## **Summary of Tests Needed for Library Management Project**

This document outlines the categories and types of tests that should be considered for the Library Management System project, based on the provided Software Requirements Specification (SRS), the SE507 Coursework guidelines, and concepts from your lectures. The aim is to develop a comprehensive testing strategy rather than a fixed number of tests.

### **I. Functional Testing (Black-Box Testing - Based on SRS)**

These tests verify that the system behaves as specified in the SRS. Techniques like **Equivalence Partitioning**, **Boundary Value Analysis**, **Use Case Testing**, and **Error Guessing** should be employed.

**A. Authentication & Authorization (SRS 3.1, 2.3)**

* **User Registration (Librarian creating users - SRS 3.1.1):**
  + Successful registration of a new Librarian.
  + Successful registration of a new Member.
  + Attempt registration with an existing email (testing uniqueness constraint).
  + Attempt registration with missing required fields (name, email, password, isAdmin).
  + Attempt registration with invalid data formats (e.g., invalid email format).
* **User Login (SRS 3.1.2):**
  + Successful login with valid Librarian credentials.
  + Successful login with valid Member credentials.
  + Attempt login with invalid email.
  + Attempt login with correct email but incorrect password.
  + Attempt login with empty email/password fields.
  + Verify redirection to the correct page based on role (Librarian to Dashboard, Member to Books).
* **User Logout (SRS 3.1.3):**
  + Verify successful logout and session termination.
  + Verify redirection to login page after logout.
* **Role-Based Access Control (RBAC - SRS 2.3, and throughout Section 3):**
  + **Librarian Access:**
    - Verify Librarians can access and use all CRUD operations for Books, Authors, Genres, Borrowals, Users, and Reviews.
    - Verify Librarians can access the Dashboard.
  + **Member Access:**
    - Verify Members can view Books, Authors, Genres.
    - Verify Members can view their own borrowal history.
    - Verify Members can create new borrowal requests for themselves.
    - Verify Members *cannot* access Librarian-only pages (e.g., User Management, Dashboard with admin stats).
    - Verify Members *cannot* perform CRUD operations reserved for Librarians (e.g., adding a new book, deleting a user).

**B. Entity Management (CRUD Operations - SRS 3.3 - 3.8)**

For each entity (Books, Authors, Genres, Users, Borrowals, Reviews), test the following operations, considering both Librarian and Member perspectives where applicable:

* **Create (Add New):**
  + Successful creation with all valid and required data.
  + Successful creation with valid optional data.
  + Attempt creation with missing required fields.
  + Attempt creation with invalid data types/formats for fields.
  + Attempt creation with data that violates constraints (e.g., duplicate unique identifiers like ISBN for books, or email for users if not handled by registration tests – testing data integrity).
  + Verify correct default values are set (e.g., isAvailable for books).
* **Read (View/Get):**
  + View list of all items (e.g., all books, all authors).
    - Test with an empty list.
    - Test with a list containing one item.
    - Test with a list containing multiple items (consider pagination if implemented).
  + View details of a specific item using its ID.
    - Test with a valid/existing ID.
    - Test with an invalid/non-existent ID.
* **Update (Edit):**
  + Successful update of an existing item with valid data.
  + Attempt to update with missing required fields.
  + Attempt to update with invalid data types/formats.
  + Attempt to update a non-existent item.
  + Verify that read-only fields (if any) cannot be updated.
  + Verify constraints are maintained upon update (e.g., cannot update an email to one that already exists for another user).
* **Delete:**
  + Successful deletion of an existing item.
  + Attempt to delete a non-existent item.
  + Consider cascading effects or restrictions and referential integrity (e.g., deleting an author with associated books – what is the expected behavior as per design? The SRS implies simple CRUD, but this is a good point for clarification/testing data integrity).

**C. Specific Feature Testing**

* **Librarian Dashboard (SRS 3.2):**
  + Verify all statistical widgets and charts are displayed.
  + Verify data (even if sample/placeholder as noted in SRS) is rendered correctly.
* **Book Management (SRS 3.3):**
  + Test linking/unlinking of authors and genres to books, ensuring data consistency.
  + Test isAvailable status functionality and its impact on borrowability.
* **Borrowal Management (SRS 3.6):**
  + Librarian creating a borrowal record.
  + Member creating a borrowal request for themselves.
  + Updating borrowal status (e.g., "Borrowed" to "Returned", "Overdue").
  + Verify due date calculations/setting.
  + Filtering borrowal history for a specific member accurately.
* **User Management (Librarian - SRS 3.7):**
  + Librarian viewing all users vs. all members.
  + Updating user roles (isAdmin flag) and verifying access changes.
* **Review Management (SRS 3.8):**
  + Test adding, viewing, updating, deleting reviews (consider both Librarian and potential Member actions if UI allows).
  + Test star rating input (valid/invalid range) and display.

