Movie Booking Web Application

Analysis and Design Document

Student: Enache Mihai

**Group: 30238**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 16/04/2023 | 1.0 |  | Enache Mihai |
| 01/05/2023 | 1.1 | Back to original project idea | Enache Mihai |
| 20/05/2023 | 1.2 | Complete this doc | Enache Mihai |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

# Project Specification

*[Present the project specification]*

The ***BookMyShow*** platform is an online website that aims to provide a platform for users to search and book movies.

1. User Requirements:

* The platform must allow users to create accounts and login.
* Users must be able to search for movies and view movie details such as name, price, rating, showtimes, theatre location.
* The web application must have a booking system that allows users to select a movie, choose a theatre, select a showtime, and book seats.

1. Technical Requirements:

* The backend of the web application will be built using Java and Spring Boot.
* The frontend of the web application will be built using Angular.
* The web application will use MySQL for the database management system.

1. Non-Functional Requirements:

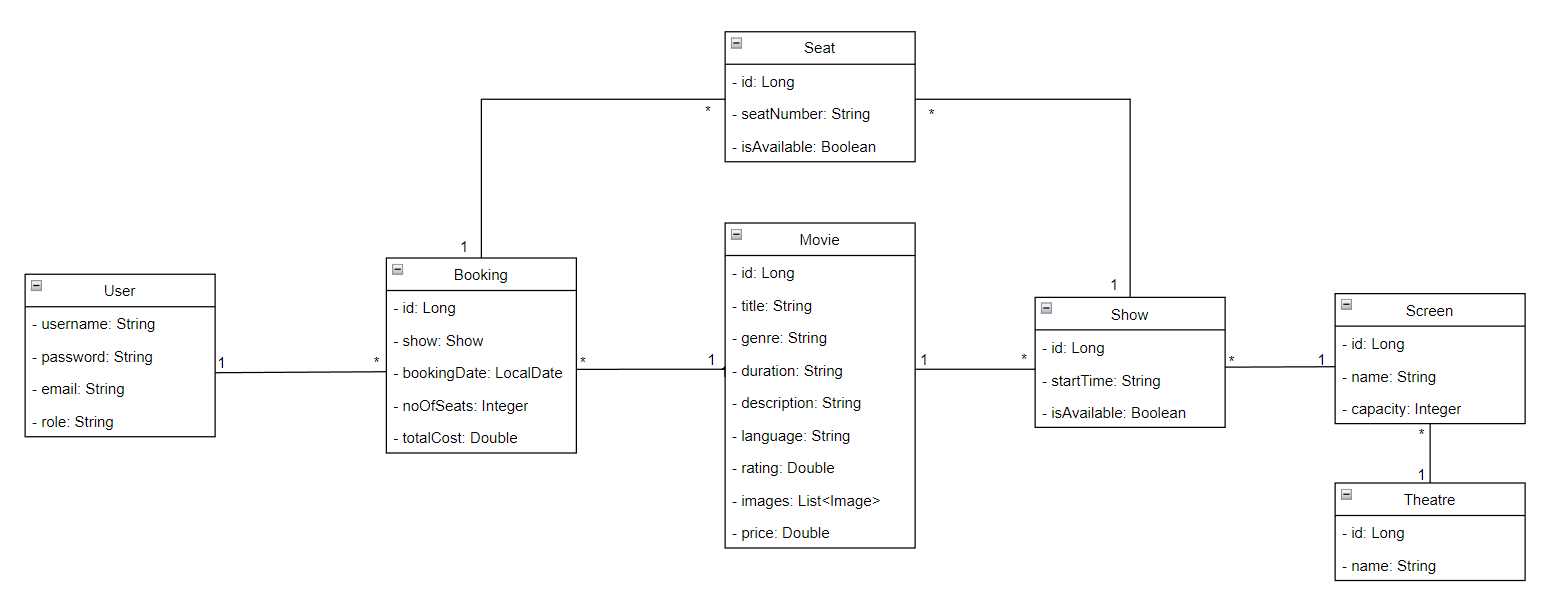
* Availability: The web application must be available 99.9% of the time.
* Performance: The web application must load quickly, with an average load time of 2 seconds or less.
* Security: The web application must follow industry-standard security protocols to protect user data and prevent unauthorized access.
* Testability: The web application must be thoroughly tested before launch to ensure all features work as intended.
* Usability: The web application must be user-friendly and easy to navigate, with a clean and modern design. The booking process must be simple and intuitive.

