Bookshop Management System
Analysis and Design Document : Assignment 1

Student: Grad Laurentiu-Calin
Group: 30435

Table of Contents

Contents

1. Requirements Analysis

1.1 Assignment Specification

The aim of this project is to develop my aptitudes in the field of software engineering, especially in the areas of system design and project management, and to become familiar with widely used programming paradigms, programming languages and frameworks. For this purpose, I chose to implement a well-known project: a bookstore management system. Even though the theme is rather simple, the focus of the project is to understand the capabilities of the technologies I will work with.

1.2 Functional Requirements

There are three main "things" users should be able to interact with: authors, books and publishers. These are the foundations of the functional requirements:

1. Allow users to read, update and delete old entries or create new entries for **publishers** by specifying: the name, the location of their headquarters and the year of foundation.
2. Allow users to read, update and delete old entries or create new entries for **authors** by specifying: the first and last name, an alias (optional) and the nationality.
3. Allow users to read, update and delete old entries or create new entries for **books** by specifying: the ISBN, the title, the authors, the publisher, the year of publication, the price and the available stock.
4. Allow admins to read, update and delete existing users of the system.
5. Allow users to register in the system by specifying: a username, a password, the first and last name and their role.
6. Allow users to login in the system.
7. Allow users to add books to their virtual cart and place orders.
8. Allow users to sort items by category.

1.3 Non-functional Requirements

Used technologies:

- Database management: PostgreSQL
- Backend development: Java & SpringBoot
- Frontend development: React & Vite

Some of the non-functional requirements:

1. Implement validation to ensure all inputs adhere to a specified format (e.g. ISBN is a 10/13-digit code) and thus keep the store information consistent.
2. Impose only highly-complexity passwords (e.g. containing at least a capital, a digit and some special symbols).
3. Use an ORM to handle database interaction.
4. Use a DI container for dependency injection.
5. Implement layered architecture for better organization and separation of concerns.
6. Use a SQL database to store required information.

