**Assignment 1 – Ticket selling system**

Student: Bumbuc Ioana

Group: 30431

**Table of Contents**

[1. Requirements Analysis 3](#_Toc130235475)

[2. Use-Case Model 4](#_Toc130235476)

[3. System Architectural Design 5](#_Toc130235477)

[4. UML Sequence Diagram 8](#_Toc130235478)

[5. Class Diagram 9](#_Toc130235479)

[6. Data Model 9](#_Toc130235480)

[7. System Testing 12](#_Toc130235481)

[8. Bibliography 14](#_Toc130235482)

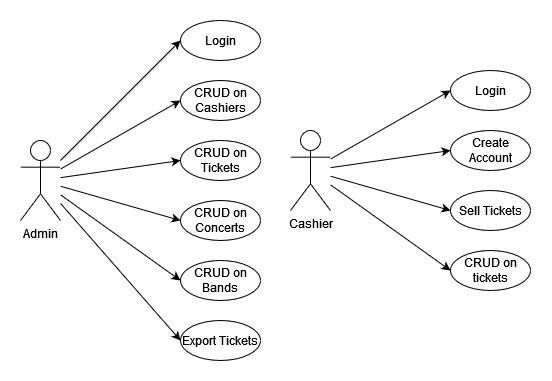
Requirements Analysis

* 1. Assignment Specification

The object of the assignment is to design and implement an application that manages tickets for concerts. The application has two types of users: cashier and administrator. The application should hold information about concerts, bands and tickets. The cashier is responsible with selling the tickets, while the administrator performs CRUD operations on the other entities.

* 1. Functional Requirements
* The application should allow administrators to create, read, update, and delete information about cashiers, concerts, bands, and tickets.
* The application should allow the admin to export the tickets for a concert in a JSON file.
* The application should allow cashiers to sell tickets to customers for specific concerts.
* The application should notify the cashier when the number of tickets for a show exceeds the available tickets.
* The application should store information about concerts, bands, and tickets in a database.
  1. Non-functional Requirements
* The application should be secure and protect sensitive information, such as passwords.
* Scalability: The application should be scalable to handle increasing amounts of data and users without sacrificing performance.
* Reliability: The application should be reliable and minimize the occurrence of errors, crashes, or data loss.

Use-Case Model



Use case: Sell Tickets

Level: Summary level

Primary Actor: Cashier

Main success scenario: The cashier gathers the data for selling a ticket (ID of concert, quantity, names of buyers). The cashier creates a post request to the endpoint *users/cashier/sell/concertId={concertId}&quantity={quantity}*,with the names of buyers in the body of the request. The response is OK and a new ticket is created and added to the database.

Extensions:

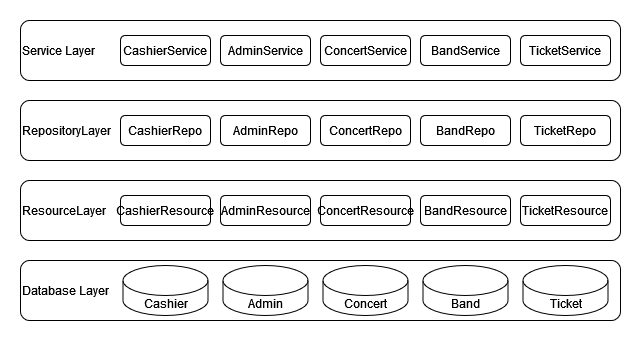
* The number of buyers the cashier provided doesn't match the number of tickets – gets a Bad Request response and an error message.
* The concert id doesn’t exist in the database – gets a Not Found response and an error message
* The quantity exceeds the number of tickets available – gets a Bad Request response and an error message.

System Architectural Design

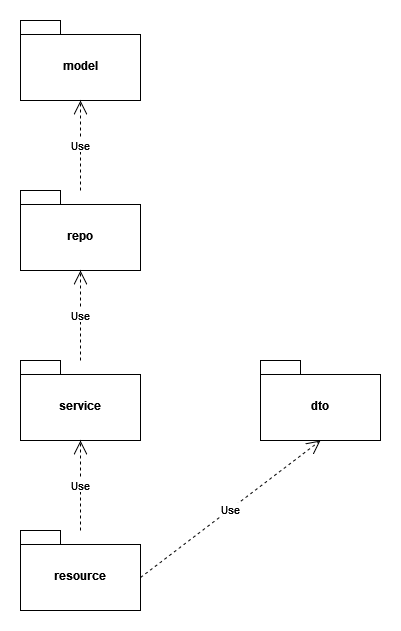
* 1. Layered Architectural Pattern

The layered architectural pattern is a common way of organizing software components. In this pattern, the application is divided into layers, where each layer performs a specific set of tasks. The service layer is responsible for implementing the business logic, while the repository layer is responsible for accessing the data store. The resource layer is responsible for handling HTTP requests and responses. The layers are designed to be loosely coupled, so changes in one layer don't affect the others. This allows for easier maintenance and testing of the application.