# Elaboration – Iteration 1.1

# Domain Model

*[Define the domain model and create the conceptual class diagrams]*

* **User**: This entity represents a registered user of the application. It contains attributes such as name, email, password and the role he has.
* **Movie**: This entity represents a movie that can be booked through the application. It contains attributes such as title, genre, language, duration, description, rating, images and a price used for booking.
* **Theatre**: This entity represents a physical location where movies can be screened. It contains attributes such as name and a list of screens.
* **Booking**: This entity represents a reservation made by a user to book a movie in a particular theatre. It contains attributes such as the user who made the booking, the movie being booked, the theatre in which the movie is being screened, and the date and time of the screening.
* **Seat**: This entity represents a seat in a theatre. It contains attributes such as the seat number and whether it is booked or available.
* **Screen**: This entity represents a physical screen or theater screen where movies are displayed. It contains attributes such as name, capacity, the theatre it is associated with and a list of shows.
* **Show**: This entity captures details about a particular movie being played on a specific screen at a given date and time. It contains attributes such as the movie being shown, the screen where it is playing, the showtime, a list of seats and a booking it is associated with.



**Conceptual class diagram**

# Architectural Design

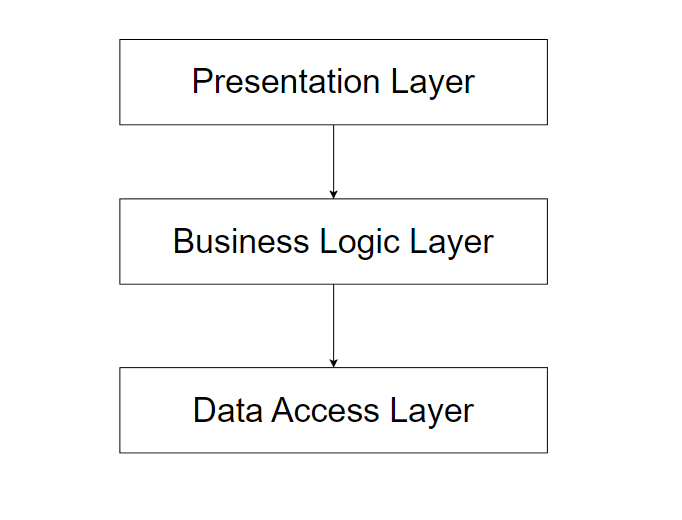
## Conceptual Architecture

*[Define the system’s conceptual architecture; use an architectural style and pattern - highlight its use and motivate your choice.]*

I’m going to follow a **Layered Architecture**, where I have a *Presentation Layer* i.e. the Frontend which I will develop using Angular Framework, the *Business Layer* and the *Data Access Layer* which will be running on a server developed using Spring Boot. For storing the data, I will use a MySQL database.

Some reasons for choosing this type of architecture are:

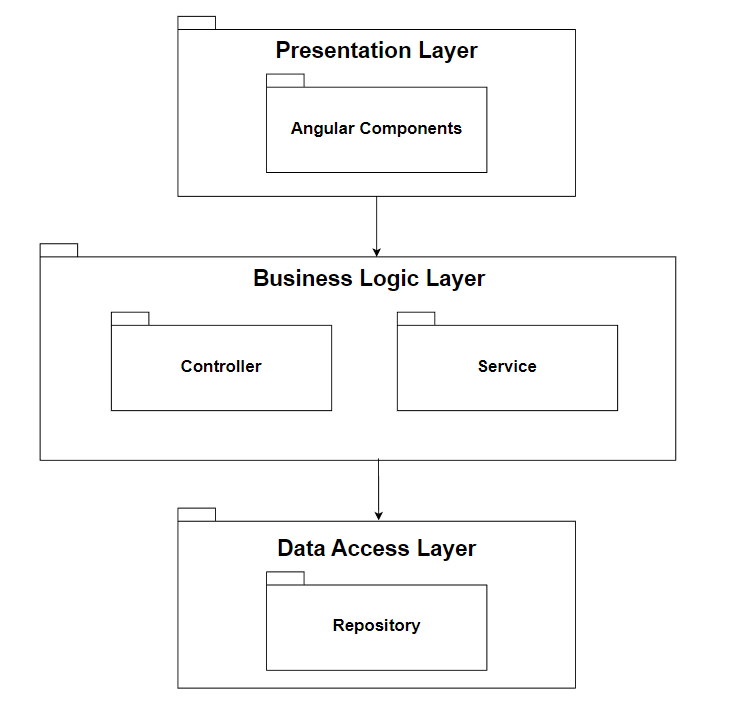
1. The framework is simple and easy to implement
2. There is reduced dependency because of the function of each layer is separate from the other layers
3. Testing is easier because of the separated components, each component can be tested individually