2. Use-Case Model

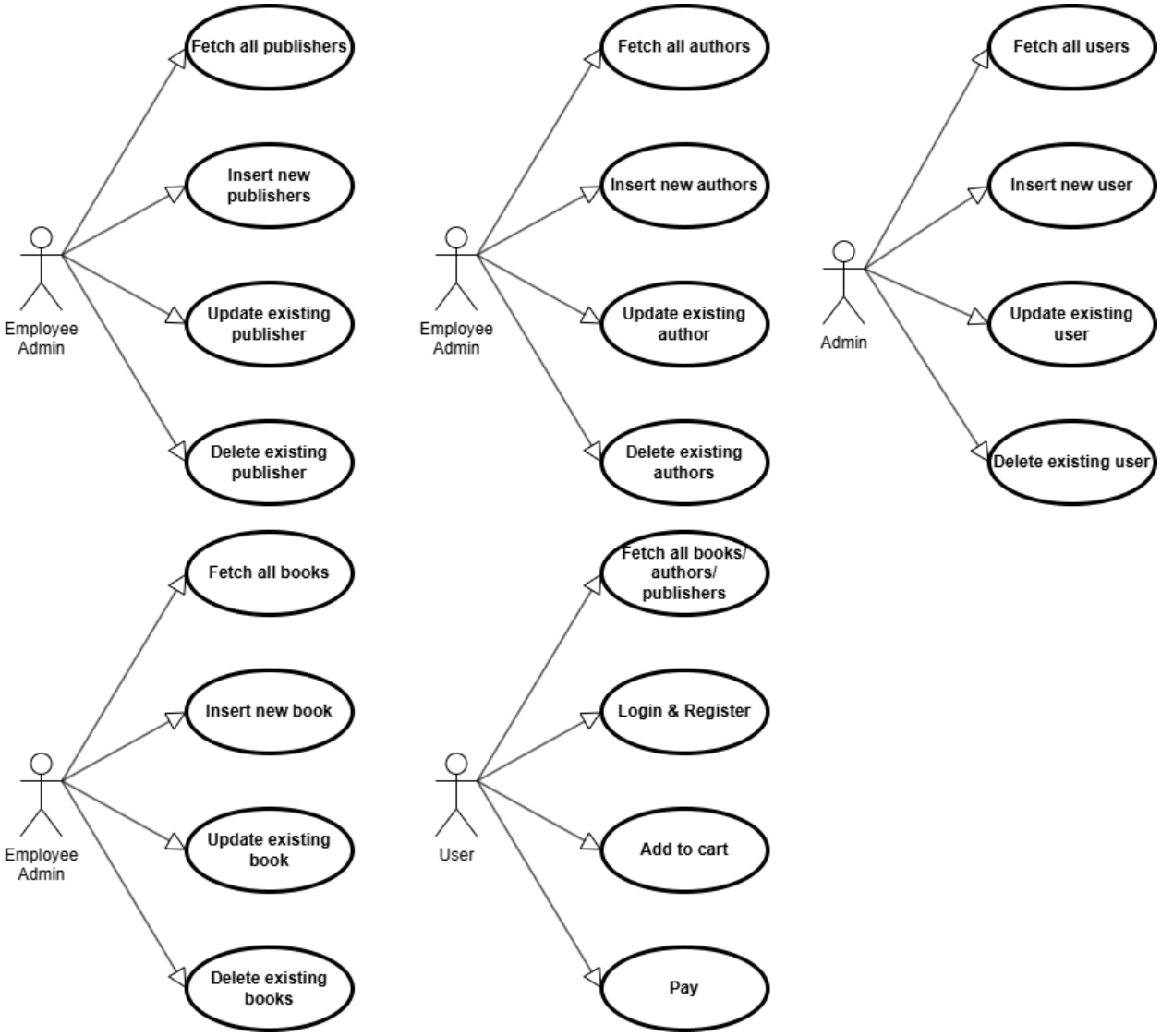


Figure 1. The uses cases of the bookstore management system.

2.1 Description of the User role:

Note: Any employee or administrator is a user.

Use case	Description	Main success scenario	Extensions
Register	The user fills in a form with some personal details and is registered by our system, from now on it can log into his personal account.	The database records a new user of the system.	Fields that do not comply with the specified standard generate errors that are displayed in the GUI.

Login	A pre-registered user accesses his account.	Successful identification based on the provided credentials.	Unsuccessful/Successful logins are signaled in the GUI.
Fetch all entities*	Fetching all the entries in the database that correspond to an entity and displaying them in a table.	Backend and database connection succeed, and the entities are displayed in an intuitive manner.	Any errors that happen during the fetching are reported in the GUI, as well as information messages ("No entities* are present!")
Add to cart	*TODO*		
Pay	*TODO*		

(*****entity = book | author | publisher)

2.2 Description of the Employee role:

Note: Administrators are employees.

Use case	Description	Main success scenario	Extensions
Insert a new entity*	Register in the database a new entity instance.	The database records a new instance of the entity type.	Any errors that appear at the database level (unique constraint violation error) or at the logic layer (validation errors) are reported in the GUI.
Update an existing entity*	Update an existing instance from the database.	The database updates the existing instance of the entity type.	Any errors that appear at the database level (unique constraint violation error) or at the logic layer (validation errors) are reported in the GUI.
Delete entities*	Delete one or more entity instances from the database.	The database drops the specified (existing) instances of the entity type.	Any errors that appear at the database level (unique constraint violation error) or at the logic layer (validation errors) are reported in the GUI.

