th>Price</th>
            <th>Availability</th>
        </tr>
        <c:forEach var="book" items="${books}">
            <tr>
                <td>${book.title}</td>
                <td>${book.author}</td>
                <td>${book.genre}</td>
                <td>${book.price}</td>
                <td>${book.inStock ? 'In stock' : 'Out of stock'}</td>
            </tr>
        </c:forEach>
    </table>

</body>
</html>
```

Admin Page (`admin.jsp`)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Admin - Add Book</title>
</head>
<body>
    <h1>Admin: Add a New Book</h1>
    <form action="BookController" method="post">
        Title: <input type="text" name="title" required /><br />
        Author: <input type="text" name="author" required /><br />
        Genre: <input type="text" name="genre" required /><br />
        Price: <input type="text" name="price" required /><br />
        In Stock: <input type="checkbox" name="isInStock" value="true"
/><br />
        <button type="submit">Add Book</button>
    </form>
</body>
</html>
```

---

5. Web Configuration (`web.xml`)

```
<web-app>
    <servlet>
        <servlet-name>BookController</servlet-name>
        <servlet-class>com.example.bookstore.BookController</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>BookController</servlet-name>
        <url-pattern>/BookController</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Explanation of the Process:

1. **Entities and Data:** The `Book` class defines the book properties. The `BookService` manages an in-memory list of books, and allows adding new books and filtering by genre.
2. **Controller:** The `BookController` processes the requests to display books and handle book additions. It supports both GET requests (for listing and filtering books) and POST requests (for adding new books).
3. **Filtering:** The genre filter is implemented using checkboxes and is processed using JSTL on the frontend.
4. **Admin Functionality:** The admin page allows administrators to add new books to the inventory using a form.
5. **JSP Pages:** The JSP pages use JSTL to display dynamic content such as book lists and error messages.

This solution provides a basic inventory management system with user and admin interfaces, allowing users to filter books by genre and administrators to add new books.

Problem Statement 2: User Registration and Login System
Background: You are developing a user registration and login system for a small web application. The application allows users to create an account and log in to access personalized content.
Requirements:

1. **Registration Page:** Create a JSP page for user registration that collects the following information:
   - Username
   - Password
   - Email
   - Confirm Password
2. **Validation:** Use JSTL to validate the user inputs. Implement checks to ensure that:
   - All fields are filled out.
   - The password and confirm password fields match.
   - The email format is valid.
3. **Login Page:** Create a separate JSP page for users to log in. The login form should ask for the username and password.
4. **Session Management:** Once a user successfully logs in, create a session and display a welcome message on a new JSP page. If the login fails, display an error message using JSTL.
5. **User Data Storage:** For simplicity, use a Java List to store user data in memory. Implement a way to check for existing usernames during registration.

ANSWER- User Registration and Login System Implementation

---

1. Project Setup

1. **Create a new Maven-based Java Web Application project.**
   - Add necessary dependencies in `pom.xml` for JSP and JSTL.

2. Dependencies (`pom.xml`)

```
<dependencies>
    <!-- Servlet API -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>
```

3. Directory Structure

```
src
│
├── main
│   ├── java
│   │   ├── com.example.usersystem
│   │   │   ├── User.java
│   │   │   ├── UserService.java
│   │   │   └── UserController.java
│   │
│   ├── webapp
│       ├── WEB-INF
│       │   └── web.xml
│       ├── index.jsp
│       ├── register.jsp
│       ├── login.jsp
│       ├── welcome.jsp
│       └── error.jsp
│
```

4. Backend Implementation

User Entity (`User.java`)

```
package com.example.usersystem;

public class User {
    private String username;
    private String password;
    private String email;

    public User(String username, String password, String email) {
        this.username = username;
        this.password = password;
        this.email = email;
```

```java
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public String getEmail() {
        return email;
    }
}
```

User Service (`UserService.java`)

```java
package com.example.usersystem;

import java.util.ArrayList;
import java.util.List;

public class UserService {
    private List<User> users = new ArrayList<>();

    public boolean registerUser(String username, String password, String
email) {
        if (isUsernameTaken(username)) {
            return false;
        }
        users.add(new User(username, password, email));
        return true;
    }

    public boolean isUsernameTaken(String username) {
        return users.stream().anyMatch(user ->
user.getUsername().equals(username));
    }

    public boolean validateLogin(String username, String password) {
        return users.stream()
                .anyMatch(user -> user.getUsername().equals(username) &&
user.getPassword().equals(password));
    }
}
```

User Controller (`UserController.java`)

```java
package com.example.usersystem;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

public class UserController extends HttpServlet {
    private UserService userService;
    @Override
    public void init() throws ServletException {
        userService = new UserService();
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String action = request.getParameter("action");

        if ("register".equals(action)) {
            handleRegistration(request, response);
        } else if ("login".equals(action)) {
            handleLogin(request, response);
        }
    }

    private void handleRegistration(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        String confirmPassword = request.getParameter("confirmPassword");
        String email = request.getParameter("email");

        if (!password.equals(confirmPassword)) {
            request.setAttribute("error", "Passwords do not match!");
            request.getRequestDispatcher("register.jsp").forward(request,
response);
            return;
        }

        boolean isRegistered = userService.registerUser(username, password,
email);

        if (isRegistered) {
            response.sendRedirect("login.jsp");
        } else {
            request.setAttribute("error", "Username already taken!");
            request.getRequestDispatcher("register.jsp").forward(request,
response);
        }
    }

    private void handleLogin(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (userService.validateLogin(username, password)) {
            HttpSession session = request.getSession();
            session.setAttribute("username", username);
            response.sendRedirect("welcome.jsp");
        } else {
            request.setAttribute("error", "Invalid username or password!");
            request.getRequestDispatcher("login.jsp").forward(request,
response);
        }
    }
}
```

5. JSP Pages

Registration Page (`register.jsp`)

```
<!DOCTYPE html>
<html>
<head>
    <title>Register</title>
</head>
<body>
    <h1>User Registration</h1>
    <form action="UserController" method="post">
        <input type="hidden" name="action" value="register" />
        Username: <input type="text" name="username" required /><br />
        Password: <input type="password" name="password" required /><br />
        Confirm Password: <input type="password" name="confirmPassword"
required /><br />
        Email: <input type="email" name="email" required /><br />
        <button type="submit">Register</button>
    </form>
    <c:if test="${not empty error}">
        <p style="color:red;">${error}</p>
    </c:if>
</body>
</html>
```

Login Page (`login.jsp`)

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h1>User Login</h1>
    <form action="UserController" method="post">
        <input type="hidden" name="action" value="login" />
        Username: <input type="text" name="username" required /><br />
        Password: <input type="password" name="password" required /><br />
        <button type="submit">Login</button>
    </form>
    <c:if test="${not empty error}">
        <p style="color:red;">${error}</p>
    </c:if>
</body>
</html>
```

