Theater Ticketing System

Software Design Specification

Version 2.1

05 October 2023

Group 2
Afnan Algharbi, Jasmine Rasmussen,
Everett Richards

Prepared for
CS 250- Introduction to Software Systems
Instructor: Gus Hanna, Ph.D.
Fall 2023

Revision History

Date	Description	Author	Comments
09/22/23	Version 1.0	Group 2	First Draft
09/28/23	Version 1.1	Group 2	Various revisions
10/05/23	Version 2.0	Group 2	Final draft for Part 2
10/16/23	Version 2.1	Group 2	Added references to test docs

Document Approval

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

Signature	Printed Name	Title	Date
	Afnan Algharbi	Software Engineer	10/05/2023
	Jasmine Rasmussen	Software Engineer	10/05/2023
	Everett Richards	Software Engineer	10/05/2023
	Dr. Gus Hanna	Instructor, CS 250	10/05/2023

Table of Contents

1. System Description

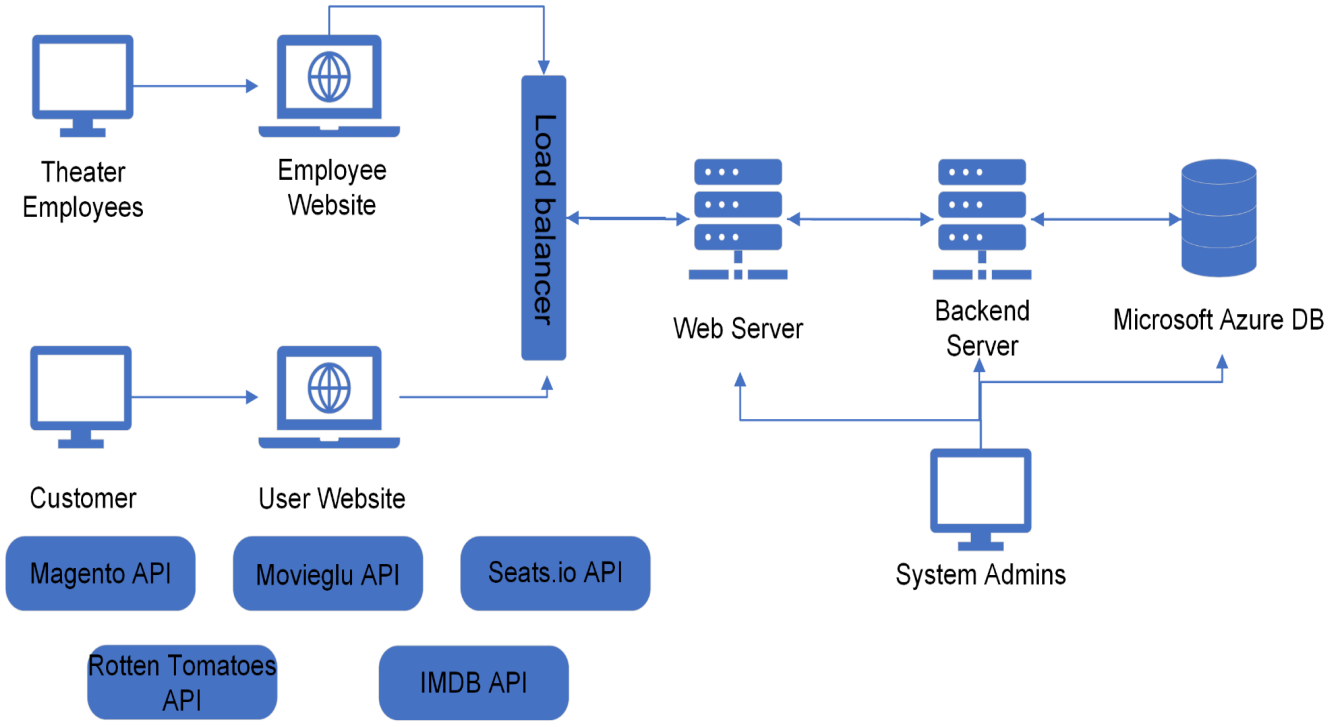
1.1 Brief overview of system

The following Modules detail the Software Design Specification for the ticketing system and offer a meticulous outline for the architectural component of the project to prepare for the implementation phase. That includes analyzing the relationships between the system's components and how the data flows between them to the safe processing of tickets, purchases, and other measures of user interactions.