**D. Use Case Testing (Based on SRS Section 7 and implicit use cases)**

* For each defined Use Case (UC-001 to UC-005) and other implicit use cases (like any CRUD operation):
  + Test the Main Flow (happy path).
  + Test all specified Alternative Flows and Exception Flows, ensuring graceful error handling and informative messages.
  + Example for UC-002 (Add New Book):
    - Main flow: Librarian successfully adds a book.
    - Alternative: Validation error (e.g., missing book name), verify clear message.
    - Alternative: API error during submission, verify user feedback and system state.

**E. State Transition Testing**

* Based on Finite State Machine (FSM) concepts, this technique is useful for testing features with distinct states and transitions.
* **Identify Entities/Features with States:**
  + Borrowal Status: (e.g., Requested -> Approved -> Borrowed -> Returned / Overdue -> Archived).
  + Book Availability: (e.g., Available -> Borrowed -> Reserved -> Available).
  + User Session: (e.g., Not Logged In -> Logged In -> Session Expired -> Logged Out).
* **Design Test Cases:**
  + Test every valid transition between states.
  + Test invalid transitions (e.g., trying to return a book that was never borrowed).
  + Test sequences of transitions (e.g., a full borrowal lifecycle).
  + Test events that trigger transitions (e.g., member requests book, librarian approves, due date passes).

### **II. API Testing**

Test the backend API endpoints directly. This can be done using tools like Postman or automated scripts. For each endpoint defined in the SRS (e.g., POST /api/auth/register, GET /api/book/getAll):

* **Valid Requests:**
  + Send requests with valid data and correct HTTP methods.
  + Verify correct HTTP status codes (e.g., 200 OK, 201 Created).
  + Verify the structure and content of the response body.
* **Invalid Requests:**
  + Send requests with missing required parameters/fields in the payload.
  + Send requests with incorrect data types or out-of-range values.
  + Test boundary values for parameters.
  + Verify appropriate error status codes (e.g., 400 Bad Request, 404 Not Found, 401 Unauthorized, 403 Forbidden).
  + Verify error messages in the response body are informative and do not expose sensitive information.
* **Authentication/Authorization:**
  + For protected endpoints, verify access is denied without a valid token/session.
  + Verify access is denied if a user with insufficient privileges attempts to access an endpoint.
* **Idempotency (for PUT, DELETE where applicable):**
  + Verify that making the same request multiple times has the same effect as making it once.

### **III. White-Box Testing (Logic Coverage - Based on Source Code)**

These tests require looking at the internal structure of the code.

* **Select Coverage Criteria:** Decide on a level of coverage (e.g., Statement, Decision/Branch, Condition, Multiple-Condition). Decision/Condition coverage is often a good balance. Path coverage for critical/complex modules can also be considered.
* **Identify Key Modules/Functions:** Focus on complex logic in controllers (e.g., authController.js, bookController.js, borrowalController.js), models, and any helper functions.
* **Design Test Cases to Cover Paths:**
  + For conditional statements (if, else if, else), ensure tests execute each branch.
  + For loops, test conditions for entering the loop, iterating within it (zero, one, multiple times), and exiting.
  + Test error handling paths (e.g., try-catch blocks), ensuring errors are caught and handled appropriately.
  + Example: In authController.loginUser:
    - Test path where user is not found.
    - Test path where user is found but password validation fails.
    - Test path where login is successful.
  + Example: In bookController.addBook:
    - Test path for successful book save.
    - Test path where book.save() throws an error.
* **Data Flow Testing (Consider for complex variable interactions):**
  + Identify definition-use pairs of critical variables.
  + Design tests to cover paths from variable definitions to their uses (all-defs, all-uses coverage).

### **IV. Non-Functional Testing**

**A. Security Testing (SRS 5.1)**

* **Password Security:** Verify passwords are not stored in plain text (SRS mentions pbkdf2Sync with SHA512 – conceptually verify this). Test password policies if any (e.g., complexity, length).
* **Session Management:** Test session expiration, secure handling of session cookies/tokens (e.g., HttpOnly, Secure flags). Test for session fixation if applicable.
* **Input Validation (Client-side and Server-side):**
  + Attempt to bypass client-side validation and submit malicious data directly to the API.
  + Test for common vulnerabilities like Cross-Site Scripting (XSS) by inputting script tags in all relevant text fields.
  + Test for injection vulnerabilities if applicable (e.g., NoSQL injection patterns for MongoDB).
* **Access Control:** Rigorously re-test role-based access to ensure no unauthorized data access or modification is possible through direct API calls or UI manipulation.
* **Sensitive Data Exposure:** Ensure no sensitive data (e.g., full error stacks, other users' private info) is leaked in responses or logs.

