Theater Ticketing Software System

Software Design Specification

Version: 3.101.1

Oct 5, 2023

Group #1

Ryan, Nardos, Rhenz

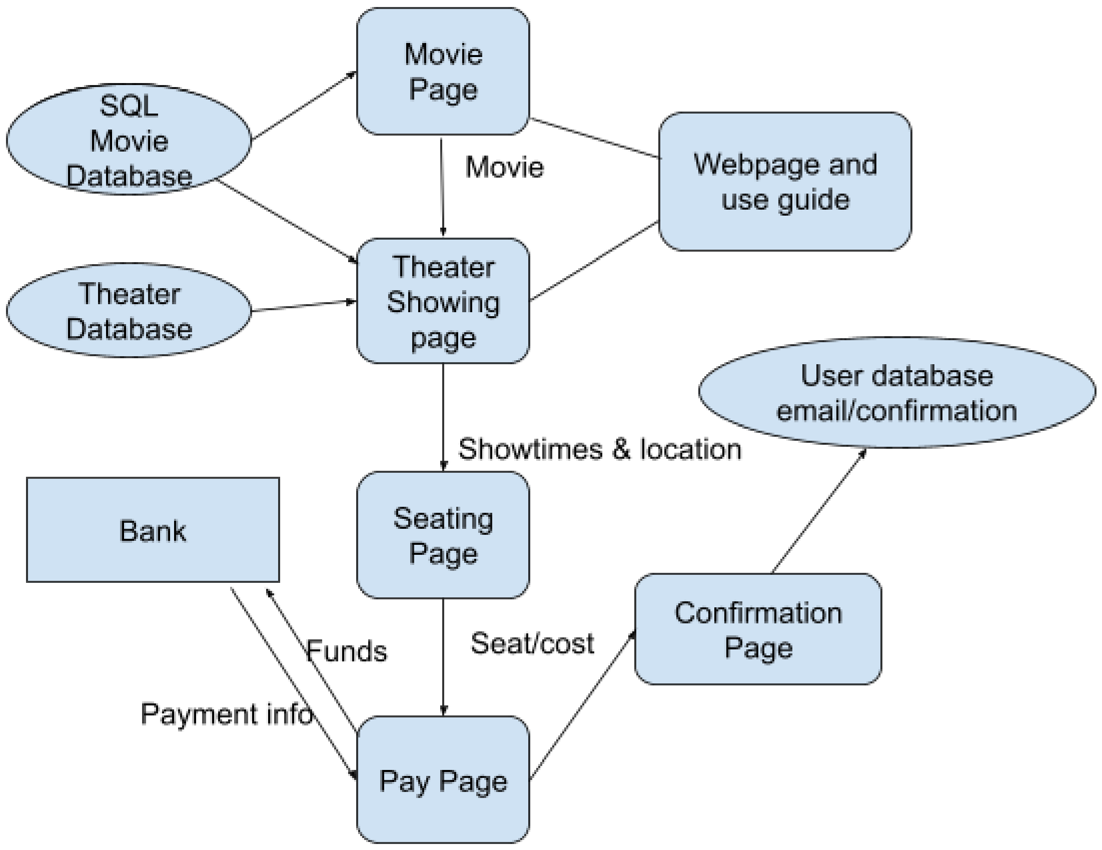
Prepared for

CS 250 - Introduction to Software Systems

Instructor: Gus Hanna, Ph.D.

Fall 2023

Movie Theater Software Architecture Diagram



Theater Ticketing Software System Architecture Diagram Description

We have several modules and databases that interact with each other depending on functionality.

Modules

- Webpage and guide - welcomes and introduces customers to movies and ticket purchase availability.
- Movie page - displays movie by genre that draws from sql database
- Theater page - displays movie showtimes at specific theater that feeds from theater database

- Seat selection - displays seats at specific movie showtimes and theater
- Pay page - displays payment information required to book selected seats, confirms funds are available from the bank and proceeds to charge payment method.
- Confirmations page - displays confirmation of all the selections made, receipt of funds and sends it to email.