Welcome Page (`welcome.jsp`)

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Welcome, ${sessionScope.username}!</h1>
    <a href="login.jsp">Logout</a>
</body>
</html>
```

Error Handling Page (`error.jsp`)

```
<!DOCTYPE html>
<html>
<head>
    <title>Error</title>
</head>
<body>
    <h1>An error occurred: ${error}</h1>
</body>
</html>
```

---

6. Web Configuration (`web.xml`)

```
<web-app>
    <servlet>
        <servlet-name>UserController</servlet-name>
        <servlet-class>com.example.usersystem.UserController</servlet-
class>
    </servlet>
    <servlet-mapping>
        <servlet-name>UserController</servlet-name>
        <url-pattern>/UserController</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Explanation

1. **Registration Page:** Collects user input and performs validation (e.g., passwords must match, email must be valid).
2. **Login Page:** Validates credentials and creates a session for the user.
3. **Session Management:** `HttpSession` manages login state and displays personalized content.
4. **Data Storage:** A simple in-memory `List` stores user data for demo purposes.
5. **JSTL:** Provides error messages and validation feedback to users.

Problem Statement 3: Task Management Application

Background: You are developing a task management application that allows users to create, update, and delete tasks. Users can also mark tasks as complete and filter tasks based on their status.

Requirements:

1. **Task Entity:** Create a Task entity class with the following attributes:
   - ID (auto-generated)
   - Title
   - Description
   - Status (Pending, In Progress, Completed)
   - Due Date
2. **Repository Layer:** Implement a Spring Data JPA repository to handle CRUD operations for the Task entity.
3. **Service Layer:** Create a service class that provides methods for:
   - Creating a new task
   - Updating an existing task
   - Deleting a task
   - Retrieving all tasks
4. **Controller Layer:** Develop a Spring MVC controller with the following endpoints:
   - GET /tasks - Retrieve all tasks.
   - GET /tasks/{status} - Filter tasks by status.
   - POST /tasks - Create a new task.
   - PUT /tasks/{id} - Update an existing task.
   - DELETE /tasks/{id} - Delete a task.
5. **View Layer:** Use Thymeleaf to create HTML templates for displaying the task list and a form for creating/updating tasks.

ANSWER-

Task Management Application Implementation

---

1. Project Setup

1. **Create a Spring Boot project using Spring Tool Suite (STS).**
   - **Dependencies:**
     - Spring Web
     - Spring Data JPA
     - Thymeleaf
     - H2 Database
     - Spring Boot Validation

---

2. Dependencies (`pom.xml`)

```
<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>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
```

---

3. Database Configuration (`application.properties`)

```
spring.datasource.url=jdbc:h2:mem:taskdb
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
```

---

4. Entity Layer

`Task` Entity

```
package com.example.taskmanagement.entity;

import jakarta.persistence.*;
import jakarta.validation.constraints.NotBlank;
import java.time.LocalDate;

@Entity
public class Task {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Title is required")
    private String title;

    private String description;

    @Enumerated(EnumType.STRING)
    private Status status = Status.PENDING;

    private LocalDate dueDate;

    public enum Status {
        PENDING,
        IN_PROGRESS,
        COMPLETED
    }

    // Getters and Setters
}
```

---

5. Repository Layer

TaskRepository

```
package com.example.taskmanagement.repository;

import com.example.taskmanagement.entity.Task;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface TaskRepository extends JpaRepository<Task, Long> {
    List<Task> findByStatus(Task.Status status);
}
```

---

6. Service Layer

TaskService

```
package com.example.taskmanagement.service;

import com.example.taskmanagement.entity.Task;
import com.example.taskmanagement.repository.TaskRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class TaskService {

    @Autowired
    private TaskRepository repository;

    public Task createTask(Task task) {
        return repository.save(task);
    }

    public Task updateTask(Long id, Task updatedTask) {
        Task task = repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Task not
found!"));
        task.setTitle(updatedTask.getTitle());
        task.setDescription(updatedTask.getDescription());
        task.setStatus(updatedTask.getStatus());
        task.setDueDate(updatedTask.getDueDate());
        return repository.save(task);
    }

    public void deleteTask(Long id) {
        repository.deleteById(id);
    }

    public List<Task> getAllTasks() {
        return repository.findAll();
    }

    public List<Task> getTasksByStatus(Task.Status status) {
        return repository.findByStatus(status);
    }
}
```

---

7. Controller Layer

TaskController

```
package com.example.taskmanagement.controller;

import com.example.taskmanagement.entity.Task;
import com.example.taskmanagement.service.TaskService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/tasks")
public class TaskController {

    @Autowired
    private TaskService service;

    @GetMapping
    public String getAllTasks(Model model) {
        model.addAttribute("tasks", service.getAllTasks());
        return "task-list";
    }

    @GetMapping("/{status}")
    public String getTasksByStatus(@PathVariable Task.Status status, Model
model) {
        model.addAttribute("tasks", service.getTasksByStatus(status));
        return "task-list";
    }

    @GetMapping("/add")
    public String showAddForm(Model model) {
        model.addAttribute("task", new Task());
        return "task-form";
    }

    @PostMapping
    public String createTask(@Valid @ModelAttribute Task task) {
        service.createTask(task);
        return "redirect:/tasks";
    }

    @GetMapping("/edit/{id}")
    public String showEditForm(@PathVariable Long id, Model model) {
        model.addAttribute("task", service.getAllTasks().stream().filter(t
-> t.getId().equals(id)).findFirst().orElse(null));
        return "task-form";
    }

    @PostMapping("/{id}")
    public String updateTask(@PathVariable Long id, @Valid @ModelAttribute
Task task) {
        service.updateTask(id, task);
        return "redirect:/tasks";
    }

    @GetMapping("/delete/{id}")
    public String deleteTask(@PathVariable Long id) {
        service.deleteTask(id);
        return "redirect:/tasks";
    }
}
```

