# Movie Theater System

# Software Requirements Specification

# Version 04

06/17/2024

**Group 5**
Sushil Rawtani
Nhu Vo
Adrian Arguelles

Prepared for
CS 250- Introduction to Software Systems
Instructor: Gus Hanna, Ph.D.
Summer 2024

## Revision History

| Date | Description | Author | Comments |
|---|---|---|---|
| 05/27/24 | Version 1 | Group 5 | First Revision |
| 06/03/24 | Version 2 | Group 5 | Added UML Diagram and Descriptions, Software Architecture Diagram and Descriptions, GitHub Accounts, Software Overview and Timeline. |
| 06/10/24 | Version 3 | Group 5 | Improved version 1 and 2 based on the feedback.<br>Added Test Plan Section |
| 06/17/24 | Version 4 | Group 5 | Added Data Management Section Added |

## Document Approval

The following Software Requirements Specification has been accepted and approved by the following:

| Signature | Printed Name | Title | Date |
|---|---|---|---|
| | Your Name | Software Eng. | |
| | Dr. Gus Hanna | Instructor, CS 250 | |
| | | | |

# Table of Contents

# 1. Introduction

## 1.1 Purpose

The purpose of this Software Requirements Specification (SRS) is to provide a comprehensive description of the Movie Theater Ticketing System. This document is intended for use by software developers, project managers, and stakeholders involved in the development and deployment of the system. It will guide them in understanding the functional and non-functional requirements, the various use cases, and the overall system architecture. The document aims to ensure that the system meets the needs of theater administrators and customers, improving the efficiency of theater operations and enhancing the customer experience.

## 1.2 Scope

The Movie Theater system will allow theater administrators to manage movie showtimes, ticket bookings, and customer information. It will also enable customers to browse movie schedules, book tickets, and manage their bookings online. The system aims to streamline theater operations and improve the customer experience.

## 1.3 Definitions, Acronyms, and Abbreviations

- SRS: Software Requirements Specification
- UI: User Interface
- DB: Database
- Admin: Administrator

## 1.4 References

- IEEE 830-1998 Recommended Practice for Software Requirements
- All documents provided in the lecture

## 1.5 Overview

This SRS document is organized into several sections. The Introduction provides the purpose, scope, and references. The General Description section outlines the product's perspective, functions, user characteristics, constraints, and dependencies. The Specific Requirements section details the external interfaces, functional requirements, and use cases. The System Overview includes UML class design, descriptions, software architecture diagrams, and the development plan. Finally, the Test Plan outlines the test cases for verification and validation

# 2. General Description

This section of the SRS should describe the general factors that affect 'the product and its requirements. It should be made clear that this section does not state specific requirements; it only makes those requirements easier to understand.

## 2.1 Product Perspective

The Movie Theater Ticketing System is a web application designed to replace manual booking systems in theaters. It will integrate with existing payment gateways, support multiple theaters, and provide a seamless user experience for both administrators and customers.

## 2.2 Product Functions

- Schedule movies
- Manage ticket bookings
- Handle customer information
- Generate reports on sales and occupancy

## 2.3 User Characteristics

- Administrators: Manage movies and showtimes, require administrative access.
- Cashiers: Handle ticket sales and customer queries, require cashier-level access.
- Customers: View movie listings, book and cancel tickets, manage personal information.

## 2.4 General Constraints

- The system must integrate with existing payment gateways.
- It should be compatible with common web browsers and mobile devices

## 2.5 Assumptions and Dependencies

- Users have basic computer literacy.
- The theater has reliable internet access.
- Payment gateway services are operational.

# 3. Specific Requirements

## 3.1 External Interface Requirements

### 3.1.1 User Interfaces

- The user interface for this website will be constructed using something like JavaScript/React. Users will be able to search for movies, filter movies by genre and title. Users will be able to select movies with available showtimes, and buy tickets for said movies, selecting where they want to sit.

### 3.1.2 Hardware Interfaces

- Working mouse and keyboard to click on buttons select movies and a keyboard to search for movies and input payment information.

### 3.1.3 Software Interfaces

- a Data Management System for storing user account information
- Integrate Paypal, to make purchases easier.

### 3.1.4 Communications Interfaces

The system shall use secure protocols for all data transmissions.

## 3.2 Functional Requirements

This section describes specific features of the software project. If desired, some requirements may be specified in the use-case format and listed in the Use Cases Section.

### 3.2.1 <User should be able to create accounts on the website, and account data should be stored and secured>

*3.2.1.1 Introduction*: The system will let user be able to create accounts on the website in order to purchase tickets

*3.2.1.2 Inputs*: Inputs include email, username, and password. Username shall be 3-20 characters, password shall be around 8 to 48 characters. Username and password will be letters, numbers, and basic symbols such as (, . : ; * & _ ). Usernames will be unique (only one person will be allowed to have a specific username).

*3.2.1.3 Processing:* The data above shall be processed and stored in the database.

*3.2.1.4 Outputs:* The account will have been created.

*3.2.1.5 Error Handling:* If the username is not unique or uses an illegal character, prevent the creation of the account and prompt the user to pick another username.