Time

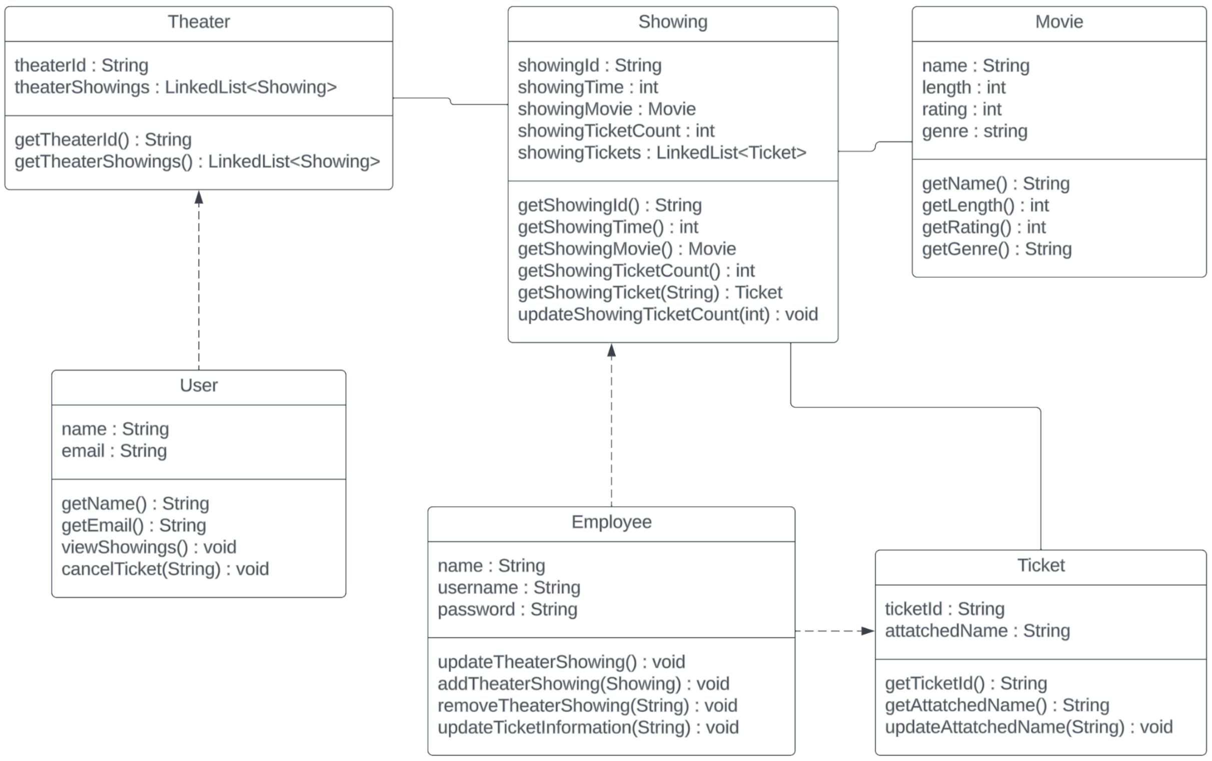
- The timeline allocated for this project is 3 months. There is flexibility to extend the timeline in accommodation of project needs.
- Estimately we will need the responsibility of Team 1 connecting the back end databases to and creating the modules deadline of 6 to 8 weeks.
- Team 2 will be doing the front end functionality of the theater database such as the confirmation page, Pay page to function correctly which will be given an estimate of 4 to 5 weeks deadline.
- Team 3 will be given UI design to make sure it is functional for any intuitive interface and visual experience for the users which will be given a 3 to 4 weeks deadline.

Teams

We will have 3 team member working on different aspects of the project

- Team 1 will have the responsibility of connecting the back end databases
- Team 2 - will work on the front end functionality.
- Team 3 - will work on the UI design to make sure it is user friendly.

UML Class Diagram of System Operation



Class Descriptions

Employee

The employee class contains 3 attributes which represent the login information of the employee. The class interacts with the *showing* class and *ticket* class. They are given 3 operations relating to the *showing* class and 1 operation relating to the *ticket* class. The employee and update theater showings, add new theater showings, and update theater showings. The updateTheaterShowing() operation takes in a *showing* object as a parameter which is constructed within the operation. The removeTheaterShowing operation and updateTicketInformation operation take in a string parameter which represents the ID of the showing, from there the showing can be updated.

