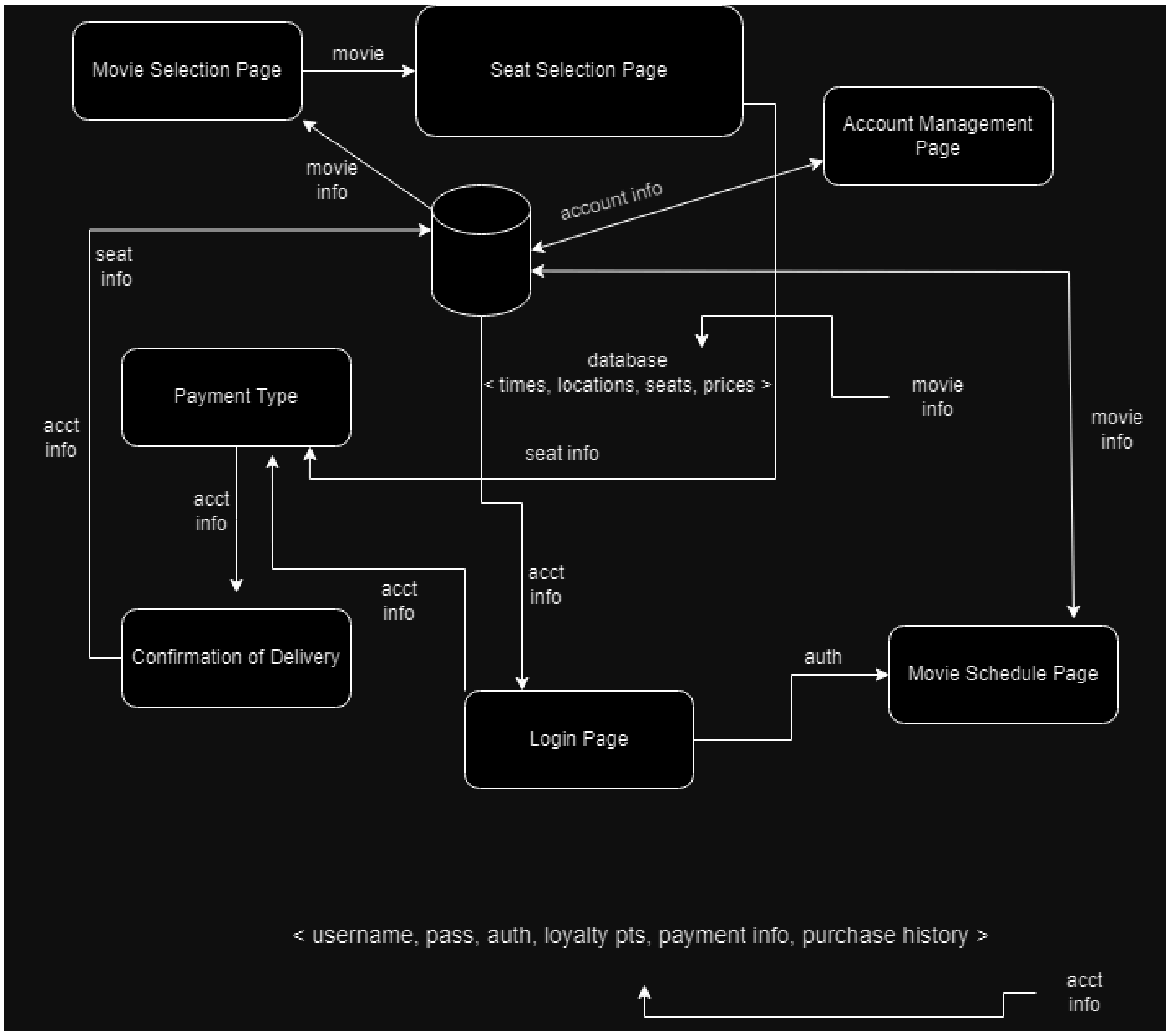
Software Design Specification
Group #15
Movie Theater Ticketing System

Bryce Rambach
Nikoli Oudo
Atah Habibi

System Description

Users and team members will be able to use the movie theater ticketing system through an interactive program presented by the software design. As listed below, it will have a number of prerequisites. Users of the program will be able to reserve cinema tickets and grant team members access to the database. Numerous alternatives are provided, including a complete payment gateway and seat selection.