**Conceptual Architecture diagram**

## Package Design

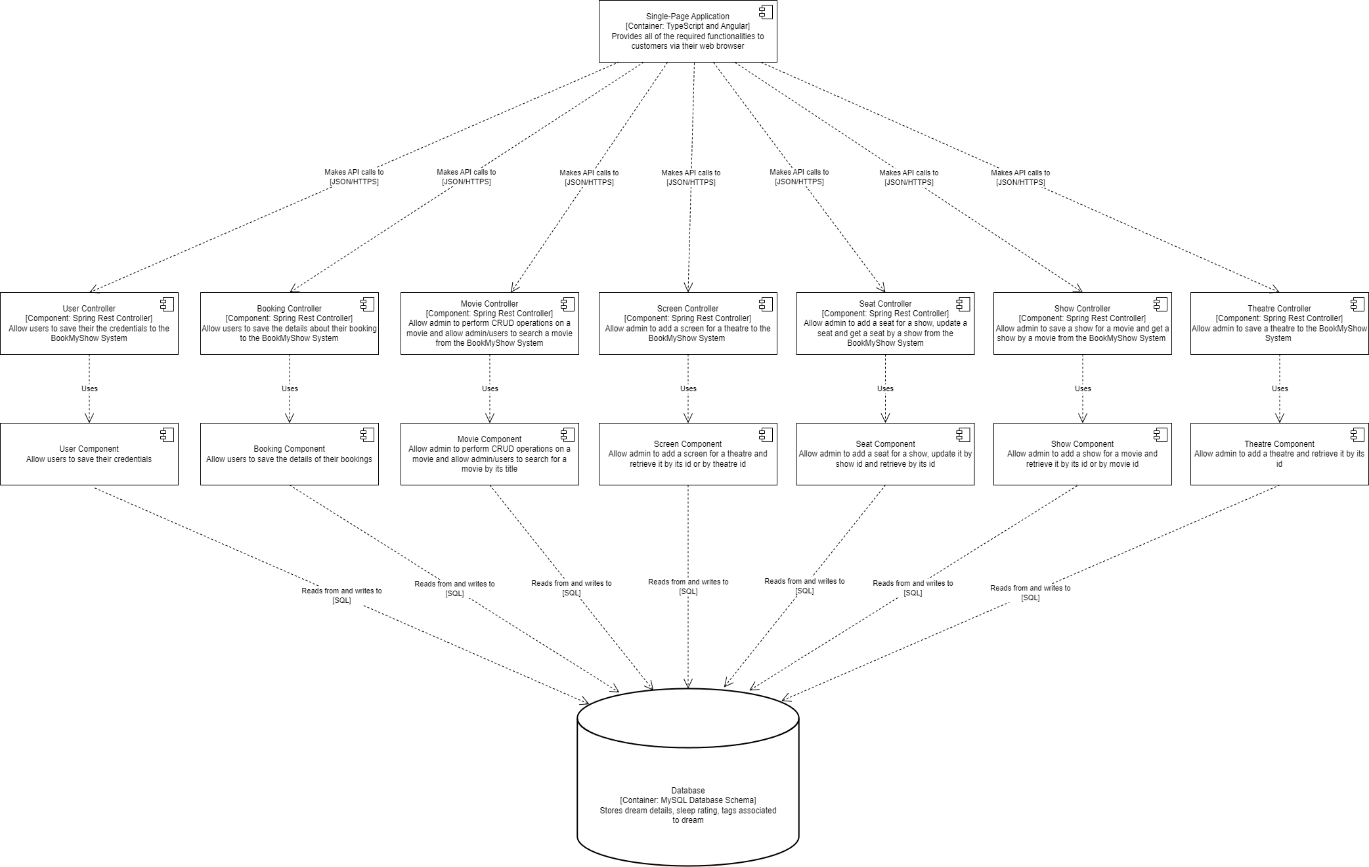
*[Create a package diagram]*



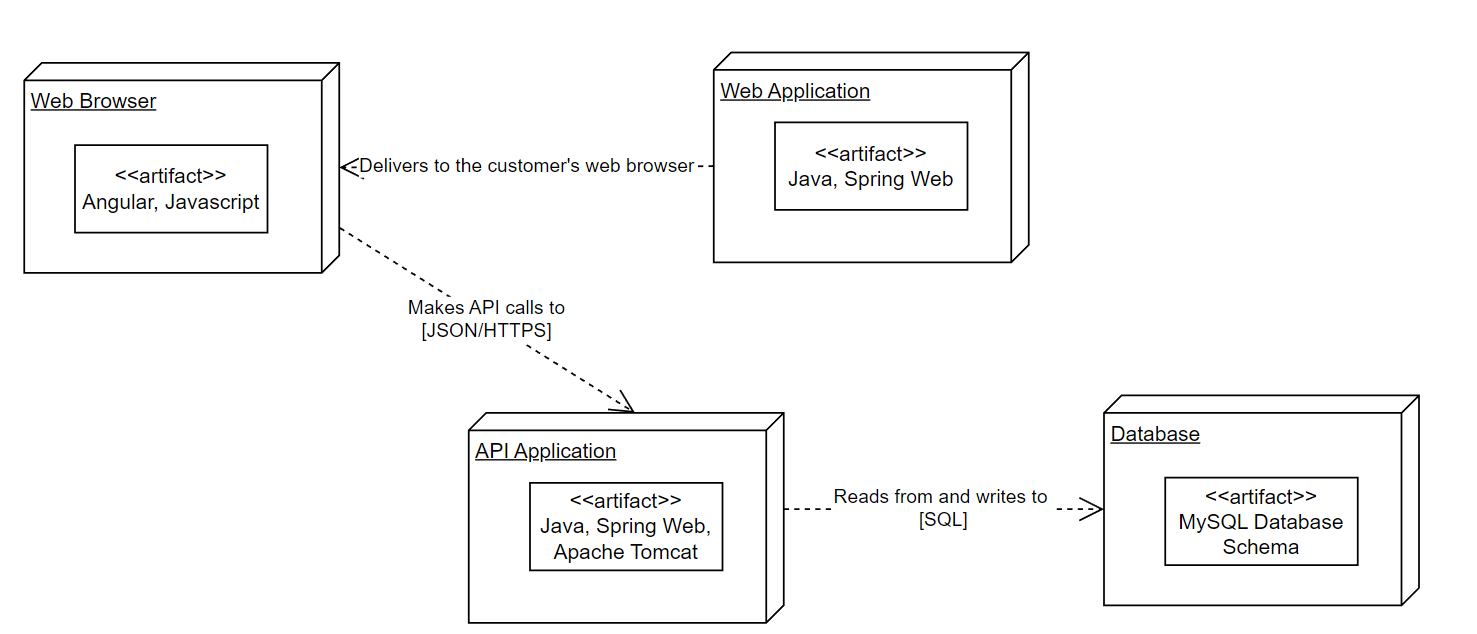
**Conceptual package diagram**

## Component