### 3.2.2 <Available movies and showtimes will be shown and cataloged on the website. >

*3.2.2.1 Introduction:* The system will contain available movies cataloged on the website, with available showtimes.

*3.2.2.2 Inputs:* Click on a movie to view details and view available showtimes.

*3.2.2.3 Processing:* The system will access the webpage/information for the movie that was clicked on.

*3.2.2.4 Outputs:* The movie's details and showtimes will be displayed.

*3.2.2.5 Error:* If the movie's details are not available, redirect to a webpage that displays "This movie is not yet available."

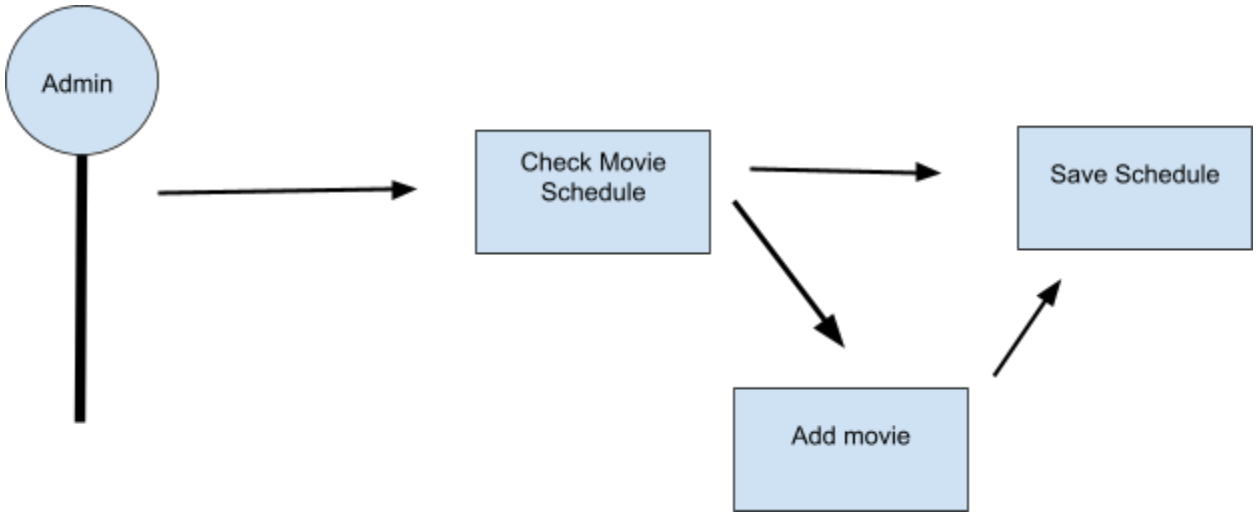
## 3.3 Use Cases

### 3.3.1 Use Case #1: Schedule Movie

- Actors: Administrator
- Description: The administrator adds a new movie schedule to the system.
- Preconditions: The administrator is logged in to the website.

- Postconditions: The new movie schedule is available for customers to view and book.

1. Administrator navigates to the scheduling section.
2. Administrator enters movie details and showtime.
3. Administrator saves the schedule.




### 3.3.2 Use Case #2: Book Ticket

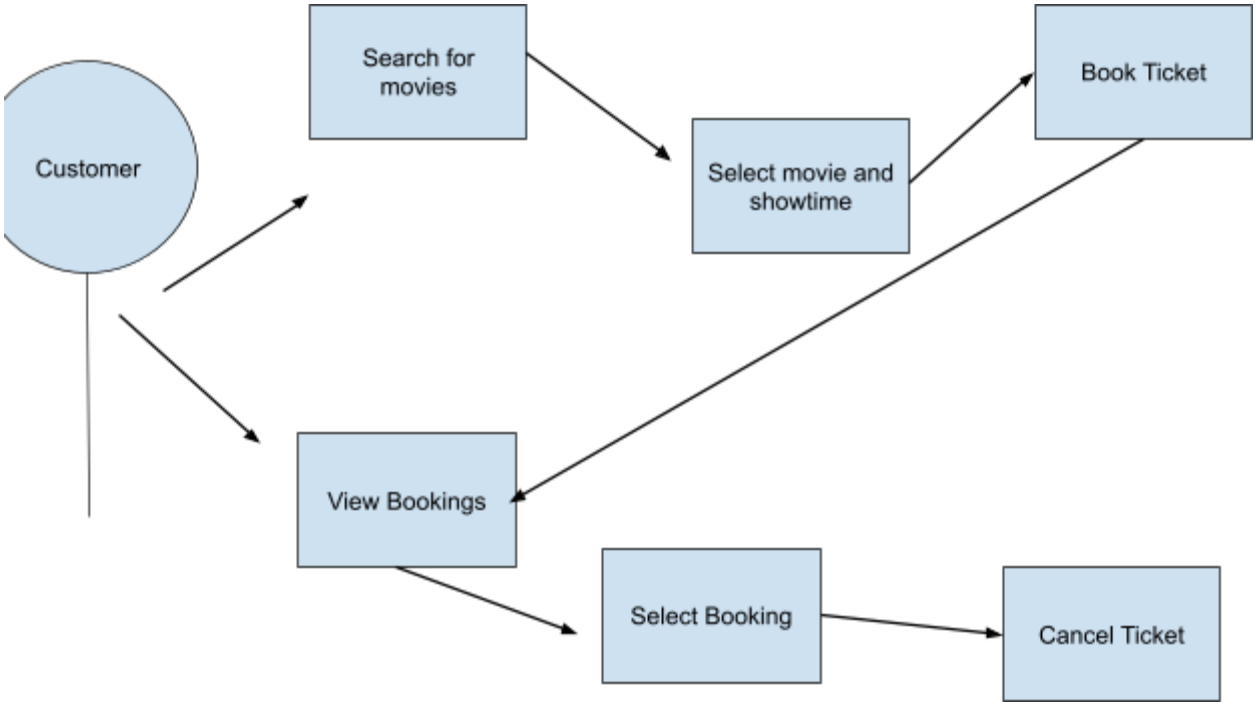
- Actors: Customer
- Description: A customer books a ticket for a movie.
- Preconditions: The customer is registered and logged in.
- Postconditions: The ticket is reserved, and the customer receives a confirmation.