Software Architecture Overview



Components Overview

1. Movie Selection Page:

- **Function:** This is where users select the movie they want to watch.
- **Connectors:**
    - Sends selected movie information to the Seat Selection Page.
    - Receives movie info from a database to display available movies.

2.Seat Selection Page:

- **Function:** Users choose their preferred seats on this page.
- **Connectors:**
    - Sends seat info to the Payment Type for transaction processing.
    - Receives account info from the Account Management Page to verify user details.
    - Receives seat info from the database to display available seats.

3. Account Management Page:

- **Function:** Handles user account details, preferences, and history.
- **Connectors**:
    - Sends account info to the Seat Selection Page for booking tickets.
    - Sends movie info to the Movie Schedule Page to display the movie timings and details.

4. Payment Type:

- **Function:** Allows users to choose a payment method and processes the payment.
- **Connectors:**
    - Sends acct info to the Confirmation of Delivery to finalize the transaction.
    - Receives seat info and acct info from the Seat Selection Page to process the payment.

5. Confirmation of Delivery:

- **Function:** Confirms the successful transaction and ticket delivery.
- **Connectors:**
    - Receives acct info from the Payment Type to confirm the transaction and update the user's account.

6. Login Page:

- **Function:** Authenticates users before they access their accounts or make a booking.
- **Connectors:**
    - Sends auth (authentication) information to the Movie Schedule Page to ensure the user is logged in before accessing movie schedules.
    - Receives acct info from users attempting to log in.

7. Movie Schedule Page:

- **Function:** Displays the schedule of movies.
- **Connectors:**
    - Receives auth from the Login Page to display schedules to authenticated users.
    - Sends movie info to the Account Management Page to allow users to manage their bookings.