and Deployment Diagrams

*[Create the component and deployment diagrams.]*



**Component diagram**

****

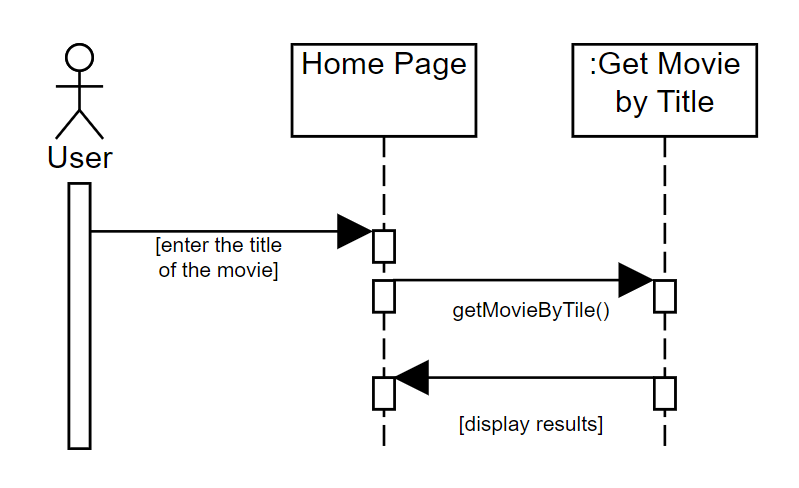
**Deployment diagram**

# Elaboration – Iteration 1.2