---

8. View Layer

Thymeleaf Templates

1. **task-list.html**

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Task List</title>
</head>
<body>
    <h1>Task List</h1>
    <a href="/tasks/add">Add Task</a>
    <table>
        <tr>
            <th>Title</th>
            <th>Status</th>
            <th>Due Date</th>
            <th>Actions</th>
        </tr>
        <tr th:each="task : ${tasks}">
            <td th:text="${task.title}"></td>
            <td th:text="${task.status}"></td>
            <td th:text="${task.dueDate}"></td>
            <td>
                <a th:href="@{/tasks/edit/{id}(id=${task.id})}">Edit</a>
                <a th:href="@{/tasks/delete/{id}(id=${task.id})}">Delete</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

29. **task-form.html**

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
```

```
    <title>Task Form</title>
</head>
<body>
    <h1>Task Form</h1>
    <form th:action="@{/tasks}" th:object="${task}" method="post">
        <input type="text" th:field="*{title}" placeholder="Title" />
        <textarea th:field="*{description}" placeholder="Description"></textarea>
        <select th:field="*{status}">
            <option value="PENDING">Pending</option>
            <option value="IN_PROGRESS">In Progress</option>
            <option value="COMPLETED">Completed</option>
        </select>
        <input type="date" th:field="*{dueDate}" />
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Explanation

1. **Entity Layer:** Defines the `Task` entity, including `Status` as an enum for better type safety.
2. **Repository Layer:** Provides data access methods, including a custom finder for `Status`.
3. **Service Layer:** Contains all business logic, such as updating and filtering tasks.
4. **Controller Layer:** Manages HTTP requests and integrates the view layer using Thymeleaf.
5. **Thymeleaf Templates:** Enable user-friendly HTML forms and dynamic task listings.

Problem Statement 4: E-Commerce Product Catalogue

Background: You are tasked with building an e-commerce application that features a product catalogue where users can view, add, update, and delete products.

Requirements:

1. **Product Entity:** Create a Product entity class with the following attributes:
   - ID (auto-generated)
   - Name
   - Description
   - Price
   - Stock Quantity
   - Category
2. **Repository Layer:** Implement a Spring Data JPA repository for managing the Product entity.
3. **Service Layer:** Create a service class that includes methods for:
   - Adding a new product
   - Updating an existing product
   - Deleting a product
   - Retrieving all products

4. **Controller Layer:** Develop a Spring MVC controller with the following endpoints:
   - GET /products - Retrieve and display all products.
   - GET /products/{id} - Retrieve product details by ID.
   - POST /products - Add a new product.
   - PUT /products/{id} - Update an existing product.
   - DELETE /products/{id} - Delete a product.
5. **View Layer:** Use Thymeleaf to create HTML templates for displaying the product catalog, product details, and forms for adding/updating products.
6. **Data Validation:** Implement validation for product fields using Spring's validation annotations.

ANSWER- **E-Commerce Product Catalogue Implementation**

---

1. Project Setup

1. **Create a Spring Boot Project using Spring Tool Suite (STS):**
   - **Dependencies:**
     - Spring Web (for RESTful APIs and MVC)
     - Spring Data JPA (for database access)
     - Spring Boot Validation (for data validation)
     - Thymeleaf (for view layer)
     - H2 Database (for development and testing)

---

2. Dependencies (`pom.xml`)

```
<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>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

3. Database Configuration (`application.properties`)

```
spring.datasource.url=jdbc:h2:mem:ecommerce
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
```

4. Entity Layer

`Product` Entity

```
package com.example.ecommerce.entity;

import jakarta.persistence.*;
import jakarta.validation.constraints.*;

@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    private String description;

    @NotNull(message = "Price is required")
    @Min(value = 0, message = "Price cannot be negative")
    private Double price;

    @NotNull(message = "Stock quantity is required")
    @Min(value = 0, message = "Stock quantity cannot be negative")
    private Integer stockQuantity;

    @NotBlank(message = "Category is required")
    private String category;

    // Getters and Setters
}
```

5. Repository Layer

`ProductRepository`

```
package com.example.ecommerce.repository;

import com.example.ecommerce.entity.Product;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

---

6. Service Layer

ProductService

```
package com.example.ecommerce.service;

import com.example.ecommerce.entity.Product;
import com.example.ecommerce.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ProductService {

    @Autowired
    private ProductRepository repository;

    public Product addProduct(Product product) {
        return repository.save(product);
    }

    public Product updateProduct(Long id, Product updatedProduct) {
        Product existingProduct = repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not
found!"));
        existingProduct.setName(updatedProduct.getName());
        existingProduct.setDescription(updatedProduct.getDescription());
        existingProduct.setPrice(updatedProduct.getPrice());

existingProduct.setStockQuantity(updatedProduct.getStockQuantity());
        existingProduct.setCategory(updatedProduct.getCategory());
        return repository.save(existingProduct);
    }

    public void deleteProduct(Long id) {
        repository.deleteById(id);
    }

    public List<Product> getAllProducts() {
        return repository.findAll();
    }

    public Product getProductById(Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not
found!"));
    }
}
```

---

7. Controller Layer

ProductController

```java
package com.example.ecommerce.controller;

import com.example.ecommerce.entity.Product;
import com.example.ecommerce.service.ProductService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/products")
public class ProductController {

    @Autowired
    private ProductService service;

    @GetMapping
    public String getAllProducts(Model model) {
        model.addAttribute("products", service.getAllProducts());
        return "product-list";
    }

    @GetMapping("/{id}")
    public String getProductById(@PathVariable Long id, Model model) {
        model.addAttribute("product", service.getProductById(id));
        return "product-details";
    }

    @GetMapping("/add")
    public String showAddForm(Model model) {
        model.addAttribute("product", new Product());
        return "product-form";
    }

    @PostMapping
    public String addProduct(@Valid @ModelAttribute Product product) {
        service.addProduct(product);
        return "redirect:/products";
    }

    @GetMapping("/edit/{id}")
    public String showEditForm(@PathVariable Long id, Model model) {
        model.addAttribute("product", service.getProductById(id));
        return "product-form";
    }

    @PostMapping("/{id}")
    public String updateProduct(@PathVariable Long id, @Valid
@ModelAttribute Product product) {
        service.updateProduct(id, product);
        return "redirect:/products";
    }

    @GetMapping("/delete/{id}")
    public String deleteProduct(@PathVariable Long id) {
        service.deleteProduct(id);
        return "redirect:/products";
    }
}
```