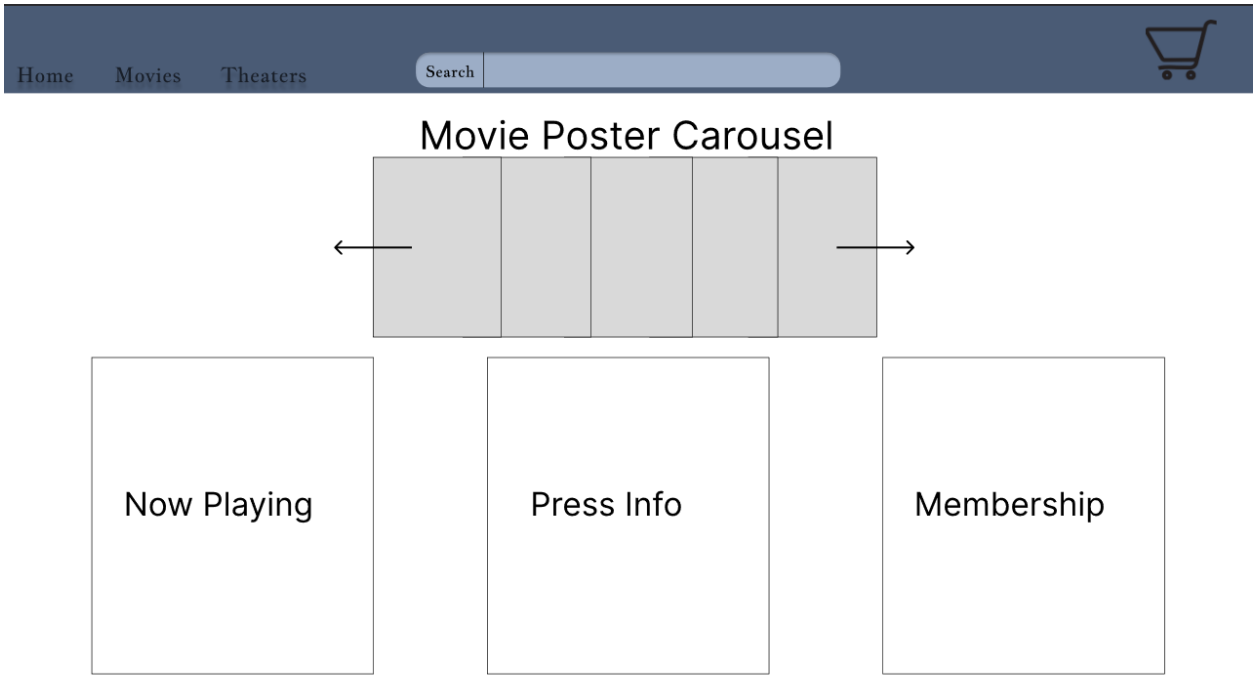
2. Software Architecture Overview

To better understand the functionality of the ticketing system, presenting a visual diagram of the system's architecture and design is crucial. This module delves into a comprehensive review of these various diagrams.

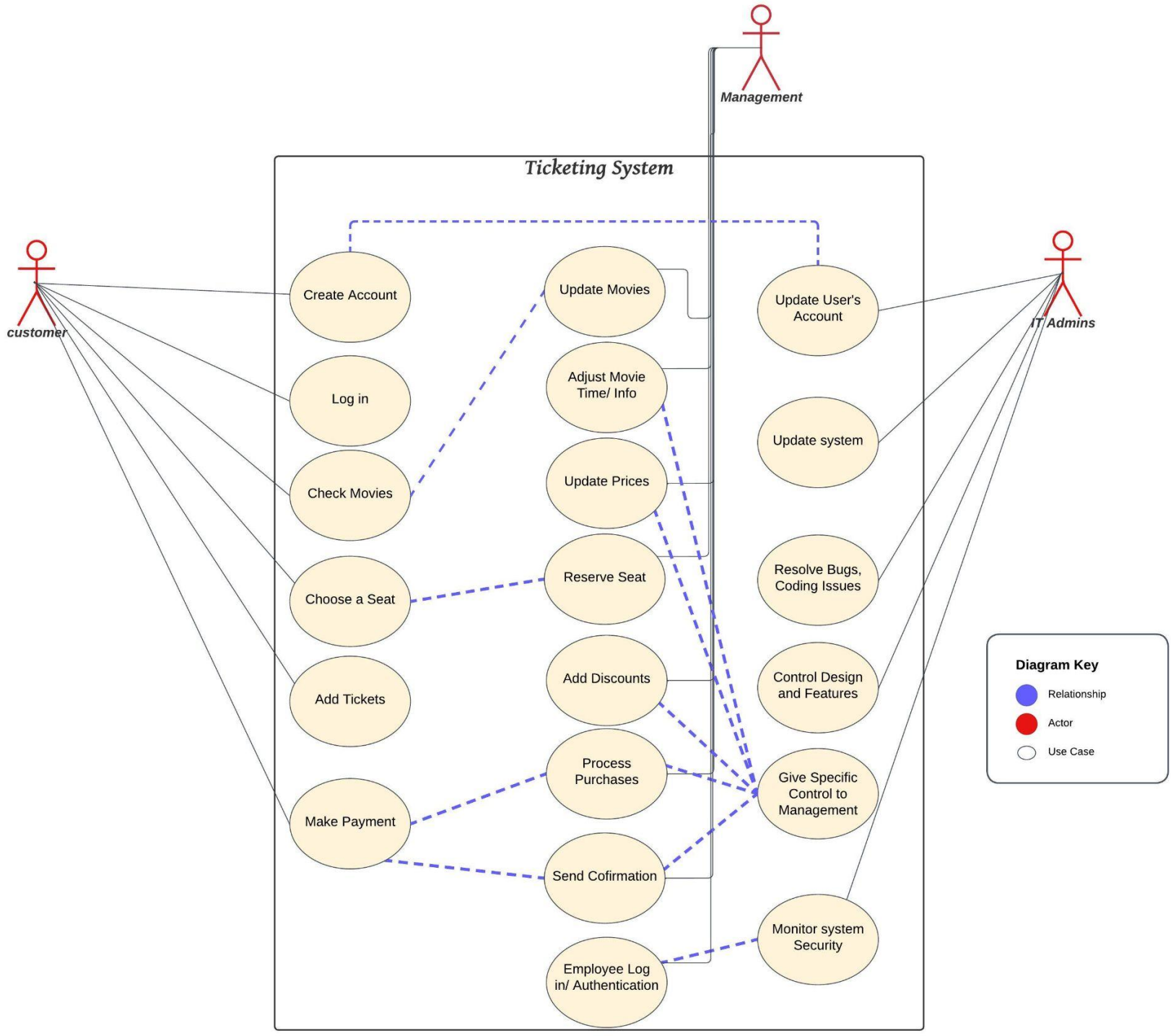
2.1 Architectural diagram of all major components



