**Assignment Title: Building a Book Store Web Application using Spring MVC and Hibernate**

**Objective:** To create a web application that uses Spring MVC as the framework and Hibernate for data persistence.

**Scenario:** You have been hired as a Full Stack Developer at XYZ Book Store. The management wants you to develop a web application to manage their store digitally. They want functionalities like adding a new book, updating book information, deleting a book, and viewing all the books in the store. They would like you to use the Spring MVC Framework for the application's architecture and Hibernate for handling the database operations.

**Tasks:**

**1. Setup and Initial Configuration:**

* Set up a Spring MVC project using Spring Boot.
* Configure Hibernate with the database of your choice. MySQL or PostgreSQL can be used.

**2. Database Design:**

* Design the "Book" entity with fields like id, title, author, ISBN, price, and numberOfCopies.
* Design a "Store" entity with fields like id, name, and location.
* Create a many-to-many relationship between the Book and Store entities to represent the books present at each store.

**3. Implement Hibernate DAO Layer:**

* Create a DAO Interface for each entity for the CRUD operations.
* Implement the interfaces using Hibernate, creating methods for adding, updating, deleting, and fetching books and stores.

**4. Implement Service Layer:**

* Create a Service Interface for each entity to define the service methods.
* Implement the Service Interfaces. Use the DAO layer methods to add, update, delete, and fetch books and stores.

**5. Create Spring MVC Controllers:**

* Create Controllers for each entity using Spring MVC.
* Include methods for adding, updating, deleting, and fetching books and stores.

**6. Design the Views:**

* Create views for adding a new book/store, updating a book/store, deleting a book/store, and viewing all the books/stores.
* Implement a search function in the book list view to quickly find a specific book.

**7. Implement Form Validation:**

* Use Hibernate Validator for form validation. Ensure all necessary fields are filled, and validate fields like ISBN and price.

**8. Implement a Role-based Access Control:**

* Implement a role-based access control mechanism. For instance, only "Admin" users should be able to add, update, or delete books/stores.

**9. Exception Handling:**

* Implement a global exception handling mechanism using **@ControllerAdvice**. Handle exceptions such as a book not found, insufficient stock, etc.

**10. Testing:**

* Perform unit testing for the DAO and Service layers using JUnit and Mockito.
* Implement integration testing for the overall application using Spring Boot Test.

**11. Logging and Monitoring:**

* Implement a logging mechanism using Logback or SLF4J.
* Implement a basic monitoring mechanism using Spring Boot Actuator.

This assignment will offer a comprehensive experience in building a web application using Spring MVC and Hibernate while covering different aspects of web development like database relationships, form validation, access control, exception handling, testing, logging, and monitoring.

**Database deign**

CREATE TABLE Book (

id INT AUTO\_INCREMENT,

title VARCHAR(255),

author VARCHAR(255),

ISBN VARCHAR(20),

PRIMARY KEY (id)

);

INSERT INTO Book (title, author, ISBN)

VALUES ('The Great Gatsby', 'F. Scott Fitzgerald', '9780743273565'),

('To Kill a Mockingbird', 'Harper Lee', '9780060935467'),

('1984', 'George Orwell', '9780451524935');

CREATE TABLE Store (

id INT AUTO\_INCREMENT,

name VARCHAR(255),

location VARCHAR(255),

PRIMARY KEY (id)

);

INSERT INTO Store (name, location)

VALUES ('Central Store', '101 Main Street'),

('North Branch', '500 Maple Avenue'),

('South Branch', '999 Oak Drive');

CREATE TABLE Store\_Book (

store\_id INT,

book\_id INT,

numberOfCopies INT,

price DECIMAL(5,2),

PRIMARY KEY (store\_id, book\_id),

FOREIGN KEY (store\_id) REFERENCES Store(id),

FOREIGN KEY (book\_id) REFERENCES Book(id)

);

INSERT INTO Store\_Book (store\_id, book\_id, numberOfCopies, price)

VALUES (1, 1, 5, 10.99),

(1, 2, 2, 8.99),

(1, 3, 10, 9.99),

(2, 1, 3, 11.99),

(2, 2, 5, 8.49),

(3, 1, 1, 12.99),

(3, 2, 3, 7.99),

(3, 3, 6, 9.49);

These SQL commands will create tables for "Book", "Store", and "Store\_Book", and populate them with some initial data. Note that MySQL automatically assigns an ID to each new book and store because we set **id** as **AUTO\_INCREMENT**