8. Database:

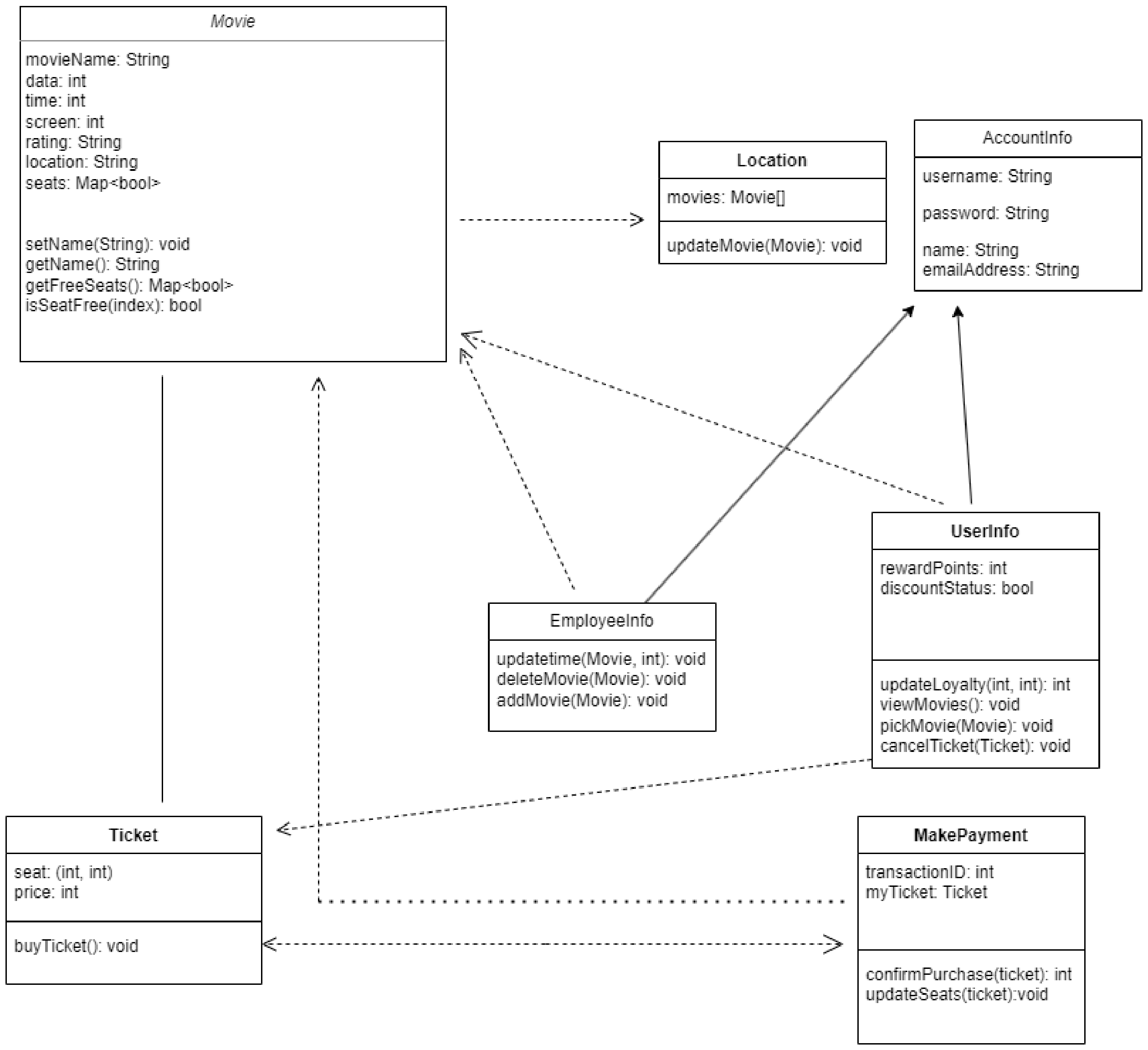
- **Function:** Stores all the data required for the system to function, like movie times, locations, seat availability, and pricing.
- **Connectors:**
    - Interacts with multiple components, providing times, locations, seats, and prices to the Seat Selection Page, and seat info to the Payment Type.

Bottom Connector (username, pass, auth, loyalty pts, payment info, purchase history):

- This represents the flow of information that is likely used throughout the system for different purposes. It connects the entire system architecture and suggests that these pieces of information are central to the system's operations and are passed between various components.

Explanation of Connectors and Flow:

- **Movie Selection to Seat Selection:** After selecting a movie, the relevant information is passed to the Seat Selection Page, so the user can choose their seats.
- Seat Selection to Payment: Once seats are selected, this information, along with the user's account details, is passed to the Payment Type to proceed with the transaction.
- **Payment to Confirmation:** After payment details are provided, the information is sent to the Confirmation of Delivery component to confirm the transaction and issue the ticket.
- **Account Management to Seat Selection and Movie Schedule:** User account details are shared with the Seat Selection for booking and the Movie Schedule Page to display personalized movie schedules.
- **Login to Movie Schedule:** Ensures that only authenticated users can access the movie schedule.
- **Database Interactions:** The database acts as a central repository, providing necessary data to the Seat Selection and Payment Type components and receiving updates from them as transactions occur.



Classes Description:

- **Movie**
  - **Attributes:**
    - **movieName: String** - A text attribute to store the name of the movie.
    - **date: int** - An integer to store the date, which is likely a timestamp or similar numerical representation.
    - **time: int** - An integer to store the time, which could also be a timestamp.
    - **screen: int** - An integer indicating which screen the movie is showing on.
    - **rating: String** - A text attribute to store the movie's rating.
    - **location: String** - A text attribute to store the location where the movie is being shown.
    - **seats: Map<bool>** - A map structure with boolean values to represent the occupancy status of each seat.