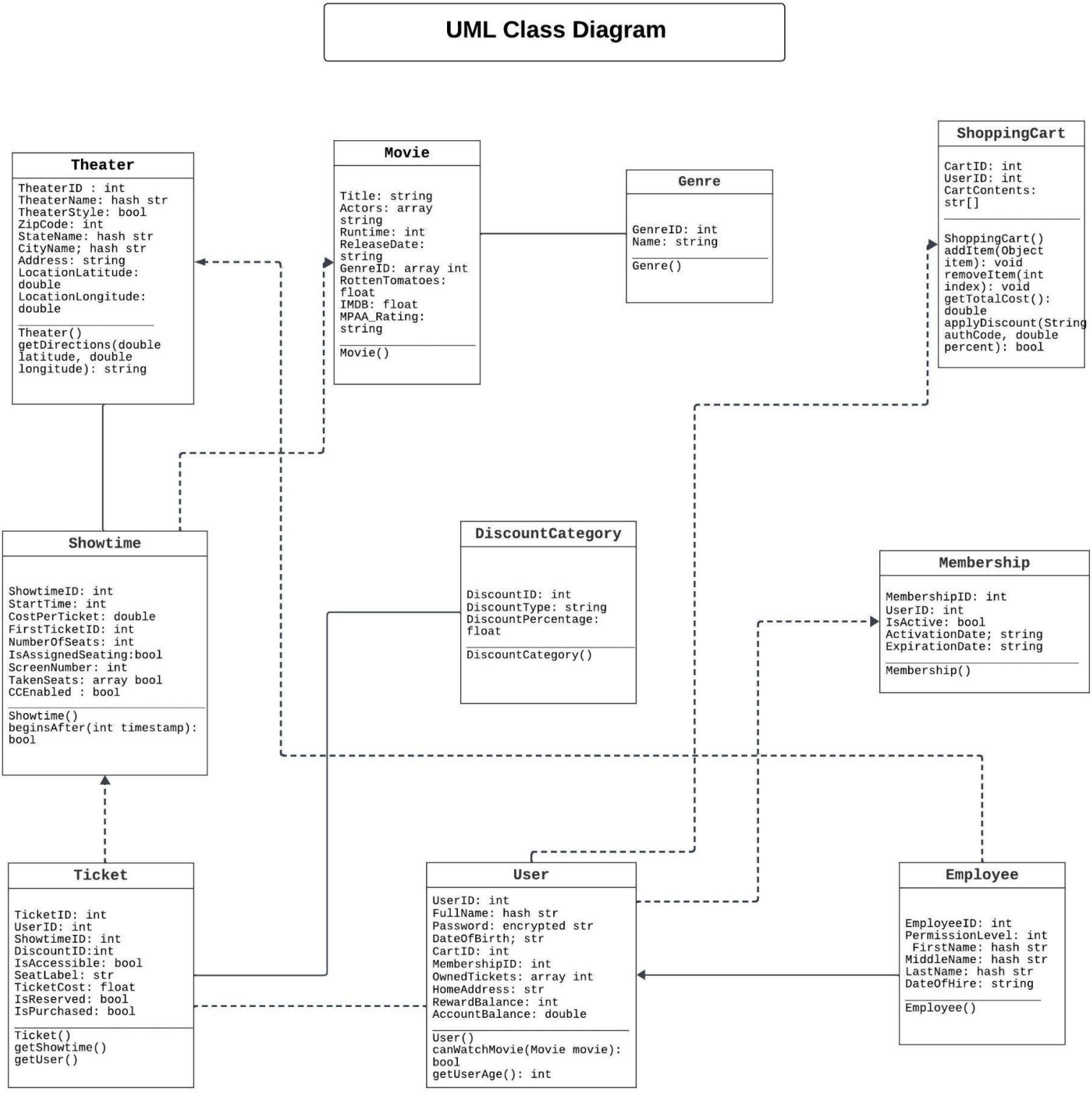
2.1.1 UI Mockup



2.1.2 Use Cases

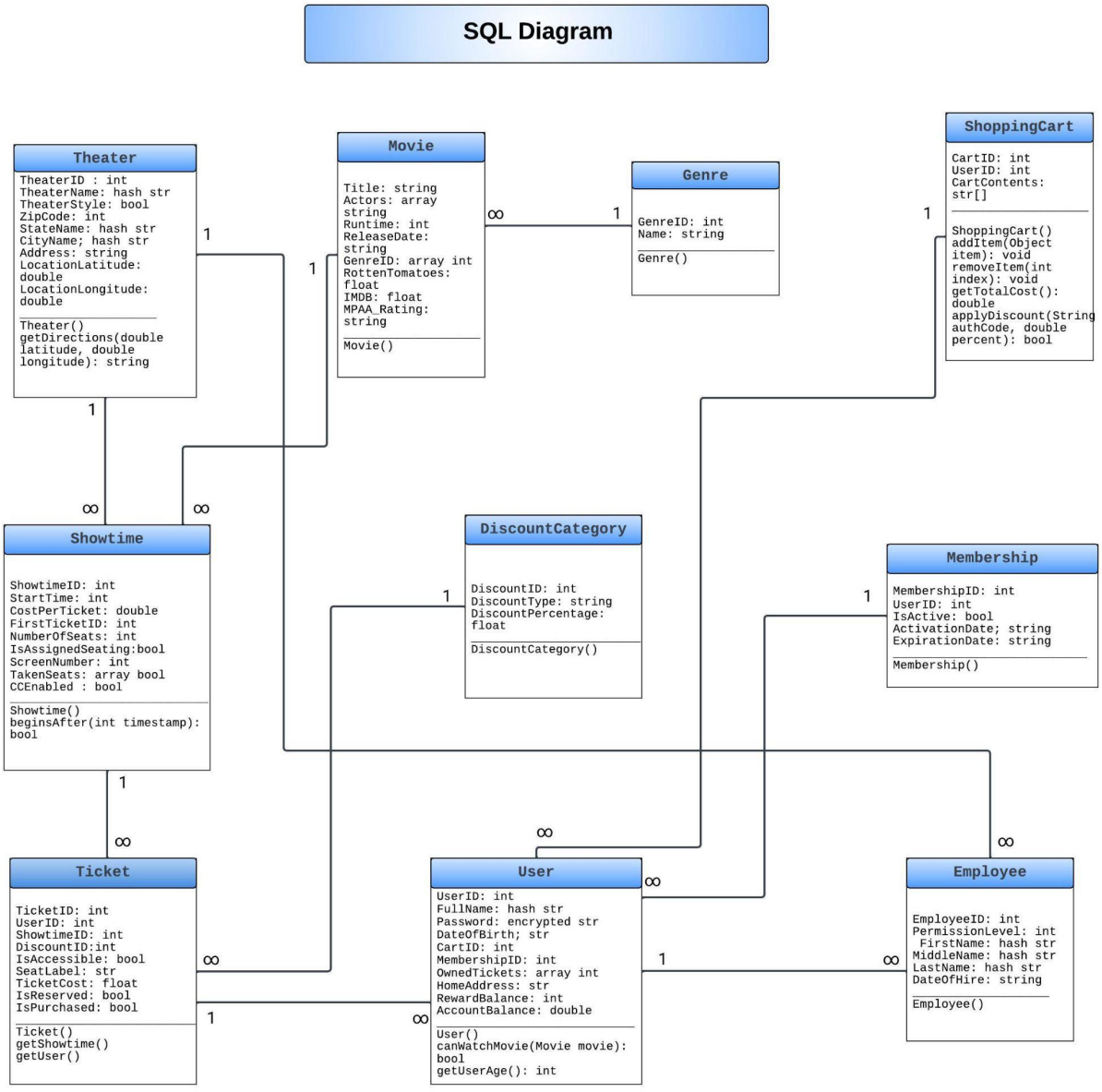


2.2 UML Class Diagram



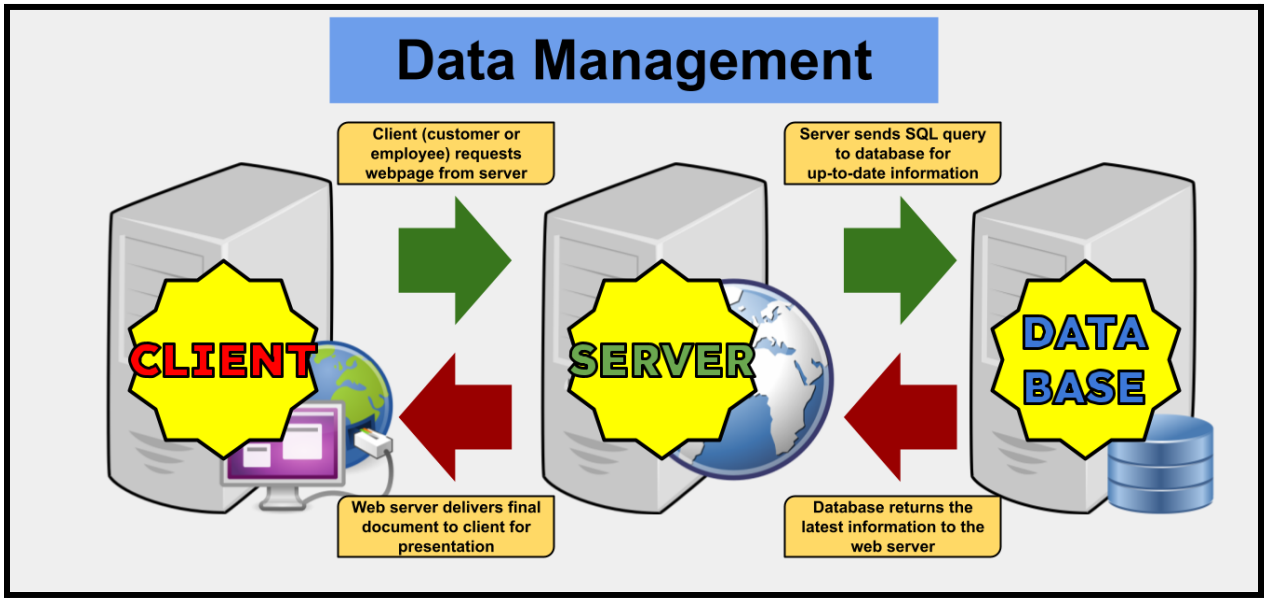
Architecture Design

3.1 Software Architecture Diagram



3.2 Data Management

3.2.1 Data Management Diagram



3.2.2 Data Management Strategy

Data will be managed using a distinct array of web servers and databases. Databases will be outsourced to Microsoft Azure hosting, which will ensure the secure and consistent availability of our data. The web servers will be hosted independently in company-owned server facilities.

When a user (either a customer or employee) makes a web request via HyperText Transfer Protocol (HTTP), our server will process the request and determine the proper course of action. If data is needed from the database, the web server will request that data using a SQL query to the database. Then, the web server will package the response and send it to the user's client to be rendered. The database can only be queried by verified company web servers, using credentials supplied by the user in the case of sensitive information such as account details and payment information.

The web server can be queried by essentially any internet-capable device, typically through a web browser such as Google Chrome or Microsoft Edge. We officially recommend that users access the website through Google Chrome for the best experience.

4. Classes, Objects, & Attributes

Described below are the classes, objects, and attributes that will be needed for our Theater Ticketing System. These objects will exist within our database, as well as within our backend and frontend processing as Java classes and JavaScript data structures.

The database will be structured using SQL and will be updated by authorized users, such as system administrators. The backend Java and frontend JavaScript programs will execute automatically according to the system's design specifications and relevant data in the database. Methods will only be executable by the client or web server, as specified, and will not exist within the database.

Objects in the database will interact with each other through specified SQL relationships, typically by referencing the unique numerical identifier(s) of the parent class(es). Objects in the internal software will interact through polymorphism and hierarchical class structures.

The prefix `(HASH STR)` refers to a String that is serialized using a two-way hash function when stored in the database, to allow for quicker indexing. In OOP, these will be represented with traditional String instances.