8. View Layer

Thymeleaf Templates

product-list.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Product List</title>
</head>
<body>
    <h1>Product List</h1>
    <a href="/products/add">Add Product</a>
    <table>
        <tr>
            <th>Name</th>
            <th>Category</th>
            <th>Price</th>
            <th>Actions</th>
        </tr>
        <tr th:each="product : ${products}">
            <td th:text="${product.name}"></td>
            <td th:text="${product.category}"></td>
            <td th:text="${product.price}"></td>
            <td>
                <a th:href="@{/products/{id}(id=${product.id})}">View</a>
                <a th:href="@{/products/edit/{id}(id=${product.id})}">Edit</a>
                <a th:href="@{/products/delete/{id}(id=${product.id})}">Delete</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

product-details.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Product Details</title>
</head>
<body>
    <h1 th:text="${product.name}"></h1>
    <p th:text="'Category: ' + ${product.category}"></p>
    <p th:text="'Price: $' + ${product.price}"></p>
    <p th:text="'Description: ' + ${product.description}"></p>
    <p th:text="'Stock: ' + ${product.stockQuantity}"></p>
    <a href="/products">Back to List</a>
</body>
</html>
```

product-form.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Product Form</title>
</head>
<body>
    <h1>Product Form</h1>
```

```html
    <form th:action="@{/products}" th:object="${product}" method="post">
        <input type="text" th:field="*{name}" placeholder="Name" />
        <input type="text" th:field="*{category}" placeholder="Category" />
        <input type="number" th:field="*{price}" placeholder="Price" />
        <input type="number" th:field="*{stockQuantity}" placeholder="Stock Quantity" />
        <textarea th:field="*{description}" placeholder="Description"></textarea>
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Explanation

1. **Entity Layer:** Defines the `Product` entity with validation annotations.
2. **Repository Layer:** Manages database access with `JpaRepository`.
3. **Service Layer:** Contains business logic.
4. **Controller Layer:** Handles HTTP requests and integrates Thymeleaf for rendering HTML views.
5. **Thymeleaf Templates:** Provides UI for listing, viewing, adding, and updating products.
6. **Validation:** Ensures correct user input using annotations like `@NotBlank` and `@Min`.

Problem Statement 5: Online Grocery Store API

Background: You are tasked with developing a RESTful API for an online grocery store. The API should allow users to manage grocery items, including searching, adding, updating, and deleting items from the inventory.

Requirements:

1. **Grocery Item Entity:** Create a GroceryItem entity class with the following attributes:
   - ID (auto-generated)
   - Name
   - Description
   - Price
   - Quantity in Stock
   - Category
2. **Repository Layer:** Use Spring Data JPA to create a repository interface for managing GroceryItem entities.
3. **Service Layer:** Implement a service class that provides methods for:
   - Adding a new grocery item
   - Updating an existing grocery item
   - Deleting a grocery item by ID
   - Retrieving all grocery items
   - Searching for items by name or category

4. **Controller Layer:** Develop a RESTful controller with the following endpoints:
   - GET /api/grocery-items - Retrieve all grocery items.
   - GET /api/grocery-items/{id} - Retrieve a specific item by ID.
   - POST /api/grocery-items - Add a new grocery item.
   - PUT /api/grocery-items/{id} - Update an existing grocery item.
   - DELETE /api/grocery-items/{id} - Delete a grocery item.
5. **Data Validation:** Implement input validation for the GroceryItem fields using Spring's validation annotations.
6. **Exception Handling:** Create a global exception handler to manage errors and return appropriate HTTP status codes and messages.

ANSWER-

Online Grocery Store API Implementation

---

1. Project Setup

1. **Create a Spring Boot Project using Spring Tool Suite (STS):**
   - **Dependencies:**
     - Spring Web (for RESTful API)
     - Spring Data JPA (for database access)
     - H2 Database (for an in-memory database)
     - Spring Boot Validation (for input validation)

---

2. Dependencies (`pom.xml`)

```
<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>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

---

3. Database Configuration (`application.properties`)

```
spring.datasource.url=jdbc:h2:mem:grocerydb
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
```

---

4. Entity Layer

`GroceryItem` Entity

```
package com.example.grocerystore.entity;

import jakarta.persistence.*;
import jakarta.validation.constraints.*;

@Entity
public class GroceryItem {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    private String description;

    @NotNull(message = "Price is required")
    @Min(value = 0, message = "Price cannot be negative")
    private Double price;

    @NotNull(message = "Quantity in stock is required")
    @Min(value = 0, message = "Quantity cannot be negative")
    private Integer quantityInStock;

    @NotBlank(message = "Category is required")
    private String category;

    // Getters and Setters
}
```

---

5. Repository Layer

`GroceryItemRepository`

```
package com.example.grocerystore.repository;

import com.example.grocerystore.entity.GroceryItem;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

public interface GroceryItemRepository extends JpaRepository<GroceryItem,
Long> {
    List<GroceryItem> findByNameContainingIgnoreCase(String name);

    List<GroceryItem> findByCategoryIgnoreCase(String category);
}
```

---

6. Service Layer

GroceryItemService

```
package com.example.grocerystore.service;

import com.example.grocerystore.entity.GroceryItem;
import com.example.grocerystore.repository.GroceryItemRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class GroceryItemService {

    @Autowired
    private GroceryItemRepository repository;

    public GroceryItem addItem(GroceryItem item) {
        return repository.save(item);
    }

    public GroceryItem updateItem(Long id, GroceryItem updatedItem) {
        GroceryItem existingItem = repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Grocery item not
found!"));
        existingItem.setName(updatedItem.getName());
        existingItem.setDescription(updatedItem.getDescription());
        existingItem.setPrice(updatedItem.getPrice());
        existingItem.setQuantityInStock(updatedItem.getQuantityInStock());
        existingItem.setCategory(updatedItem.getCategory());
        return repository.save(existingItem);
    }

    public void deleteItem(Long id) {
        repository.deleteById(id);
    }

    public List<GroceryItem> getAllItems() {
        return repository.findAll();
    }

    public GroceryItem getItemById(Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new RuntimeException("Grocery item not
found!"));
    }

    public List<GroceryItem> searchByName(String name) {
        return repository.findByNameContainingIgnoreCase(name);
    }

    public List<GroceryItem> searchByCategory(String category) {
        return repository.findByCategoryIgnoreCase(category);
    }
}
```

