Bookshop Management System
Analysis and Design Document : Project

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 login or register in the system by specifying: a username, a password, the first and last name and their role.
6. Allow users to sort items by category.
7. Limit application functionality based on role.
8. Added JWT authentication.
9. Implemented 2-step password recovery functionality over e-mail.
10. Dockerized the application.
11. Users can purchase items and receive e-mail confirmation; users can see their order history.

1.3 Non-functional Requirements

Used technologies:

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

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 high-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 injections.
5. Implement layered architecture for better organization and separation of concerns.
6. Use a SQL database to store required information.

2. Use-Case Model

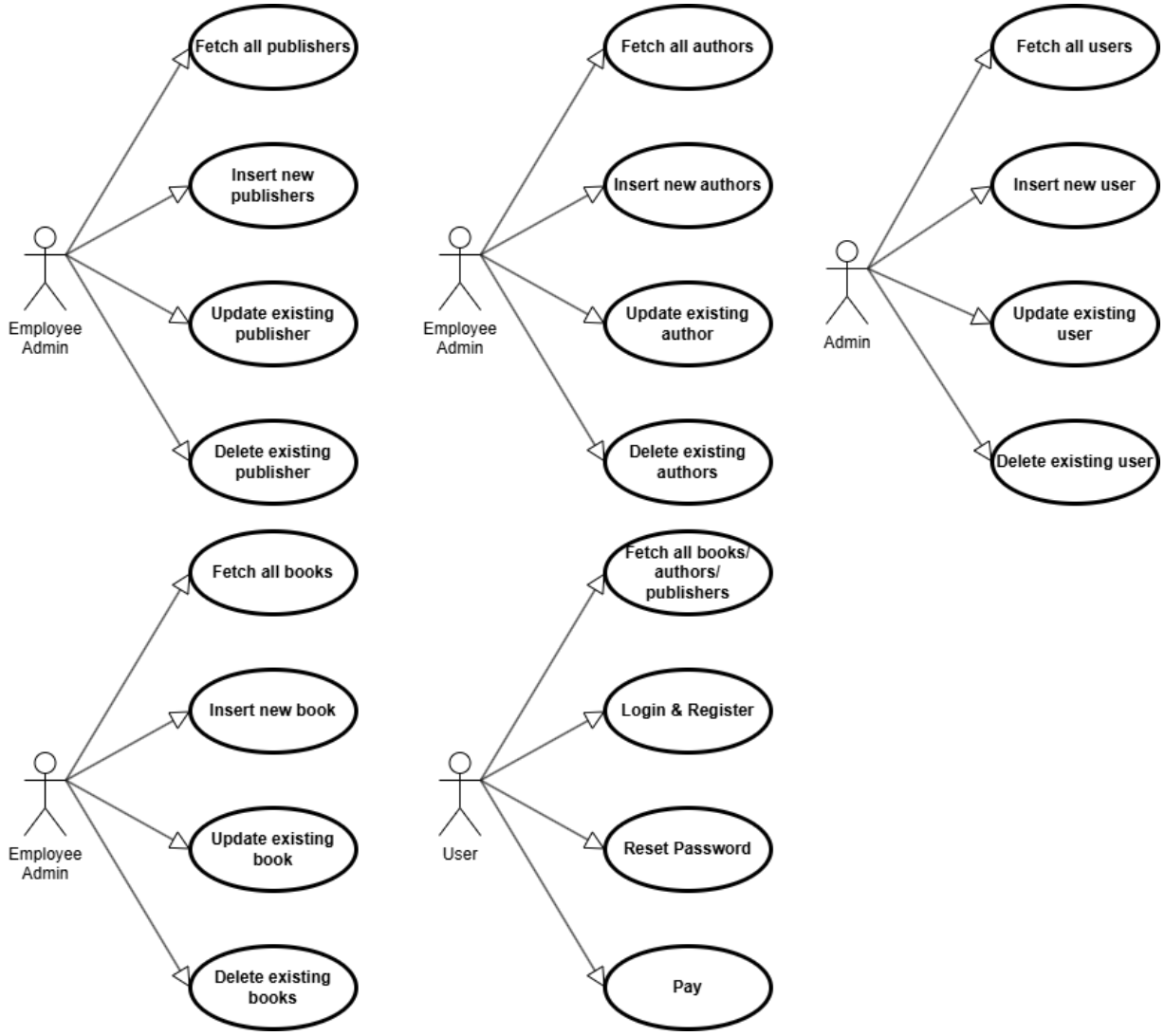

