MovieMania

Software Requirements Specification

Version 4

8/04/2025

Group #1
Suber Ebrahim and Deryn Cabana

Link to group GitHub Repository
https://github.com/dcabana777/CS250-Project/tree/main

Prepared for
CS 250- Introduction to Software Systems
Instructor: Gus Hanna, Ph.D.
Summer 2025

Revision History

Date	Description	Author	Comments
7/15/2025	Version 1	Suber, Deryn	First Revision
7/22/2025	Version 2	Suber, Deryn	Added “Software Design Specification”
7/29/2025	Version 3	Suber, Deryn	Added “Test Plan”
7/29/2025	Version 3.1	Suber, Deryn	Modified “Test Coverage”
8/04/2025	Version 4	Suber, Deryn	Added “Architecture Design & Data Management”

Document Approval

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

Signature	Printed Name	Title	Date
Suber, Deryn,	Suber, Deryn	Software Requirements.	7/15/2025
	Dr. Gus Hanna	Instructor, CS 250	

Table of Contents

Software Requirements Specification

Software Design Specification

Test Plan

Architecture Design & Database Management

1. Introduction

This Software Requirement Specification (SRS) document provides all the expectations for this movie theater ticketing system. It will show guidelines for the whole system and it will also show the limitations and the necessary functions and the necessary misc.

1.1 Purpose

This movie ticketing system will carry out various tasks in order to ensure a stable and easy to use environment for the user while simultaneously accounting for as many scenarios as possible. This system puts emphasis on local movie theater connection while gearing up and preparing for future features and expansions.

1.2 Scope

- *This system is to be accessible both online(on browser) and on the app.*
- *Although this system is to be accessible on both desktop and mobile the main focus will be on the mobile side of the system.*
- *The system should work only locally for now but should be implemented with an expandable framework that can be later expanded to accommodate national and international support*
- *Showtimes are showcased two weeks in advance*
- *Showtime are 4-10 day*
- *Special discounts for military, senior, students, and children if accompanied by senior*
- *Limit on individual ticket buyers is 7 for first week of premier and unlimited after.*

1.3 Definitions, Acronyms, and Abbreviations

- VIP: seating that costs extra
- Standard seats: regular seats price is dependant on the movie
- Recliners: seats with extra comfort and legroom for extra cost
- Senior seats: seats for people who need help or easier access

1.4 References

- *Software requirements template*
- *IEEE SRS standard 1998*
- *Lecture use cases*
- *Theater ticketing system*
- *Theater ticketing requirements*
- *Theater ticketing questions*

1.5 Overview

This subsection should:
(1) Describe what the rest of the SRS contains
(2) Explain how the SRS is organized.

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

This system will always recommended the closest theatres to users and also have the option to call for any questions or issues.

2.2 Product Functions

- *Shows movies and their showtimes*
- *Shows price for all movies*
- *Shows premier dates for upcoming movies*
- *Empty and taken seats for movies*
- *Promotions for vip seating*
- *Showcase any sales events*
- *Showcase any discounts for seniors or vets etc*

2.3 User Characteristics

- *Allow guest users but with limitations*
- *Logged in users have access to everything*
- *Movie theater kiosk users will have an extra option to call for assistance from worker*

2.4 General Constraints

- *This system can't be used outside of the local 100 mile radius*
- *This system is on primarily on mobile experience on desktop is not good*
- *One user per account*
- *Limit on how many tickets can be purchased by one person*
- *Bot purchase detection*

2.5 Assumptions and Dependencies

- *Internet access*
- *Online account if purchasing*
- *If you qualify for a discount you must provide proof for example seniors must prove age.*
- *Ability to pay online if purchasing online*

3. Specific Requirements

This will be the largest and most important section of the SRS. The customer requirements will be embodied within Section 2, but this section will give the D-requirements that are used to guide the project's software design, implementation, and testing.

