Technical University of Applied Sciences
Würzburg-Schweinfurt

Backend Systems Portfolio Assignment 04

Library Management System

Kristian Popov and Enrico Ebert
Instructor: Prof. Dr. Peter Braun

February 12, 2025

Submission Deadline: Saturday, February 15th, 2025, 6 pm CET
Submitted by: Kristian Popov and Enrico Ebert

This document describes our Library Management System project, detailing our implementation of distributed systems using a hexagonal architecture and REST APIs.

1 Introduction

This document describes our **Library Management System**, focusing on its design and implementation with a *hexagonal architecture* (ports and adapters). The system streamlines essential library operations—book and author management, user registration, and borrowing transactions—and exposes functionality via a **REST** interface.

1.1 Use Cases

- **Book Management**: Create, retrieve, update, or delete books. Librarians can search by ISBN, title, author, or ID.
- **Author Management**: Create and modify author records (names, IDs) used to enrich book information.
- **Customer Management**: Library users can register as customers, update personal info, or be removed if necessary.
- **Transaction Management**: Borrow and return books, keeping a historical record of all previous transactions for each customer.

2 Software Architecture

We apply the **hexagonal architecture** to separate our core domain logic from external concerns (inbound requests, database, configuration, etc.). This approach emphasizes:

- **Ports**: Interfaces that the domain uses to communicate with the outside world.
- **Adapters**: Concrete implementations of these ports (e.g., REST controllers, database connectors).
- **Domain**: Encapsulates entities, business rules, and core services.

2.1 Overall Package Structure

Our main packages are arranged under `app` as follows:

- `app.adapters` **in**: Inbound adapters such as REST controllers or any other entry points that handle incoming requests. **out.H2**: Outbound adapters specific to H2 persistence.
- `app.domain` **models**: The core entities (`Book`, `Author`, `Customer`, `Transaction`) capturing fundamental data and domain rules. **port**: Interfaces (`BookDao`, `AuthorDao`, `CustomerDao`, `TransactionDao`) that define how the domain layer interacts with external systems (database). **services**: Business logic classes (`BookService`, `AuthorService`, etc.) that implement domain-specific rules (e.g., checking book availability).
- `app.infrastructure` **config**: System-level or cross-cutting configurations (e.g., security, caching, or database seeding). **exceptions**: Custom exception classes or global exception handlers.

This arrangement enforces a **clear boundary** between the domain (core logic) and the adapters (the means by which data enters or leaves the core).

2.2 Domain Layer: Models, Ports, and Services

- **Models** (`app.domain.models`): Each entity (e.g., `Book`, `Author`, `Customer`, `Transaction`) represents core concepts. They typically include fields (ID, title, names, etc.) and may hold simple validation logic (e.g., "page count must be non-negative").
- **Services** (`app.domain.services`): Examples: `BookService`, `TransactionService`. They contain business rules like "ensure a book is available before loaning it out." These services do *not* contain infrastructure-specific code; instead, they rely on the port interfaces to interact with, for instance, a database.
- **Ports** (`app.domain.port`): Interfaces such as `BookDao`, `AuthorDao`, `CustomerDao`, and `TransactionDao` define the operations required by the domain layer (e.g., "save a book," "find a transaction by ID"). They are unaware of the actual persistence mechanism (H2, H2, etc.).

2.3 Adapters Layer

2.3.1 Inbound Adapters: `app.adapters.in`

Inbound adapters are typically the user-facing entry points:

- **Controllers**: Classes responsible for handling HTTP endpoints (e.g., `BookController`, `AuthorController`). They map incoming HTTP requests to domain service calls, convert domain objects to DTOs if necessary, and return appropriate responses.
- **Other Inbound Interfaces**: If the system eventually supports messaging or scheduled tasks, those would also reside here as inbound adapters.

2.3.2 Outbound Adapters: `app.adapters.out.H2`

Outbound adapters take the port interfaces from `domain.port` and implement them with a specific technology—in this case, H2:

- **DAO Implementations**: `AuthorDaoAdapter` provides an implementation for the `AuthorDao` interface. In doing so, domain objects (e.g., `Author`) are converted into entities (e.g., `AuthorEntity`) and persisted in the H2 database using Spring Data.

2.4 Challenges

Implementing the separation of concerns in hexagonal architecture was initially challenging, particularly in distinguishing domain logic from infrastructure concerns. However, understanding the pattern improved our design thinking.

3 API Technology

3.1 Why REST?

We chose REST (Representational State Transfer) for our Library Management System because it is widely adopted, easy to implement, and follows a stateless, client-server model. Each request carries all necessary information, facilitating efficient scaling. Standard HTTP methods (GET, POST, PUT, DELETE) and response codes ensure clear, human-readable communication.

3.2 Advantages

- **Ease of Integration**: Comprehensive support across platforms simplifies connecting with third-party services.
- **Flexibility**: REST allows seamless interaction with various client types (web, mobile), enabling straightforward feature expansion.
- **Caching**: Built-in HTTP caching reduces redundant requests and enhances performance.

