## Book Renting Application

**Problem Statement:** Develop a user-friendly book renting application that enables users to browse, rent, and read books digitally. The application should cater to a diverse audience, including students, professionals, and casual readers, offering a wide range of genres and titles

1. **Book Catalog and Search Functionality:**
   a. Create a comprehensive and easily navigable catalog of books.
   **b.** Develop advanced search and filter options to help users find specific titles or genres quickly.
2. **Rental Process:**
   a. Design a seamless rental process that allows users to rent books for a specified period using an in-app wallet.
   **b.** Include options for extending rental periods and returning books.
3. **User Reviews and Ratings:**
   a. Enable users to leave reviews and ratings for books.
   b. Implement a system to highlight popular and highly rated books based on user activity.
4. **Notifications and Reminders:**
   a. Send notifications for due dates, new arrivals, and personalized recommendations. Store the generated notifications in a Table to send at a later point in time.
   b. Provide reminders for rental expirations.
5. **Admin Panel:**
   a. Develop an admin panel for managing the book catalog, user accounts, and rental transactions.
   b. Include spring actuator monitor application health.

## Gradle test generation

Create a Gradle task that generates test cases for each public Java method, using reflection to inspect the methods of my classes and generate corresponding test files if they do not already exist?

Notes:

- **Reflection**: The task uses Java reflection to inspect each class and its public methods.
- **Test File Generation**: For each class, it generates a corresponding test class if it doesn't already exist.

- **Test Methods**: It creates a placeholder test method for each public method in the original class.
-

This setup will help you automatically generate basic test files and methods, which you can then fill in with actual test logic.

## Hierarchical Graph Solver

Create a Spring Boot application that enables users to create, manage, and solve problems involving hierarchical graphs. The application should allow various graph operations, including adding nodes, defining relationships, and querying specific information. There's no need to use a database for storing the graph; instead, read an existing graph from a classpath resource and store it in memory. Package the application as a Spring Boot jar and launch it from the command line.

1. **Graph Management:**
   a. Provide endpoints to create and manage nodes and edges in the graph.
   b. Support operations to add, update, and delete nodes and relationships.
2. **Graph Querying:**
   a. Implement endpoints to query the graph for specific information, such as:
      i. Finding the path between two nodes.
      ii. Calculating the depth of a node.
      iii. Identifying the common ancestor of two nodes.
3. **Error Handling and Validation:**
   a. Implement robust error handling and input validation.
   b. Ensure the application gracefully handles invalid operations and provides meaningful error messages.

## Example Endpoints:

1. **Create Node:**
   a. **POST /nodes**
   b. Request Body: { "name": "Node1", "parentId": "ParentNodeId" }
2. **Get Node:**
   a. **GET /nodes/{id}**
3. **Add Relationship:**
   a. **POST /relationships**

    b. Request Body: { "parentId": "ParentNodeId", "childId": "ChildNodeId" }
4. **Find Path:**
    **a. GET /paths**
    b. Query Parameters: startNodeId=Node1&endNodeId=Node2
5. **Calculate Depth:**
    **a. GET /nodes/{id}/depth**
6. **Find Common Ancestor:**
    **a. GET /common-ancestor**
    b. Query Parameters: nodeId1=Node1&nodeId2=Node2

SAMPLE DATA:

- **CEO**
    - **CTO**
        - **Engineering Manager**
            - **Software Engineer 1**
            - **Software Engineer 2**
        - **QA Manager**
            - **QA Engineer 1**
            - **QA Engineer 2**
    - **CFO**
        - **Accountant 1**
        - **Accountant 2**
    - **COO**
        - **Operations Manager**
            - **Operations Specialist 1**
            - **Operations Specialist 2**

```
{
  "nodes": [
    { "id": "1", "name": "CEO" },
    { "id": "2", "name": "CTO", "parentId": "1" },
    { "id": "3", "name": "CFO", "parentId": "1" },
    { "id": "4", "name": "COO", "parentId": "1" },
    { "id": "5", "name": "Engineering Manager", "parentId": "2" },
    { "id": "6", "name": "QA Manager", "parentId": "2" },
    { "id": "7", "name": "Software Engineer 1", "parentId": "5" },
    { "id": "8", "name": "Software Engineer 2", "parentId": "5" },
    { "id": "9", "name": "QA Engineer 1", "parentId": "6" },
    { "id": "10", "name": "QA Engineer 2", "parentId": "6" },
    { "id": "11", "name": "Accountant 1", "parentId": "3" },
    { "id": "12", "name": "Accountant 2", "parentId": "3" },
    { "id": "13", "name": "Operations Manager", "parentId": "4" },
    { "id": "14", "name": "Operations Specialist 1", "parentId": "13" },
    { "id": "15", "name": "Operations Specialist 2", "parentId": "13" }
  ],
  "relationships": [
    { "parentId": "1", "childId": "2" },
    { "parentId": "1", "childId": "3" },
    { "parentId": "1", "childId": "4" },
    { "parentId": "2", "childId": "5" },
    { "parentId": "2", "childId": "6" },
    { "parentId": "5", "childId": "7" },
    { "parentId": "5", "childId": "8" },
    { "parentId": "6", "childId": "9" },
    { "parentId": "6", "childId": "10" },
    { "parentId": "3", "childId": "11" },
    { "parentId": "3", "childId": "12" },
    { "parentId": "4", "childId": "13" },
    { "parentId": "13", "childId": "14" },
    { "parentId": "13", "childId": "15" }
  ]
}
```

## Example Queries:

1. **Find Path**: Find the path from "CEO" to "Software Engineer 1".
2. **Calculate Depth**: Calculate the depth of "QA Engineer 2".
3. **Find Common Ancestor**: Find the common ancestor of "Software Engineer 1" and "QA Engineer 1".