(*****entity = book | author | publisher)

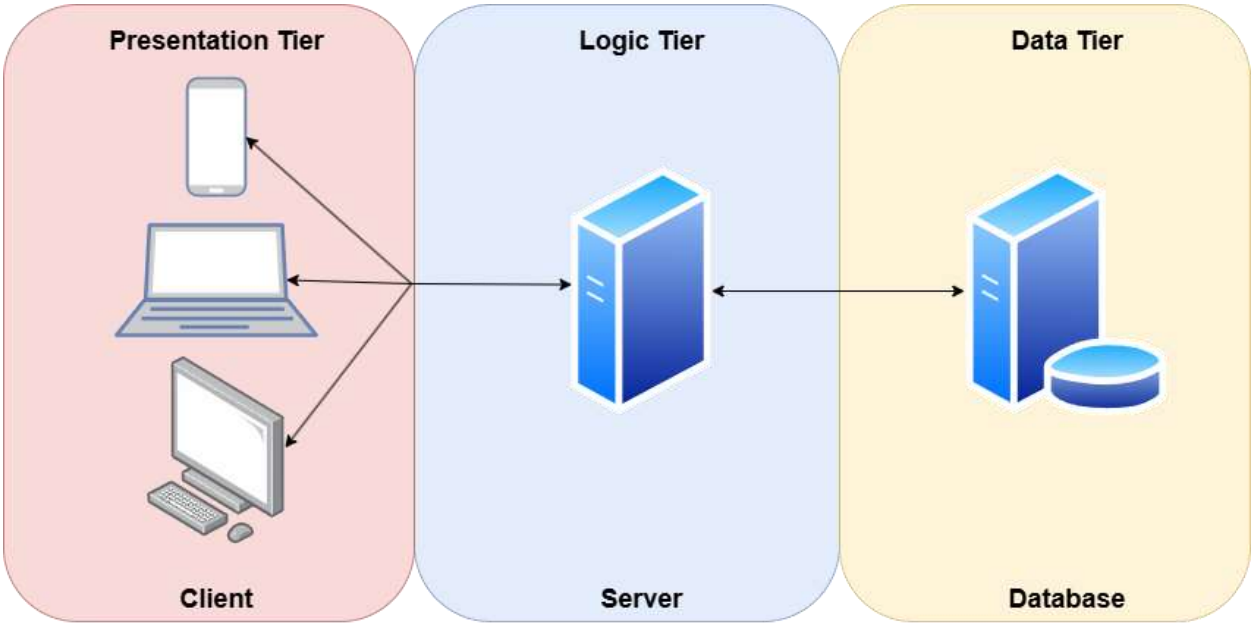
2.3 Description of the Admin role:

Use case	Description	Main success scenario	Extensions
Fetch all users	Fetching all the user entries in the database and displaying them in a table.	The user entries are displayed in a table	Any errors that happen during the fetching are reported in the GUI. There should always be at least one admin (user) of the system.
Insert a new user	Register in the database a new user instance.	The database records a new user with the specified credentials.	Any errors that appear at the database level (unique constraint violation error) or at the logic layer (validation errors) are reported in the GUI.
Update an existing user	Update an existing user from the database.	The database updates the existing user.	Any errors that appear at the database level (unique constraint violation error) or at the logic layer (validation errors) are reported in the GUI.
Delete users	Delete one or more users from the database.	The database drops the specified (existing) users.	Any errors that appear at the database level (unique constraint violation error) or at the logic layer (validation errors) are reported in the GUI.

3. System Architectural Design

3.1 Architectural Pattern Description

The system is designed based on the three-server architectural pattern: the data tier, the logic tier and the presentation tier are run concurrently. The theoretical advantages of this strategy are portability – tiers can be deployed on different platforms without major changes; scalability – each tier scales independently of the others; maintainability – updates in one tier do not affect the structure of the others; security – each layer can integrate different layers of security, thus restricting the access to sensitive data; reusability – usually, the business logic can be reused across several systems; flexibility – different technologies and frameworks can be used for each tier.



The client side is done in React and Vue. Its main purpose is to ensure good user experience. React is used for building the core structure of the system, especially because it supports dynamic components and state management. React's component-based structure allows you to create reusable UI elements, making it easier to manage updates and rendering. The key features of this layer:

1. **Component-based architecture** – The client-side application is built using React's component-based architecture.
2. **Routing and navigation** – React Router is utilized to handle client-side navigation. This allows users to navigate between different views or pages within the application without requiring a full page reload. This improves the overall performance and user experience by providing instant transitions.
3. **React hooks** – React Hooks are leveraged to manage lifecycle events and state in functional components, replacing traditional class-based components. Hooks such as useState, useEffect, and useContext simplify component logic, improve readability, and make the codebase more concise and maintainable.