- **Operations:**
    - **setName(String): void** - A method to set the movie's name, takes a string as a parameter.
    - **getName(): String** - A method to get the movie's name, returns a string.
    - **getFreeSeats(): Map<bool>** - A method to get a map of the free seats, returns a map with boolean values.
    - **isSeatFree(index): bool** - A method to check if a seat is free, takes an index as a parameter and returns a boolean.

- **Ticket**
    - **Attributes**:
        - **seat: (int, int)** - A tuple of two integers representing the seat location (row, column).
        - **price: int** - An integer to store the price of the ticket.
    - **Operations:**
        - **buyTicket(): void** - A method to initiate the purchase of a ticket.
- **Location**
    - **Attributes**:
        - **movies: Movie[]** - An array of Movie objects.
    - **Operations**:
        - **updateMovie(Movie): void** - A method to update a movie, takes a Movie object as a parameter.
- **AccountInfo**
    - **Attributes**:
        - **username: String** - A text attribute to store the username.
        - **password: String** - A text attribute to store the password.
        - **name: String** - A text attribute to store the name of the account holder.
        - **emailAddress: String** - A text attribute to store the email address.
    - **Operations**: None specified.
- **UserInfo**
    - **Attributes**:
        - **rewardPoints: int** - An integer to store the reward points.
        - **discountStatus: bool** - A boolean to store the status of a discount.
    - **Operations**:
        - **updateLoyalty(int, int): int** - A method to update loyalty points, takes two integers as parameters and returns an integer.
        - **viewMovies(): void** - A method to view movies.
        - **pickMovie(Movie): void** - A method to select a movie, takes a Movie object as a parameter.
        - **cancelTicket(Ticket): void** - A method to cancel a ticket, takes a Ticket object as a parameter.

- **EmployeeInfo**
  - **Operations**:
    - **updateMovie(Movie, int): void** - A method to update movie information, takes a Movie object and an integer as parameters.
    - **deleteMovie(Movie): void** - A method to delete a movie, takes a Movie object as a parameter.
    - **addMovie(Movie): void** - A method to add a new movie, takes a Movie object as a parameter.
- **MakePayment**
  - **Attributes**:
    - **transactionID: int** - An integer to store the transaction ID.
    - **myTicket: Ticket** - A Ticket object.
  - **Operations**:
    - **confirmPurchase(ticket): int** - A method to confirm the purchase of a ticket, takes a Ticket object as a parameter and returns an integer.
    - **updateSeats(ticket): void** - A method to update the seats after purchase, takes a Ticket object as a parameter.

Relationships Description:

- The **Location** class has a one-to-many relationship with the **Movie** class, indicating that one location can have multiple movies.
- The **UserInfo** class is related to **AccountInfo**, **EmployeeInfo**, and **MakePayment**. The exact nature of these relationships isn't specified but could indicate inheritance (UserInfo is a type of AccountInfo) or association (UserInfo interacts with EmployeeInfo and MakePayment).
- The **MakePayment** class is associated with the **Ticket** class, indicating that the payment process is related to purchasing tickets.

Development Plan and Timeline

Bryce Rambach will focus on the backend development, dealing with the Database design and integration, ensuring all data interactions are optimized for performance and security.

Nikoli Oudo will take on the front-end development, crafting the user interface for the Movie Selection Page, Seat Selection Page, and ensuring a seamless user experience across the platform.

Atah Habibi will handle the business logic layer, implementing the class operations for Movie, Ticket, and Payment processing, ensuring the system's functionality aligns with business requirements.

Each member will also collaborate on the overall system architecture, ensuring that all components work together cohesively. Regular meetings will be set to review progress, address any integration issues, and ensure the project stays on track according to the timeline established.

