Project Report
RESTful Bookstore API

Introduction

The RESTful Bookstore API is a backend application developed to manage books and authors in a bookstore. It provides endpoints for creating, retrieving, updating, and deleting records. The project follows RESTful principles and includes advanced features like filtering, pagination, and sorting for efficient handling of large datasets. Swagger UI is integrated to document and test the APIs.

Abstract

This project demonstrates the design and implementation of a Spring Boot–based RESTful API for bookstore management. The system supports CRUD operations with request/response handling through dedicated POJO classes. These POJOs are mapped to entity classes using ModelMapper, ensuring clean separation between persistence and API layers. An H2 in-memory database is used for development and testing. Robust exception handling has been added through custom exceptions, enums for error codes, and centralized error handling with @ControllerAdvice, ensuring consistent and meaningful error responses.

Tools Used

- Java 17 – Programming language.
- Spring Boot 3.5.5 – Framework for building the REST API.
- Spring Data JPA & Hibernate – ORM for database operations.
- H2 Database – Lightweight in-memory database for development/testing.
- Springdoc OpenAPI – API documentation with Swagger UI.
- ModelMapper – Mapping between POJOs and entities.
- Lombok – Reduces boilerplate code.
- Eclipse STS – IDE for development.
- Maven – Build and dependency management.

Steps Involved in Building the Project

1. Project Setup: Initialized a Maven-based Spring Boot project in Eclipse STS.
2. Entity Design: Defined BookEntity and AuthorEntity for persistence.
3. POJO Layer: Created request and response POJOs in a dedicated .pojo package to structure API input and output.
4. Model Mapping: Used ModelMapper to convert between POJOs and entities automatically.
5. Repository Layer: Implemented repositories using Spring Data JPA for database operations.
6. Service Layer: Wrote business logic for managing books and authors.

7. Controller Layer: Exposed REST endpoints (/books, /authors) for CRUD operations.
8. Validation: Applied input validation using annotations from spring-boot-starter-validation.
9. Filtering, Pagination & Sorting: Leveraged Spring Data JPA features for dynamic queries and efficient data retrieval.
10. Database Integration: Used H2 in-memory database for persistence during development.
11. Exception Handling: Implemented centralized exception handling using Controller Advice with custom exceptions such as ConflictException, ResourceNotFoundException, and handling DataIntegrityViolationException.
12. Error Codes & Enums: Created ErrorCodeEnum with standardized error codes (e.g., 1000 Invalid Request, 1001 Resource Not Found, 1002 Data Integrity Violation, 2000 Internal Server Errorr) to provide consistent error responses.
13. API Documentation: Integrated Springdoc OpenAPI to auto-generate API docs at /v3/api-docs and interactive Swagger UI at /swagger-ui/index.html.
Swagger UI : http://localhost:8080/swagger-ui/index.html#/
API docs: http://localhost:8080/v3/api-docs
14. Testing: Verified repositories, services, and controllers with spring-boot-starter-test.

📂 Project Structure Overview

- `config` → For configuration classes (e.g., Swagger config, CORS config).
- `constants` → Centralized constants (error codes, messages).
- `controller` → REST controllers exposing endpoints (`/books` , `/authors`).
- `entity` → JPA entities mapped to DB tables.
- `exception` → Custom exceptions + global exception handler.
- `pojo` → Request/Response POJOs mapped via ModelMapper to entities.
- `repo` → Spring Data JPA repositories (`AuthorRepository` , `BookRepository`).
- `service.interfaces` → Service layer interfaces defining business contracts.
- `service.impl` → Implementations of service interfaces containing business logic.

Conclusion

The RESTful Bookstore API demonstrates how to build a robust backend system with Spring Boot. By separating request/response handling into POJO classes and mapping them to entities, the project ensures clean architecture and maintainability. Features like filtering, pagination, and sorting make the API production-ready for real-world use cases. With the addition of centralized exception handling, error code enums, and meaningful error responses, the API is resilient and client-friendly. With Swagger integration, the API is easy to test and document, forming a strong foundation for further enhancements such as external database support, authentication, and deployment.