- Attributes
  1. Username: username of employee account
  2. Password: password of employee account
  3. Name: proper name attached to employee account
- Operations
  1. updateShowing: allows employee to update showing info such as showtime and showing moving

2. addShowing: allows employee to add a new showing to a theater
3. removeShowing: allows employee to remove existing showing from a theater
4. updateTicketInformation: allows employee to adjust ticket information

User

The *User* class seems to have two attributes which mainly represents the email of the user's account including their personal information like first and last name. They are given 2 operations which is getEmail which mainly retrieves the User's email address. And the getName operation retrieves the name of the User and a view object which allows the User to view all of the showings that are on in the ticketing system.

- Attributes
    1. Email : email of the user's account
    2. Name : first and last name of the user
- Operations
    1. getEmail: retrieve the Users email
    2. getName: retrieve the name of the Users
    3. viewShowings: allows Users to view all showings

Theater

The Theater class contains two attributes which represent the theater id and theater showing. It interacts with *showing* class and *user* class. It has two operations that retrieve theater Id and showings list for the theater. The getThaterID takes in a string parameter.

- Attributes
    1. theaterId: unique identifier of the theater represented as a 4 character string
    2. theaterShowings: a linked list comprised of all showings at the specific theater
- Operations
    1. getTheaterId: retrieves theater ID
    2. getTheaterShowings: retrieves list of all showing for the theater

Showing

The *showing* class contains 5 attributes that represent the unique information regarding the showing along with ticket information. It contains 5 get operations and 1 update operations. The getShowingId, getShowingTime, getShowingMovie, and getShowingTicketCount directly retrieves the related attribute information. The getShowingTicket takes a string as a parameter which represents the unique ID of the strings that is desired for retrieval, it then returns that ticket. The updateShowingTicketCount takes in an int as a parameter which represents the adjustment to the showingTicketCount attribute (i.e. -10 decrements the count by 10)

- Attributes
    1. showingID: allows to retrieve the unique identifier for time show movies
    2. showingTime: allows to retrieve the time movies of the shows
    3. showingMovie: contains the movie attached to the showing
    4. showingTicketCount: contains the number of remaining tickets for the showing
    5. showingTickets: contains a list of all tickets for the showing
- Operations
    1. getShowingId: retrieves showing ID
    2. getShowingTime: retrieves showing time
    3. getShowingMovie: retrieves movie played at the showing
    4. getShowingTicketCount: retrieves count of tickets remaining
    5. getShowingTicket: retrieves a specific ticket within it's ticket list
    6. updateShowingTicketCount: updates the number of tickets

Movie

The Movie class has four attributes that display information about the movies. It interacts with the *Showings* class. It contains four operation that retrieves the name, genre, length and rating of movie

- Attributes
    1. Name: shows movie Name
    2. Genre: genre of move
    3. Length: length of move in minutes
    4. Rating: rating of movie
- Operations
    1. getName: retrieves name of movie
    2. getGenre: retrieves genre of movie
    3. getRating: retrieves rating of movie
    4. getLength: retrieves length of movie

Ticket

The *Ticket* class has two attributes which mainly represents a ticketID which identifies the person with the ticket and an attachedName that represents the user who purchased the ticket. And the class has 2 similar Operations which has the getTicketId() which is able to retrieve the ticket ID including the getAttachedName() which would be able to retrieve the name attached to the ticketing. And the updateAttachedName() Operation would show the ticketing of the attached name and update it.

1. ticketId: unique identifier of the ticket represented as a 16 character string
2. attachedName: name of user who purchased the ticket

- Operations
    1. getTicketId: retrieves ticket ID
    2. getAttachedName: retrieves name attached to ticket

3. updateAttachedName: updates the attached name to ticket

Verification Test Plan for Theatre Ticketing Software System

This document generally describes our verification test plan by taking advantage of test cases to get the best possible outcome for our software. We are using black box structures for relevant inputs by employing all granularities(units, function, system).

Features to Test and methods of testing

1. The feature employee login enables an employee to login to the theater management page. The test will be to verify that when an employee enters a valid username, password the employee and clicks login the user will reach the theater management page. The login button will call on the employee_login method.

2. The feature user search bar enables a user to search for a desired movie. The test will be to verify that when a user enters a movie title into the search bar the desired movie/movies is returned. This is done through utilizing the search bar

3. The feature add theater showing enables the employee to place the desired showing time to the user view list. The test will verify when an employee adds a theater showing time to the list that it is viewed on the updated list. This is done by calling the addTheaterShowing() method.