Entity classes

import javax.persistence.\*;

import java.util.Set;

@Entity

@Table(name = "Book")

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column(name = "title")

private String title;

@Column(name = "author")

private String author;

@Column(name = "ISBN")

private String isbn;

@OneToMany(mappedBy = "book")

private Set<StoreBook> storeBooks;

// Constructors, Getters, and Setters omitted for brevity

}

import javax.persistence.\*;

import java.util.Set;

@Entity

@Table(name = "Store")

public class Store {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column(name = "name")

private String name;

@Column(name = "location")

private String location;

@OneToMany(mappedBy = "store")

private Set<StoreBook> storeBooks;

// Constructors, Getters, and Setters omitted for brevity

}

import javax.persistence.\*;

@Entity

@Table(name = "Store\_Book")

public class StoreBook {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@ManyToOne

@JoinColumn(name = "store\_id", nullable = false)

private Store store;

@ManyToOne

@JoinColumn(name = "book\_id", nullable = false)

private Book book;

@Column(name = "numberOfCopies")

private int numberOfCopies;

@Column(name = "price")

private double price;

// Constructors, Getters, and Setters omitted for brevity

}

Note: Due to the **numberOfCopies** and **price** fields being specific to each store, a typical many-to-many relationship is not used. Instead, a **@ManyToOne** association from the **StoreBook** class to both **Book** and **Store** is created. This way, you can better handle the attributes specific to the association of a book with a store.

Next, you would create DAO, Service, and Controller classes for each entity to handle the database operations and web requests. The complete code for these classes would be too lengthy to include here, but each DAO class would have methods for CRUD operations, each Service class would use the DAO methods, and each Controller class would handle the web requests and responses.

Don't forget to include appropriate Hibernate and Spring annotations in your code to facilitate the ORM and MVC operations.

Dao classes

import java.util.List;

public interface BookDAO {

List<Book> getBooks();

Book getBook(int id);

void saveBook(Book book);

void deleteBook(int id);

}

import org.hibernate.Session;

import org.hibernate.SessionFactory;

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

import org.springframework.stereotype.Repository;

import javax.transaction.Transactional;

import java.util.List;

@Repository

public class BookDAOImpl implements BookDAO {

@Autowired

private SessionFactory sessionFactory;

@Override

public List<Book> getBooks() {

Session currentSession = sessionFactory.getCurrentSession();

return currentSession.createQuery("from Book", Book.class).getResultList();

}

@Override

public Book getBook(int id) {

Session currentSession = sessionFactory.getCurrentSession();

return currentSession.get(Book.class, id);

}

@Override

public void saveBook(Book book) {

Session currentSession = sessionFactory.getCurrentSession();

currentSession.saveOrUpdate(book);

}

@Override

public void deleteBook(int id) {

Session currentSession = sessionFactory.getCurrentSession();

currentSession.delete(currentSession.get(Book.class, id));

}

}

import java.util.List;

public interface StoreDAO {

List<Store> getStores();

Store getStore(int id);

void saveStore(Store store);

void deleteStore(int id);

}

import org.hibernate.Session;

import org.hibernate.SessionFactory;

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

import org.springframework.stereotype.Repository;

import javax.transaction.Transactional;

import java.util.List;

@Repository

public class StoreDAOImpl implements StoreDAO {

@Autowired

private SessionFactory sessionFactory;

@Override

public List<Store> getStores() {

Session currentSession = sessionFactory.getCurrentSession();

return currentSession.createQuery("from Store", Store.class).getResultList();

}

@Override

public Store getStore(int id) {

Session currentSession = sessionFactory.getCurrentSession();

return currentSession.get(Store.class, id);

}

@Override

public void saveStore(Store store) {

Session currentSession = sessionFactory.getCurrentSession();

currentSession.saveOrUpdate(store);

}

@Override

public void deleteStore(int id) {

Session currentSession = sessionFactory.getCurrentSession();

currentSession.delete(currentSession.get(Store.class, id));

}

}

Remember to annotate with **@Transactional** on a service class or method when you will interact with the database. The **@Transactional** annotation is responsible for ensuring that a method or class participates in a transaction. It will start a new transaction if one isn't already running, or it will use the currently running transaction if one is already started.

Please ensure that these classes are in the right packages so Spring can discover them during component scanning. In the real world, you would typically include exception handling in these DAO implementations as well.