4. **Integration with backend (SpringBoot)** – React communicates with the backend (Spring Boot) using Fetch API to make HTTP requests to RESTful endpoints. This allows the frontend to send and retrieve data, such as books, authors, and publishers, dynamically without needing to reload the entire page.
5. **User interaction and feedback** – Real-time interaction elements such as form validation, loading spinners, and error messages are integrated into the UI to ensure users are informed throughout their interactions. Immediate feedback (errors, informations or confirmations) is displayed textually in the UI.

The logic level of the Bookstore Management System is implemented using Spring Boot, a powerful and flexible framework that simplifies the development of Java-based backend applications. Spring Boot handles the core business logic, data processing, and interaction between the client-side (React/Vue) and the database. The logic tier is responsible for performing operations such as managing book inventories, author and publisher details, and user authentication. The key features of this layer are:

1. **RESTful APIs** – The backend exposes a set of RESTful APIs that the frontend communicates with to retrieve and manipulate data. Endpoints include CRUD operations for books, authors, publishers, and user accounts.
2. **Business logic implementation** – handling book transactions, validating ISBNs, calculating prices, and managing stock levels, as well as custom logic for managing relationships between authors, publishers, and books is implemented in the service layer.
3. **Database interaction** – Spring Data JPA is used for efficient data handling through the repository layer, where entities like Book, Author, Publisher and User are mapped to the database tables. Spring Boot's automatic configuration minimizes the need for boilerplate code.
4. **Error handling and validation** – Exception handling is implemented globally to provide meaningful error messages for various failure scenarios, ensuring a robust and user-friendly experience. Input validation, including ISBN format validation, is handled using custom annotations and Spring's validation framework to ensure data integrity.

The database level is implemented in PostgreSQL. It serves as the foundation for data storage and management, handling the persistence of entities such as books, authors, publishers, and users. The key features of this layer:

1. **Data modelling** – The database schema is designed to accurately reflect the relationships between entities in the system. Tables are created to represent Books, Authors, Publishers, Users, and other relevant entities.
2. **Relational database** – PostgreSQL supports a fully relational model, where tables are linked through foreign keys. For example, each book is linked to its author and publisher through foreign key constraints, ensuring referential integrity.
3. **Data integrity** - Data integrity is enforced using primary keys to uniquely identify records and foreign keys to enforce relationships between entities. Unique constraints are used to ensure that fields like ISBN numbers for books and usernames remain unique.

4. **Database integration using SpringBoot** – The database is seamlessly integrated with the Spring Boot application using Spring Data JPA, which provides an abstraction layer over raw SQL queries and simplifies database interaction. Hibernate is the JPA implementation used for ORM (Object-Relational Mapping), allowing entities in the Spring Boot application to map directly to PostgreSQL tables.

3.2 Diagrams

The system implementation follows the layered architecture principles. Logically, the internal structure is divided into three layers, each mapped one-to-one to the tiers defined in the previous section. This separation ensures maintainability, scalability, and clear responsibility demarcation.

The **presentation layer** is responsible for the user interface and experience. It handles user interactions, displaying information, and collecting input. It communicates with the domain layer to fetch and update data. While the data exchange between these layers is bidirectional, the dependency is unidirectional meaning the presentation layer depends on the domain layer but not vice versa. This separation allows flexibility in modifying the frontend without impacting the business logic.

The **domain layer** encapsulates the business logic and enforces application rules. It validates incoming data from the frontend to maintain database consistency. Additionally, this layer acts as an intermediary between the presentation and data layers, ensuring that user requests are processed through structured database queries. This layer also supports business operations, including data transformations, rule enforcement, and workflow execution.