1. Customer searches for the movie.
2. Customer selects a showtime.
3. Customer enters payment details.
4. System confirms booking.

### 3.3.3 Use Case #3: Cancel Ticket

- Actors: Customer
- Description: A customer cancels a previously booked ticket.
- Preconditions: The customer is logged in and has a valid booking.
- Postconditions: The booking is canceled, and the seat is available for others.

1. Customers navigate to their bookings.
2. Customer selects the ticket to cancel.
3. System processes the cancellation.




## 3.4 Classes / Objects

### 3.4.1 Book Ticket

**Attributes:**

- Username: The username of the account that purchased the ticket
- Movie: The movie being booked
- Seat: What seat was booked
- Date of Movie: The date the movie is playing
- Price: The price of the ticket
- Purchase Details: The purchase details, to identify the specific purchase.
-

**Functions:**

- startBooking(): make a new booking
- getPrice(): Get and calculate the price of the ticket
- confirmBooking(): Confirm the booking of the ticket
- getDetails(): get the details of the booking.

### 3.4.2 Movie

**Attributes:**

- movie: The name of the movie

- genre: the genre of the movie
- cast: the cast of the movie
- rating: the age rating of the movie
- synopsis: what the movie is about

**Functions:**

- createMovie() - create movie with all the details;
- updateMovie() - update an existing movies details
- getMovieDetails() - get the details of the movie

## 3.5 Non-Functional Requirements

### 3.5.1 Performance

The system will be designed to perform 90% of user requests within 1-2 seconds, and create bookings within 3 seconds.

### 3.5.2 Reliability

The system will be designed to keep backups of user data and the catalog of movies in order to maintain reliability. Any errors that occur during the operation of the system will be handled within one hour - two hours.

### 3.5.3 Availability

The system shall be available at all times, except for when the website needs full maintenance.

### 3.5.4 Security

The system will be designed in a way so that user account data is secure in the database. Since we are dealing with account data and payment information, we will prioritize a high level of security in our system to prevent sensitive data being stolen.

### 3.5.5 Maintainability

The system will be designed with maintainability in mind to ensure it can be easily updated and modified as needed. Specific metrics and criteria for maintainability include:

- Modular Design: The system will be built using a modular architecture, allowing individual components to be updated or replaced without affecting the entire system.
- Code Documentation: Comprehensive documentation will be provided for all code, including inline comments and detailed descriptions of functions and modules, to facilitate understanding and modifications by future developers.
- Version Control: The system's codebase will be managed using a version control system (e.g., Git), enabling tracking of changes, rollbacks, and collaborative development.
- Error Logging: Implement robust error logging to help diagnose and address issues quickly.
- Testing Framework: Use a testing framework to ensure that updates do not introduce new bugs. Regular automated tests will be run to verify the integrity of the system.
- Scalable Architecture: Design the system architecture to accommodate future growth, including the potential addition of new features and increased user load.

The team will follow best practices for software development and maintenance to ensure the system remains reliable and functional over time.