1. Verification Test Plan Overview

Purpose: The purpose of this Verification Test Plan (VTP) is to validate the functional and non-functional requirements of the Movie Theater Ticketing System as specified in the Software Design Specification (SDS).

Scope: The scope of this VTP includes:
- Front-end components: Movie Selection Page, Seat Selection Page, Account Management Page, Login Page, and Movie Schedule Page.
- Back-end components: Payment Processing, Confirmation of Delivery, and Database interactions.
- Security, performance, and usability aspects of the system.

2. Test Categories

Unit Testing:
- Target: Individual classes and methods.
- Test Sets/Vectors: Simulated inputs for methods to test each class independently.
- Coverage: Testing setters/getters, constructors, business logic within methods.
- Failures Covered: Incorrect logic, data handling errors, and boundary conditions.

Functional Testing:
- Target: Each page and its functionalities as a whole.
- Test Sets/Vectors: Use case scenarios where the system's functions can be tested end-to-end.
- Coverage: Interaction between components, data flow, and user interface.
- Failures Covered: Integration issues, incorrect workflow operations, and interface defects.

System Testing:
- Target: Entire system.
- Test Sets/Vectors: Complete user actions from login to ticket purchase.
- Coverage: Overall system performance, security, and reliability.
- Failures Covered: System crashes, security vulnerabilities, and performance bottlenecks.

3. Detailed Test Cases

Unit Tests:

- Test Movie Class Set/Get Methods: Validate the ability to set and retrieve movie information.
- Test Seat Availability: Check the logic for determining seat availability.

Functional Tests:

- Test Movie Selection Workflow: Follow the flow from selecting a movie to displaying available seats.
- Test Account Management Operations: Validate account creation, update, and deletion functionalities.

System Tests:

- Test End-to-End Ticket Purchase: Execute a full ticket purchase process and validate the output.
- Test System Recovery: Simulate failures and verify system's ability to recover.

4. Test Execution

- Environment Setup: The test environment will mimic the production environment with a dedicated testing database.
- Data Preparation: Populate the test database with a variety of data scenarios for movies, seats, user accounts, etc.
- Execution Schedule: Tests will be scheduled to run automatically after every major build, with manual testing as required for specific scenarios.

5. Test Evaluation

Each test case will be evaluated against the expected results. Success criteria will be predefined, and any deviation from the expected results will be recorded as a defect. The defect will then be triaged to determine its impact and the urgency of a fix.