---

7. Controller Layer

GroceryItemController

```
package com.example.grocerystore.controller;

import com.example.grocerystore.entity.GroceryItem;
import com.example.grocerystore.service.GroceryItemService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import jakarta.validation.Valid;
import java.util.List;

@RestController
@RequestMapping("/api/grocery-items")
public class GroceryItemController {

    @Autowired
    private GroceryItemService service;

    @PostMapping
    public GroceryItem addItem(@Valid @RequestBody GroceryItem item) {
        return service.addItem(item);
    }

    @PutMapping("/{id}")
    public GroceryItem updateItem(@PathVariable Long id, @Valid
@RequestBody GroceryItem updatedItem) {
        return service.updateItem(id, updatedItem);
    }

    @DeleteMapping("/{id}")
    public void deleteItem(@PathVariable Long id) {
        service.deleteItem(id);
    }

    @GetMapping
    public List<GroceryItem> getAllItems() {
        return service.getAllItems();
    }

    @GetMapping("/{id}")
    public GroceryItem getItemById(@PathVariable Long id) {
        return service.getItemById(id);
    }

    @GetMapping("/search/name")
    public List<GroceryItem> searchByName(@RequestParam String name) {
        return service.searchByName(name);
    }

    @GetMapping("/search/category")
    public List<GroceryItem> searchByCategory(@RequestParam String
category) {
        return service.searchByCategory(category);
    }
}
```

---

8. Global Exception Handling

`GlobalExceptionHandler`

```
package com.example.grocerystore.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(RuntimeException
ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGenericException(Exception ex) {
        return new ResponseEntity<>("An error occurred: " +
ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Testing

- **Run the application** and use **Postman** or **cURL** for API testing:
  - **Add a new item:**
    `POST /api/grocery-items`
  - `{`
  - `    "name": "Apple",`
  - `    "description": "Fresh red apples",`
  - `    "price": 3.5,`
  - `    "quantityInStock": 100,`
  - `    "category": "Fruits"`
  - `}`
  - **Update an item:**
    `PUT /api/grocery-items/1`
  - **Search by name:**
    `GET /api/grocery-items/search/name?name=apple`
  - **Search by category:**
    `GET /api/grocery-items/search/category?category=fruits`

Explanation

1. **Entity Layer:** Defines `GroceryItem` with attributes and validation.
2. **Repository Layer:** Provides default CRUD methods and custom queries.

3. **Service Layer:** Contains business logic for handling inventory operations.
4. **Controller Layer:** Exposes RESTful endpoints.
5. **Global Exception Handler:** Manages runtime and generic exceptions.
6. **Validation:** Ensures input correctness with annotations like `@NotBlank` and `@Min`.

Problem Statement 6: Employee Management System

Background: You are developing an Employee Management System to manage employee records within a company. The system should allow HR personnel to perform CRUD operations on employee data.

Requirements:

1. **Employee Entity:** Create an Employee entity class with the following attributes:
   - ID (auto-generated)
   - First Name
   - Last Name
   - Email
   - Phone Number
   - Position
   - Salary
2. **Repository Layer:** Use Spring Data JPA to create a repository interface for managing Employee entities.
3. **Service Layer:** Implement a service class that provides methods for:
   - Adding a new employee
   - Updating an existing employee
   - Deleting an employee by ID
   - Retrieving all employees
   - Retrieving an employee by ID
4. **Controller Layer:** Develop a RESTful controller with the following endpoints:
   - GET /api/employees - Retrieve all employees.
   - GET /api/employees/{id} - Retrieve a specific employee by ID.
   - POST /api/employees - Add a new employee.
   - PUT /api/employees/{id} - Update an existing employee.
   - DELETE /api/employees/{id} - Delete an employee.

ANSWER-

Employee Management System Implementation

---

1. Project Setup

1. **Create a new Spring Boot Project in Spring Tool Suite (STS):**
   - **Dependencies:**
     - **Spring Web**: For building RESTful APIs.

- **Spring Data JPA**: For database access and persistence.
- **H2 Database**: For using an in-memory database.

---

2. Dependencies (`pom.xml`)

```
<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>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

---

3. Database Configuration (`application.properties`)

```
spring.datasource.url=jdbc:h2:mem:employeedb
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
```

---

4. Entity Layer

`Employee` Entity

```
package com.example.employee.entity;

import jakarta.persistence.*;

@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    private String position;
    private double salary;

    // Getters and Setters
}
```

5. Repository Layer

EmployeeRepository

```
package com.example.employee.repository;

import com.example.employee.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

6. Service Layer

EmployeeService

```
package com.example.employee.service;

import com.example.employee.entity.Employee;
import com.example.employee.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    public Employee addEmployee(Employee employee) {
        return employeeRepository.save(employee);
    }

    public Employee updateEmployee(Long id, Employee updatedEmployee) {
        Employee existingEmployee = employeeRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Employee not
found!"));
        existingEmployee.setFirstName(updatedEmployee.getFirstName());
        existingEmployee.setLastName(updatedEmployee.getLastName());
        existingEmployee.setEmail(updatedEmployee.getEmail());
        existingEmployee.setPhoneNumber(updatedEmployee.getPhoneNumber());
        existingEmployee.setPosition(updatedEmployee.getPosition());
        existingEmployee.setSalary(updatedEmployee.getSalary());
        return employeeRepository.save(existingEmployee);
    }

    public void deleteEmployee(Long id) {
        employeeRepository.deleteById(id);
    }

    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }
    public Employee getEmployeeById(Long id) {
        return employeeRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Employee not
found!"));
    }
}
```

---

7. Controller Layer

EmployeeController

```
package com.example.employee.controller;

import com.example.employee.entity.Employee;
import com.example.employee.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @PostMapping
    public Employee addEmployee(@RequestBody Employee employee) {
        return employeeService.addEmployee(employee);
    }

    @PutMapping("/{id}")
    public Employee updateEmployee(@PathVariable Long id, @RequestBody
Employee updatedEmployee) {
        return employeeService.updateEmployee(id, updatedEmployee);
    }

    @DeleteMapping("/{id}")
    public void deleteEmployee(@PathVariable Long id) {
        employeeService.deleteEmployee(id);
    }