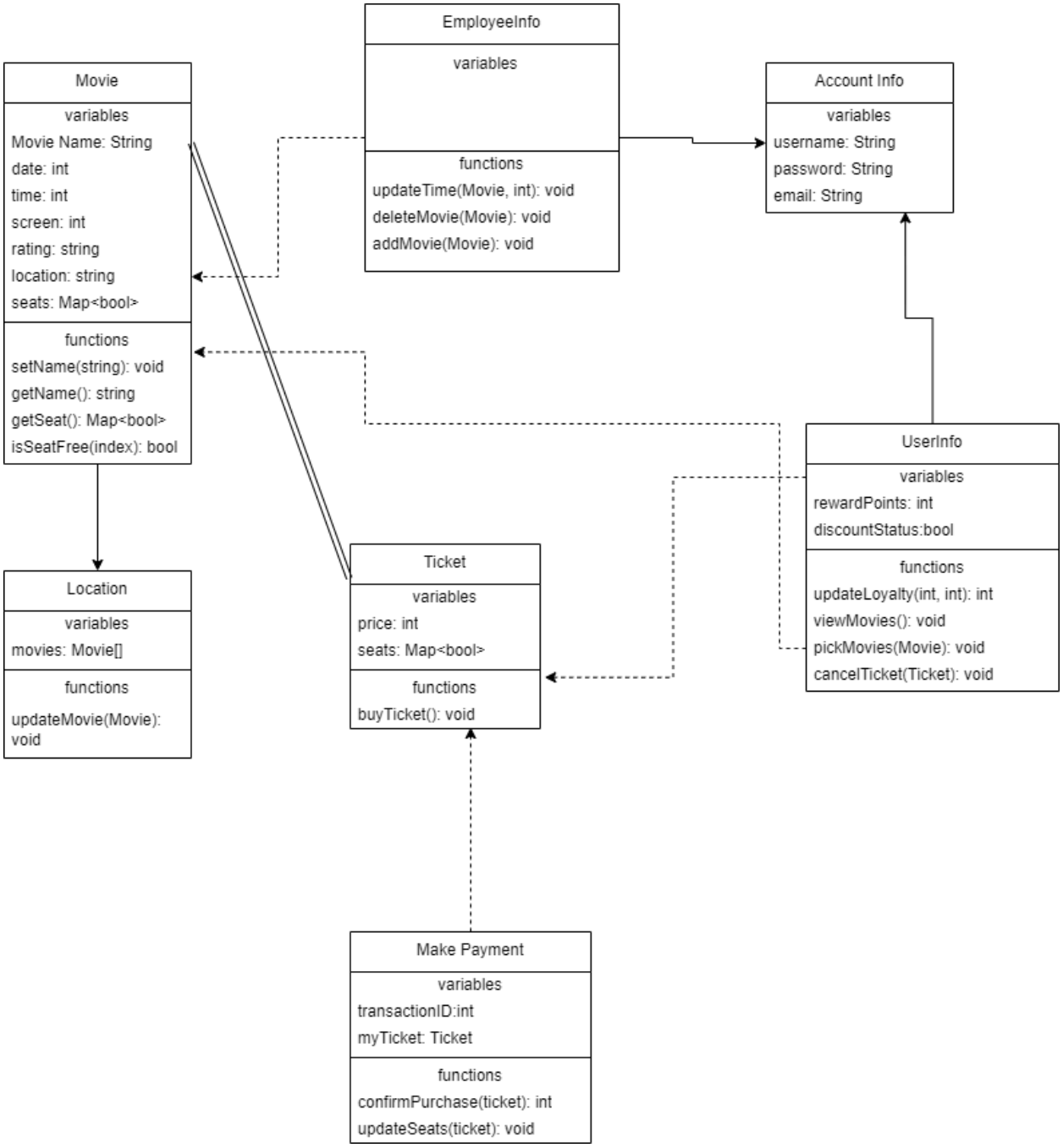
### 3.5.6 Portability

The system will be designed to work and be accessible on common web browsers.

# 4. System Overview

The Movie Theater Ticketing System is designed as a web application to facilitate movie ticket booking for customers and to streamline theater management operations. The system will allow users to browse movie listings, book tickets, manage their bookings, and make payments online. Administrators will be able to manage movie schedules and handle customer information. The overall workflow involves various interactions between the user interface, backend services, and databases.

### 4.1 UML Class Design




## 4.2 UML Class Design Descriptions

The above diagram shows our class diagram, consisting of 7 classes. This section will go in depth of each class and their variables and parameters. The Movie Ticket Booking System is designed to manage and facilitate the various operations involved in booking movie tickets. This

system encompasses several key classes, each responsible for different aspects of the process, from managing movie details and user accounts to handling ticket purchases and payments.

1. The Movie class contains variables, relevant details that are needed for a movie. This includes, of course, the movie name (string), the date(int), time(int), rating(int), etc. of a given movie. This class contains simple functions such as a setter and getter for the movie name, getSeat(), a boolean function isSeatFree() to check if a seat is available.

2. The Location class simply contains a list of movies available at a current location, and a function, updateMovie(Movie) to update the movies at a given location.

3. The EmployeeInfo class is responsible for managing the details and operations related to employees who handle movie data. This class contains various functions to update, delete, and add movies. Specifically, it includes the following functions: updateTime(Movie, int) : void is the function that updates the time of a specified movie. deleteMovie(Movie) : void is the function that deletes a specified movie from the system. addMovie(Movie) : void is function adds a new movie to the system.