Figure 1. The uses cases of the bookstore management system.

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	Fetching all the entries	Backend and database connection	Any errors that happen during

entities*	in the database that correspond to an entity and displaying them in a table.	succeed, and the entities are displayed in an intuitive manner.	the fetching are reported in the GUI, as well as information messages ("No entities* are present!")
Reset password	Generate a security code that is sent over email to the user and is used to validate the identity when resetting the password.	A random code is generated, saved in the database and the user fills the form correctly, yielding a successful password reset.	Any errors that happen during the process are displayed in the GUI.

(***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

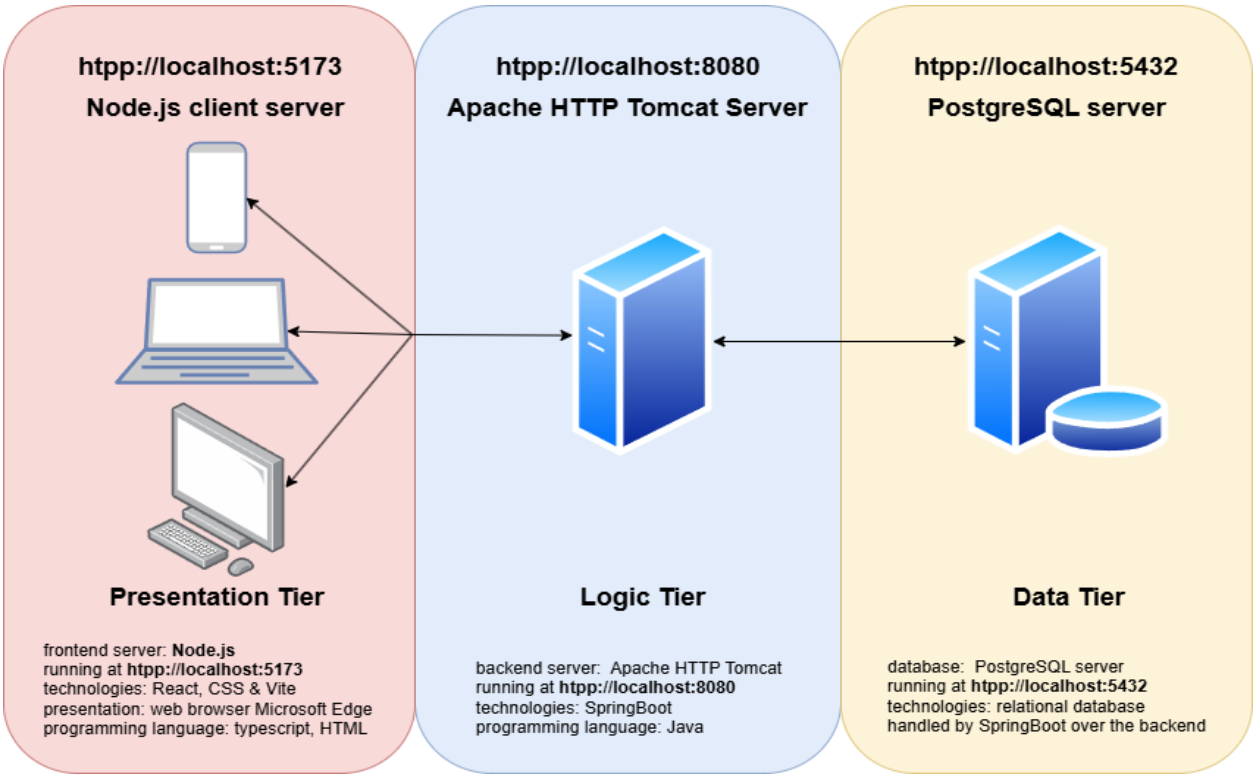

Figure 2. The diagram of the three-server architecture.