Each requirement in this section should be:

- *Correct*
- *Traceable (both forward and backward to prior/future artifacts)*
- *Unambiguous*
- *Verifiable (i.e., testable)*
- *Prioritized (with respect to importance and/or stability)*
- *Complete*
- *Consistent*
- *Uniquely identifiable (usually via numbering like 3.4.5.6)*

Attention should be paid to the carefuly organize the requirements presented in this section so that they may easily accessed and understood. Furthermore, this SRS is not the software design document, therefore one should avoid the tendency to over-constrain (and therefore design) the software project within this SRS.

3.1 External Interface Requirements

3.1.1 User Interfaces

- App based with support for desktop and browser.
- Realtime map showing taken and available seating
- Realtime movies posters for movies with showtime that day
- Confirmation for payments

3.1.2 Hardware Interfaces

- Phones

3.1.3 Software Interfaces

-

3.1.4 Communications Interfaces

- Must provide a section for QNA and phone number and email for support

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 <Functional Requirement or Feature #1>

3.2.1.1 Introduction

3.2.1.2 Inputs

- Must work on mobile, desktop, and online
- Must limit tickets purchasing correctly

3.2.1.3 Processing

- System must be able to accommodate at least 1 million current users
- Must handle discounts
- Must handle cancellations and refunds
- Must stop underage users from R rated movies.
- Must not allow purchase if user declined terms of service

3.2.1.4 Outputs

- Must output payment status after payment is made
- Must output movie missed if ticket is not used in time
- Must output movie ticket availability when browsing

3.2.1.5 Error Handling

- System must give user directions to fix error and the option to contact help

3.2.2 <Functional Requirement or Feature #2>

…

3.3 Use Cases

3.3.1 Use Case #1

Buying ticket while underage

- Look for movie
- User selects R rated movie
- User attempts to buy a ticket to the movie
- System checks if they are underage
- Systems displays message to user that they aren't allowed to buy the movie ticket because they are underage

3.3.2 Use Case #2

Using the system while declining terms of service

- Terms of service is shown to user
- User declines terms
- System automatically closes the app or puts up error if in browser
- User must agree to the terms of service if they want to proceed message is put up in the app once the user returns.

…

3.4 Classes / Objects

3.4.1 <Class / Object #1>

3.4.1.1 Attributes
3.4.1.2 Functions
<Reference to functional requirements and/or use cases>

3.4.2 <Class / Object #2>

…

3.5 Non-Functional Requirements

Non-functional requirements may exist for the following attributes. Often these requirements must be achieved at a system-wide level rather than at a unit level. State the requirements in the following sections in measurable terms (e.g., 95% of transaction shall be processed in less than a second, system downtime may not exceed 1 minute per day, > 30 day MTBF value, etc).

3.5.1 Performance

- System must give user the contact for help if stuck in loading for more than 10 minutes

3.5.2 Reliability

- System must be online all the time with scheduled downtime reported two days in advance

3.5.3 Availability

- Must be available all the time

3.5.4 Security

- Must encrypt all user date

3.5.5 Maintainability

3.5.6 Portability

3.6 Inverse Requirements

*State any *useful* inverse requirements.*

3.7 Design Constraints

Specify design constrains imposed by other standards, company policies, hardware limitation, etc. that will impact this software project.

3.8 Logical Database Requirements

Will a database be used? If so, what logical requirements exist for data formats, storage capabilities, data retention, data integrity, etc.

3.9 Other Requirements

Catchall section for any additional requirements.

4. Analysis Models

List all analysis models used in developing specific requirements previously given in this SRS. Each model should include an introduction and a narrative description. Furthermore, each model should be traceable the SRS's requirements.

4.1 Sequence Diagrams

4.3 Data Flow Diagrams (DFD)

4.2 State-Transition Diagrams (STD)

5. Change Management Process

Identify and describe the process that will be used to update the SRS, as needed, when project scope or requirements change. Who can submit changes and by what means, and how will these changes be approved.

A. Appendices

Appendices may be used to provide additional (and hopefully helpful) information. If present, the SRS should explicitly state whether the information contained within an appendix is to be considered as a part of the SRS's overall set of requirements.

Example Appendices could include (initial) conceptual documents for the software project, marketing materials, minutes of meetings with the customer(s), etc.