**B. Usability Testing (SRS 5.2)**

* **User Interface (UI) and Navigation:**
  + Is the navigation intuitive and consistent?
  + Are forms easy to understand and fill? Are labels clear?
  + Are error messages clear, helpful, and user-friendly (avoiding technical jargon)?
  + Is feedback provided for user actions (e.g., success/error toasts, loading indicators)?
* **Responsiveness (SRS 4.1):** Test the application on different screen sizes (desktop, tablet, mobile – using browser dev tools or emulators).
* **Learnability:** How easy is it for a new user (Librarian or Member) to understand and use the system? (Can be assessed through exploratory testing or by having someone unfamiliar with the system try it).
* **Efficiency:** Can users complete common tasks efficiently?

**C. Performance Testing (Basic)**

* **Page Load Times:** Observe if pages and key components load within a reasonable timeframe under typical conditions.
* **API Response Times:** For critical API calls, check if responses are quick.
* **Resource Consumption (Client-side):** Basic check for excessive browser memory or CPU usage during common operations.
* **Stress/Volume (Conceptual):** While full-scale stress testing might be out of scope, consider scenarios with a higher number of records (e.g., many books, many users) to see if performance degrades noticeably in list views or searches.
* (Note: Extensive performance/load testing is likely beyond the scope of this coursework but basic observations are useful).

**D. Maintainability (SRS 5.3)**

* This is primarily assessed through Static Testing (see Section VIII), focusing on code quality, structure, and documentation.

E. Browser Compatibility Testing (SRS 2.4)

\* Test core functionalities on the latest stable versions of browsers specified or implied by "Modern web browser" (e.g., Chrome, Firefox, Edge, Safari as per SRS 2.4).

\* Check for consistent UI rendering and behavior across these browsers.

### **V. Integration Testing**

* **Client-Server Integration:** Verify that the React frontend correctly communicates with the Node.js backend API.
  + Test data submission from forms and data display from API responses, ensuring data integrity.
  + Verify correct handling of API success and error responses by the client.
* **Module Integration (Backend):**
  + Test interactions between different backend modules/services (e.g., when a book is borrowed, does its isAvailable status update correctly? When a user is created, is it correctly reflected in other parts of the system that might query users?).
* **Database Integration:** Ensure data is correctly persisted to and retrieved from MongoDB, and that schema constraints are enforced.

### **VI. Regression Testing**

* **Purpose:** To ensure that new code changes (bug fixes, new features, modifications as required by coursework) do not negatively impact existing functionalities.
* **Strategy:**
  + After every significant change, re-run a subset (or all) of previously passed test cases, especially those related to the modified area and critical functionalities.
  + Automate regression tests where possible, particularly for unit and API tests.
  + Create new test cases for the changed/new functionality and add them to the regression suite.
* This is an ongoing activity throughout the development and modification phase.

### **VII. Installation/Deployment Testing (If applicable)**

* If using Docker (as suggested by docker-compose files), test if the application can be set up and run correctly in the Docker environment as per instructions (e.g., README.md).
* Verify all services (client, server, database) start and connect correctly.

### **VIII. Static Testing (Non-Execution Based)**

This involves examining the code and documentation without executing the program. It aligns with concepts from Week 3 lectures (Inspections, Walkthroughs, Reviews).

* **Code Reviews/Inspections:**
  + **Purpose:** To identify defects, inconsistencies, and areas for improvement in the source code.
  + **Process:** Team members (or peers) systematically review code against checklists, coding standards, and design documents.
  + **Focus Areas:**
    - Logic errors.
    - Security vulnerabilities.
    - Non-adherence to coding standards.
    - Poor maintainability (complex code, lack of comments, poor structure - relates to SRS 5.3).
    - Inconsistent error handling.
    - Testability of the code.
* **Walkthroughs:**
  + Less formal than inspections, where the developer leads reviewers through the code, explaining logic and soliciting feedback.
* **Documentation Review:**
  + Review SRS, design documents, and user manuals (if any) for clarity, consistency, and completeness.

### **Conclusion**

This summary provides a framework for the types of tests needed. Your team will need to:

1. **Design specific test cases** for these areas, including:
   * Test Case ID
   * Description/Purpose
   * Preconditions
   * Test Steps
   * Test Data
   * Expected Results
   * Actual Results
   * Pass/Fail Status
2. **Document these test cases** thoroughly as required by the coursework deliverables (Section 6 of SE507-SoftwareTandT-Spring2025-Coursework.pdf).
3. **Prioritize tests** based on risk, criticality of features, and complexity.
4. Remember the principles from your lectures: aim to find errors, and understand that complete testing is often impossible. The goal is effective and well-justified testing.