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.

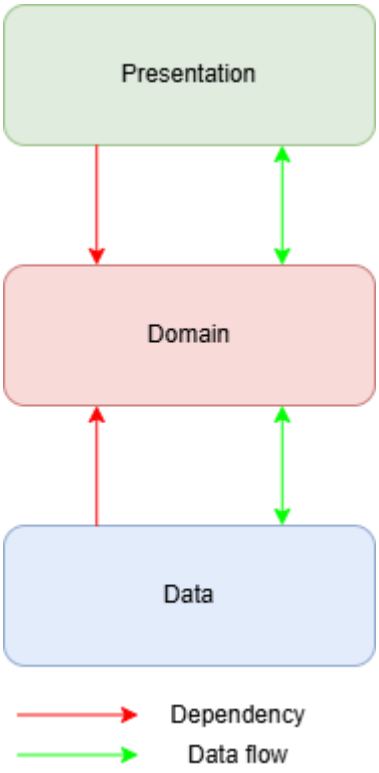

Figure 3. The layers of the system.

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

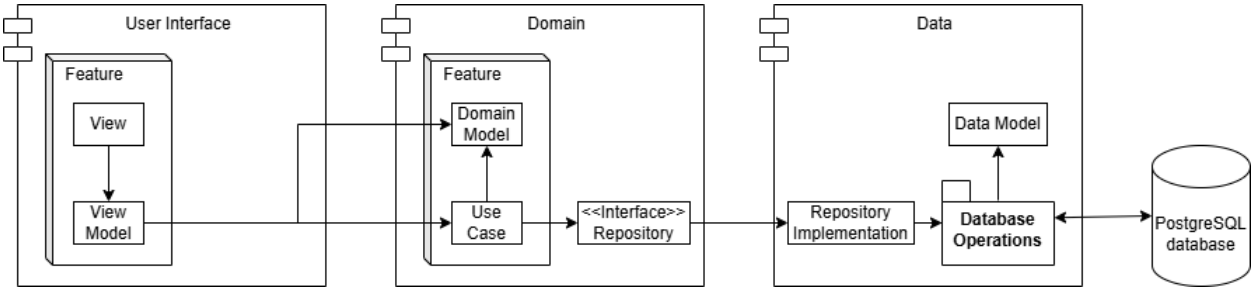

Figure 4. Package diagram of the system.