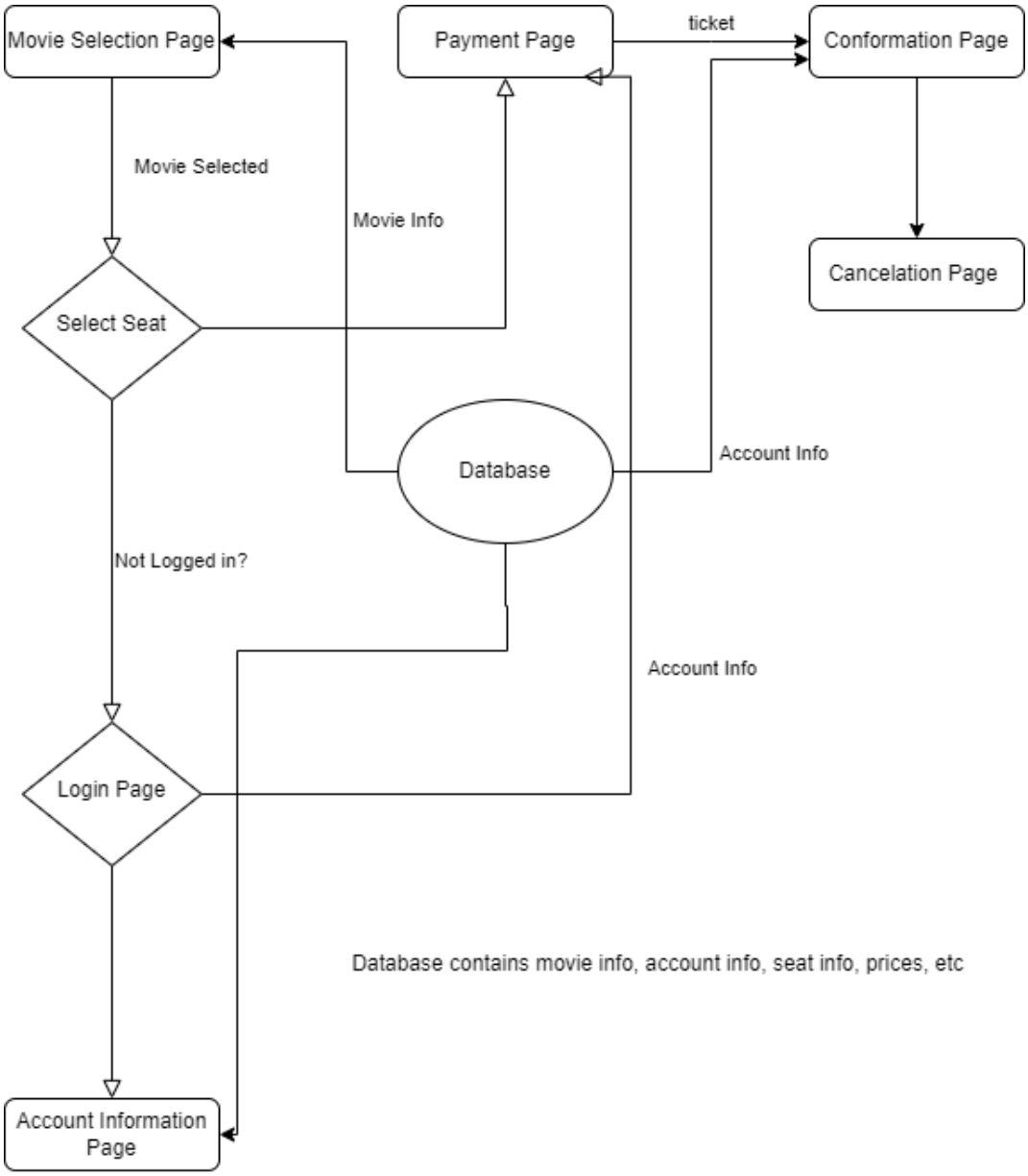
4. The Account Info class manages the login credentials of the users. It contains variables that store the username, password, and email address of the account holder: username : String is the username of the account holder. password : String is the password associated with the account. email : String is the email address linked to the account.

5. The Ticket class manages the details related to a movie ticket. This includes the ticket price and a map representing the availability of seats. The class includes a function to buy a ticket: price( int) is the price of the ticket. seats : Map<bool> is a map indicating the availability of seats. buyTicket() : void is the function that facilitates the purchase of a ticket.

6. The UserInfo class manages user-related details and operations. It includes variables to store the user's reward points and discount status. Additionally, it provides several functions for user operations: rewardPoints : int is the reward points accumulated by the user. discountStatus : bool is the discount status of the user. updateLoyalty(int, int) : int is the function that updates the user's loyalty points. viewMovies() : void allows the user to view available movies. pickMovies(Movie) : void  enables the user to select a movie. cancelTicket(Ticket) : void allows the user to cancel a ticket.

7. The Make Payment class handles the payment process for purchasing tickets. It includes variables to store the transaction ID and the ticket associated with the transaction. The class also provides functions to confirm the purchase and update seat availability: transactionID : int: is the ID of the transaction. myTicket : Ticket is the ticket associated with the transaction. confirmPurchase(ticket) : int to confirm the purchase of a ticket and returns a transaction ID. updateSeats(ticket) : void to update the seat availability after a purchase.

## 4.3 Software Architecture Diagram




## 4.4 Software Architecture Diagram Descriptions

The above diagram shows the components of the movie theater ticketing system and how they interact with each other. The user interacts with the movie theater ticketing system through a webpage and the webpage changes based on user input and user data.

1. **User Interface:**
- Users interact with the system through a web application.
- The interface allows users to search for movies, book tickets, and manage their accounts.

2. **Backend Services:**
- Handles business logic, data processing, and interactions between the user interface and databases.
- Includes services for user authentication, movie management, booking management, and payment processing.

