Ticket Selling System

Student: Ratoi Razvan-Valeriu

**Group: 30431**

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 3

3. System Architectural Design 4

4. UML Sequence Diagrams 6

5. Class Design 6

6. Data Model 6

7. System Testing 8

8. Bibliography 9

1. Requirements Analysis

# Assignment Specification

This assignment’s purpose is to implement an application that would serve as a ticket selling system for the Untold festival, managing shows, tickets, users and so on.

# Functional Requirements

The functional requirements based on the role of admin are the following:

* Perform CRUD operations on cashiers
* Perform CRUD operations on shows
* Export into an xml file all the tickets sold for a specific show

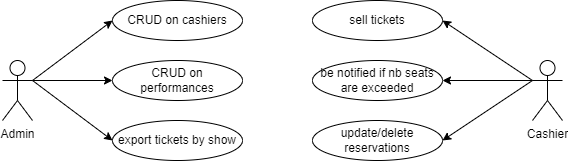
The functional requirements based on the role of cashier:

* Perform CRUD operations on tickets

# Non-functional Requirements

The application should be easy to use, intuitive, accept valid inputs from the user. Some other non-functional requirements would be storing the data into a relational database, a layered architecture should be used, the passwords should be encrypted when stored in the database, unit tests should be provided for some cases, some validations should also be implemented to check various cases and finally, for exporting the tickets, the factory method design pattern should be used.

2. Use-Case Model



The use case diagram is represented below:

Use case: Exporting tickets by show ID

Level: User-goal level

Primary actor: Administrator

Main success scenario:

1. The admin selects the option to export tickets by show ID.
2. The system prompts the box office manager to enter the show ID for which they want to export tickets.
3. The box office manager enters the show ID.
4. The system retrieves all the ticket information for the specified show ID and exports it in a XML file.

Extensions:

1a. If the show ID entered by the box office manager is invalid or does not exist, the system displays an error message and prompts the box office manager to enter a valid show ID.

1b. If there are no tickets sold for the specified show ID, the system displays a message indicating that there are no tickets to export.

4a. If there are any errors or issues with retrieving the ticket information, the system displays an error message and prompts the box office manager to try again later or contact technical support.

3. System Architectural Design

**3.1 Architectural Pattern Description**

For implementing this application, I used the Layered Architecture, which involved having 3 layers: Data Access layer (in the source code found as repositories), Business Logic layer (services) and presentation layer (controllers). The communication between them is realized only between presentation and business logic, business logic and data access. The presentation layer gets the request and inputs from the user and sends them down to the business logic; then the data is processed in the current layer and sent down again to the data access where the database is accessed.

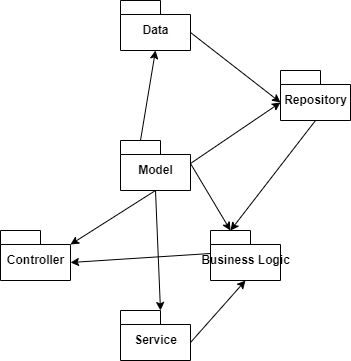
**3.2 Diagrams**

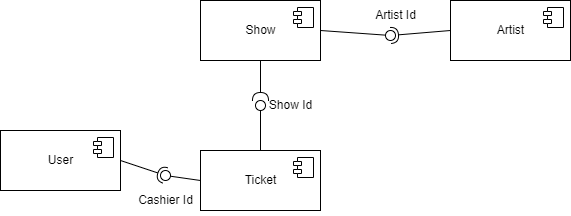
*Diagram

Description automatically generated Create package, component and deployment diagrams]*

System's Architecture

Architecture design diagram:

 Package diagram:

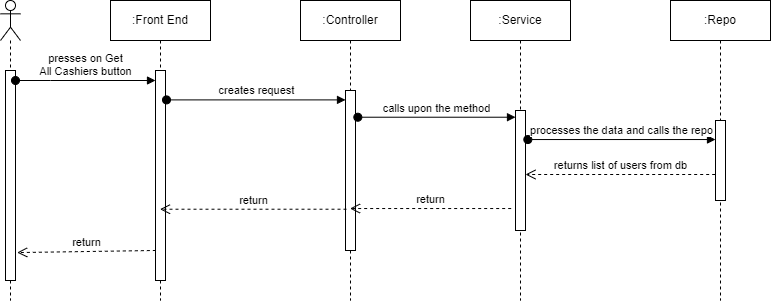
 Component diagram:

Deployment diagram:



4. UML Sequence Diagrams

Scenario: Getting all cashiers’ details

**

5. Class Design

**5.1 Design Patterns Description**

For the implementation of this project, I used the Factory Method Design Pattern, which is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. I implemented it by creating an interface AbstractCreator which would have the method createFile, and another class XmlCreator that implemented the previous one mentioned, which would have the method implemented as well.

**5.2 UML Class Diagram**

Below is the class diagram, where it can be seen that i implemented the Factory Method design pattern by having an interface AbstractCreator and a class that implements it XmlCreator. In the CreateFile method found in the TicketService class, an object of type AbstractCreator is created, this time being an XmlCreator object.