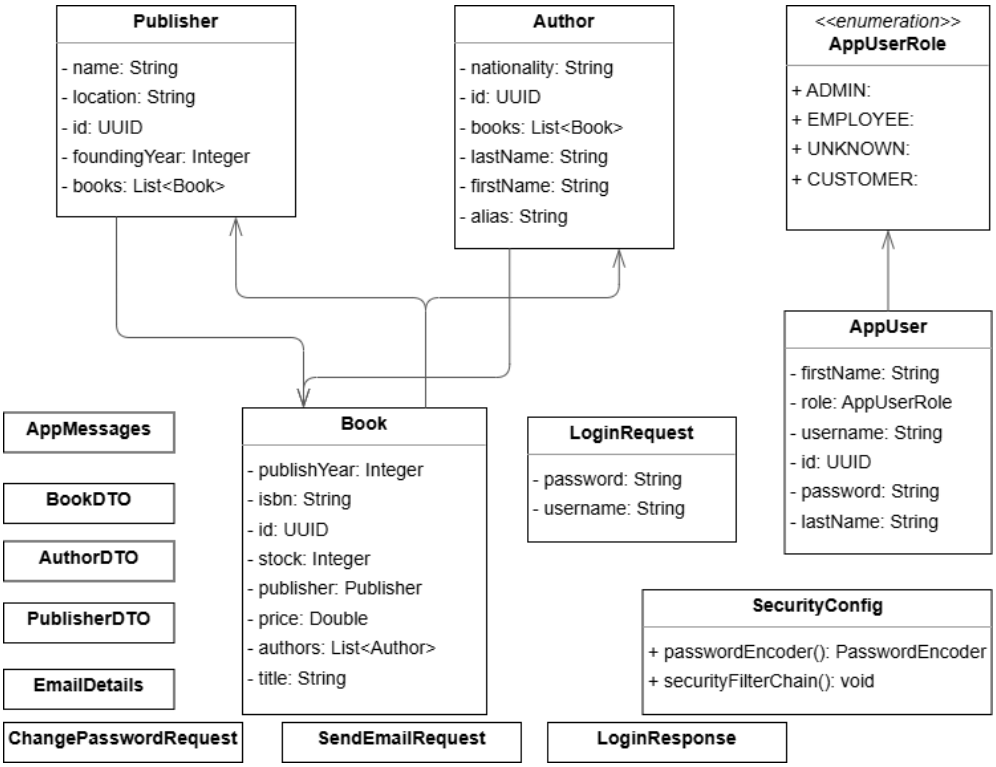

Figure 5. The model classes of the system.

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.