A.1 Appendix 1

A.2 Appendix 2h

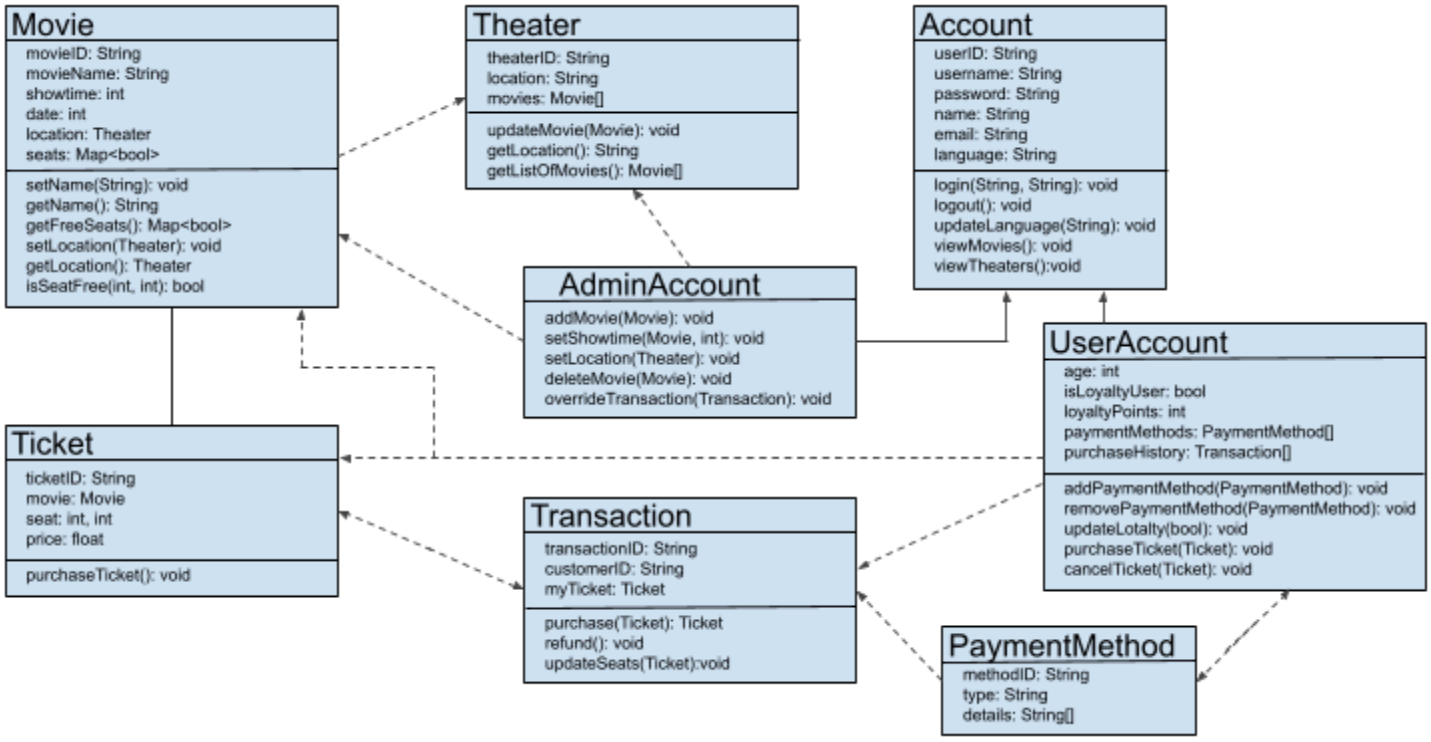
Software Design Specification

1. System Description

The theater ticketing system is a web-based application designed to facilitate the discovery, selection, and purchase of movie tickets across a network of 20 theaters in the San Diego area. The system supports both regular and deluxe theaters, enforces purchase constraints, and accommodates optional user accounts. Key features include secure digital and physical ticket issuance. This system consists of two main sides, the user interface through accounts, and the theater system database, containing theaters, movies, tickets, and more. The system is developed using a modular architecture with distinct classes responsible for user management, ticketing, payments, and administration. It employs centralized data storage partitioned by theater, with APIs to external services for payment and more.

