“TriTicketing” Ticketing-System

Software Requirements Specification

March 6, 2025

Group 4
Elias Diaz
Cecilia Nothstein
Tibelya Yalcin

Prepared for
CS 250- Introduction to Software Systems
Instructor: Dr. Gus Hanna
Spring 2025

Revision History

Date	Description	Author	Comments
02.20.2025	Version 1.0	Cecilia Nothstein	First Part of specific requirements and 3.5, 3.6
02.20.2025	Version 1.0	Elias Diaz	Use Cases and Class Diagrams
02.20.2025	Version 1.0	Tibelya Yalcin	Introduction and General Description
06.03.2025	Version 2.0	Elias Diaz	UML Class Diagram Software Architecture Diagram
06.03.2025	Version 2.0	Tibelya Yalcin	Description of UML Class Diagram Explanation of Class Attributes and Operations
06.03.2025	Version 2.0	Cecilia Nothstein	Description of Software Architecture Diagram Explanation of SAD and Refinement and Documentation
03.20.2025	Version 3.0	Tibelya Yalcin	UPDATE: UML is Updated with new classes! - TicketVerification Class + Description
03.20.2025	Version 3.0	Cecilia Nothstein	UPDATE: UML is Updated with new classes! - NotificationClass + Description
03.20.2025	Version 3.0	Elias Diaz	UPDATE: UML is Updated with new classes! - LogEntry Class + Description
03.20.2025	Version 3.0	Elias Diaz, Cecilia Nothstein, Tibelya Yalcin	Test Cases, Descriptions, Excel -> in a group session, all of us
09.04.2025	Version 4.0	Elias Diaz, Cecilia Nothstein, Tibelya Yalcin	Brainstorming about Strategy and updated Architecture Diagram

09.04.2025	Version 4.0	Elias Diaz	Update Architecture Diagram, Update Description, Logical Database
09.04.2025	Version 4.0	Tibelya Yalcin	Database Management Strategy, Database Management Overview
09.04.2025	Version 4.0	Cecilia Nothstein	Database Management overview by Domain

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

The introduction of the Software Requirements Specification (SRS) provides an overview of the complete document. This document contains all the information needed by software engineers to design and implement the Theater Ticketing-System effectively. It outlines the purpose, scope, key definitions, references and overall structure of the SRS to ensure clarity and completeness in defining system requirements.

1.1 Purpose

The purpose of this document is to define and specify the requirements for the Theater Ticketing-System. This system will allow customers to purchase and reserve theater tickets online, via mobile devices, or through digital kiosks at the theater. It will provide support for multiple languages, various payment methods and secure ticketing mechanisms to prevent fraud and unauthorized resale. Additionally, the system will offer advanced features such as dynamic pricing, discount eligibility and personalized recommendations based on user history.

The intended audience for this document includes:

- **Software Developers** - Responsible for designing and implementing the system.
- **Testers** - Ensuring compliance with functional and non-functional requirements.
- **Project Managers** - Overseeing development progress and requirement fulfillment.
- **Stakeholders** - Business owners, theater operators and other parties involved in the project.

1.2 Scope

The Theater Ticketing-System is a **web-based and mobile-compatible application** designed to facilitate the seamless reservation, purchase and management of movie tickets for theaters in **San Diego**. The system will offer the following core functionalities:

1. Ticket Purchase & Reservation: Customers can browse available showtimes, select seats and complete transactions.
2. Multi-Platform Support: The system will be accessible via web browsers, mobile applications and theater kiosks.
3. User Accounts: Optional customer accounts for storing payment details, purchase history, loyalty points and notifications.
4. Secure Payments: Transactions supported via credit card, PayPal, Bitcoin and theater gift cards.
5. Fraud Prevention: Each ticket will be uniquely generated and non-replicable (NFT-based security).
6. Theater Management: Administrative tools for managing showtimes, pricing, customer service and promotions.
7. Scalability: The system must support up to 100,000 concurrent users.
8. Dynamic Pricing: Ticket prices can vary based on demand, seat selection and peak hours.

9. Personalized Experience: AI-driven recommendations based on user preferences and past purchases.
10. Accessibility Features: Screen reader support, adjustable font sizes and high-contrast mode for visually impaired users.