3. **Database:**
- Stores user information, movie details, booking records, and payment transactions.
- Ensures data integrity and supports query operations for retrieving and updating information.

4. **Payment Gateway:**
- Integrates with external payment services (e.g., PayPal) to handle financial transactions securely.
- Ensures that payment data is processed and stored in compliance with security standards.

**Data Flow:**
- Users send requests via the web interface.
- The backend processes requests, interacts with the database, and integrates with the payment gateway as needed.
- Responses are sent back to the user interface for display.

**Architectural Patterns:**
- The system follows a Model-View-Controller (MVC) pattern to separate concerns and facilitate maintenance and scalability.
- Uses RESTful APIs for communication between the frontend and backend.

## 4.5 Development Plan

Nhu Vo:
- Create Movie Selection Page (~2 weeks)
- Retrieve movie info from database (~1 week)
- Send user to login page or payment page (~1 week)

Sushil Rawtani:
- Create Payment Page (~2 weeks)
- Retrieve account and movie info (~1 week)
- Send user to confirmation page (~1 week)

Adrian Arguelles:

- Create Database (~2 weeks)
- Create Login Page (~1 week)
- Send user to account information page and payment page (~1 week)

# 5. Test Plan

This section contains our test plan for verification and validation. This test plan will include methods, different types of tests, and ten test cases to be executed to ensure the system meets its functional and nonfunctional requirements.

## 5.1 Introduction to Test Plan

The test plan aims to ensure that all features and functionalities of the Movie Theater Ticketing System are thoroughly tested. The tests will cover various aspects of the system, including unit tests, functional tests, and system tests. Each test will be designed to validate specific features and to identify any potential issues or failures.

## 5.2 Test Methodology

We will use a combination of manual and automated testing methods. The testing will be conducted in three phases:

- Unit Testing: Testing individual components or functions to ensure they work as intended.
- Functional Testing: Testing the system's functionality against the requirements.
- System Testing: Testing the complete and integrated system to evaluate the system's compliance with the specified requirements.

Each test will be detailed with the target feature, the test sets, and expected outcomes. The test cases will cover various scenarios, including normal operations, edge cases, and failure modes

The test strategy includes both the verification and validation processes:

1. Verification includes:  Analysis of requirements, design, and code
2. Validation includes: Testing including the three granularities: unit, functional, and system.

## 5.3 Test Cases Samples

## *Test Set #1(Functional): Registration and Login*

**Test Case 1: Registration for a new user**

**Description:** Verify that a new user can make an account on the website.

**Prerequisites:**
1) Have access to the registration page
2) Test data for a new user, including valid email, password, and username

**Test Steps:**
1) Navigate to the registration page
2) Enter email, password, and username
3) Click Create Account Button
4) Send an email to the email provided, and provide a link to confirm the email address
5) Redirect to email confirmation page from the link provided.

**Expected Result:**
1) The user receives an email that redirects them to the confirmation page
2) The user is automatically logged in with their newly created account'

**Test Case 2: Logging into an existing account.**

**Description**: Verify that an existing user can login to the website.

**Prerequisites**:
1) Have access to the login page
2) An existing user account with valid credentials (username and password).

**Test Steps:**
1) Navigate to the movie ticketing website's login page.
2) Enter the valid username in the username field and valid password in the password field.
3) Click the Login button.
4) Redirect to the home page, with a small window on the side saying successfully logged in.

**Expected Result:**
1) The user is successfully logged in.
2) The user is redirected to the home page.

## *Test Set #2(Functional): Managing Bookings*

**Test Case 3: Book a Ticket**

**Description**: Verify a user can book a ticket

**Prerequisites**:

1) Have access to the movie selection page
2) Must be logged into the website.

**Test Steps:**

1) Navigate to the movie ticketing website's movie selection page
2) Select any movie available.
3) Click on available showtimes for movie
4) Select a showtime
5) Choose seats
6) Purchase Ticket
7) View ticket details

**Expected Result:**

1) The ticket is successfully purchased
2) The user receives a confirmation email with booking details.

**Test Case 4: Cancel a Booking**

**Description:** Verify that a user can cancel a previously booked ticket.

**Prerequisites:**

1) The user must be logged in.
2) The user must have an existing booking.

**Test Steps:**

1) Navigate to the user's booking page.
2) Select the booking to cancel.
3) Click the "Cancel Booking" button.
4) Confirm the cancellation.

**Expected Result:**

1) The booking was successfully canceled.
2) The user receives a confirmation email of the cancellation.
3) The seat is marked as available.

**Test Case 5: View Booking History**

**Description:** Verify that a user can view their booking history.

**Prerequisites:** The user must be logged in.

**Test Steps:**
1) Navigate to the user's account page.
2) Click on the "Booking History" link.
3) View the list of past bookings.