4. The feature view showings enables a user to see the available showings. The test will verify when a user presses the view showing button the available showing are on the screen. This is done by calling the viewShowings() method.

5. The feature of getName enables the user to search for a specific movie. The test will verify when a user enters a movie name that specific movie name will be viewed on the screen. This is done by calling the getName() methods.
6. The feature Ticket ID is a ticket identifier number that is issued on the ticket for identification and tracking purposes. The test will verify that a ticket has an ID. This is done by calling the getTicketId() method.

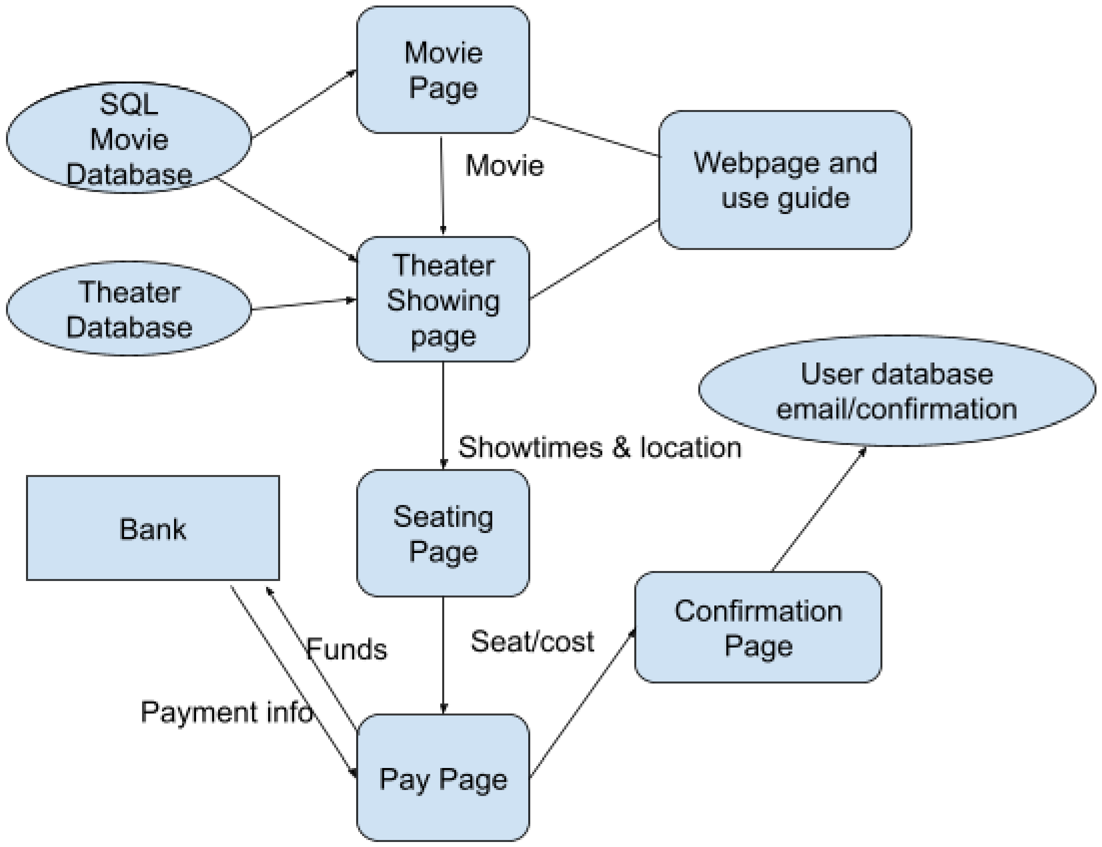
7. The feature Theater showing enables the user to view showings for specific theaters. The test will verify that when a user presses the theater showing button, the showings are returned. This is done by calling the getTheaterShowing() method.
8. The feature ticket count enables employees to track the amount of tickets issued/remaining for organization purposes. The test will verify that the ticket count is returned. This is done by calling on the getCount() method.

9. The feature email confirmation enables the user to get a confirmation of ticket purchased for desired movie showing via email. The test will verify that a confirmation email is

received for the purchased ticket. This is done by creating a tester email and purchasing a ticket.

10. The feature showings time enables users to see the times allotted for movie showings. The test will verify the showings time is returned when requested. This is done by calling the getShowing() method.