The system will **not** support:

- Movie streaming or on-demand viewing.
- Physical ticket delivery outside of in-theater pickup.

1.3 Definitions, Acronyms and Abbreviations

- POS - Point of Sale.
- Dynamic Pricing - A pricing strategy where ticket costs adjust based on factors like demand and time until showtime.
- AI-driven Recommendations - A system that suggests movies based on user preferences and previous transactions.

1.4 References

The following documents and sources are referenced for this project:

- IEEE 830-1998: Recommended Practice for Software Requirements Specifications.
- Existing theater ticketing-system documentation.
- Online payment gateway APIs (PayPal, Bitcoin, Credit Card, Theater Gift Cards).
- Regulatory compliance standards for digital transactions and data protection.
- ADA Compliance Guidelines for accessibility.

1.5 Overview

This document is structured as follows:

- Section 2: General Description - Provides a broad overview of the system's functionality, user roles and constraints.
- Section 3: Specific Requirements - Details the system's functional and non-functional requirements, including security, scalability and performance specifications.
- Section 4: Analysis Models - Includes diagrams illustrating system workflows, data processing and user interactions.
- Section 5: Change Management Process - Defines the methodology for modifying requirements as the system evolves.
- Appendices: Additional references and supporting documentation.

This document follows the IEEE SRS standards and serves as a **guiding reference** for the entire software development lifecycle of the Theater Ticketing-System.

2. General Description

This section describes the general factors that influence the Theater Ticketing-System and its requirements. It does not specify detailed requirements but provides the necessary context for understanding them.

2.1 Product Perspective

The Theater Ticketing-System is an independent web-based and mobile-accessible application designed to improve the current ticket reservation and purchasing experience. It integrates with theater management systems to fetch real-time showtimes and seat availability. Additionally, it provides APIs for external systems such as payment gateways and loyalty programs. The system enhances customer convenience by offering a seamless online and in-person booking process.

2.2 Product Functions

The key functions of the system include:

- Real-time showtime and seat availability lookup
- Online and in-person ticket purchasing
- Secure payment processing through multiple channels
- User account management with loyalty program integration
- Dynamic pricing adjustment based on demand and user preferences
- Fraud prevention through unique digital ticket generation
- Administrative tools for managing showtimes, pricing and promotions

2.3 User Characteristics

The system will be used by:

- Customers - Individuals purchasing tickets online or at kiosks with varying levels of technical expertise.
- Theater Staff - Employees handling in-person sales and resolving customer inquiries.
- Administrators - Personnel responsible for updating showtimes, pricing and monitoring transactions.

2.4 General Constraints

- The system must support 100,000 concurrent users.
- Users can purchase a maximum of 20 tickets per transaction.
- The system must comply with PCI DSS for secure payments.
- Single device login policy for user accounts.
- Limited to theaters within San Diego.

2.5 Assumptions and Dependencies

- The system requires an active internet connection for transactions.
- Third-party payment providers are needed for processing transactions.
- Movie schedule data will be provided by theater management databases.
- The system will run on modern web browsers and digital kiosks.

This section establishes the foundational understanding required for defining the specific requirements of the Theater Ticketing-System.

3. Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

The user interface for the Movie Theater Ticketing-System shall be accessible via standard web browsers, including Google Chrome, Mozilla Firefox, Safari and Microsoft Edge. The system shall be optimized for both desktop and mobile devices, ensuring a responsive design that adapts to different screen sizes.

The user interface shall be implemented using web technologies such as **HTML5, CSS, JavaScript (React/Vue.js) and Bootstrap**, ensuring an intuitive and user-friendly experience. The system shall support accessibility features, including screen readers and keyboard navigation, to accommodate users with disabilities.

