Backend Systems - Portfolio Assignment 04: Library System Design Document

Bauer, Moritz; Elsner, Merle; Wetzel, Alexander

Contents

1 Introduction

This document provides an outline of our distributed library system's design, creation, and implementation as a semester-long academic project. The hexagonal architecture paradigm, which separates the main application logic from supporting infrastructure, is the method we used to build our system. This makes the software ecosystem more flexible and maintainable. GraphQL is used as the API communication protocol through this architecture to guarantee smooth, effective, and adaptable client-server interactions. This library system is built to manage key entities such as books, authors, and users, each playing a significant role in day-to-day library operations.

This system's functionality is based on the use cases it supports. Users can borrow books that are available in the library's catalog. In order to maintain accuracy and operational flow, this procedure involves updating the book's availability status and making sure the catalog represents accurate data. Users have the option to make reservations in case a book is not available.

Additionally, the system allows users to return books they have borrowed, therefore changing the book's status to available to make it available for other customers. The system is also improved by the tools it offers for managing user profiles, which allows users to create, edit, and remove personal data.

The system uses search capabilities to improve discoverability and increase user interaction. Usability is enhanced through the ability for users to browse the library's collection and filter books according to different criteria such as author, genre, or availability.

2 Project Overview - Hexagonal Architecture in the Library Management System

Our library management system is built on a hexagonal architecture, which aims to produce a software solution that is modular, manageable, and scalable. This method improves flexibility while simplifying maintenance by separating the fundamental domain logic from external concerns like databases and interfaces.

2.1 Domain, Business Logic, and Infrastructure Components

Domain: This component contains the library's primary data structures and business logic, including entities such as `Book`, `Author`, and `User`. These components outline fundamental library concepts, concentrating only on business rules while remaining disregardant of technical implementations.

Business Logic: This layer manages application-specific workflows, encapsulated

within services like `BorrowService`, `ReservationService`, and `ReturnService`. These services coordinate tasks related to borrowing, reserving, and returning books.

Infrastructure: This includes technological elements that use the `GraphQLServer` and repository implementations to process API requests, enable database operations, and interact with the outside world.

2.2 Ports and Services in the Library Management System

In our library management system, ports and services are crucial for maintaining a separation between the core application logic and external systems. They support seamless communication and integration while keeping core logic stable despite external changes.

Ports:

- *Inbound Ports:* These interface with external actors, handling actions like borrowing books or checking availability. Service interfaces act as inbound ports, initiating domain operations based on these interactions.
- *Outbound Ports:* Used by the domain to manage external data storage or retrieval, such as through the `BookRepository` interface, which abstracts database operations.

Services as Adapters:

Services implement these ports, functioning as adapters that bridge external requests with domain processes.

- *Inbound Adapters:* Services like `BorrowService` convert client inquiries into domain-specific actions, ensuring execution aligns with business rules.
- *Outbound Adapters:* These services manage data exchanges with repositories, as in `BorrowService` updating book status by interacting with `BookRepository`.

By using ports and services as adapters, our library management system ensures flexibility and adaptability. External changes, like database updates or UI enhancements, occur without altering core logic, supporting long-term resilience and scalability.

The library management system is kept flexible by this division into separate parts. Hexagonal design is appropriate for projects that prioritize long-term maintenance and scalability because it allows database or user interface updates without altering the essential business logic.

Effectively managing the dependencies to maintain our domain logic's true decoupling from external systems was one of the challenges we encountered when putting the hexagonal architecture into practice. Additionally, in order to avoid performance bottlenecks and guarantee the success of the architecture, it was crucial that our team coordinate their comprehension and application of these concepts to preserve system cohesiveness.

3 Explanation of the API Technology

We decided to use GraphQL as the API technology when designing our library management system. This choice was made in consideration of several factors crucial for our system's requirements that support the dynamic data interaction needs of our system.

Because it provides a very adaptable and effective framework for handling complex queries—a critical need for our library system—we chose GraphQL. It offers a single endpoint that can handle a variety of queries, in contrast to REST, which might result in over-fetching or under-fetching by requiring several endpoints or delivering unnecessary data. This is especially useful in a library setting because users may use a single, efficient query to request specified information such as book authors, author biographies, or user reservations. Efficient resource management is achieved by lowering client and server overhead through the ability to request accurate data.