3.3 Limitations

- **Overhead**: For high-frequency or large data transfers, REST may introduce more overhead than protocols like gRPC.
- **Versioning**: Maintaining backward compatibility while evolving the API can be complex.
- **No Native Streaming**: REST does not inherently support real-time streaming, limiting continuous data exchange.

4 Implementation Details

4.1 Spring Boot as the Framework

We use **Spring Boot** for its rich ecosystem (Spring Data, Security, Actuator) and convention-over-configuration approach. This simplifies:

- **Dependency Management**: Gradle or Maven handles the required libraries with minimal setup.
- **Auto-Configuration**: Reduces boilerplate code, letting us focus on domain logic.
- **Integration**: Provides straightforward ways to integrate with databases, caches, or security modules.

4.2 Authentication

Authentication was implemented using a token-based mechanism with **JSON Web Tokens (JWT)**. When a user logs in, the system validates their credentials and generates a JWT, which is then used for all subsequent requests. This stateless approach aligns with REST principles, ensuring scalability and security by eliminating the need for server-side session management.
The authentication configuration is defined using Spring Security. Key components include:

- **Authentication Manager**: Configured to validate user credentials against a custom `UserDetailsService`.
- **Token Filter**: A custom filter intercepts incoming requests to validate the JWT, ensuring only authenticated users can access secured endpoints.
- **Password Encryption**: Passwords are securely hashed using `BCryptPasswordEncoder` to protect user credentials.

4.3 Database Integration (H2)

In `app.adapters.out.H2`, we define H2-specific repositories or DAOs that implement the port interfaces. These classes might leverage:

- **Spring Data JPA** or pure JDBC for database queries.
- **Entity Mappings**: If using JPA, domain models are annotated (e.g., `@Entity`). Alternatively, we might map them separately, returning domain objects from the adapter layer.

5 Testing Strategy

5.1 Unit Tests

- **Objective**: Verify individual methods or classes in isolation, focusing on domain logic (for instance, ensuring `AuthorService` or `TransactionService` correctly enforce business rules).
- **Implementation**: We rely on **JUnit** and **Mockito** to create focused tests that mock out dependencies, such as repositories. This guarantees that each test targets one piece of functionality without requiring a running web server or real database access.

5.2 Integration Tests

- **Objective**: Ensure controllers, services, and the database layer work together as expected. This involves simulating real HTTP requests and checking both responses (status codes, JSON payloads) and the final database state.
- **Implementation**:

- We use **Spring Boot Test** annotations like `@SpringBootTest` and `@AutoConfigureMockMvc` to stand up the entire application context and inject a `MockMvc` instance.
- `WithMockUser` allows testing secured endpoints by providing a mock user context.
- For JSON handling, we rely on `com.fasterxml.jackson.databind.ObjectMapper` to serialize and deserialize request/response objects (e.g., `CreateNewAuthor`).

5.3 Benefits of Combined Testing

A layered testing approach (unit + integration) helps detect **both** logic-level issues (like incorrect validations or missing fields) and system-level issues (e.g., misconfigured endpoints, erroneous DB queries). This provides high confidence that the application behaves correctly under real usage scenarios.

6 Learning Outcomes and Reflection

6.1 Key Learnings

- **Hexagonal Architecture**: Clear boundaries between domain logic and infrastructure adaptors simplified changes and improved testability.
- **REST Best Practices**: Emphasized proper use of HTTP methods, status codes, and JSON for request/response.
- **Spring Boot Ecosystem**: Leveraged built-in features for rapid setup, easy integration, and less boilerplate code.
- **Database Abstraction**: By using port interfaces, we kept the domain decoupled from H2-specific details.

6.2 Reflections

- **What Worked Well**:
  - The use of Hexagonal Architecture ensured a clean and logical separation of concerns, making it easier to modify or extend individual components.
  - Spring Boot streamlined backend development, allowing us to focus on core functionality without being overwhelmed by boilerplate code.
  - Collaboration within the team was highly effective, with clear communication and task distribution enabling smooth progress.
- **What Could Be Improved**:
  - While our testing strategy was robust, incorporating automated performance and load testing would provide additional insights into system reliability under high-demand scenarios.
  - A deeper exploration of asynchronous programming and event-driven architectures could improve the system's responsiveness and scalability.

6.3 Team Collaboration and Hexagonal Architecture

Team collaboration was a key strength of this project. Regular standups, clear role assignments, and effective use of version control systems like Git ensured that all team members stayed aligned and productive. The adoption of Hexagonal Architecture proved to be a game-changer, providing a clear and consistent structure that enabled each team member to focus on specific layers without causing conflicts or confusion. This approach facilitated parallel development, improved code readability, and made testing more manageable.

Acknowledgments

This article was drafted and refined using GPT-4 based on an outline containing related information. The GPT-4 output was reviewed, revised, and enhanced with additional content. It was then edited for improved readability and active tense, partially using Grammarly.