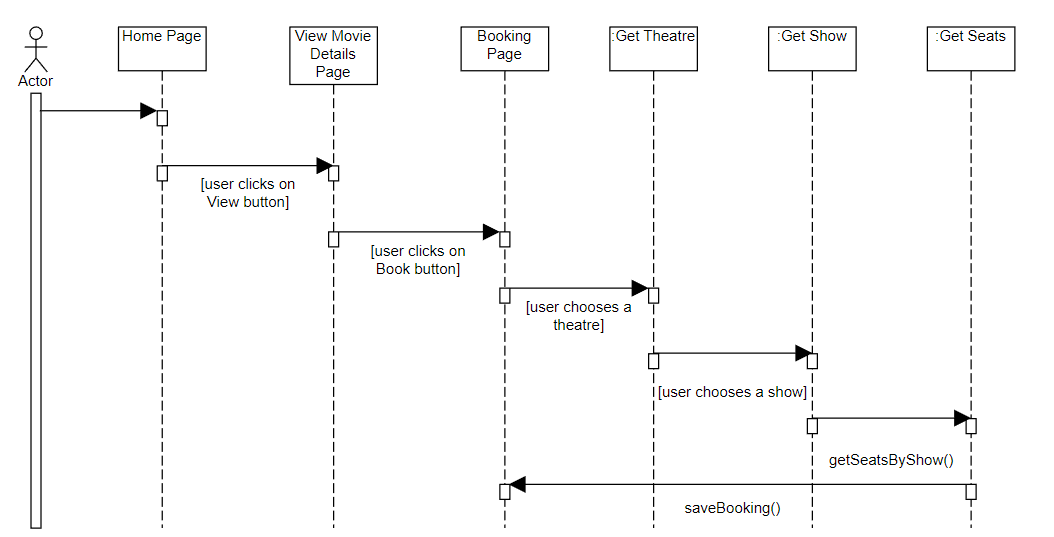
# Design Model

## Dynamic Behavior

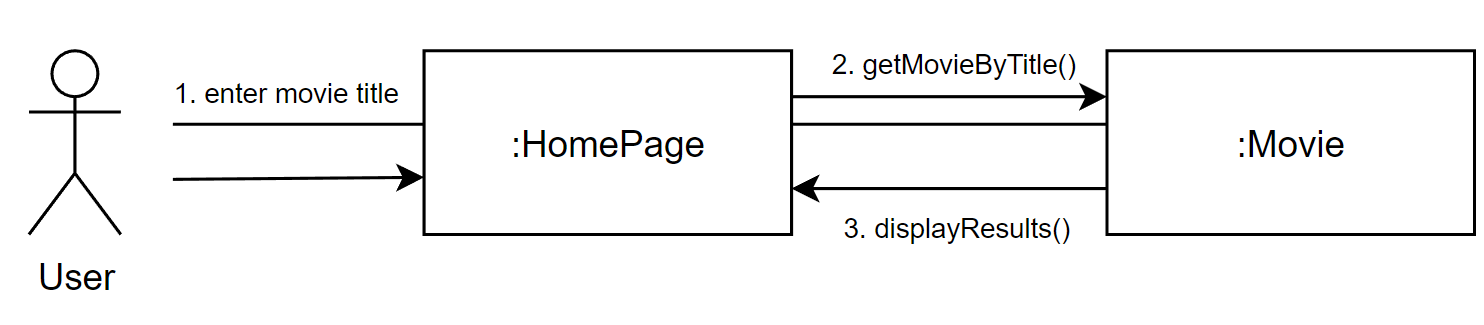
*[Create the interaction diagrams (1 sequence, 1 communication diagrams) for 2 relevant scenarios]*