    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }

    @GetMapping("/{id}")
    public Employee getEmployeeById(@PathVariable Long id) {
        return employeeService.getEmployeeById(id);
    }
}
```

---

8. Running and Testing the Application

- **Run the Application:**
  Launch the application from Spring Tool Suite (STS).
- **Access H2 Console:**
  Navigate to `http://localhost:8080/h2-console` to view the database. Use the following credentials:
  - **JDBC URL:** `jdbc:h2:mem:employeedb`
  - **Username:** `sa`
  - **Password:** `password`
- **Test API Endpoints:**
  Use a tool like **Postman** or **cURL**:
  - Add an employee:
    `POST /api/employees`
    Request Body:
  - `{`
  - `    "firstName": "John",`
  - `    "lastName": "Doe",`
  - `    "email": "john.doe@example.com",`
  - `    "phoneNumber": "1234567890",`
  - `    "position": "Software Engineer",`
  - `    "salary": 75000`
  - `}`
  - Update an employee:
    `PUT /api/employees/1`
    Request Body:
  - `{`
  - `    "firstName": "Jane",`
  - `    "lastName": "Doe",`
  - `    "email": "jane.doe@example.com",`
  - `    "phoneNumber": "0987654321",`
  - `    "position": "Team Lead",`
  - `    "salary": 90000`
  - `}`
  - Delete an employee:
    `DELETE /api/employees/1`
  - Retrieve all employees:
    `GET /api/employees`
  - Retrieve an employee by ID:
    `GET /api/employees/1`

Explanation of the Process

1. **Project Setup:**
   - Created a Spring Boot project with dependencies for web, JPA, and H2 database.
2. **Entity Design:**
   - Designed the `Employee` entity with relevant fields.
3. **Repository Layer:**
   - Created a repository interface extending `JpaRepository` for CRUD operations.
4. **Service Layer:**
   - Implemented the business logic for employee CRUD operations.

5. **Controller Layer:**
    - Developed RESTful endpoints for managing employees.
6. **Database Interaction:**
    - Used H2 in-memory database for lightweight testing and development.
7. **Testing:**
    - Verified API functionality using Postman and inspected the database using the H2 console.

This modular architecture ensures clear separation of concerns, making the system maintainable and scalable.

Problem Statement 7: Library Management System

Background: You are tasked with developing a library management system that allows users to manage books and their associated authors. The system should facilitate operations like adding new books, updating existing ones, and retrieving information about books and authors.

Requirements:

1. **Entities:**
    - **Book Entity:** Create a Book entity class with the following attributes:
        - ID (auto-generated)
        - Title
        - ISBN
        - Published Date
        - Number of Copies
    - **Author Entity:** Create an Author entity class with the following attributes:
        - ID (auto-generated)
        - Name
        - Biography
        - List of Books (one-to-many relationship with Book)
2. **Mapping:** Use Hibernate annotations to establish a one-to-many relationship between Author and Book. An author can write multiple books, while a book has only one author.
3. **Repository Layer:** Create a repository interface for both entities to perform CRUD operations using Hibernate.
4. **Service Layer:** Implement a service class that provides methods for:
    - Adding a new book and author
    - Updating book details
    - Retrieving all books and their authors
    - Searching for books by title or author name
5. **Transaction Management:** Ensure that all operations that modify data are wrapped in a transaction to maintain data integrity.

Answer-

Code Implementation: Library Management System

---

1. Project Setup

1. **Create a new Spring Boot Project in Spring Tool Suite (STS):**
   - **Dependencies:**
     - **Spring Web**: For RESTful APIs.
     - **Spring Data JPA**: For database operations.
     - **H2 Database**: For an in-memory database.

---

2. Dependencies (`pom.xml`)

```
<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>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

---

3. Database Configuration (`application.properties`)

```
spring.datasource.url=jdbc:h2:mem:librarydb
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
```

---

4. Entity Layer

`Book` Entity

```
package com.example.library.entity;

import jakarta.persistence.*;
import java.util.Date;

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String isbn;
    @Temporal(TemporalType.DATE)
    private Date publishedDate;
    private int numberOfCopies;

    @ManyToOne
    @JoinColumn(name = "author_id", nullable = false)
    private Author author;

    // Getters and Setters
}
```

`Author` Entity

```
package com.example.library.entity;

import jakarta.persistence.*;
import java.util.List;

@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    @Lob
    private String biography;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Book> books;

    // Getters and Setters
}
```

---

5. Repository Layer

`AuthorRepository`

```
package com.example.library.repository;

import com.example.library.entity.Author;
import org.springframework.data.jpa.repository.JpaRepository;

public interface AuthorRepository extends JpaRepository<Author, Long> {
}
```

`BookRepository`

```
package com.example.library.repository;

import com.example.library.entity.Book;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface BookRepository extends JpaRepository<Book, Long> {
    List<Book> findByTitleContaining(String title);
    List<Book> findByAuthorNameContaining(String authorName);
}
```

---

6. Service Layer

LibraryService

```
package com.example.library.service;

import com.example.library.entity.Author;
import com.example.library.entity.Book;
import com.example.library.repository.AuthorRepository;
import com.example.library.repository.BookRepository;
import jakarta.transaction.Transactional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class LibraryService {

    @Autowired
    private AuthorRepository authorRepository;

    @Autowired
    private BookRepository bookRepository;

    @Transactional
    public Book addBook(Book book, Long authorId) {
        Author author = authorRepository.findById(authorId)
                .orElseThrow(() -> new RuntimeException("Author not
found!"));
        book.setAuthor(author);
        return bookRepository.save(book);
    }

    public Author addAuthor(Author author) {
        return authorRepository.save(author);
    }

    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }

    public List<Book> searchBooksByTitle(String title) {
        return bookRepository.findByTitleContaining(title);
    }

    public List<Book> searchBooksByAuthor(String authorName) {
        return bookRepository.findByAuthorNameContaining(authorName);
    }
}
```

7. Controller Layer

`LibraryController`

```
package com.example.library.controller;

import com.example.library.entity.Author;
import com.example.library.entity.Book;
import com.example.library.service.LibraryService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/library")
public class LibraryController {

    @Autowired
    private LibraryService libraryService;

    @PostMapping("/authors")
    public Author addAuthor(@RequestBody Author author) {
        return libraryService.addAuthor(author);
    }

    @PostMapping("/books/{authorId}")
    public Book addBook(@RequestBody Book book, @PathVariable Long
authorId) {
        return libraryService.addBook(book, authorId);
    }

    @GetMapping("/books")
    public List<Book> getAllBooks() {
        return libraryService.getAllBooks();
    }

    @GetMapping("/books/title")
    public List<Book> searchBooksByTitle(@RequestParam String title) {
        return libraryService.searchBooksByTitle(title);
    }

    @GetMapping("/books/author")
    public List<Book> searchBooksByAuthor(@RequestParam String authorName)
{
        return libraryService.searchBooksByAuthor(authorName);
    }
}
```

8. Running and Testing the Application

- **Run the Application:**
  Launch the application from STS.