6. Reporting and Feedback

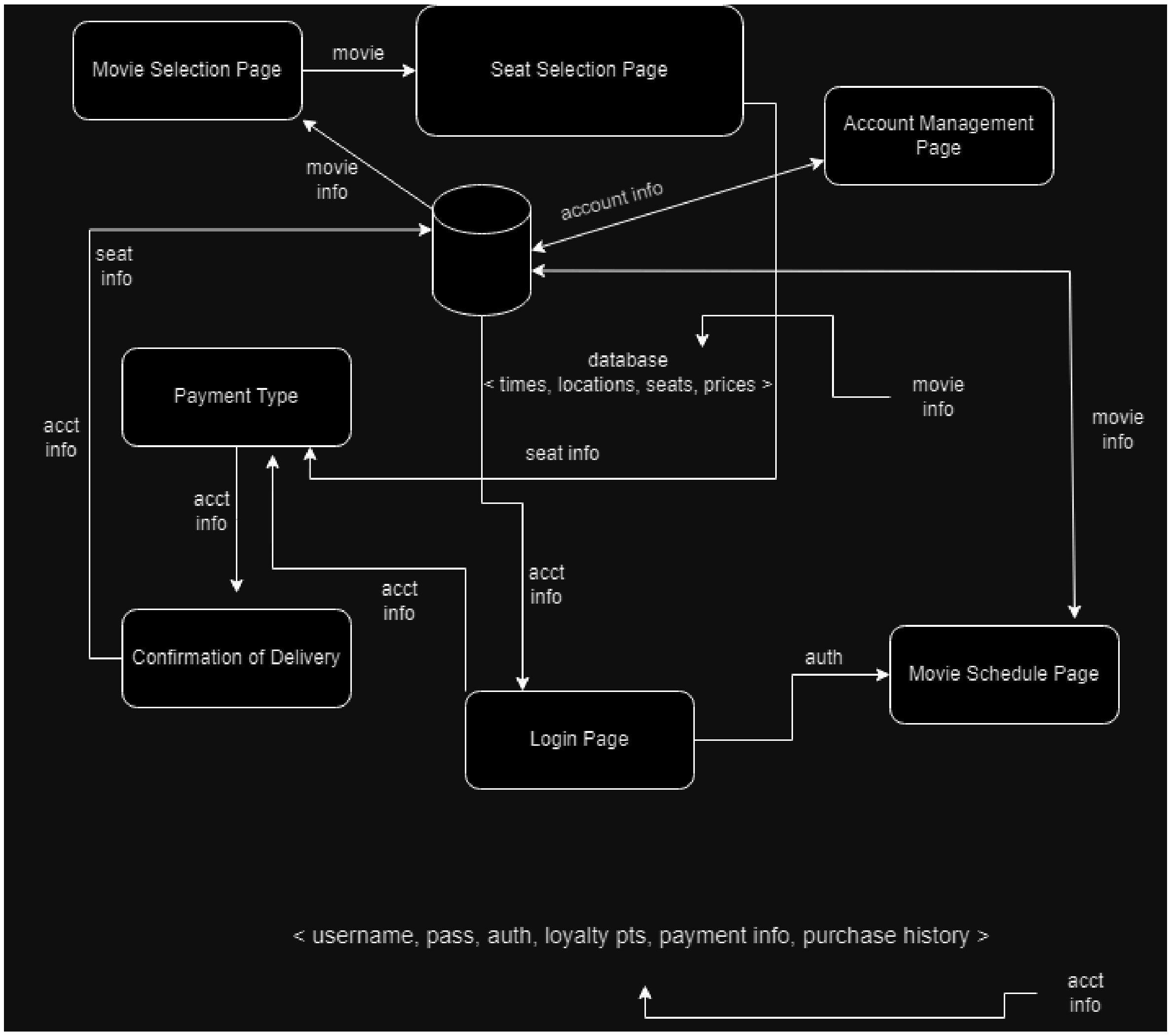
The results of the testing will be documented in detailed reports, providing visibility into the health of the system. These reports will be reviewed by the development team and used to inform decisions on addressing issues and making improvements.

7. Conclusion

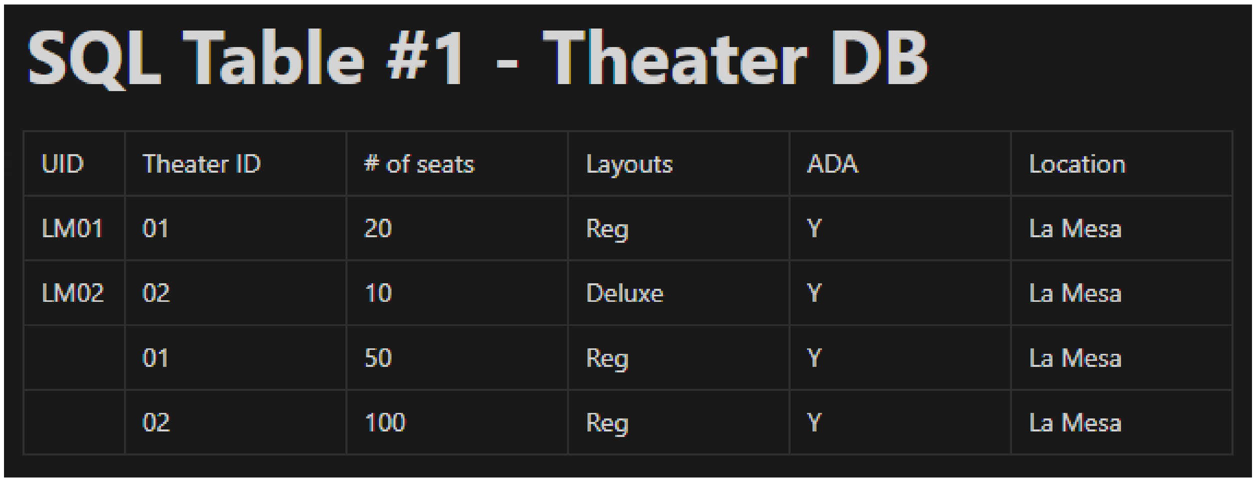
This Verification Test Plan provides a structured approach to ensure that the Movie Theater Ticketing System meets the design specifications and user requirements. Through a mix of automated and manual testing, the plan aims to identify defects early, reduce risks, and maintain high-quality standards throughout the development lifecycle.

