Book Management Application

Student: Casiana Stefana Campean

**Group: 30238**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 5

4. UML Sequence Diagrams 8

5. Class Design 8

6. Data Model 9

7. System Testing 10

8. Bibliography 10

1. Requirements Analysis

# Assignment Specification

For this assignment we need to build a book management service.

A user should be able to create an account, choose a payment plan and log in to search the book library.

Payments can be done via a cash only policy and need to be validated by library staff.

The library is managed by staff and can be filtered by release date, author, title, genre.

If a book is available, a user can add it to it’s library. If not, the user can join a waiting list. Once a book has been read by a user, it can be returned via the online library return function. This assign the book to the next user in the waiting list after validation by the library staff.

The service also provides with dynamic recommendations based on the latest trends (popular borrowed books) or user defined interests by genre or topic.

# Functional Requirements

The functional requirements are:

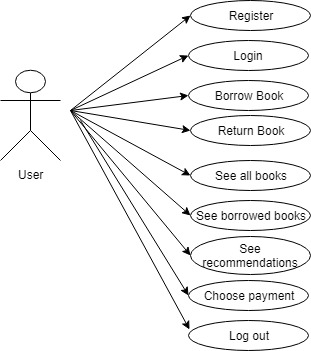
* to implement and test the application
* to use layered architecture
* use Decorator Design Pattern
* Store data in a database
* Validate inputs before submitting he data and saving it in the database
* Send notification to users when a book is available

# Non-functional Requirements

The functional requirements are:

* To commit the work to my Git repository and to do it iteratively as I progress
* To use OOP language
* Use CQRS architecture with a mediator

2. Use-Case Model



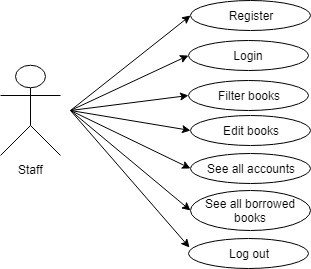
Use case: Borrow Book

Level: User-goal level

Primary actor: User

Main success scenario: If he doesn’t have an account, he register and he choose a payment plan. Then he login, he selects See all books button, he selects Borrow button, he introduces the correct book’s id, then a book is added to his borrowed books list. He can see his new borrowed book if he presses the Back button, he presses the show My Books button, then he sees all his borrowed books and also there he can return any book.

Extensions: He presses the Borrow button, he introduces an incorrect book id, then he gets a message saying the book id is invalid.



Use case: Filter books

Level: Staff-goal level

Primary actor: Staff

Main success scenario: If he doesn’t have an account, he choose to register. Then he enter his account, he presses filter by button, he introduce the correct text in the title, author, date or genre text field and then he presses the filter button by title, author, date or genre and then he can see that data.

Extensions: He presses the Filter by button and he doesn’t introduce the existent values in the text fields, so he gets a message that the data is incorrect.

3. System Architectural Design

**3.1 Architectural Pattern Description**

**CQRS**

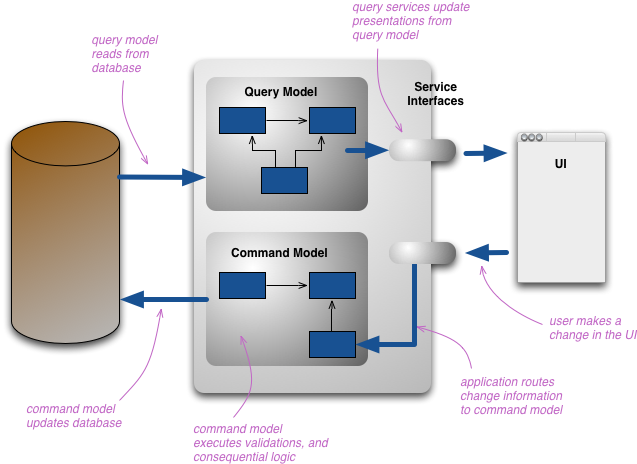
Command-query responsibility segregation (CQRS) is a pattern in system architecture inspired by CQS. It divides the system in two distinct parts, separating the components used for writing (executing commands) from those for querying. In such a system we can find two kinds of requests corresponding to these two models.

Firstly, there are commands – ordering the system do something or change its state. A piece of business logic updates the domain model (or rejects the command) and lets the client know that the change has been accepted (or not).

The main idea of command-query separation (CQS) is that all operations are either:

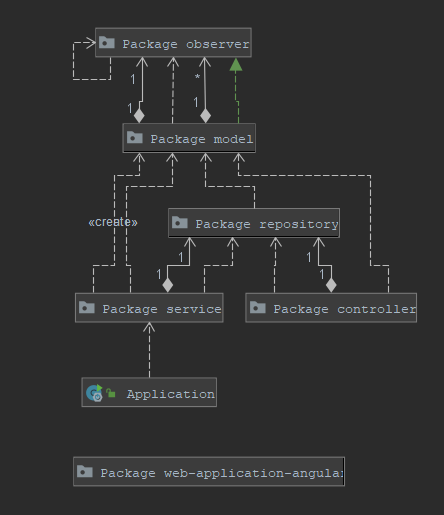
* **commands**, changing state of the system,
* **queries**, getting some information from the system.

Either one or the other, never both. For example, if a command changes anything in the system, it should not be used to read its state. Mutation-free read access should always be possible. Asking a system to change something to read a value seems plain wrong, and queries inadvertently changing state are very confusing, surprising and leading to bugs at best.

