Full-Stack Online Library with AI Insights Challenge

Overview

You will build an online library management system with two main components:

1. **Backend API (Spring Boot):**
   Develop a Spring Boot application that provides RESTful endpoints for managing books. In addition to standard CRUD operations, the API will integrate with an external AI service (e.g., OpenAI) to generate insights (such as taglines or summaries) for each book.
2. **Frontend UI (Next.js):**
   Create a simple Next.js application that communicates with your Spring Boot API. The UI will display the list of books, allow users to view detailed information about each book, and display AI-generated insights when requested.

Part 1: Spring Boot Backend

1. Domain Model

- **Book Entity:**
  Create a `Book` entity with the following fields:
  - `id`: Auto-generated unique identifier.
  - `title`: The title of the book.
  - `author`: The author of the book.
  - `isbn`: The International Standard Book Number.
  - `publicationYear`: The year the book was published.
  - `description`: A brief summary or description of the book (used for AI processing).
- Use appropriate JPA annotations to map the entity to your database.

2. Data Persistence

- Use **Spring Data JPA** for data access.
- Configure an **H2 in-memory database** for persistence.

3. REST Endpoints

Implement a controller (e.g., `BookController`) that exposes the following endpoints:

- **Create a New Book**
  - **Endpoint:** `POST /books`
  - **Description:** Accept a JSON representation of a book (without an `id`, which should be auto-generated) and save it to the database.
- **Retrieve All Books**
  - **Endpoint:** `GET /books`

    - **Description:** Return a list of all books. *(Optional: add pagination/sorting)*
- **Retrieve a Single Book by ID**
    - **Endpoint:** `GET /books/{id}`
    - **Description:** Return the book with the given ID, or respond with 404 if not found.
- **Update an Existing Book**
    - **Endpoint:** `PUT /books/{id}`
    - **Description:** Update the details of the book with the specified ID. Validate input and return an error if the book does not exist.
- **Delete a Book**
    - **Endpoint:** `DELETE /books/{id}`
    - **Description:** Remove the book with the given ID from the database.
- **Search for Books**
    - **Endpoint:** `GET /books/search`
    - **Description:** Allow searching by title and/or author using query parameters.
    - **Examples:**
        - `GET /books/search?title=Spring`
        - `GET /books/search?author=Smith`
        - `GET /books/search?title=Spring&author=Smith`
- **AI-Powered Book Insights**
    - **Endpoint:** `GET /books/{id}/ai-insights`
    - **Description:**
        - Retrieve the specified book using its ID.
        - Build a prompt using the book's `description` (and optionally its title and author).
        - Integrate with an external AI service (e.g., OpenAI) by making an HTTP call to generate a short, engaging tagline or summary.
        - Return the AI-generated insights along with the book's details.
    - **Notes:**
        - Externalize API keys and endpoints in your configuration (e.g., using `application.properties` or `application.yml`).
        - Use Spring's `RestTemplate` or `WebClient` for the HTTP call.
        - Gracefully handle errors (e.g., API timeouts or failures) with appropriate HTTP statuses and error messages.

4. Input Validation & Exception Handling

- **Validation:**
  Use **JSR-303 Bean Validation** annotations (e.g., `@NotNull`, `@Size`) on the `Book` entity. Ensure that invalid inputs return meaningful error messages (HTTP 400).

- **Global Exception Handling:**
  Implement a global exception handler with `@ControllerAdvice` to manage:
    - Resource not found errors (404).
    - Validation errors.
    - Errors from the AI API integration.

5. Testing

- Write **integration tests** for your API endpoints using Spring Boot's testing tools (e.g., **MockMvc** or **TestRestTemplate**).
- Include tests for both successful operations and error cases (e.g., non-existent book, AI API failure).
- Consider mocking the external AI API calls using tools like WireMock or Spring's `@MockBean`.

6. Documentation (Optional)

- Integrate **Swagger/OpenAPI** to generate interactive API documentation.
- Provide clear instructions in a `README.md` file on how to build and run your application, including any configuration needed for the AI API integration.

Part 2: Next.js Frontend

1. UI Requirements

Develop a simple Next.js application with the following pages/features:

- **Home / Books List Page:**
    - Display a list of all books retrieved from the backend API (`GET /books`).
    - Each book item should display key details (e.g., title, author, publication year).
    - Allow users to click on a book to view more details.
- **Book Details Page:**
    - Show detailed information about a selected book by calling `GET /books/{id}`.
    - Provide a button or link to request AI insights for the book.
- **AI Insights Feature:**
    - When the user clicks to get AI insights, call the `GET /books/{id}/ai-insights` endpoint.
    - Display the AI-generated tagline or summary alongside the book details.
    - Handle loading states and potential errors from the API call gracefully.
- **Search Functionality (Optional):**
    - Add a search input on the Home page to allow users to filter books by title or author using the `GET /books/search` endpoint.

2. API Integration

- Use Next.js's data fetching methods (e.g., `getServerSideProps`, `getStaticProps`, or client-side fetching with `useEffect`) to communicate with the Spring Boot API.
- Ensure your Next.js application can correctly handle API responses, including error states.

3. UI/UX Considerations

- Keep the UI simple and focused on functionality.
- Use any styling solution you prefer (e.g., CSS modules, styled-components, Tailwind CSS).
- Ensure the application is responsive and provides clear user feedback (e.g., loading spinners, error messages).

General Evaluation Criteria

- **Correctness:**
  The application should meet all the functional requirements for both the backend and frontend components.
- **Code Quality:**
  Your code should be well-organized, readable, and maintainable. Follow best practices in both Java and JavaScript/React.
- **Integration:**
  Effective and robust integration with an external AI service from the backend, along with seamless communication between the frontend and backend.
- **Validation & Exception Handling:**
  Robust handling of errors, both from user input and external API calls.
- **Testing:**
  Adequate test coverage for your backend endpoints (including AI integration) and any key frontend functionality.
- **Documentation:**
  Provide a clear `README.md` with instructions on how to set up, run, and test both the Spring Boot and Next.js applications. If Swagger/OpenAPI is integrated, include directions on how to access the API documentation.

Deliverables

- **Git Repository:**
  A public repository link containing your Spring Boot backend and Next.js frontend projects. Make sure these are deployed separately on 2 Github links.
- **README:**
  A comprehensive `README.md` with:
  - Build and run instructions for both parts.
  - Configuration details for the AI API integration.
  - An overview of your design decisions.
  - (Optional) Details on accessing API documentation if Swagger/OpenAPI is used.

Time Allocation

Aim to complete this challenge within **4–6 hours**. Focus on building a well-structured, maintainable full-stack application with robust error handling and clear documentation.

Good luck, and happy coding!