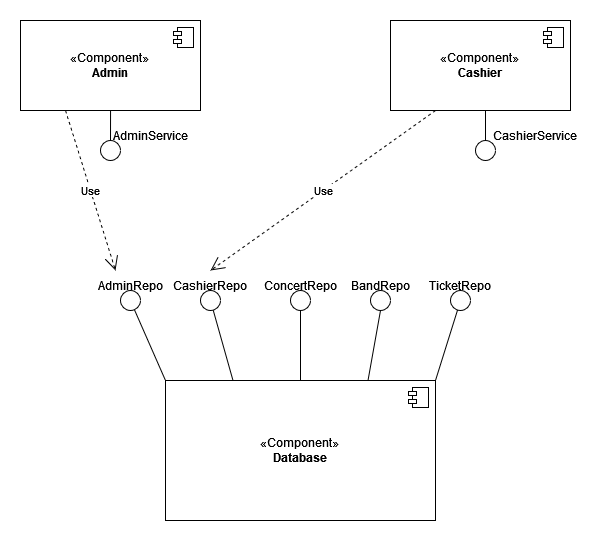
* 1. Diagrams
* Architectural Diagram



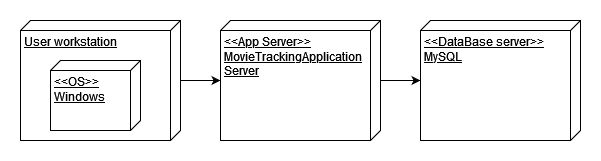
* Package Diagram



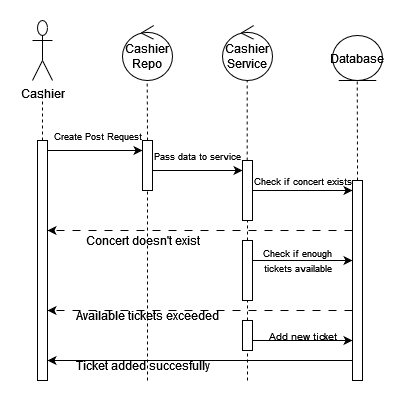
* Component Diagram



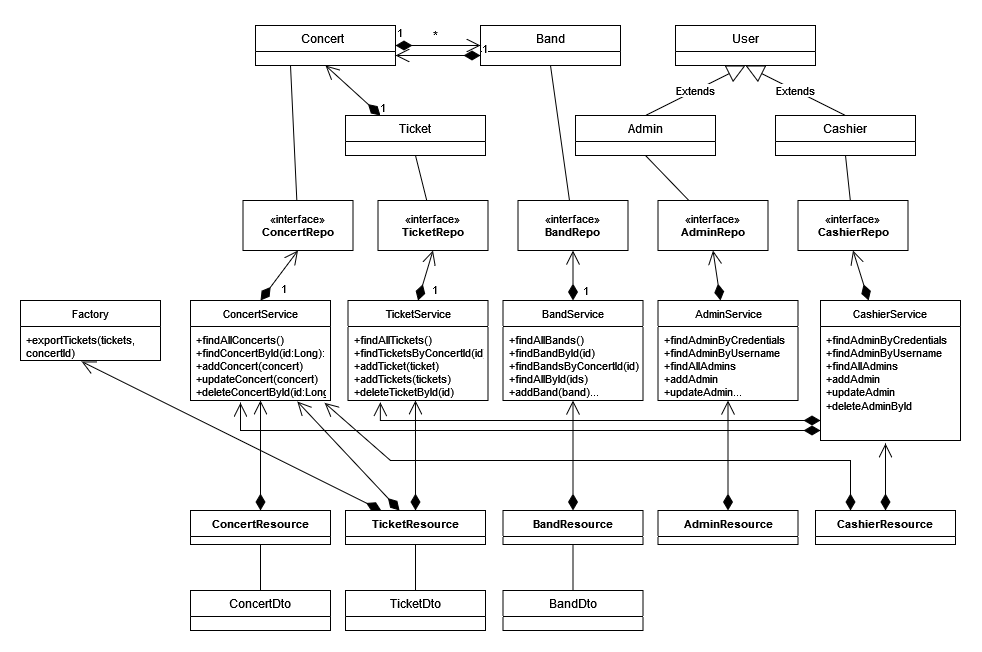
* Deployment Diagram



UML Sequence Diagram

****

Class Diagram

****

Data Model

* 1. **User Abstract Class**

**O imagine care conține text

Descriere generată automat**

The passwords are encrypted before saving them to the database using the PrePersist annotation from spring. Then, the checkPassword method checks if a given password is the same as the encrypted one from the DB.

**O imagine care conține text

Descriere generată automat**

* 1. **Cashier and Admin**

Cashier and Admin are two classes that extend the User abstract class. In the database the users are saved in a “user” table which has a type column. Each user has in the type column either “Cashier” or “Admin”.

* 1. **Concert**

**O imagine care conține text

Descriere generată automat**

In Concert, there is a many to many relationship to Bands, because one concert has multiple bands and one band can play at multiple concerts.

* 1. **Band**

**O imagine care conține text

Descriere generată automat**

* 1. **Ticket**

O imagine care conține text

Descriere generată automat

Each Concert has multiple tickets, thus the many to one relationship.

System Testing

I created unit tests for selling tickets and for checking if a user’s password is encrypted correctly when saved into the database.

* 1. **Test password encryption**

O imagine care conține text

Descriere generată automat

For password encryption, I used the same Base64 module to encrypt a given password. Then, I compare the new password given by the prePersist() function with the expected encrypted password. The test passes.

* 1. **Test selling tickets**

O imagine care conține text

Descriere generată automat

O imagine care conține text

Descriere generată automat

I tested every possible outcome of the sellTickets method. I tested that the tickets are saved correctly when all data is correct. Then I tested the error cases: when the number of buyers doesn’t match the quantity, when the concert doesn’t exist or when they are not enough available tickets. All cases pass.

Bibliography

* JPA Many to Many example with Hibernate in Spring Boot - <https://www.bezkoder.com/jpa-many-to-many/>
* JPA One To Many example with Hibernate and Spring Boot - <https://www.bezkoder.com/jpa-one-to-many/>
* Spring Boot and Angular Full Stack Development | 4 Hour Course - <https://youtu.be/8ZPsZBcue50>