Every non-static class attribute will be `private` unless otherwise specified, and will be readable / writable using "get"/"set" methods, respectively. For example, the get/set methods for the "Theater" class are outlined below:

- `[void] set*Attribute*(*type* value)` – e.g. `setZipCode(int zip_code)`
    - Assigns the value of `*Attribute*` to `value`
    - Valid attributes: (INT) TheaterID, (STR) TheaterName, (BOOL) TheaterStyle, (INT) ZipCode, (STR) StateName, (STR) CityName, (STR) Address
- `[*type*] get*Attribute*()` – e.g. `getZipCode()`
    - Returns the value of `*Attribute*`
    - Valid attributes: `(INT) TheaterID, (STR) TheaterName, (BOOL) TheaterStyle, (INT) ZipCode, (STR) StateName, (STR) CityName, (STR) Address, (DOUBLE) LocationLatitude, (DOUBLE) LocationLongitude`

4.1 Theater

4.1.1 Attributes

- *(INT) TheaterID – Unique sequential numerical identifier for a theater, starting at 1
- (HASH STR) TheaterName – Unique string representation of a theater's name, i..e "AMC Mission Valley"
- (BOOL) TheaterStyle – The "style" of theater. 0 = standard, 1 = dine-in
- (INT) ZipCode – The geographic ZIP code in which the theater exists
- (HASH STR) StateName – The name of the state in which the theater exists
- (HASH STR) CityName – The name of the municipality in which the theater exists

- (STRING) Address – The physical address of the movie theater, i.e. “1234 Hanna Road”
- (DOUBLE) LocationLatitude – The geographical latitude of the theater’s physical address
- (DOUBLE) LocationLongitude – The geographical longitude of the theater’s physical address
- (SHOWTIME[])

4.1.2 Functions

- Used to identify a specific “showtime” object based on its physical location
- Used to search for nearby theaters based on various criteria, such as state, city, and ZIP

4.1.3 Methods

- `[Theater] Theater()` – Constructor. Creates a blank “Theater” instance.
- `[String] getDirections(double latitude, double longitude)` – e.g. `getDirections(49.2842,-10.329)`
    - Returns a hyperlink, in `String` format, to the device’s native navigation application (Google Maps in a web browser by default), set up to provide directions from the user’s current location to the theater’s location
        - If the user’s current location is not specified, the map application will direct the user to enter their location

4.2 Showtime

4.2.1 Attributes

- *(INT) ShowtimeID – Unique sequential numerical identifier for a showtime, starting at 1
- (INT) StartTime – UNIX time stamp for the showtime’s beginning.
- (DOUBLE) CostPerTicket – Undiscounted cost (in $) for an adult ticket for this showing
- (INT) FirstTicketID – The ID # of the first ticket associated with this showtime.
- (INT) NumberOfSeats – The number of seats in the theater where this Showtime is being held
- (BOOL) IsAssignedSeating – 0 = seating is first-come-first-serve, 1 = seating is reserved
- (INT) ScreenNumber – The number of the actual theater screening room within a Theater
- (ARRAY BOOL) TakenSeats – ordered list of boolean values representing whether each seat has been purchased already or not
- (BOOL) CCEnabled – whether or not Closed Captions are enabled for this showing

4.2.2 Functions

- “Belongs” to a Movie
- “Belongs” to a Theater
- Used to organize Tickets based on the specific showtime they refer to
    - i.e. if *FirstTicketID* is 950, and *NumberOfSeats* is 100, then Tickets 950-1049 will belong to this Showtime.
- Used to query nearby showings of a certain movie, so that users can search for showings by time and date within a specified geographic area

4.2.3 Methods

- `[Showtime] Showtime()` – Constructor. Creates a blank “Showtime” instance.

- `[Bool] beginsAfter(int timestamp)` – e.g. `beginsAfter(1604360704)`
    - Returns `True` if the showtime begins after the specified Unix timestamp. Returns `False` otherwise.
- `[Int] getNumberOfTicketsSold()`
    - Returns the total number of tickets sold by indexing `TakenSeats[]`.
- `[String] getStartDate()`
    - Returns the showtime's date in `Weekday, MM/DD/YYYY` format.
- `[String] getStartTime()`
    - Returns the showtime's start time in `HH:MM (AM/PM)` format.

4.3 Movie

4.3.1 Attributes

- *(INT) MovieID – Unique sequential numerical identifier of the movie
- (HASH STR) Title – The name of the movie, hashed for more efficient indexing
- (ARRAY STR) Actors – CSV list of main actors appearing in the movie
- (INT) Runtime – The duration of the movie in minutes
- (STR) ReleaseDate – MM/DD/YYYY, the date the movie was first released in theaters
- (ARRAY INT) GenreID – The ID #(s) of the movie's Genre(s)
- (FLOAT) RottenTomatoes – Consensus rating for the movie on Rotten Tomatoes. Updates daily based on continuous feedback.
- (FLOAT) IMDB – Consensus rating for the movie on IMDB. Updates daily based on continuous feedback.
- (STR) MPAA_Rating – The movie's rating by the MPAA, such as PG-13 or R.