Architecture Design w/Data Mgmt.

Software Architecture Diagram

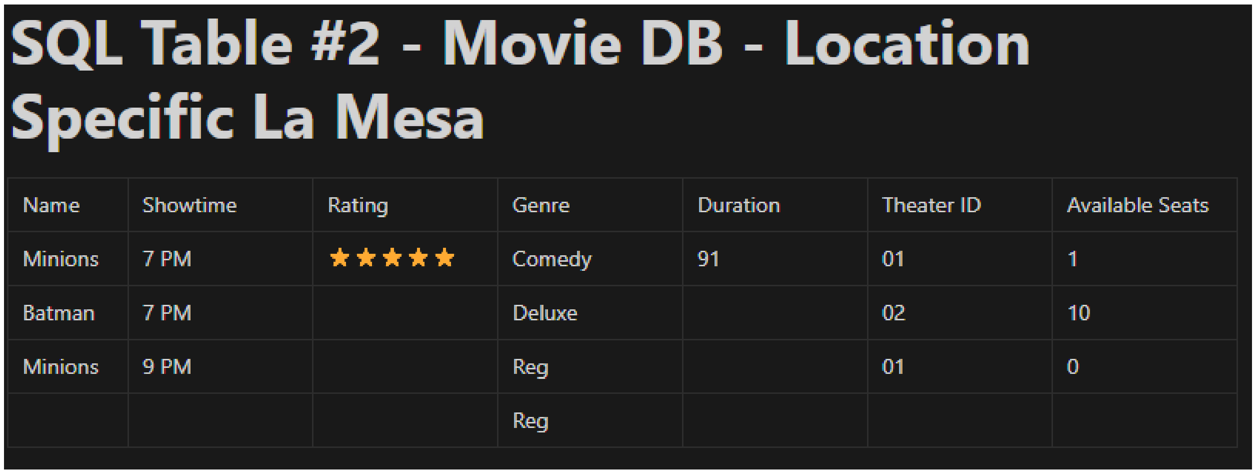


Data Management Strategy



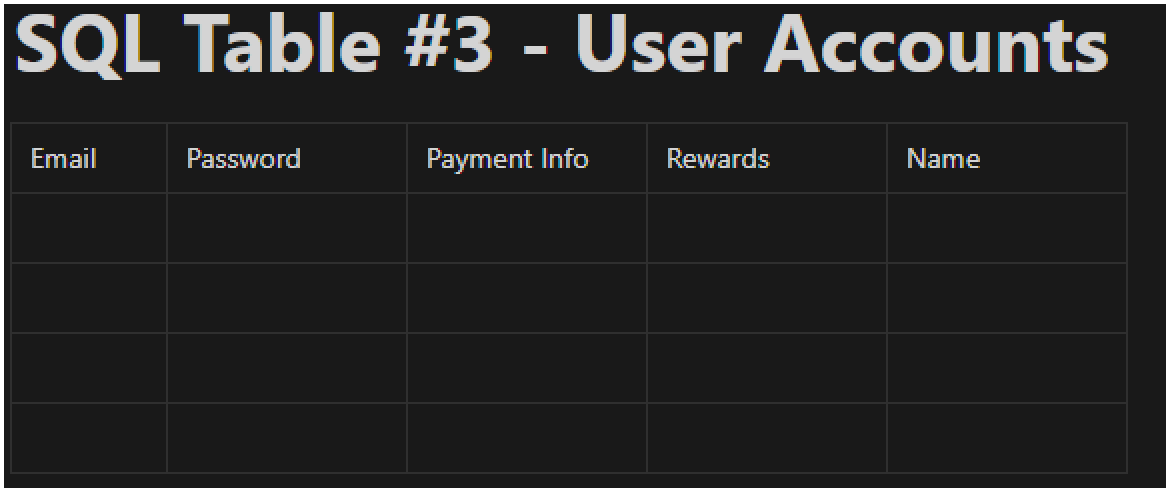
SQL Table #1 - Theater DB

UID	Theater ID	# of seats	Layouts	ADA	Location
LM01	01	20	Reg	Y	La Mesa
LM02	02	10	Deluxe	Y	La Mesa
	01	50	Reg	Y	La Mesa
	02	100	Reg	Y	La Mesa



SQL Table #2 - Movie DB - Location Specific La Mesa

Name	Showtime	Rating	Genre	Duration	Theater ID	Available Seats
Minions	7 PM	★★★★★	Comedy	91	01	1
Batman	7 PM		Deluxe		02	10
Minions	9 PM		Reg		01	0
			Reg			



SQL Table #3 - User Accounts

Email	Password	Payment Info	Rewards	Name

Discussion:

- **Centralization vs. Decentralization:** We have chosen a centralized approach where all related data resides within a single database system but in separate tables. This makes it easier to manage and ensure data integrity through referential constraints and transactions.
- **Security:** For the User Accounts table, ensure the protection of sensitive data like passwords (which should be hashed and salted) and payment information (which should be encrypted and possibly tokenized).

Design Decisions:

- **Three Tables for Different Domains:** The decision to have separate tables for movies, theaters, and users helps segregate the data logically. This aids in data management and can help with performance if properly indexed.
- **Normalization:** It seems the tables are designed with normalization in mind, reducing redundancy by, for example, referencing theater IDs instead of repeating theater information.

Logical Data Split:

- **Movies and Theaters Relationship:** The MovieDB table includes a foreign key to TheaterDB, suggesting a one-to-many relationship (one theater can have many movies shown).
- **Normalization Trade-offs:** While normalization minimizes redundancy and ensures data integrity, it can sometimes lead to more complex queries and potentially slower performance on joins.

Technology Alternatives:

- **SQL Alternatives:** Using a different SQL database system might provide performance or feature benefits. For example, PostgreSQL offers advanced data types and Oracle Database might provide more robust enterprise features.
- **NoSQL Alternatives:** If scalability becomes an issue or if the data becomes more unstructured, we might consider NoSQL databases. For instance, a document-based NoSQL database could store all movie and theater information in a single document, simplifying queries at the cost of transactional integrity.

Tradeoffs:

- **SQL Constraints vs. Flexibility:** By using SQL databases, we're opting for a system with strong consistency and ACID (Atomicity, Consistency, Isolation, Durability) properties, which is excellent for transactions. However, this can come at the cost of scalability and flexibility compared to NoSQL databases.
- **Performance Optimization:** Depending on the size and query patterns, a single, well-indexed SQL database could be optimized for performance. However, as your dataset grows, you might encounter scaling issues that NoSQL databases can handle more gracefully.