2. Software Architecture Overview

2.1 UML Diagram



2.1.1 Class: Movie

Purpose: Represents an individual film with associated showtime, seating, and other metadata.
Attributes:
movieID: String - Unique identifier for the movie
movieName: String - Title of the movie
showtime: int - Start time of the movie in a simplified integer format
date: int - Date of the movie showing in an integer format
location: Theater - The theater location where the movie is being shown
seats: Map<bool> - Data structure representing seat availability, with seat ID mapped to availability
Operations:
setName(String): void - Updates the movie title
getName(): String - Retrieves the movie title
getFreeSeats(): Map<bool> - Returns a map of available seats
setLocation(Theater): void - Updates the theater the movie is showing at
getLocation(): Theater - Retrieves the movie location
isSeatFree(int,int): bool - Checks is a specific seat is available, by row and column

2.1.2 Class: Theater

Purpose: Represents a physical theater, with multiple movies playing.
Attributes:
theaterID: String - Unique identifier for the movie
location: String - Physical location of the theater (likely a city address)
movies: Movie[] - Array of movies playing at the theater
Operations:
updateMovie(Movie): void - Updates a movie through the operations available in the Movie class
getLocation(): String - Retrieves the location of the theater
getListOfMovies(void): Movie[] - Returns a list of movies playing at this theater

2.1.3 Class: Account

Purpose: Base class for all users who log into the system, either customers or administrators.
Attributes:
userID: String - Unique user ID
username: String - username for login
password: String - Password for authentication
name: String - Full name of the user
email: String - User's email address
language: String - Preferred language for UI localization
Operations:
login(String, String): void - Authenticates user login using credentials. The first parameter is either their username or email; the second parameter is their password
logout(): void - Ends the user session

updateLanguage(String): void - Changes the user's language preference
viewMovies(): void - Allows the user to browse available movies
viewTheaters(): void - Allows the user to browse supported theater locations

2.1.4 Class: UserAccount

Purpose: Represents a registered customer with optional loyalty features. Inherits from Account.
Attributes:
age: int - User's age, used for age-restricted content or discounts
isLoyaltyUser: bool - Indicates if the user is enrolled in a loyalty program
loyaltyPoints: int - Points accrued through purchases with a loyalty program
paymentMethods: PaymentMethod[] - List of stored payment options
purchaseHistory: Transaction[] - List of past ticket purchases and related data
Operations:
addPaymentMethod(PaymentMethod): void - Adds a new payment method to the account
removePaymentMethod(PaymentMethod): void - Removes a stored payment method from the account
updateLoyality(bool): void - Allows the user to enroll in or unsubscribe from the loyalty program
purchaseTicket(Ticket): void - Initiates the ticket purchase process
cancelTicket(Ticket): void - Cancels a previously purchased ticket (if permitted)

2.1.5 Class: AdminAccount

Purpose: Represents system administrators with management permissions. Inherits from Account.
Operations:
addMovie(Movie): void - Adds a new movie to the database
setShowtime(Movie, int): void - Sets or updates the showtime for a specific movie
setLocation(Theater): void - Assigns or changes the theater for a movie
deleteMovie(Movie): void - Removes a movie from the system
overrideTransation(Transaction): void - Manually corrects, cancels, or refunds a transaction

2.1.6 Class: Ticket

Purpose: Represents a ticket for a specific movie and seat.
Attributes:
ticketID: String - Unique identifier for the ticket
movie: Movie - Associated movie object
seat: int, int - Row and column of the seat the ticket is for
price: float - Total price of the ticket after discounts
Operations:
purchaseTicket(): void - Moves onto transaction for the ticket purchase process