4.3.2 Functions

- Allows several Showtimes to point to the same Movie in memory, to share common attributes
- Allows users to search for a movie before determining which theater/showtime to select

4.3.3 Methods

- `[Movie] Movie()` – Constructor. Creates a blank "Movie" instance.

4.4 User

4.4.1 Attributes

- *(INT) UserID – Unique identifier for a user with an account
- (HASH STR) FullName – The full name of the user, hashed for easy access
- (ENCRYPTED STR) Password – The user's password, with one-way hash encryption
- (STR) DateOfBirth – MM/DD/YYYY, the user's birth date (used for age verification based on movie rating)
- (INT) CartID – Reference to the user's cart
- (INT) MembershipID – The unique identifier of the user's Premium Membership. 0 if no membership.
- (ARRAY INT) OwnedTickets – List of ticket ID's owned/purchased by the user

- (STR) HomeAddress – The user's full address, such as 1234 Hanna Blvd. San Diego, CA 92182.
- (INT) RewardBalance – The number of reward points the user has. 1 point = $0.01
- (DOUBLE) AccountBalance – The user's current account balance ($), i.e. from gift cards and returns/refunds

4.4.2 Functions

- Facilitates the purchase of tickets and the handling of transactions
- Allows for administrators to review an individual's order history and premium membership

4.4.3 Methods

- `[User] User()` – Constructor. Creates a blank "User" instance.
- `[Bool] canWatchMovie(Movie movie)` – e.g. `canWatchMovie(StarWars)`
  - Returns `True` if the user is old enough to purchase tickets for the specified movie, based on its MPAA rating (R, PG, PG-13, etc.). Returns `False` otherwise.
- `[Int] getUserAge()`
  - Returns the user's age in years, based on their account's DateOfBirth.

4.5 ShoppingCart

4.5.1 Attributes

- *(INT) CartID
- (INT) UserID – The ID of the user to which the cart belongs
- (STR[]) CartContents – Array of all items in cart. Ex: {Tix.305,Tix.306,Tix.307,PremMbrsp}

4.5.2 Functions

- Allows a user to add several tickets / other items to their "cart" and check out all at once

4.5.3 Methods

- `[ShoppingCart] ShoppingCart()` – Constructor. Creates a blank "ShoppingCart" instance.
- `[void] addItem(Object item)`
  - Adds an item of type Object (any subclass) to the cart's list of contents.
- `[void] removeItem(int index)`
  - Removes the item in the shopping cart at index `[index]`
- `[double] getTotalCost()`
  - Returns the total cost of everything in the cart.
- `[Bool] applyDiscount(String authCode, double percent)`
  - Applies a `[percent]%` discount to the ShoppingCart, with authorization code `[authCode]` to represent the discount's identifier (i.e. coupon code, military discount).
  - Returns `True` if the discount is successfully applied. Returns `False` otherwise.

4.6 Membership

4.6.1 Attributes

- *(INT) MembershipID – The unique ID # of the Membership
- (INT) UserID – The ID # of the user owning the membership
- (BOOL) IsActive – Whether or not the membership is currently active (0=false, 1=true)
- (STR) ActivationDate – MM/DD/YYYY
- (STR) ExpirationDate – MM/DD/YYYY, when the membership did/will expire
    - Can be modified by renewing the membership (can be automatic)

4.6.2 Functions

- Contains basic information about a user's premium membership, if they have one
- Exists as a distinct object to handle expired memberships and enable transferral of membership between users, with administrator approval

4.6.3 Methods

- `[Membership] Membership()` – Constructor. Creates a blank "Membership" instance.

4.7 Ticket

4.7.1 Attributes

- *(INT) TicketID – Unique numerical sequential identifier
- (INT) UserID – The ID of the user who owns the ticket
- (INT) ShowtimeID – The ID of the showtime that the ticket refers to
- (INT) DiscountID – The ID of the discount applied to this ticket. 0 for no discount or ticket not yet purchased.
- (BOOL) IsAccessible – Whether the seat is wheelchair accessible (will display as such when purchasing ticket). 0 = not wheelchair accessible, 1 = wheelchair accessible
- (STR) SeatLabel – The labeled letter/number of a seat, such as "A6" or "F12"
    - Used by the system to display the seat's location within the theater, when purchasing tickets
- (FLOAT) TicketCost – Default, undiscounted cost of the ticket. May vary within a certain showing, based on variation in seating, i.e. luxury vs. standard seats
- (BOOL) IsReserved – Whether or not the ticket is temporarily "reserved", meaning that a user has selected it and added it to their cart (but hasn't purchased it yet). The user gets 5 minutes to purchase the ticket before it becomes available to other users.
- (BOOL) IsPurchased – Whether or not the ticket has been purchased by a user, thereby making it unavailable.