**Expected Result:** The user sees a list of their past bookings with details such as movie name, showtime, and seat number

### *Test Set #3 (System): Payment Processing*

**Test Case 6: Successful Payment Processing**

**Description:** Verify that the payment gateway processes a payment successfully.

**Prerequisites:**
1) User is on the payment page.
2) Valid payment details are available.

**Test Steps:**
1) Enter valid payment details.
2) Click the "Pay Now" button.
3) Confirm the payment.

**Expected Result:**
1) Payment is processed successfully.
2) User receives a payment confirmation message and email.

**Test Case 7: Payment Failure Handling**

**Description:** Verify that the system handles payment failures appropriately.

**Prerequisites:**
1) User is on the payment page.
2) Invalid payment details (e.g., expired cards) are used.

**Test Steps:**
1) Enter invalid payment details.
2) Click the "Pay Now" button.

**Expected Result:**

1) Payment fails.
2) The user receives an error message indicating the reason for the failure.
3) The user can retry the payment with different details.

## *Test Set #4 (System): Admin Functionality*

**<u>Test Case 8:</u> Adding a movie**

**Description:** Make sure an admin can add a new movie to the system.

**Prerequisites:**

1) Admin's account has admin privileges.
2) Admin can access the admin control panel on the website.

**Test Steps:**

1) Navigate to the admin control panel.
2) Select the option to add a new movie.
3) Enter movie details.
4) Click the "Add" button.

**Expected Result:**

1) The new movie is added to the database.
2) The movie appears in the list of available movies on the movie selection page.

**<u>Test Case 9:</u> Deleting a movie**

**Description:** Make sure an admin can delete a movie to the system.

**Prerequisites:**

1) Admin's account has admin privileges.
2) Admin can access the admin control panel on the website.

**Test Steps:**

1) Navigate to the admin control panel.
2) Select the movie that you want to delete.
3) Click the "Manage" button next to the movie.
4) Click the "Delete" button on the movie's management panel.
5) Confirm the deletion.

**Expected Result:**

1) The selected movie is successfully removed from the database.
2) The movie appears in the list of available movies on the movie selection page.

### *Test Set #5 (Unit): User Authentication*

**Test Case 10: Valid Email**

**Description:** Ensure that the email format is valid.

**Prerequisites:** User registration/login has a field for email.

**Test Steps:**

1) Input an invalid email or invalid email format.
2) Check the if valid/invalid.

**Expected Result:** The system rejects invalid emails and accepts valid emails.

**Test Case 11: Valid Password**

**Description:** Ensure that the password is valid (meets all password requirements, no illegal characters)

**Prerequisites:** User registration/login has a field for password.

**Test Steps:**

1) Input an invalid password.
2) Check if it is valid/invalid.

**Expected Result:** The system rejects the invalid password and asks the user to enter another password.

## *Test Set #6 (Unit): Backend Functionality*

**Test Case 12: Retrieve User Information**

**Description:** Verify that the backend correctly retrieves user information from the database.

**Prerequisites:** User account exists in the database.

**Test Steps:**
1) Send a request to the API endpoint to retrieve user information.
2) Include the user ID or username as a parameter.
3) Validate the response received from the backend.

**Expected Result:**

1) The response contains the correct user information, such as username, email, and booking history.
2) Data integrity is maintained and matches the records in the database.

**Test case 13: Update Movie Details**

**Description:** Verify that the backend allows administrators to update movie details correctly.

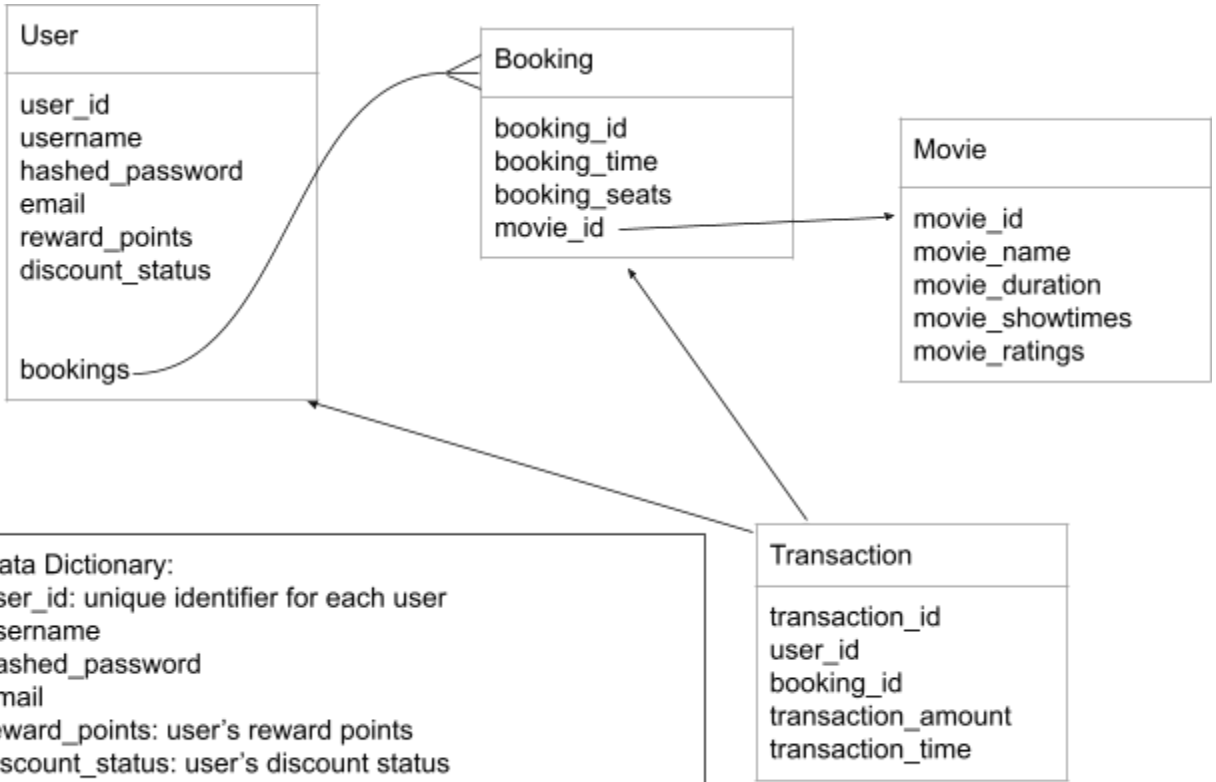
**Prerequisites:**
1) Admin account with appropriate privileges.
2) Existing movie records in the database.