2.1.7 Class: Transaction

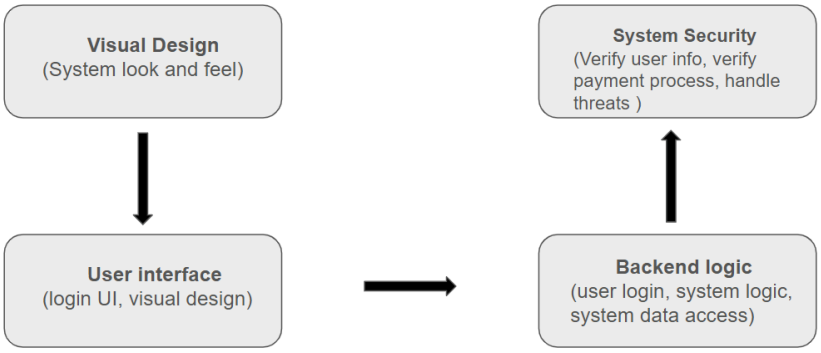
Purpose: Represents a complete payment and booking activity tied to a customer and ticket.
Attributes:

transactionID: String - Unique transaction identifier
customerID: String - ID of the user who made the transaction
myTicket: Ticket - The ticket associated with this transaction
Operations:
purchase(Ticket): Ticket - Performs the purchase logic through the user's account and payment method, and returns the resulting ticket
refund(): void - Refunds this transaction (if permitted)
updateSeats(Ticket): void - Updates the seat availability for the seat associated with the purchased ticket

2.1.8 Class: PaymentMethod

Purpose: Represents a stored payment method used for purchases.
Attributes:
methodID: String - Unique ID for the payment method
type: String - type of payment (e.g. "CreditCard", "PayPal", "Bitcoin")
details: String[] - List of stored details for the payment method (e.g. card number, expiry, etc.)

2.2 SWA Diagram



2.2.1 SWA Description

- The SWA chart illustrates the relationships between different components of the system. Starting with Visual Design (system look and feel) influences the User Interface (login UI, visual design). The Frontend User Interface connects with the Backend Logic to carry out functions such as(user login, system logic, system data access), this system as a whole is inspected at at every level for threats and leaks by the System Security (verify user info, verify payment process, handle threats)

3. Development Plan & Timeline

- Week 1 Collect information and resources for SRS documents.
- Week 2 - 5 Core backend/foundational requirements,
- Week 5 - 7 User authentication and initial frontend design.
- Week 8 Debug and test any bugs with user log in.
- Week 9 Movie showcasing system and general user interface.
- Week 10 Work on payment system.
- Week 11-12 Comprehensive system testing and security test
- Week 13 Add misc features such as holiday sales, non recurring discounts etc.

3.1 Partitioning of Tasks

	Khalid(Backend)	**Marcus(Frontend)**	**Kawhi(System security)**	**Kent(Visual designer)**
Week 1	**Collect info for srs**	**Assists Khalid and Kent with UI**	**Collect info for srs**	**Works on initial mock ups**
Week 2-5	**Work on core functional requirements**	**Begins work on frontend framework**	**Works closely with khalid on foundational security requirements**	**Designs initial frontend design**
Week 5-7	**Implements user authentication**	**Develop login pages and work with khalid to connect to backend**	**Test user authentication for bugs or exploits**	**Inspect UI for early visual bugs**
Week 8	**Fix bugs and test**	**Debug any frontend bugs**	**Fix or flag any bugs found in the system**	**Compare notes with team members and make sure art works with the system functions**
Week 9	**Builds movie times and other UI features and logic**	**Implement movie showcase UI**	**Implement more complex system specific security features such as one user per account.**	**Implement movie showcase art style and animation**
Week 10	**Works on payment processing**	**Build payment UI**	**Add a 3rd party system to verify user**	**Implement look for payment screen**

			payment. And user info	
Week 11-12	**Optimize all features and debug**	**Debug any UI bugs or mismatches**	**Check for any new bugs with focus on user info vulnerability.**	**Final check on UI and system art style for any visual bugs**
Week 13	**Build framework for future implementation of promotional discounts and events**	**Finalize UI and implement promotional/discounts UI**	**Finalize system security debug**	**Implement promotional/special event art style and UI**

3.2 Team Member Responsibilities

1. Khalid Ayman, Backend developer. Responsible for implementing core functional system requirements, and testing and debugging

2. Marcus Smith, Frontend developer. Responsible for connecting building the frontend UI while closely working with Khalid(backend) and Kent(visual designer)

3. Kawhi Leonard, System Security. Responsible for system security, responsibilities include working with other team members to test and debug

4. Kent Bazemore, Visual Designer. Responsible for the overall system look and works closely with Marcus(frontend)

Test Plan

1. Introduction

1.1 Purpose

The purpose of this test plan is to define the strategies, scope, and coverage used to verify and validate the functional and non-functional components of the online web browser-based theater ticketing system. The testing effort is made to ensure that all components and modules behave as specified in the Software Requirements Specification and Software Design Specification, and that the system delivers a secure, consistent, and reliable user experience.