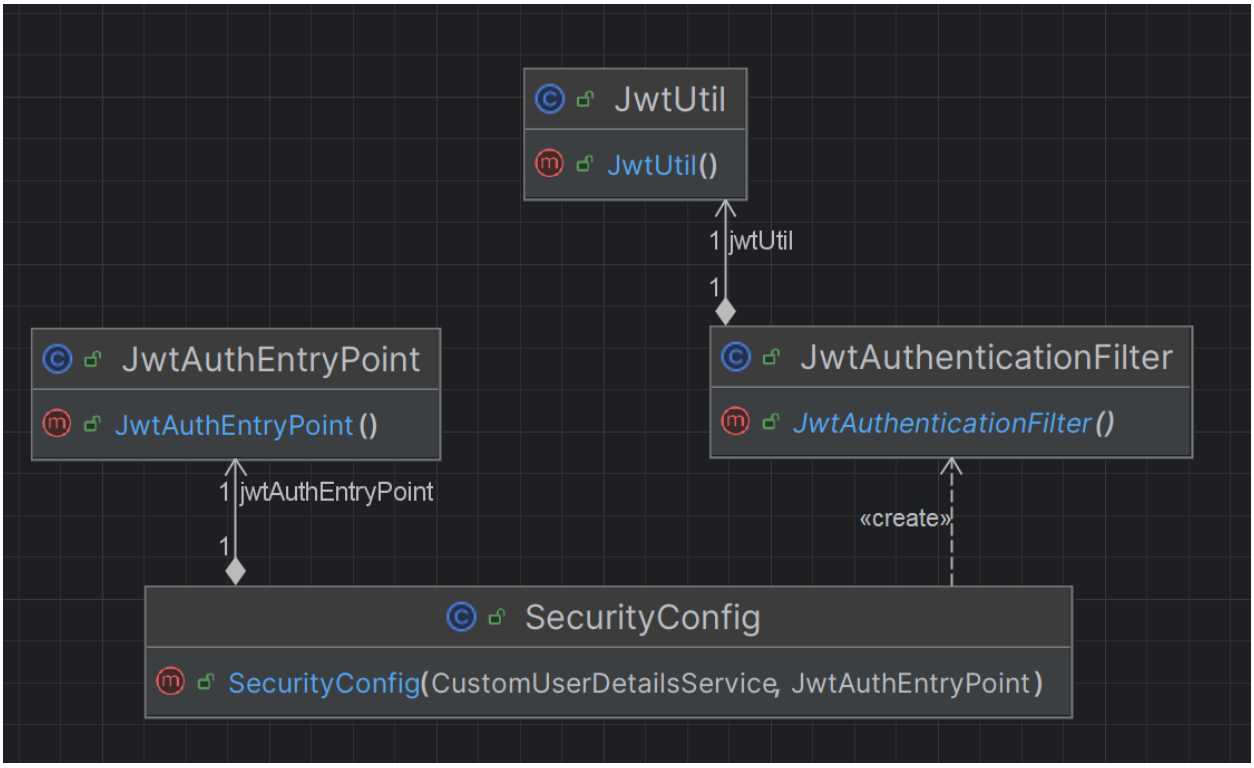

Figure 6. The diagram of the Spring security configuration implementation.

The diagram illustrates a common pattern for implementing JWT-based authentication within a Spring Security context in Java. The *SecurityConfig* class is a Spring configuration class that takes dependencies on both a *CustomUserDetailsService* (for fetching user details) and a *JwtAuthEntryPoint* (for handling unauthorized access attempts). The *SecurityConfig* is responsible for creating a *JwtAuthenticationFilter*, which is the core component for intercepting incoming requests, extracting the JWT, and attempting to authenticate the user based on its contents. The *JwtAuthenticationFilter* utilizes a *JwtUtil* class, a utility component responsible for JWT-related operations such as generating, validating, and extracting information from the tokens. Finally, the *JwtAuthEntryPoint* is invoked when an unauthenticated user tries to access a protected resource, typically sending back an "Unauthorized" response. This setup ensures that requests are authenticated using JWTs, and unauthorized access is properly handled within the application's security framework.

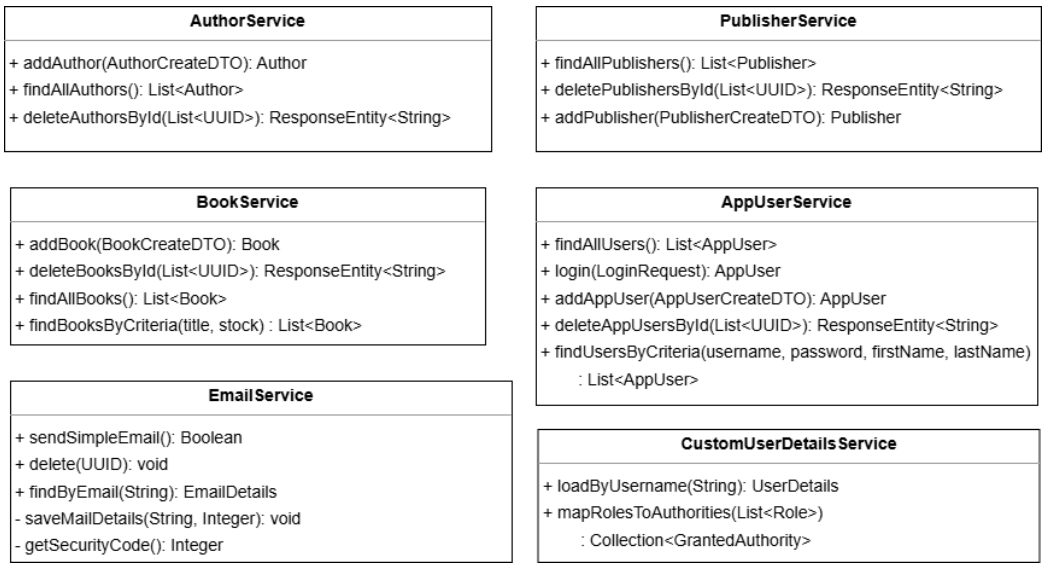

Figure 7. The service classes provide basic CRUD functionalities.