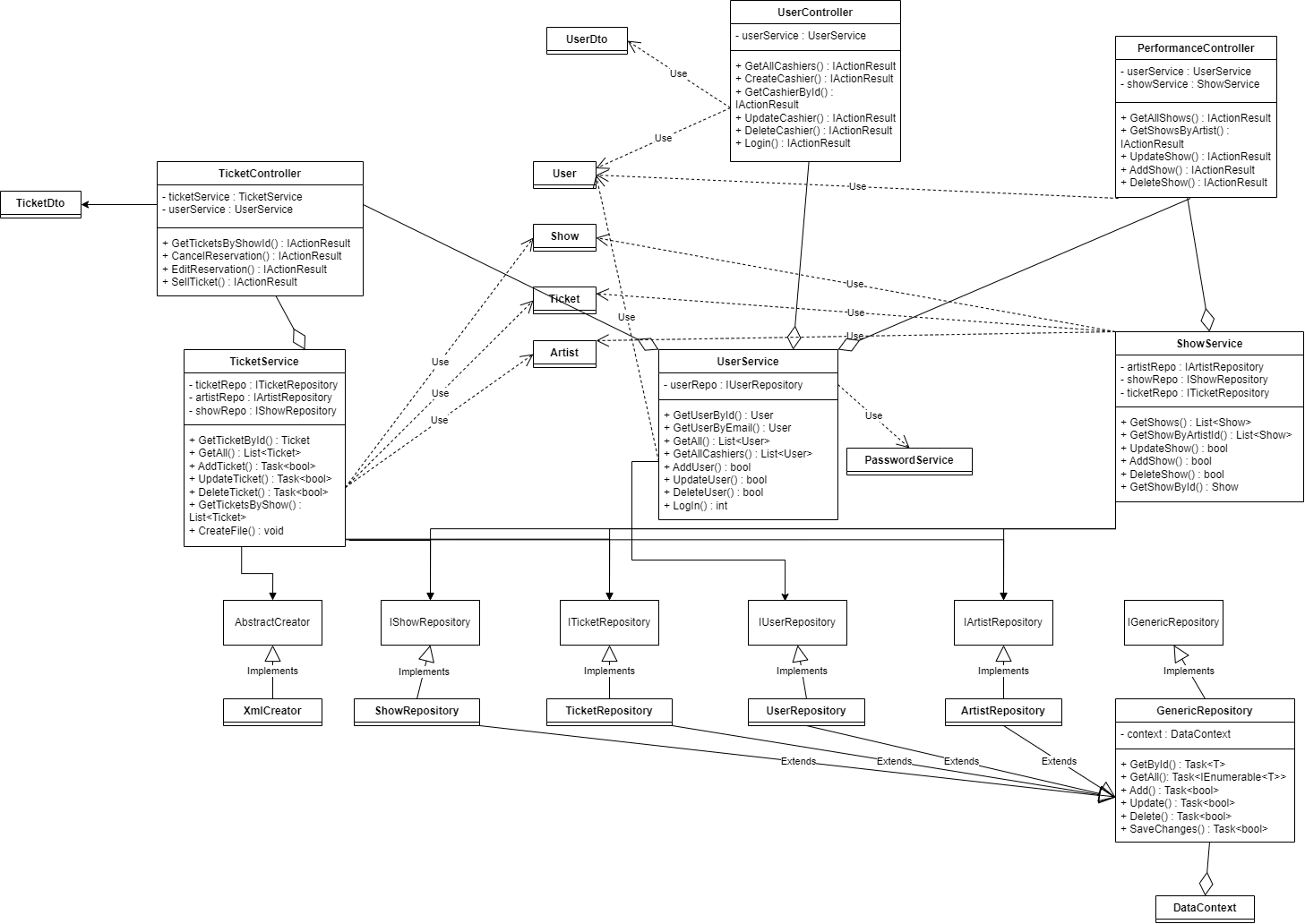
6. Data Model

The first data model used in this implementation is the user, that has the fields:

* Id
* Username
* Password
* Email

Class diagram

* Phone number
* Role (1 for Cashier, 2 for Admin).

**I also used a Data Transfer Object for the user, which has only the username and password fields, used for the log in use case.

For the tickets:

* Id
* CashierId
* ShowId
* Places (number of entries per ticket).

Similarly, I used a DTO for this model as well, having only the show title and number of entries, so the request made when selling a ticket is easier than searching for the id of each show and so on.

Show:

* Id
* Title
* DateAndTime
* ArtistId
* MaxNbTickets

Artist:

* Id
* Name
* Genre

7. System Testing

*[Present the used testing strategies (unit testing, integration testing, validation testing) and testing methods (data-flow, partitioning, boundary analysis, etc.).]*

For testing the application, one method of testing was to call each request (that matched to one use-case) in a service like Insomnia (or Postman) to check what message would be sent and what the output would look like. An example would be the following image, where a GET request is sent (with the id of the user in the route) to get all the cashiers.

We can see that the message is 200 (OK) and the content is the list of all cashiers from the database.

Testing requests

For unit testing I created a class that would have 2 REST API requests, each which would call upon a function that would test one criteria: password hashing and checking if the number of tickets requested is greater than the ones available. The first function mentioned stores an already encrypted password and one that is not encrypted. Then the second one is encrypted with the same method that is found in the project, and finally are compared. If the 2 are equal, a 200 (OK) Http status code would be sent, otherwise, Internal Server Error (500). The second one takes the number of tickets available of a show and requests a ticket having that number incremented. Because the difference between them would be -1, it gives an error telling us that the ticket could not be added in the database.

Graphical user interface, text

Description automatically generated For validation, I checked the user’s role before processing any data so that it would match with the authorization from the requirements. If the role is okay, the function continues, otherwise it gives a Bad Request status code. The same is done for checking if there are any tickets available for a show, having the same output.

8. Bibliography

https://learn.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022

https://learn.microsoft.com/en-us/dotnet/api/system.convert.tobase64string?view=net-5.0

https://www.c-sharpcorner.com/UploadFile/mahesh/create-xml-in-C-Sharp/

https://www.baeldung.com/cs/layered-architecture#:~:text=Layered%20architectures%20are%20said%20to,a%20single%20unit%20of%20software.

https://moodle.cs.utcluj.ro/course/view.php?id=548