4.7.2 Functions

- The system creates "Tickets" for each showtime according to the number of seats required for that showtime. Tickets may then be purchased by users. A ticket represents an individual seat in an individual showing at a theater.
- Allows a user to keep track of the tickets they own

- Allows the theater to keep track of available seating and to track statistics

4.7.3 Methods

- `[Ticket] Ticket()` – Constructor. Creates a blank "Ticket" instance.
- `[Showtime] getShowtime()`
  - Returns the `Showtime` associated with this `Ticket`
- `[User] getUser()`
  - Returns the `User` who purchased this `Ticket`

4.8 DiscountCategory

4.8.1 Attributes

- *(INT) DiscountID – Unique numerical sequential identifier
- (STR) DiscountType – i.e. "Military", "Youth", "Student"
- (FLOAT) DiscountPercentage – Number from 0 to 100, i.e. 33.3

4.8.2 Functions

- Allows for consistent application of discounts, which are applied to each ticket according to the characteristics of the user who the ticket is for.

4.8.3 Methods

- `[DiscountCategory] DiscountCategory()` – Constructor. Creates a blank "DiscountCategory" instance.

4.9 Genre

4.9.1 Attributes

- *(INT) GenreID – Unique numerical sequential identifier
- (STR) Name – The name of the genre, i.e. "Comedy", "Drama", "Horror"

4.9.2 Functions

- Allows a movie to "belong" to one or more Genre

4.9.3 Methods

- `[Genre] Genre()` – Constructor. Creates a blank "Genre" instance.

4.10 Employee

4.10.1 Attributes

- *(INT) EmployeeID – Unique numerical sequential identifier
- (INT) PermissionLevel – Refers to the employee's level of permission. 0 = no permissions (i.e. suspended / terminated), 1 = low-level employee (i.e. cashier), 2 = theater manager, 3 = system administrator, 4 = corporate executive
- (HASH STR) First Name
- (HASH STR) Middle Name
- (HASH STR) Last Name
- (STR) DateOfHire

4.10.2 Functions

- Allows employees to handle certain tasks that are unavailable to customers

- Allows for permissions to be delegated among employees according to the requirements of their job duties
- Allows for secure entry to and administration of databases and web servers when necessary

4.10.3 Methods

- `[Employee] Employee()` – Constructor. Creates a blank “Employee” instance.

5. Task Management and Development Timeline

It is necessary to partition tasks among different development teams and individual developers to ensure that adequate progress is made in a timely manner. (Note: Pretend we have a fake development team)

5.1.1 Beginning Development

- Includes unit tests
- Consultations with the UX/UI team to establish a visual mockup and functional relationship between different parts of the system. After review, our Frontend and Backend team will collaborate to execute this vision. Design is subject to change throughout the process
- Each module and the interaction between them will be tested by QA to ensure satisfactory final product
- Security will be thoroughly tested to ensure final product is safe for consumers to use
- Live customer service agents will be available 9 AM - 5 PM to handle reports with appropriate response time

5.1.2 Full System Tests

- Functional:
    - Unit testing
    - Integration Testing
- Non-Functional:
    - Performance testing
    - Usability testing
    - Load testing
    - Security Testing
    - Scalability testing
    - Functionality testing
    - Recovery testing
- Maintenance

5.1.3 Completion of Minimum Viable Product

In order to deliver a Minimal Viable Product that achieves our vision, the following will be taken into account to complete it:

- User research to find what features of similar systems work/don't work
- Small launch with a handful of theaters to gain real-time user feedback
- Estimated final system completion is 1 month after feedback; subject to change based on unknown risks
- Analysis of costs necessary to construct the system with our current team

5.1.4 Final Product Due

- The project's timeline incorporates building a viable prototype that is estimated to take 4 months to examine user testing
- Including the testing interval, the expected deadline for release will be anywhere from 10 months to a year from now

5.2 Delegation of Tasks

5.2.1 UX/UI

- Research and analyze user requirements
- Responsible for making GUI to meet those requirements
- Collaborate with rest of team to ensure seamlessness

5.2.2 Frontend Team

- Responsible for developing frontend code that will be rendered in a user's web browser
- Integration of APIs from the front end, such as the shopping cart and email system, to fit visual expectations
- Responsible for executing the final vision from consultations with UX/UI team
- Implement proper user authentication and require security checks and cool-downs for bolstered security; collaborate with Back end for seamlessness

5.2.3 QA Testers

- Oversees all testing
- Collaborate with appropriate teams to provide feedback for bug and QoL fixes

5.2.4 Backend Team

- Responsible for developing software architecture that operates on the web server
- Responsible for ensuring the security of the system as the intermediary between the database and the web client

5.2.5 Database Team

- Responsible for setting up database infrastructure and working with the Backend Team to achieve parity between internal classes and database classes

5.2.6 Customer Service

- Responsible for establishing communication with users/theater employees
- Mediary for resolution of issues