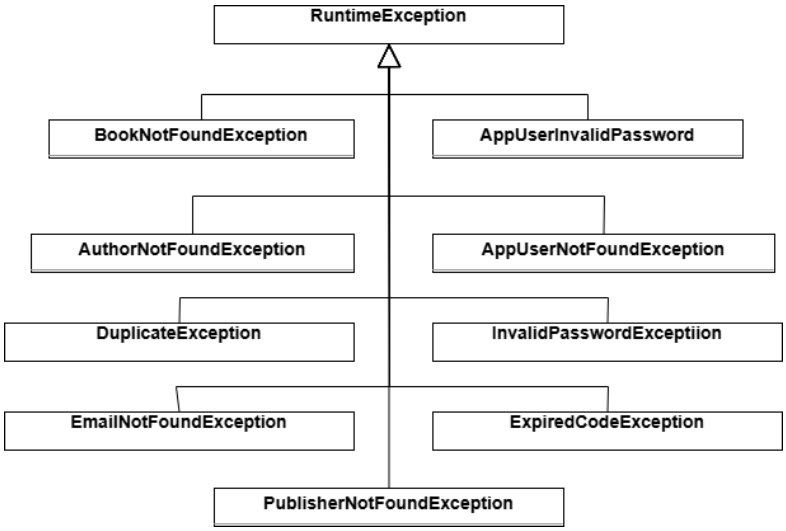

Figure 8. The custom exception hierarchy.

5. Data Model

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

Figure 9. The database diagram.

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.

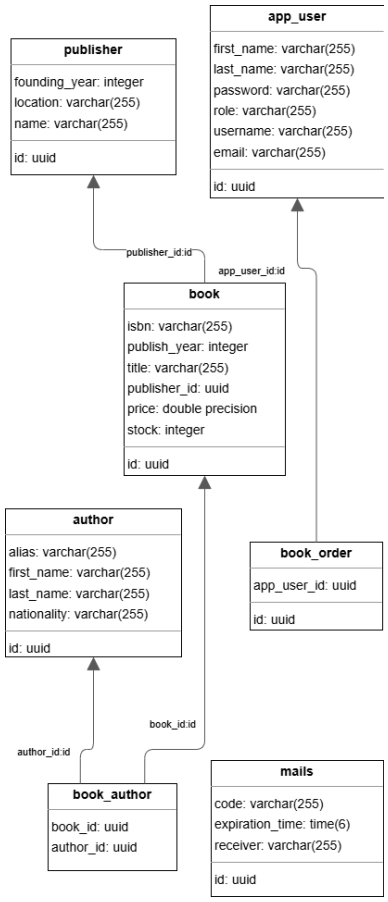

Figure 10. The database diagram of the system.

6. Testing

6.1 Unit testing

This unit test suite for the AppUserService focuses on verifying the core business logic for managing AppUser entities in isolation. By employing mocking with Mockito, the tests simulate the behavior of the AppUserRepo and PasswordEncoder dependencies, allowing for a concentrated examination of the service methods such as findAllUsers, addAppUser, deleteByIds, and updateAppUser. The tests assert that the service correctly interacts with the repository (e.g., saving, finding, deleting), properly encodes passwords using the PasswordEncoder before persistence, and accurately processes user data transfer objects (DTOs) to perform create and update operations. These tests ensure that the service layer functions as expected, independent of the underlying data access implementation, contributing to a more robust and maintainable application.

This unit test suite for the AuthorService meticulously examines the service's operations for managing Author entities in isolation. By leveraging Mockito to mock the AuthorRepo, the tests precisely control the behavior of the data access layer. The test cases cover essential

functionalities such as retrieving all authors, adding new authors, deleting authors individually and in batches, and updating existing author information. Assertions within each test verify that the service correctly interacts with the repository methods (findAll, save, deleteById, findById) and that the data transformations and operations within the service logic are performed as expected, ensuring the reliable management of author data within the application's domain.

6.2 Integration testing

Database seeding from JSON ensures a consistent and repeatable testing environment. Before running tests, the application reads the JSON, parses the data, and inserts it into the database. This allows tests to operate on known data, making test outcomes predictable and easier to verify. It's a common practice for integration and end-to-end testing where interactions with the database are involved.