Key user interface features include:

- **Movie Listings**: Display available movies with showtimes, trailers and descriptions.
- **Seat Selection**: Interactive seat maps allowing users to choose specific seats in deluxe theaters.
- **Ticket Purchase Flow**: Step-by-step booking process with discount selection (student, military, senior) and real-time seat availability updates.
- **Payment Integration**: Secure checkout supporting credit cards, PayPal and Bitcoin transactions.
- **Account Management**: User registration, login with social authentication (Google, Facebook) and profile management.
- **Booking History**: Users can view past and upcoming reservations.
- **Multi-language Support**: Interface available in **English, Spanish and German**.
- **Digital and Printable Tickets**: QR code generation for digital tickets and option to print physical tickets.
- **Customer Feedback System**: Post-purchase rating system using smiley faces for quick feedback.
- **Real-time Notifications**: Confirmation emails and SMS reminders for upcoming showtimes.

The system shall also prevent automated bots from mass-purchasing high-demand tickets through CAPTCHA validation and fraud detection mechanisms. Furthermore, a **five-minute session timer** shall be implemented to hold seats temporarily before automatic release.

3.1.2 Hardware Interfaces

Since the Movie Theater Ticketing-System must operate both online and in physical theater locations, the hardware interfaces shall include all required components for internet connectivity and in-person ticket transactions.

The required hardware includes:

- **Network Infrastructure**: The system shall operate over a stable internet connection, requiring hardware such as **modems, routers, WAN-LAN networks and Ethernet cables** to ensure reliable online access.
- **Point-of-Sale (POS) Terminals**: The system shall integrate with POS terminals at theater locations to handle in-person transactions via **credit/debit cards, NFC payments (Apple Pay, Google Pay), Bitcoin and cash payments**.
- **Digital Ticketing Kiosks**: Self-service kiosks shall be installed in theaters for users to **browse showtimes, select seats and purchase tickets**.
- **Ticket Printers**: Thermal printers shall be used for generating physical tickets at the theater's box office and self-service kiosks.
- **Barcode and QR Code Scanners**: Handheld and mounted scanners shall be used to validate digital and printed tickets at theater entrances.
- **Mobile and Desktop Devices**: The system shall be accessible on **smartphones, tablets and computers** through web browsers, ensuring a responsive experience.
- **Display Screens**: Digital signage screens within theaters shall display **real-time seat availability and movie showtimes**, synchronized with the ticketing system.
- **Server Infrastructure**: A robust **cloud-based or on-premises server** infrastructure shall be used to handle database queries, transactions and high user concurrency (up to 100,000 users).
- **Security Hardware**: The system shall incorporate **firewalls, biometric access for admin accounts and surveillance integration** to ensure security in theaters and online.

These hardware components ensure smooth ticket sales, seamless user experience and secure transactions for both online and in-theater operations

3.1.3 Software Interfaces

The Movie Theater Ticketing-System shall integrate with various external and internal software components to ensure smooth operation, secure transactions and efficient management.

- The ticketing system shall communicate with the **Showtime Management System** to retrieve and update available movies, showtimes and seat availability in real-time.

- The system shall integrate with a **Payment Gateway** (e.g., Stripe, PayPal, Bitcoin) to validate payments and process transactions securely.
- The system shall interface with a **Loyalty and Membership System** to manage AMC membership cards, discounts and loyalty rewards.
- The system shall connect with an **Authentication Service** (e.g., OAuth, Google, Facebook) to provide secure login and user account management.
- The system shall integrate with a **Customer Relationship Management (CRM) System** to handle customer inquiries, refunds and support requests.
- The system shall communicate with an **Accounting System** to log ticket sales, revenue tracking and tax calculations.
- The system shall interact with an **Email and SMS Notification System** to send booking confirmations, reminders and promotional offers.
- The system shall fetch **Movie Ratings and Reviews** from third-party services such as **Rotten Tomatoes, IMDB and Metacritic** to display critic reviews alongside movie listings.
- The system shall include a **Fraud Detection System** to prevent bots from mass-purchasing high-demand tickets and ensure ticket uniqueness.
- The system shall communicate with a **Data Analytics Platform** to track sales trends, customer behavior and system performance for business insights.
- The system shall provide an **API for third-party integration**, allowing external theater management systems to interact with ticketing and showtime data.
- The system shall incorporate **Secure Transaction Verification Software** (e.g., Verisign, SSL/TLS encryption) to protect user data and payment processing.

These software integrations ensure that the ticketing system operates efficiently, securely and provides a seamless experience for users and administrators alike.