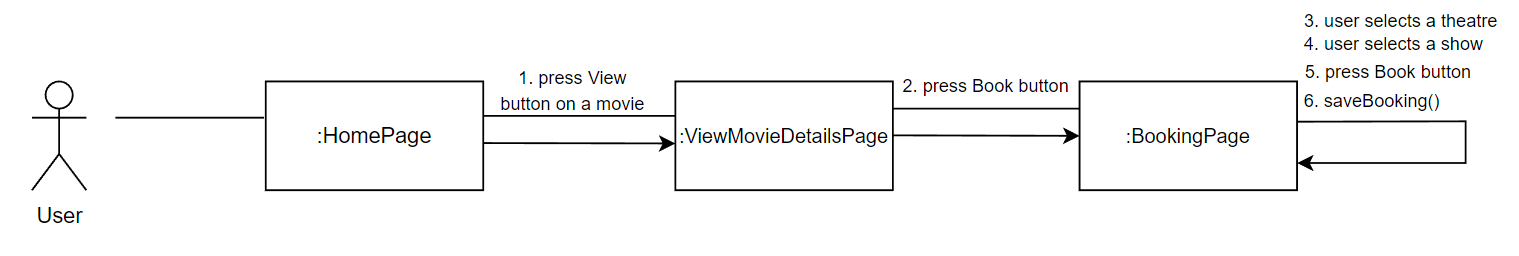
**Sequence diagram for the *Search movie* scenario**

****

**Sequence diagram for *Booking a movie* scenario**

****

**Communication diagram for *Search movie* scenario**

****

**Communication diagram for *Booking a movie* scenario**

## Class Design

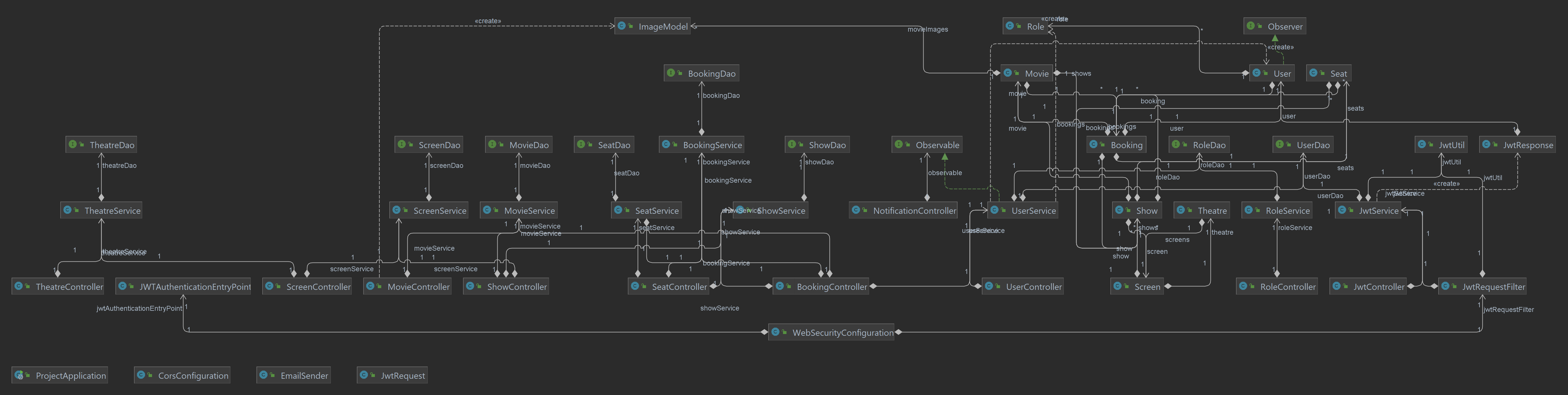
*[Create the UML class diagram; apply GoF patterns and motivate your choice]*

For this assignment I have implemented three design pattern, each belonging to a design pattern category.