**Test Steps:**
1) Log in as an admin.
2) Send a request to the API endpoint to update movie details.
3) Include the movie ID and updated details (e.g., new showtime, updated rating) as parameters.
4) Validate the response received from the backend.
5) Retrieve the updated movie details from the database to confirm the changes.

**Expected Result:**
1) The movie details are updated successfully in the database.
2) The updated information is accurately reflected when queried.

Spreadsheet for Test Cases

# 6. Data Management Plan




Data Dictionary:
user_id: unique identifier for each user
username
hashed_password
email
reward_points: user's reward points
discount_status: user's discount status
bookings: list of bookings that user has

booking_id: unique identifier for each booking
booking_time: time that showing starts
booking_seats: seats in theater that were booked

movie_id: unique identifier for each movie
movie_name: name of movie
movie_duration: length of movie
movie_showtimes: available times that movie is showing
movie_ratings: ratings for movie

transaction_id: unique identifier for each transaction
transaction_amount: amount that user payed for booking
transaction_time: time that transaction occurred

## 6.1 Database Choices:

- Use SQL database for structured data storage and complex queries.

- Single database for users, movies, bookings, and payments to simplicity in management and maintains data integrity.

## 6.2 Design Decisions:

- **User Table**: Stores user profiles, login credentials (hashed passwords), and personal information.
- **Movie Table:** Stores movie details, showtimes, ratings, and other metadata.
- **Booking Table:** Stores booking information, seat selections, and related details.
- **Payment Table:** Stores payment transactions, status, and history.

## 6.3 Data Storage and Retrieval:

- Normalization: Normalize the database to avoid redundancy and ensure data integrity.
- Indexing: Use indexing for faster search and retrieval,
- Partitioning: Partition the database to improve performance and manageability.

## 6.4 Data Security:

- Encrypt sensitive data such as payment information.
- Use secure communication protocols (HTTPS) for data transmission.
- Implement access controls and regular audits.

## 6.5 Data Integrity and Backup:

- Database transactions will be ACID Compliant in order to maintain data integrity
- Full Backups of the database will happen every night.
- Maintain uniqueness for things like email, username, etc
- Redundancy: There will be a copy of the database on a different server.
- There will be tools to monitor the database to ensure everything is in check, and an alert system to notify admins of issues that arise.
-

## 6.6 Scalability and Performance:

- Perform database maintenance tasks regularly and clean up unused data to maintain performance.
- Vertical scalability: Increase the resources of the existing database(ie; cpu, ram) for better performance.
- Horizontal scalability: Implement database sharding to distribute data across multiple servers

## 6.7 Trade-offs and Alternatives:

Alternatives:

- NoSQL
- Multiple databases
- Single Table

NoSQL:

SQL uses relational databases, whereas NoSQL uses non-relational databases. One reason we chose SQL over NoSQL is to have a database that relates the tables for users, movies, bookings, and payments. SQL is also best suited for structured data, whereas NoSQL is best suited for unstructured data. While new fields may have to be added to the database in the future, the overall structure of the Movie Theater Ticketing System database is unlikely to change, hence SQL best suits our needs.

Multiple Databases:

The main reason that we chose to use a single database rather than multiple databases is that it is harder to access two units of data that are related to each other if they are on separate databases. For example, if a system administrator is granted access to a user's data, and wants to see the user's booking, the admin will also need to be granted access to the bookings database if it is separate from the user database. Splitting the data into multiple databases also makes backups more complicated because the databases have to be synchronized. If one database is backed up before another, the other database can become outdated.

Single Table:

Although a database with a single table may run faster than a database with multiple tables, it is not worth the tradeoff because it makes maintaining data integrity much more difficult. For example, since a user can have multiple bookings, each booking needs to be in the same format. Also, multiple people can have a booking for the same movie which created redundancy. By separating the data into different tables and using unique IDs to relate the data, the consistency of the data is easier to maintain and redundant data is eliminated.

## GitHub Accounts

Nhu Vo: https://github.com/nhuvo931/CS250
Sushil Rawtani: https://github.com/srawtani6872/Movie-Theater-CS-250
Adrian Arguelles: https://github.com/aarguelles207/cs250-movie-theater-system/