Data Management



General Description

This section of the specification will go over the organization of the software associated databases, the data management strategies deployed to handle the data, and the reasonings and justifications for the chosen management strategies.

Organization

Each software application will have access to three databases, the Movie Database, the Theater Database, and the User Database. The Movie Database and User Database are offsite, meaning they will not have any correspondence to a specific theater and instead, operate as accessible databases to all theaters. However each theater will have its own database, represented here as the Theater Database.

Movie Database

The Movie Database will contain all movie relevant information.
This information (with type specification) will include:

- Title { VARCHAR (255) }
- Genre { VARCHAR (255) }
- Length { TINYINT [unsigned] }
- Rating { TINYINT [unsigned] }

As well as additional supporting information which will include:

- ID { SMALLINT [unsigned] }
- Date Added { MEDIUMINT [unsigned] }

The organization of the data within the database will look like the following example:

ID	DATE_ADDED	TITLE	GENRE	LENGTH	RATING
12	230721	Barbie	Comedy,Fantasy	114	88

User Database

The User Database will contain all user relevant information.
This information (with type specification) will include:

- Email { VARCHAR (255) }
- Password { VARCHAR (255) }
- Name { VARCHAR (255) }

As well as additional supporting information which will include:

- ID { INT [unsigned] }
- Date Added { MEDIUMINT [unsigned] }

The organization of the data within the database will look like the following example:

ID	DATE_ADDED	EMAIL	PASSWORD	NAME
478	230916	janeDoe@gmail.com	04bsc91bfkq3n2np	Jane

Theater Database

Each Theater Database will contain all theater relevant information separated into two partitions.
The information in Partition 1 (with type specification) will include:

- TicketID { SMALLINT [unsigned] }
- AttachedName { VARCHAR (255) }

As well as additional supporting information which will include:

- Date Added { MEDIUMINT [unsigned] }
- Attached ID { INT [unsigned] }

The organization of the data within Partition 1 will look like the following example:

TICKET_ID	DATE_ADDED	ATTATCHED_NAME	ATTATCHED_ID
928	231014	Jane	478

The information in Partition 2 (with type specification) will include:

- Username {VARCHAR (255) }
- Password {VARCHAR (255) }
- Name {VARCHAR (255) }

As well as additional supporting information which will include:

- ID { INT [unsigned] }
- Date Added { MEDIUMINT [unsigned] }

The organization of the data within Partition 2 will look like the following example:

ID	DATE_ADDED	USERNAME	PASSWORD	NAME
37	220209	mRobert	jd93hlgak39d3kf0	Robert

Reasoning, Justification, and Discussion

The decision to provide two off-site databases accessible by all theaters and a single database for each theater was to minimize complexity and ensure unwanted accessibility. Separating the Movie Database from the User Database was made to simplify read and write queries to each database and enable individual database optimization. The same can be said for the Theater Databases but the primary reasoning for each theater to have one is to prevent cross-talk. If the Theater database was an off-site database that all theaters would query into, then it opens up the possibility for one theater to modify the data of another theater, causing potential errors. It would also increase code complexity to write safe measures that ensure such faults don't occur.

Some drawbacks of the database organization involve outages and maintenance. The two off-site databases present a potential hazard in the case of an outage or required maintenance. Since all theaters rely on these two databases, if either one goes offline due to the reasons listed previously, then all theaters will lose all functionality in that specific field. A solution to this problem would involve adding a duplicate database for each one, where one represents the main and the other represents the sub. In the event the main is put offline, then the sub will take the main position until the original main returns online.

The addition of supporting information which doesn't relate to the respective classes for which the data is attached to, such as DATE_ADDED are used for general database optimization to prevent extensive query operations.

Without being too verbose, the reasoning for the specific data type of each stored value was minimizing storage usage and accounting for enough overhead. An example is the TITLE field under the Movie Database which is represented as a VARCHAR(255), essentially a 256 character string. It is expected that no movie title will realistically exceed 255 characters in length, let alone reach anywhere near this value, but it is set to 255 characters to provide enough overhead. Another example would be the DATE_ADDED field under each database which is represented as a MEDIUMINT[unsigned], essentially a 24 bit integer that ranges from 0 to 16777215. The date is stored as an integer representation which can be parsed by the software. The data type is unsigned since negative numbers shouldn't represent a parsable date and the size allows for dates that will very likely never be reached.