1.2 Scope of Testing

This testing process will cover all critical modules and interactions outlined in the UML and SWA diagrams under the Software Design Specification. The primary focuses will be on:

- Account creation and login/logout (for users and admins)
- Movie discovery and filtering
- Ticket purchasing workflow
- Payment handling and refund processing
- Loyalty program logic
- Seat reservation and availability tracking
- Administrator-level operations (such as setting showtimes, overriding transactions)

The following are considered out of scope for this test plan:

- Integration with third party review sites
- Scalability testing beyond 1000 concurrent users
- UI-specific animation or styling defects
- Physical kiosk hardware interactions
- Any and all future or planned features

2. Test Strategy

To validate system functionality and reliability, different types and levels of testing will be employed.

2.1 Unit Testing

Focuses on individual classes and operations as defined in the UML diagram. Each operation (e.g., *purchaseTicket()* or *addPaymentMethod()*) will be tested in isolation using mock objects and dependencies where applicable.

The objectives are to verify and validate correctness of data manipulation and return values, and account for edge cases that fall outside the intended purpose of the software.

2.2 Functional Testing

Evaluates interactions between components and tests full workflows from a user's perspective. Covers UI actions, form inputs, error messages, and proper flow enforcement (e.g. blocking underage users from restricted films). The internal structure of the code is not tested here.
The objectives are to confirm integration between account, movie, ticket, and transaction systems, verify that UI and backend respond correctly to inputs, and test logical constraints (e.g. ticket limits, loyalty benefits, language selection).

2.3 System Testing

Simulates real-world end-to-end behaviour across all components, including backend logic, frontend display, and API communication. Includes both typical user paths and exceptional workflows.
The objectives are to verify that the system meets its overall specifications, confirm security and stability under realistic conditions, and check synchronization across modules.

2.4 Security Testing

In addition to the above tests regarding detailed implementation, test inputs will be run on security-risk cases such as the reuse of tickets, submit malformed or invalid data, accessing admin functions as a user account, and other relevant non-functional security requirements outlined in the system design.

3. Test Coverage

Each class, feature, or component defined in the UML and SWA diagrams are targeted by one or more testing level:

Component / Feature	**Test Levels**	**Covered Tests**
Account / UserAccount / AdminAccount	Unit, Functional, System	Login/logout, user roles, language updates, individual user operations
Movie	Unit, Functional	Seat availability, showtime updates
Theater	Unit, Functional	Location mapping, movie listings
Ticket	Unit, Functional	Ticket purchasing, seat assignment
Transaction	Unit, Functional, System	Payment and refund logic, seat availability updates
PaymentMethod	Unit	Add/remove/validate methods
Notification	System, Functional	Reminder notifications and their timing
Data Privacy	System, Functional	Ensure user data is deleted successfully
App Updates	System	User prompted to update

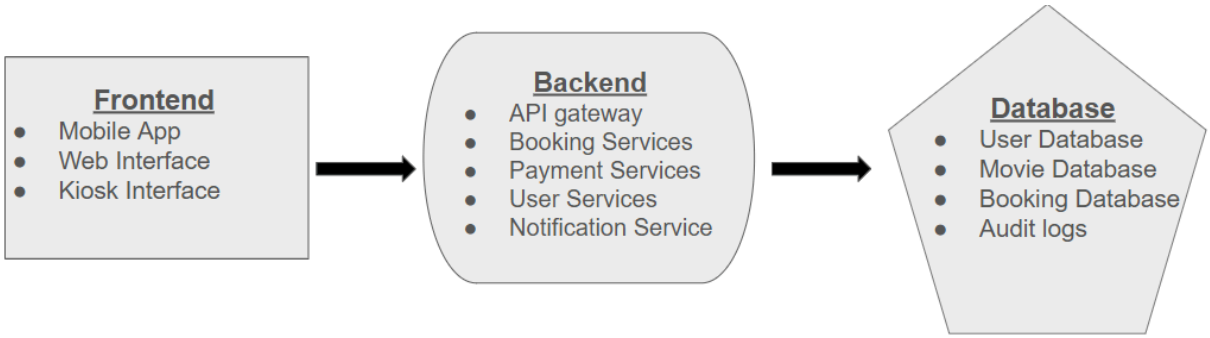
App Logout	System, Functional	Logout logic, logout security, and flow
UI & Backend	Functional, System	Frontend to backend synchronization, user interaction