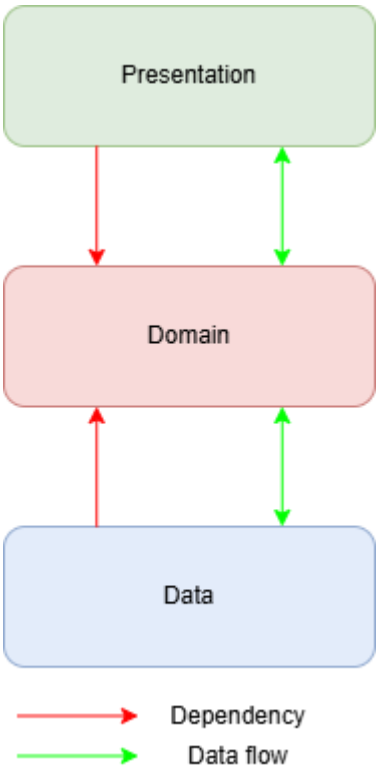


Figure 2. The layers of the system.

The **data layer** corresponds to the relational database, which stores and manages application data. It is accessed by the domain layer using SQL queries, ensuring secure and optimized data retrieval and modification. This layer abstracts the database management system (DBMS) details, allowing smooth adaptation to different database technologies without affecting the upper layers.

By following this layered architecture, the system achieves a modular and organized structure, making it easier to develop, maintain, and scale as requirements evolve.

4. Class Design

4.1 Package and class diagrams

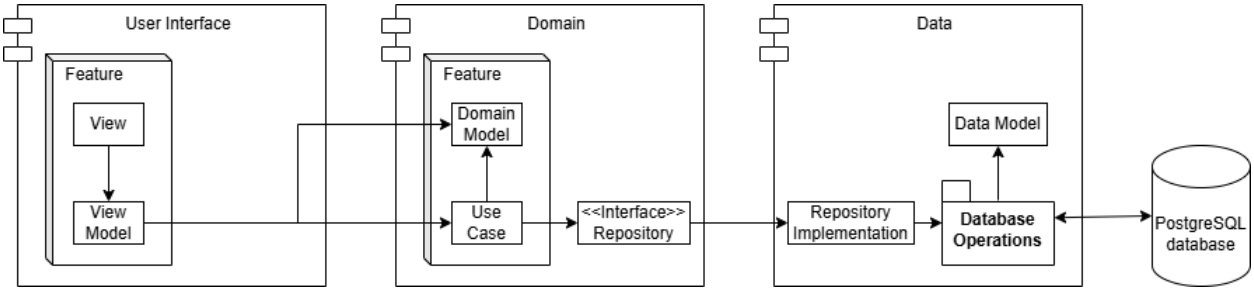


Figure 5. Package diagram of the system.

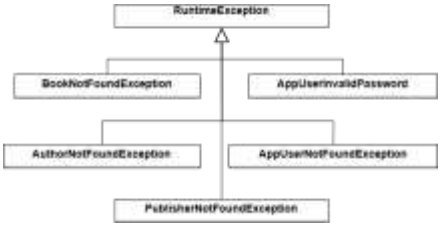


Figure 3. The custom exception hierarchy.

