**Exercise 13: Online Bookstore - Unit Testing REST Controllers**

Business Scenario:

Write unit tests for your bookstore's REST controllers using JUnit and Mockito.

1. **JUnit Setup:**

* **JUnit and Mockito Dependencies: Start by adding the necessary dependencies for JUnit and Mockito in your pom.xml (for Maven) or build.gradle (for Gradle). This will include spring-boot-starter-test, which provides built-in support for JUnit and Mockito.**
* **Annotations: Use the @RunWith(SpringRunner.class) annotation to enable Spring support in JUnit tests. The @WebMvcTest annotation can be used to focus on testing the web layer.**

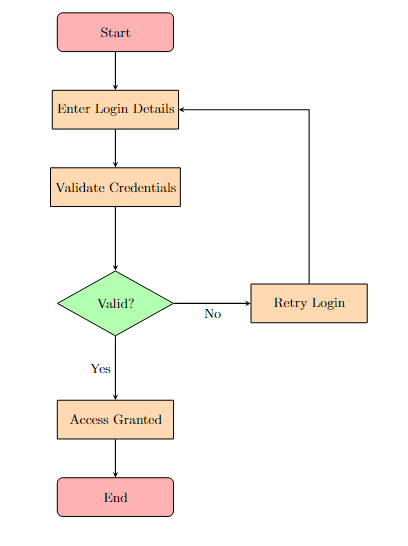
**2. MockMvc for Testing REST Controllers:**

* **MockMvc Setup: Use MockMvc to simulate HTTP requests to your REST controllers. You can set up MockMvc using @AutoConfigureMockMvc or injecting it directly into your test class.**
* **Testing HTTP Methods: Use methods like mockMvc.perform(get("/books")) to simulate GET requests. Chain additional methods like .andExpect(status().isOk()) to verify the response status.**
* **Controller Layer Isolation: Leverage @MockBean to mock the service layer, ensuring your tests are focused solely on the controller logic. This isolates the controller layer from the service and repository layers.**

**3. Test Coverage and Best Practices:**

* **Comprehensive Test Cases: Ensure that you write tests for all major controller endpoints, including edge cases. For example, test for cases like valid and invalid input, missing resources (404 errors), and successful/unsuccessful operations (200, 400, 500 status codes).**
* **Descriptive Test Methods: Write clear and descriptive method names in your test classes to indicate what each test is validating. For instance, shouldReturnListOfBooksWhenGetBooksIsCalled.**
* **Assertions: Use assertions like assertThat() or assertEquals() to validate the response content and behavior of your controllers.**

**FLOWCHART :**



Explanation :

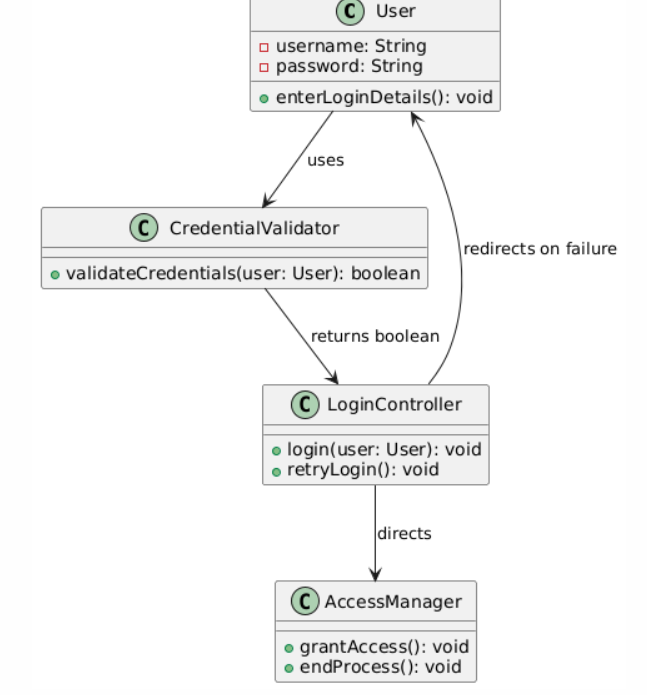
* 1. **Start:** The process begins with the "Start" node.
  2. **Enter Login Details:** The user enters their login details (e.g., username and password).
  3. **Validate Credentials:** The system checks if the entered credentials are valid.
  4. **Decision (Valid?):** A decision is made based on the credential validation:

**Yes:** If the credentials are valid, the system grants access.

**No:** If the credentials are invalid, the user is prompted to retry login.

* 1. **Access Granted:** If access is granted, the process moves to the "End".
  2. **Retry Login:** If the login fails, the user is redirected to re-enter their login details.
  3. **End:** The process concludes at the "End" node.

**CLASS DIAGRAM :**



**Explanation:**

1. **Classes:**
   * **User:** Represents the user entity with username and password attributes. The enterLoginDetails() method allows the user to input their credentials.
   * **CredentialValidator:** Handles the logic of validating the user's credentials through the validateCredentials() method.
   * **LoginController:** Manages the login process, including invoking the login() method and handling retries with retryLogin().
   * **AccessManager:** Handles access control, including granting access and ending the process.
2. **Relationships:**
   * **User → CredentialValidator:** The User class uses the CredentialValidator to validate credentials.
   * **CredentialValidator → LoginController:** The CredentialValidator returns a boolean to the LoginController to indicate whether the login was successful.
   * **LoginController → AccessManager:** The LoginController directs the process to the AccessManager if login is successful.
   * **LoginController → User:** The LoginController redirects the user back to input login details if validation fails.