The integration test suite for the AppUserController thoroughly validates the API endpoints for managing AppUser entities. It covers scenarios such as retrieving all users, adding new users with both valid and invalid payloads (including password validation and duplicate user checks), deleting users by ID, updating existing user information with valid and invalid data (including non-existent users and password validation), and performing user login with correct and incorrect credentials. By interacting with the actual controller and relying on an embedded test database populated with seed data, these tests ensure the correct behavior of the API endpoints, data validation, database interactions, and the application's response to various requests, including status codes and response body content.

The integration tests for the BookController meticulously verify the functionality of the book-related API endpoints, ensuring proper handling of book creation, retrieval, update, and deletion. These tests validate the correct persistence and retrieval of book details, including their ISBN, title, publication year, price, and stock. Furthermore, they rigorously examine the relationships with associated entities such as Authors and Publishers, confirming the integrity of the many-to-many and one-to-many associations. By testing both valid and invalid data inputs, including scenarios with missing authors or non-existent publishers, these tests guarantee the robustness of the API and its adherence to defined business rules and data constraints, ultimately ensuring the reliability of the book management features within the application.

7. Backend data filtering

In essence, Spring provides the Specification API as a first-class citizen for building dynamic and type-safe queries within the data access layer. This approach avoids the need for writing numerous repository methods for every possible filter combination, promoting cleaner, more maintainable, and more flexible data retrieval logic. The framework handles the translation of these specifications into actual database queries, freeing the developer to focus on the filtering logic itself.

8. Security Configuration

This Spring Security configuration class defines the security settings for the application. The @EnableWebSecurity annotation enables Spring Security's web security features, while @EnableMethodSecurity allows for method-level security annotations. The securityFilterChain bean configures how HTTP requests are handled. In this specific setup, CSRF protection is disabled, and all incoming requests are permitted without authentication or authorization checks

due to ".anyRequest().permitAll()". Additionally, HTTP Basic authentication is enabled with default settings. The passwordEncoder bean registers a BCryptPasswordEncoder, which is a strong hashing algorithm used to securely store user passwords by generating a salt and applying a one-way cryptographic function, making it computationally infeasible to reverse the process and obtain the original password from its hashed form. This mechanism ensures that even if the password database is compromised, the actual user passwords remain protected.

9. 2-step Password Reset

When a user initiates a password reset, the application first generates a unique, temporary 6-digit code. This code is then associated with the user's account and stored, with a 5-minute time stamp. Next, an email containing this security code and instructions on how to proceed with the password reset is sent to the user's registered email address. Upon receiving the email, the user enters the 6-digit code on a dedicated verification page within the application. The Java backend then validates this entered code against the stored code for that user, also checking for its expiration. If the code is valid and hasn't expired, the user is then allowed to set a new password, completing the two-step verification process and enhancing the security of the password reset flow.

10. JWT Authentication

JSON Web Tokens (JWTs) offer a stateless and secure method for authenticating users and transmitting information between parties. Upon successful login, the server generates a JWT, which is a compact, URL-safe string comprising three parts: a header specifying the token type and signing algorithm, a payload containing claims (user information and metadata), and a signature calculated using a secret key to verify the token's integrity and authenticity. This JWT is then typically sent back to the client and stored in the local storage. For subsequent requests, the client includes the JWT in the Authorization header (as a Bearer token). The server can then verify the signature without needing to query a database for each request, and if the signature is valid, it can trust the claims within the payload, thus authenticating the user and granting access to protected resources. This approach simplifies backend architecture and enhances scalability.

11. Changes for the project

As part of the ongoing development of the project, I have introduced two new functionalities that enhance the realism and usability of the system as a bookshop platform:

- **Book Navigation and Purchasing** →Users can now browse through the list of available books and select titles they wish to purchase. This interaction mirrors a typical online bookstore experience, where users can make informed buying decisions based on the available catalog.
- **Order Placement and Email Confirmation** → Once a user places an order, the backend generates a unique order number used to identify the transaction. The order is also linked to the username of the individual who placed it, ensuring traceability. After the order is successfully processed, a confirmation email is automatically sent to the user to acknowledge receipt of the purchase.
- **Order History** → Users can easily access their past purchases through a dedicated view, just one click away. This order history feature enhances user experience by offering

transparency and a sense of personal account management.