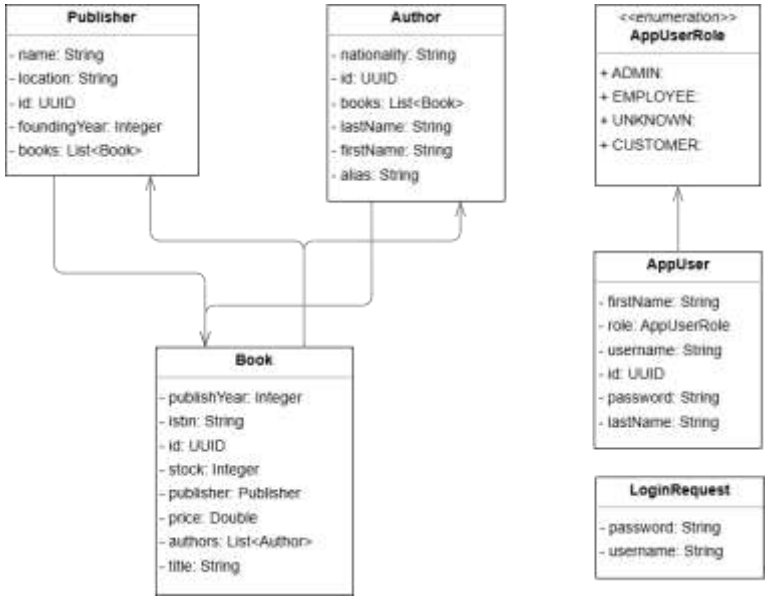


Figure 4. The model classes of the system.

Book, Author, Publisher and AppUser all have a corresponding "Create Data Transfer Object" that handles the validation by means of SpringBoot annotations. SpringBoot's Global Exception Handler class ensures that exception handling is done consistently

Figure 6. The service classes provide basic CRUD functionalities.

5. Data Model

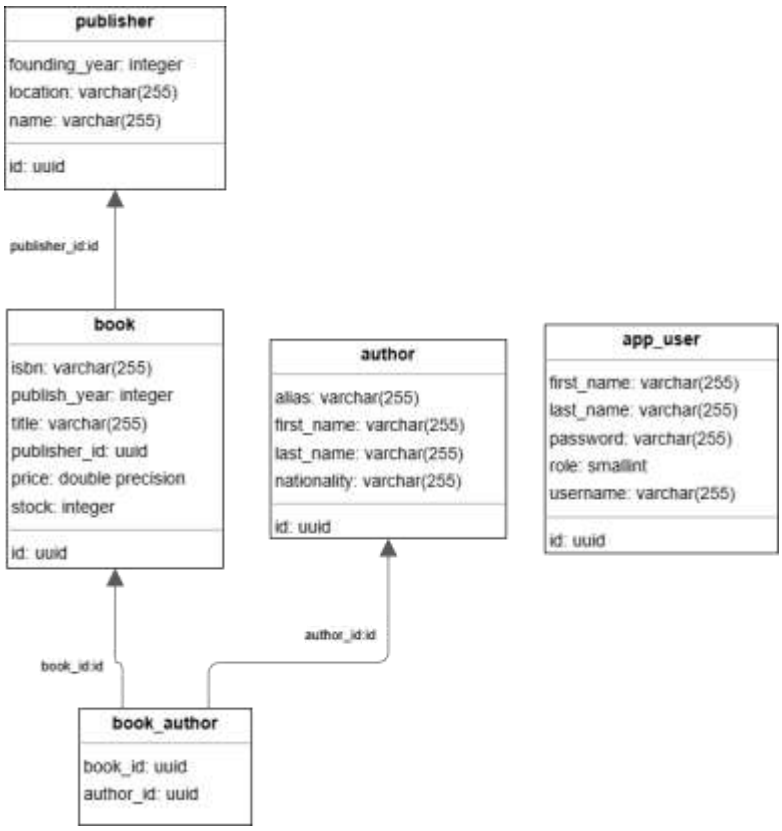


Figure 7. The database diagram.

There are three core entities: book, author and publisher that are linked based on the chosen relationships:

- A book can be published by multiple publishers, but a publisher can publish multiple books: 1:M relationship.
    - Solution: the *book* table is updated with a foreign key referencing the *publisher* table.
- A book can be written by multiple authors, and one author may write more books: M:M relationship.
    - Solution: create a junction table *book_author* that has two foreign keys: one referencing the book table and the other referencing the author table.

The *app_user* table stores information about the users of the system – admins, employees, customers etc.

6. Bibliography

1. https://www.npmjs.com/package/react-multi-select-component
2. https://www.npmjs.com/package/react-data-table-component/v/7.0.0-rc2
3. https://medium.com/trendyol-tech/how-to-write-a-spring-boot-library-project-7064e831b63b
4. https://react.dev/learn/thinking-in-react
5. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
6. https://www.dhiwise.com/post/exploring-react-datatable-everything-you-need-to-know
7. https://app.diagrams.net/