- **Access H2 Console:**
  Navigate to `http://localhost:8080/h2-console` to view the database. Use the following credentials:
  - **JDBC URL:** `jdbc:h2:mem:librarydb`
  - **Username:** `sa`
  - **Password:** `password`
- **Test API Endpoints:**
  - Add an author:
    `POST /api/library/authors`
    Request Body:
  - `{`
  - `    "name": "J.K. Rowling",`
  - `    "biography": "Author of Harry Potter series"`
  - `}`
  - Add a book:
    `POST /api/library/books/1`
    Request Body:
  - `{`
  - `    "title": "Harry Potter and the Philosopher's Stone",`
  - `    "isbn": "9780747532699",`
  - `    "publishedDate": "1997-06-26",`
  - `    "numberOfCopies": 10`
  - `}`
  - Search for books by title:
    `GET /api/library/books/title?title=Harry`

Explanation of the Process

1. **Project Setup:**
   - Initialized a Spring Boot project with required dependencies.
   - Configured H2 as the in-memory database.
2. **Entity Relationships:**
   - Defined `Author` and `Book` entities with a **one-to-many relationship** using Hibernate annotations.
   - Mapped the relationship bidirectionally for ease of use.
3. **Repository and Service Layers:**
   - Created repository interfaces for basic CRUD operations.
   - Implemented business logic in the service layer with transactional consistency.
4. **RESTful Endpoints:**
   - Built a REST API for adding and retrieving books and authors.
   - Included search functionality for books by title and author name.
5. **Testing:**
   - Used Postman to test endpoints.
   - Verified database schema and data integrity via the H2 Console.

This system is modular, scalable, and easy to integrate with a frontend or external systems.

Problem Statement 8: Online Course Registration System

Background: You are developing an online course registration system that allows students to enroll in courses. The system should manage courses, students, and their registrations efficiently.

Requirements:

1. **Entities:**
   - **Course Entity:** Create a Course entity class with the following attributes:
     - ID (auto-generated)
     - Title
     - Description
     - Credits
     - List of Students (many-to-many relationship)
   - **Student Entity:** Create a Student entity class with the following attributes:
     - ID (auto-generated)
     - Name
     - Email
     - List of Courses (many-to-many relationship)
2. **Mapping:** Use Hibernate annotations to establish a many-to-many relationship between Student and Course. Each student can enroll in multiple courses, and each course can have multiple students.
3. **Repository Layer:** Create repository interfaces for both entities to perform CRUD operations using Hibernate.
4. **Service Layer:** Implement a service class that provides methods for:
   - Enrolling a student in a course
   - Unenrolling a student from a course
   - Retrieving all courses for a student
   - Retrieving all students enrolled in a course
5. **Transaction Management:** Ensure that all enrollment and unenrollment operations are wrapped in a transaction to maintain consistency.

Project Setup

1. **IDE**: Use **Spring Tool Suite (STS)**.
2. **Dependencies**: Add the following dependencies to your `pom.xml`:

```
xml
Copy code
<dependencies>
    <!-- Spring Boot Starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <!-- Spring Boot Starter Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- H2 Database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

3. **Application Properties** (`src/main/resources/application.properties`):

```
properties
Copy code
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Code Implementation

1. Entities

- **Student Entity**

```
java
Copy code
package com.example.demo.entity;

import jakarta.persistence.*;
import java.util.Set;

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @Column(unique = true)
    private String email;

    @ManyToMany(mappedBy = "students")
    private Set<Course> courses;

    // Getters and Setters
}
```

- **Course Entity**

```
java
```

```
Copy code
package com.example.demo.entity;

import jakarta.persistence.*;
import java.util.Set;

@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;
    private String description;
    private int credits;

    @ManyToMany
    @JoinTable(
        name = "course_student",
        joinColumns = @JoinColumn(name = "course_id"),
        inverseJoinColumns = @JoinColumn(name = "student_id")
    )
    private Set<Student> students;

    // Getters and Setters
}
```

---

2. Repository Layer

- **Student Repository**

```
java
Copy code
package com.example.demo.repository;

import com.example.demo.entity.Student;
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<Student,
Long> {
}
```

- **Course Repository**

```
java
Copy code
package com.example.demo.repository;

import com.example.demo.entity.Course;
import org.springframework.data.jpa.repository.JpaRepository;

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

---

3. Service Layer

```
java
Copy code
package com.example.demo.service;

import com.example.demo.entity.Course;
import com.example.demo.entity.Student;
import com.example.demo.repository.CourseRepository;
import com.example.demo.repository.StudentRepository;
import jakarta.transaction.Transactional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class CourseRegistrationService {
    @Autowired
    private StudentRepository studentRepository;

    @Autowired
    private CourseRepository courseRepository;

    @Transactional
    public Student enrollStudentInCourse(Long studentId, Long courseId) {
        Optional<Student> studentOpt =
studentRepository.findById(studentId);
        Optional<Course> courseOpt = courseRepository.findById(courseId);

        if (studentOpt.isPresent() && courseOpt.isPresent()) {
            Student student = studentOpt.get();
            Course course = courseOpt.get();

            student.getCourses().add(course);
            course.getStudents().add(student);

            studentRepository.save(student);
            courseRepository.save(course);

            return student;
        }
        throw new RuntimeException("Student or Course not found!");
    }

    @Transactional
    public void unenrollStudentFromCourse(Long studentId, Long courseId) {
        Optional<Student> studentOpt =
studentRepository.findById(studentId);
        Optional<Course> courseOpt = courseRepository.findById(courseId);

        if (studentOpt.isPresent() && courseOpt.isPresent()) {
            Student student = studentOpt.get();
            Course course = courseOpt.get();

            student.getCourses().remove(course);
            course.getStudents().remove(student);

            studentRepository.save(student);
            courseRepository.save(course);
        } else {
            throw new RuntimeException("Student or Course not found!");
        }
    }

    public List<Course> getCoursesForStudent(Long studentId) {
        return studentRepository.findById(studentId)
                .map(Student::getCourses)
                .map(List::copyOf)
                .orElseThrow(() -> new RuntimeException("Student not
found!"));
    }

    public List<Student> getStudentsForCourse(Long courseId) {
        return courseRepository.findById(courseId)
                .map(Course::getStudents)
                .map(List::copyOf)
                .orElseThrow(() -> new RuntimeException("Course not
found!"));
    }
}
```

4. Controller Layer