The first design pattern that I implemented is **Singleton**. The Singleton pattern is a creational design pattern that restricts the instantiation of a class to a single object. It ensures that only one instance of the class can be created and provides a global point of access to that instance. I implemented this pattern in the context of sending email to users when a new movie is uploaded by the admin. I have the *EmailSender* which is responsible for sending email to users. I chose to implement the Singleton pattern for this class so that I won’t instantiate an email for each user because the content of the email is the same, only the user email is different. To implement this pattern, I created a private static variable which will be the instance of the class. In order to restrict multiple instantiations of the class, I made the constructor private and I have a getter method to get the instance of the class, method that will check if there is not already an instance of the class. In this way, I ensure that only one instance of the class will be created. This design pattern was implemented on the backend side of the application.

The second design pattern that I implemented is **Observer**. The Observer pattern is a behavioral design pattern that establishes a one-to-many relationship between objects. In this pattern, when the state of one object (called the subject or observable) changes, all dependent objects (called observers) are automatically notified and updated. I implemented this pattern by defining two interfaces that are needed in the implementation of this pattern: *Observer* and *Observable*. The entity class *User* will implement the *Observer* interface which contains a method called *notify().* This method is responsible to send the emails to all users when admin notifies the users that a new movie has been uploaded to the website. The *UserService* class will implement the *Observable* interface which contains a method called *setUpdate().* This method is responsible for notifying all the observers. This pattern was implemented on the backend side of the application.

The third design pattern that I implemented is **Decorator**. The Decorator pattern is a structural design pattern that allows behavior to be added to an object dynamically without modifying its original class. It provides a flexible alternative to subclassing for extending the functionality of an object at runtime. I used this pattern in the context of showing an icon to a movie using the movie rating. If the movie rating is greater than 7, then a smile face will appear next to the rating, suggesting that the movie is a good one. If the movie rating is between 5 and 7, then a neutral face will appear next to the rating, suggesting that the movie is good enough to watch. Lastly, if the movie rating is lower than 5, than a sad face will appear next to the rating, suggesting that the movie is not so good. To implement this pattern, I created a *Decorator* interface which has two methods: *getIcon()* and *setIcon()*. Then, I created a *RatingDecorator* class which will implement the *Decorator* interface. In the *setIcon()* method, I will check which condition the rating holds. Lastly, in the *MovieViewDetails* component, I used the decorator to set the icon based on the movie rating. This pattern was implemented on the frontend side of the application.



**UML Class Diagram**

# Data Model

*[Create the data model for the system.]*

1. **User Entry Model**: This model stores information about a user and is used for managing user-related functionalities, authentication and access control within the system.
2. **Booking Entry Model**: This model is used to track and manage reservation and facilitate ticketing and seat allocation. This model stores information about the booking a user made.
3. **Movie Entry Model**: This model is used to store and manage movie-related data, including retrieval, display, and search functionalities within the movie booking system.
4. **Seat Entry Model**: This model stores information about a specific seat and it is used to manage seat availability, allocation, and booking-related operations within the movie booking system.
5. **Screen Entry Model**: This model is used to manage screen-related operations, such as scheduling shows and screen-specific information.
6. **Theatre Entry Model**: This model is used to manage theatre-related operations, such as adding screens and it used to store information related to a theatre.
7. **Show Entry Model**: This model is used to manage and schedule movie shows, track showtimes, and facilitate booking and seat allocation for specific showing. This model stores information related to a show.

# Unit Testing

*[Present the used testing methods and the associated test case scenarios.]*

For this project, the system has been tested using manual testing. I performed exploratory testing and verified that the software meets the specified requirements and user expectations. Even though I did not use any automated testing strategies (such as unit testing or validation testing), I made all the validations necessary so that the system is correctly implemented. Every user input is validated before sending the information to the database. For the validation of the user input I used method like data-flow, partitioning and boundary analysis to ensure that the designed system has no errors.

# Elaboration – Iteration 2

# Architectural Design Refinement

*[Refine the architectural design: conceptual architecture, package design (consider package design principles), component and deployment diagrams. Motivate the changes that have been made.]*

# Design Model Refinement

## *[Refine the UML class diagram by applying class design principles and GRASP; motivate your choices. Deliver the updated class diagrams.]*

# Construction and Transition

# System Testing

*[Describe how you applied integration testing and present the associated test case scenarios.]*

# Future improvements

*[Present future improvements for the system]*

# Bibliography