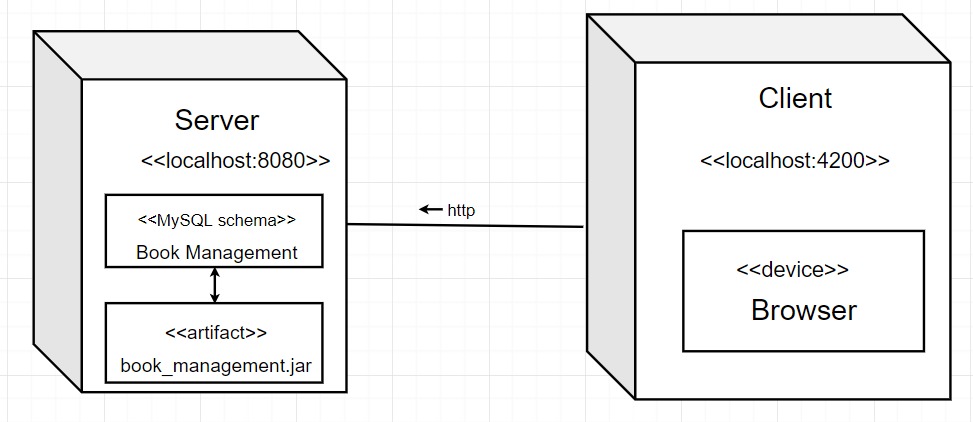


**3.2 Diagrams**

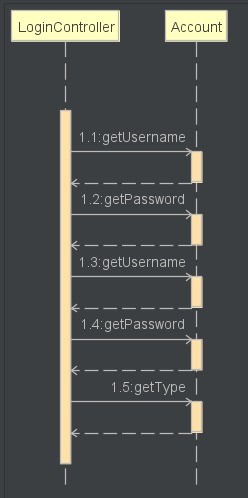
***Package diagram***



***Deployment diagram***



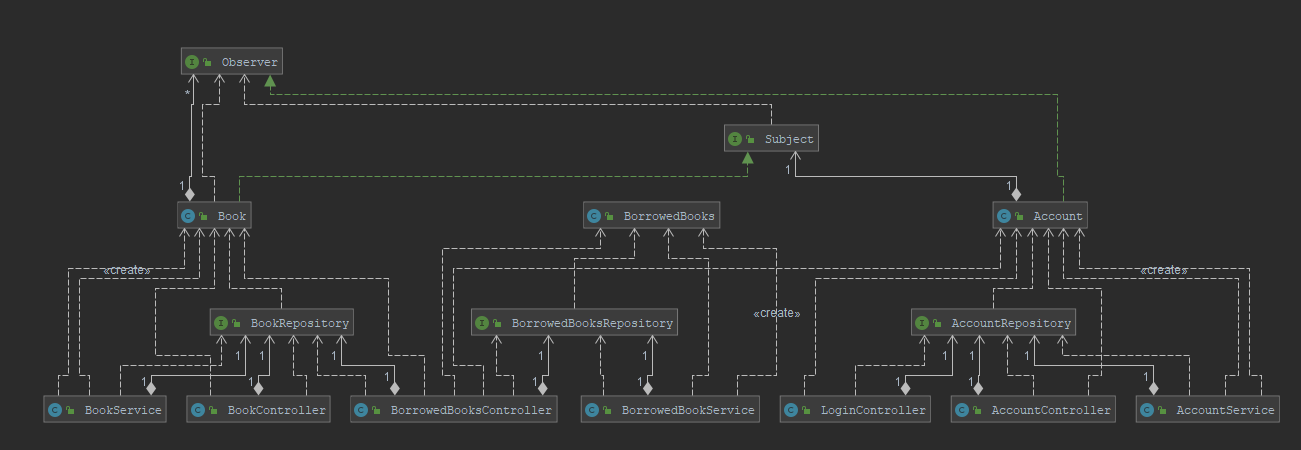
4. UML Sequence Diagrams



This sequence diagram represents a case in the application. The user of this application is the user, who login into the application. Here the user will be identified.

5. Class Design

**5.2 UML Class Diagram**



6. Data Model

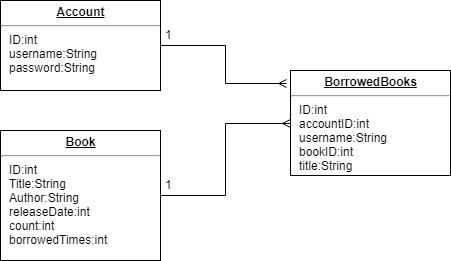
The data model is represented as Java classes and as database tables.

In this project, the data model is represented by these classes in Java/tables in DB:

-Account

-Book

-BorrowedBooks



This diagram represents the relation between tables. The borrowedBooks table contains all the details about the existent borrowed books in the library application. Between book and borrowedBooks and account and borrowedAccount is a one to many relationship, because more than one book or account can appear in the borrowedBooks table.

7. System Testing

8. Bibliography

[1] Hibernate Tutorial: <https://www.youtube.com/playlist?list=PLEAQNNR8IlB7fNkRsUgzrR346i-UqE5CG>

[2]: Angular tutorial:

<https://angular.io/tutorial>

[3] Building a Web Application with Spring Boot and Angular

<https://www.baeldung.com/spring-boot-angular-web>

[4] Angular 7 + Spring Boot Login Authentication

<https://www.javainuse.com/spring/ang7-login>

<https://www.javainuse.com/spring/ang7-basic>

[5] CQRS Architecture

<https://www.baeldung.com/cqrs-for-a-spring-rest-api>

[6] Decorator Design Pattern

<https://www.baeldung.com/java-decorator-pattern>