```
java
Copy code
package com.example.demo.controller;

import com.example.demo.entity.Course;
import com.example.demo.entity.Student;
import com.example.demo.service.CourseRegistrationService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class CourseRegistrationController {
    @Autowired
    private CourseRegistrationService service;

    @PostMapping("/enroll/{studentId}/{courseId}")
    public String enrollStudent(@PathVariable Long studentId, @PathVariable
Long courseId) {
        service.enrollStudentInCourse(studentId, courseId);
        return "Student enrolled successfully!";
    }

    @DeleteMapping("/unenroll/{studentId}/{courseId}")
    public String unenrollStudent(@PathVariable Long studentId,
@PathVariable Long courseId) {
        service.unenrollStudentFromCourse(studentId, courseId);
        return "Student unenrolled successfully!";
    }

    @GetMapping("/courses/{studentId}")
    public List<Course> getCoursesForStudent(@PathVariable Long studentId)
{
        return service.getCoursesForStudent(studentId);
    }
    @GetMapping("/students/{courseId}")
    public List<Student> getStudentsForCourse(@PathVariable Long courseId)
{
        return service.getStudentsForCourse(courseId);
    }
}
```

Running the Project

1. **Run the Application**
   Use the main application class:

   ```
   java
   Copy code
   @SpringBootApplication
   public class OnlineCourseRegistrationSystemApplication {
       public static void main(String[] args) {

   SpringApplication.run(OnlineCourseRegistrationSystemApplication.class
   , args);
       }
   }
   ```

2. **Access the H2 Console**
   Visit `http://localhost:8080/h2-console`. Use the default credentials in `application.properties`.
3. **Test Endpoints**
   Use tools like **Postman** or **cURL** to interact with the REST endpoints.

Explanation

Here's a detailed explanation of what has been done in the development of the **Online Course Registration System**:

---

1. Setting Up the Project

The system is built using **Spring Boot** in **Spring Tool Suite (STS)**. Key tasks included:

- **Dependencies:**
  Added dependencies for Spring Boot Web, Data JPA, and H2 Database in the `pom.xml`. These libraries provide the foundation for creating REST APIs, managing database interactions, and running an embedded database for testing and development.
- **Database Configuration:**
  Configured H2 in-memory database via `application.properties`. This allows rapid development and testing without needing an external database.

---

2. Defining Entities

The main components of the system are **Courses** and **Students**, each represented as an entity.

- **Student Entity:**
  Represents a student with attributes like ID, name, and email. A student can enroll in multiple courses.
  - `@Entity`: Marks this class as a database table.
  - `@ManyToMany(mappedBy = "students")`: Establishes a bidirectional many-to-many relationship with the `Course` entity.
- **Course Entity:**
  Represents a course with attributes like title, description, credits, and a list of enrolled students.
  - `@ManyToMany`: Establishes a many-to-many relationship with the `Student` entity using a join table named `course_student`.

These annotations ensure the database schema is auto-generated with the appropriate relationships when the application runs.

---

3. Repository Layer

Repositories were created for both `Student` and `Course` entities using **Spring Data JPA**:

- `StudentRepository` and `CourseRepository` extend `JpaRepository`, providing CRUD operations out of the box.
- No custom methods are added initially, as the basic operations (save, findById, delete) suffice for this application.

---

4. Service Layer

The **Service Layer** contains the business logic to handle core operations:

- **Enroll a Student in a Course:**
  Fetches the student and course by their IDs, establishes the relationship in both directions (student -> course and course -> student), and saves the updated entities.
- **Unenroll a Student from a Course:**
  Removes the student-course relationship and persists the changes.
- **Retrieve Courses for a Student:**
  Fetches all courses associated with a given student.
- **Retrieve Students for a Course:**
  Fetches all students associated with a given course.
- **Transaction Management:**
  Annotated methods with `@Transactional` to ensure database consistency. If any operation fails, all changes within that transaction are rolled back.

---

5. Controller Layer

The **Controller Layer** exposes RESTful endpoints to interact with the service layer:

- **Endpoints for Enrollment Management:**
  - `POST /api/enroll/{studentId}/{courseId}`: Enrolls a student in a course.
  - `DELETE /api/unenroll/{studentId}/{courseId}`: Removes a student from a course.
- **Endpoints for Data Retrieval:**
  - `GET /api/courses/{studentId}`: Retrieves all courses for a given student.
  - `GET /api/students/{courseId}`: Retrieves all students enrolled in a given course.

These endpoints allow clients (e.g., Postman, frontend apps) to interact with the system using HTTP requests.

---

6. Testing the Application

- **H2 Console:**
  Accessed via `http://localhost:8080/h2-console` to view and manipulate the database tables created during runtime.
  - Username: `sa`
  - Password: `password`
- **Postman/Curl:**
  Used to test the endpoints by sending HTTP requests. Example requests:
  - **Enroll a student:**
    `POST http://localhost:8080/api/enroll/1/2`
  - **Get courses for a student:**
    `GET http://localhost:8080/api/courses/1`

---

7. Features Implemented

1. **Bidirectional Many-to-Many Relationship:**
   Students and courses are linked via a join table (`course_student`). This relationship ensures that changes to one side reflect on the other.
2. **Transactional Consistency:**
   Operations like enrollment and unenrollment are wrapped in transactions, ensuring no partial updates occur.
3. **RESTful API:**
   Exposed endpoints for CRUD operations and retrievals, adhering to REST principles.
4. **In-Memory Database:**
   H2 database provides an easy-to-use and lightweight data store for development and testing.

5. **Auto-Generated Schema:**
   Spring Boot and JPA automatically generate the database schema based on the entity definitions.

How It Works

1. **Data Initialization (Optional):**
   - Add some sample data by defining `CommandLineRunner` beans in the main application class to populate the database on startup.
2. **Enroll Students:**
   - Use the `POST /api/enroll/{studentId}/{courseId}` endpoint to associate students with courses.
3. **Fetch Data:**
   - Retrieve a student's courses or a course's students using the provided endpoints.
4. **Unenroll Students:**
   - Use the `DELETE /api/unenroll/{studentId}/{courseId}` endpoint to remove a student from a course.

Summary

This system is a simple yet powerful demonstration of managing many-to-many relationships in Spring Boot with Hibernate:

- **Efficiency:** Leverages Spring Data JPA for seamless CRUD operations.
- **Scalability:** Can easily be extended with new features (e.g., user authentication, UI integration).
- **Maintainability:** The clear separation of concerns (entities, repositories, service, and controllers) ensures clean and maintainable code.

Let me know if you need more clarifications or enhancements!