- **Stock Validation** → During the purchase process, the system verifies that the requested quantity for each book does not exceed the available stock. If a user attempts to buy more units than are in stock, those specific books are excluded from the transaction, and their stock levels remain unchanged. This ensures data integrity and prevents inventory underflow.
- **Improved Data Handling via DTOs** → Several new Data Transfer Objects (DTOs) have been added to streamline the communication between the frontend and backend. These DTOs help to cleanly map and parse incoming JSON payloads, improving code maintainability and reducing the likelihood of runtime errors.
- **Dockerization** → the backend and frontend have been containerized, a PostgreSQL container has been integrated, yielding a container that performs consistent execution across different environments.

These enhancements bring the project closer to a functional and user-friendly bookshop application, aligning the system's behavior with real-world expectations.

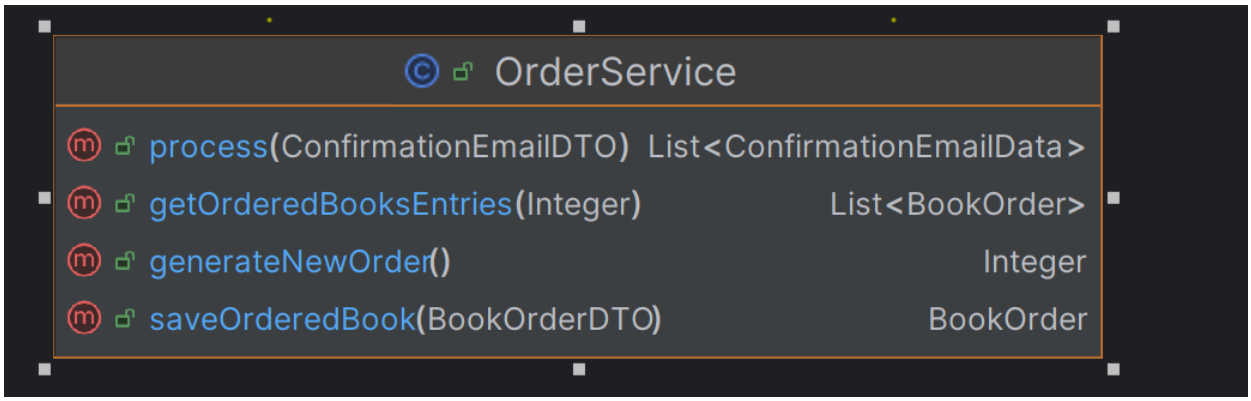

Figure 12. The Order Service class diagram.

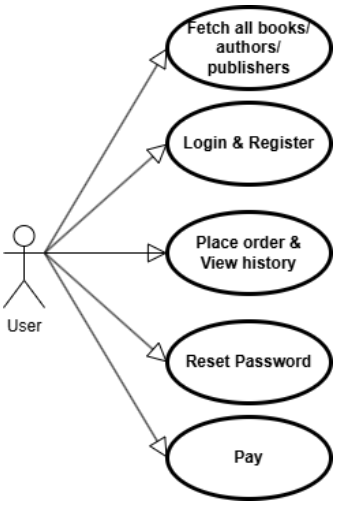

Figure 11. The updated use case diagram for the user actor.

12. 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/
8. https://spring.io/guides/gs/securing-web
9. https://www.baeldung.com/spring-boot-security-autoconfiguration
10. https://tharushkaheshan.medium.com/password-hashing-using-bcrypt-in-spring-security-part-8-d867a83b8695
11. https://www.geeksforgeeks.org/java-current-date-time/
12. https://www.geeksforgeeks.org/java-time-localtime-class-in-java/
13. https://www.geeksforgeeks.org/spring-boot-sending-email-via-smtp/