Our library system benefits significantly from GraphQL's ability to retrieve data precisely and efficiently. Whether users want a list of books that are available, information about a certain author, or both, users can be quite specific about what they are looking for. In order to ensure a seamless user experience in our system, this specialized nature improves performance by limiting data transfer, lowering bandwidth usage, and accelerating response times. Additionally, because developers can quickly and confidently understand and adjust API interactions, GraphQL's robust type and schema support fastened development and easy adaptation to changing library needs.

GraphQL is our choice for our library management system because of its significant benefits, despite some challenges. It allows clients to precisely request data, optimizing network performance by reducing data transfer and bandwidth use. This results in faster response times and a better user experience. GraphQL supports complex queries through a single endpoint, enabling dynamic interactions without multiple API calls. Overall, GraphQL's adaptability and efficiency make it well-suited for scaling our library system to meet user demands.

4 API Technology and Implementation Details

In our library management system project, we chose GraphQL as the API technology due to its modern and effective approach to managing client-server data interactions. GraphQL stands out against traditional RESTful architectures by addressing common issues such as over-fetching and under-fetching. It allows clients to specify exactly the data they need, thereby optimizing network performance, reducing unnecessary data transfer, and minimizing latency.

A critical implementation choice involved utilizing GraphQL's schema and resolvers to define the API structure. The schemas were crafted to align with the domain entities, such as `Book` and `Author`, facilitating precise access to library data. Resolvers handle the core logic for data retrieval and manipulation, bridging API requests with backend

services.

We implemented our solution using a servlet to serve as the central point for handling incoming GraphQL requests. The use of a servlet was driven by its lightweight nature and ability to integrate seamlessly with the Java ecosystem, which is well-suited for building scalable web applications.

Choosing GraphQL was driven by its adaptability and efficiency in processing varied data queries through a single endpoint. Unlike REST, which often requires multiple endpoints to deliver data, GraphQL provides a unified framework that simplifies client-server interactions, making it suited for the diverse data access requirements of a library management system.

Our technical setup involved defining GraphQL controllers as inbound services that implemented the business logic while interacting directly with the domain services. GraphQL's strong typing ensures that both clients and the server work together predictably, reducing runtime errors due to data type mismatches. The introspective schema offers comprehensive documentation, allowing developers to quickly understand and adapt the API, preserving application stability through seamless evolution.

Efficient network usage is another major advantage of GraphQL. By enabling multiple related resources to be queried in a single transaction, GraphQL minimizes the network round trips needed between the client and server. This capability significantly reduces response times and simplifies client-side code, making it ideal for the complex data structures and varied requests typical of a library management system.

Overall, this implementation leverages GraphQL and servlet technology to ensure our library management system remains efficient, scalable, and future-ready. These technologies, paired with our design considerations, create a robust foundation that provides data interactions.

5 Testing Strategy

On one hand, our testing technique focuses on unit testing, which looks at the fundamental logic of the component and service levels. We were able to achieve thorough coverage of the application's business logic by using JUnit tests. The tests were planned to encompass the entire range of behaviors anticipated in different scenarios, such as standard use cases, edge cases, and error-handling processes. Even in the face of unusual conditions and unexpected input data, this focus made sure that every service operated as planned.

On the other hand, to validate the interactions between various system components, integration testing was conducted with a focus on GraphQL endpoints and their related database communication. By confirming that data was appropriately fetched, processed, and stored across all system components, these tests were essential in confirming the accuracy of complete request-response cycles. Integration tests replicated complex interaction scenarios and real-world usage patterns.

As a result, we were able to find and fix problems early in the development process. In addition, because of this thorough testing approach, the possibility of serious failures after deployment was reduced. This systematic method guided the library system's development and preserved the system's integrity and coherence, further offering helpful details about possible future areas of improvement.

6 Learning Outcomes and Reflection

While developing our library management system, we gathered several important insights, particularly regarding the use of hexagonal architecture. This approach was crucial in helping us maintain separation between our core business logic and additional concerns, thereby enhancing both modularity and testability. We also learned a great deal about data management and deployment, utilizing tools like Docker and a servlet for consistent deployment across various environments.

Reflecting on the project, we experienced numerous successes but also identified areas for improvement. Implementing GraphQL as a framework in our application worked exceptionally well, enabling efficient data handling and interaction processes. However, the absence of an authentication mechanism became apparent as a security vulnerability.

One challenge we encountered was agreeing on a unified logic and accurately implementing a hexagonal structure. This required careful consideration to ensure that dependencies were clearly structured and that modularity was maintained throughout the system. The complexity of this task highlighted the need for clear communication and alignment among team members to successfully achieve our architectural goals.

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.