4. Sample Tests

Below is a link to a sheet containing a set of 10 sample test cases:
https://docs.google.com/spreadsheets/d/11J1JmunpzDH_MKtwa8Wb0apteGOjKgG0i_CmF8SCs1I/edit?usp=sharing

Architecture Design & Data Management

1. Software Architecture Design



2. Data Management Strategy

2.1 Overview

This online web browser-based theater ticketing system is designed to handle user accounts, movie showtimes, ticket purchases, payment methods, and administrative actions. To support these requirements, a relational SQL database model to store and manage persistent data.
This model was selected due to its widespread support, consistency, and compatibility with common web frameworks. The structure it provides is sufficient for this system's purposes, and also allows for clear enforcement of relationships between entities, making it the best fit for the intents of this system.

2.1.1 Considered Alternative

Although we chose to utilize a relational SQL database, the alternative of a NoSQL database was considered for its ease of scaling and schema flexibility. However, NoSQL is generally worse at managing strict relational logic and has higher complexity in ensuring transactional integrity. These tradeoffs were not beneficial enough to justify using a NoSQL database. However, it remains an option if the system is later intended to expand to a larger scale.

2.2 Structure of Database

Only a single primary SQL database will be used, logically partitioned into related tables roughly corresponding to major classes defined in the UML diagram.

- **Accounts**: Contain user information (payment methods, etc.), admin info, and authentication data
- **Movies**: Stores movie metadata (title, rating, showtime, description, seating availability, etc.)
- **Theaters**: Physical location data (layout type, seating numbers, etc.)
- **Tickets**: Purchase records tied to users, movies, seating assignments, etc.
- **Transactions**: Payment-related records including status, refund availability, etc.

Each table will include primary keys, foreign keys for relational inquiries, and indexes for fields frequency queried (userID, movieID, theaterID, etc.).
This single-database structure is sufficient for the current system scope and avoids the overhead of managing multiple database instances. Logical separation through tables allows for modularity without high complexity infrastructure. Theaters operate locally but data is centrally controlled, avoiding issues of duplication or inconsistency across distributed databases.
Comparing a single database vs. multiple databases, the tradeoff is reduced redundancy and simpler design vs. slightly less flexibility for independent scaling.

2.3 Data Security and Privacy

- User passwords will be hashed using a secure hashing algorithm, avoiding direct storage of user passwords in the database itself.
- Payment details (e.g., credit card information) will not be stored directly. Payment credentials via external payment gateways will be stored as tokens instead.
- Role-based access control will be enforced to ensure ordinary users cannot access or manipulate administrative data or perform administrative actions.
- Audit logs and transaction logs will be implemented for traceability of user and admin actions.
- Scheduled nightly backups will be supported to ensure disaster recovery capability.

2.4 Tradeoffs and Future Considerations

Advantages:

- Strong structure and integrity enforcement
- Easy to query, join, and analyze relational data
- Supported by most frameworks and hosting providers

Tradeoffs:

- Less flexible for unstructured data (e.g., scraped movie reviews, user feedback)
- Scaling writes and reads likely requires distribution via sharding if the system expands too much.

Future Considerations:

If the system expands beyond the initial stated scale of San Diego, migrating to a distributed SQL setup or using a hybrid model with NoSWL for high-read modules may be considered. Caching strategies could be layered to reduce read pressure on high frequency queries (e.g. seat maps, showtimes).