3.1.4 Communications Interfaces

The Movie Theater Ticketing-System shall utilize various communication protocols to ensure secure, efficient and reliable data exchange between clients, servers and third-party integrations.

1. The system shall use the **HTTPS protocol** for secure communication over the internet, ensuring encrypted transactions and data protection.
2. The system shall utilize the **TCP/IP protocol suite** for internal network communication between the ticketing system servers, databases and third-party services.
3. The system shall support **WebSockets** for real-time seat selection and ticket availability updates, ensuring a seamless booking experience.
4. The system shall use **RESTful APIs** to integrate with external services such as payment gateways, movie review providers and CRM systems.
5. The system shall enable **SMTP and SMS gateways** to send transactional emails and text message confirmations for ticket purchases.
6. The system shall support **firewall-secured VPN communication** for administrative and backend management access to the system.
7. The system shall implement **load balancing and CDN (Content Delivery Network) services** to ensure optimal performance under high traffic conditions.

8. The system shall maintain a **high-priority queuing mechanism** for handling peak-hour ticket purchases and prevent system overload.
9. The system shall support **OAuth 2.0 authentication** for secure login and third-party authentication (Google, Facebook, Apple ID).
10. The system shall log and monitor all communication traffic through a **centralized logging and analytics platform** for security auditing and performance optimization.

These communication protocols ensure a robust, scalable and secure environment for all ticketing operations, user interactions and administrative tasks.

3.2 Functional Requirements

3.2.1 Ticket Purchasing and Reservation

3.2.1.1 Introduction

The system shall allow users to browse available movies, select showtimes, choose seats (for deluxe theaters) and complete ticket purchases online or at in-person kiosks.

3.2.1.2 Inputs

- User-selected movie and showtime
- User seat selection (if applicable)
- User authentication (if logged in)
- Payment details (credit card, PayPal, Bitcoin, etc.)

3.2.1.3 Processing

- Validate user input and seat availability in real-time
- Process payment securely via integrated payment gateways
- Generate and store digital tickets linked to user accounts
- Update seat availability for the selected showtime

3.2.1.4 Outputs

- Confirmation page displaying purchased ticket details
- Email and SMS notifications with ticket QR code
- Update of seat availability in real-time

3.2.1.5 Error Handling

- Display error messages for invalid payment or seat selection conflicts
- Prevent duplicate purchases of the same seat
- Auto-release seats after five minutes of inactivity in the booking process

3.2.2 Provide Movie and Show Details

3.2.2.1 Introduction

The system shall display all available movies, their descriptions, trailers, ratings and user reviews.

3.2.2.2 Inputs

- Movie database information
- Third-party reviews (IMDB, Rotten Tomatoes)

3.2.2.3 Processing

- Fetch and display movie details dynamically
- Sync ratings and reviews from external sources

3.2.2.4 Outputs

- List of currently available movies and upcoming releases
- Movie details, including trailer, showtimes and ratings

3.2.2.5 Error Handling

- Handle missing data by displaying default placeholders
- Show error messages if external rating services are unavailable

3.2.3 User Profile Management

3.2.3.1 Introduction

The system shall allow users to create and manage profiles for personalized ticket booking and history tracking.

3.2.3.2 Inputs

- User registration details (name, email, password)
- Payment information (optional for faster checkout)

3.2.3.3 Processing

- Validate user credentials and store profile information securely
- Sync ticket purchase history to user profile

3.2.3.4 Outputs

- Display user’s booking history
- Provide saved payment methods for future use

3.2.3.5 Error Handling

- Prevent duplicate account creation with the same email
- Display errors for incorrect login credentials

3.2.4 Customer Support and Ticket Modifications

3.2.4.1 Introduction

The system shall allow users to contact support and modify or cancel tickets before the showtime.

3.2.4.2 Inputs

- User-selected order modification request
- Refund reason (if applicable)

3.2.4.3 Processing

- Check eligibility of ticket modification or refund
- Notify in-person support staff for manual refund processing

3.2.4.4 Outputs

- Confirmation page for successful modification
- Refund status updates for eligible cancellations

3.2.4.5 Error Handling

- Prevent modifications after the cancellation deadline
- Display customer support contact if automated changes are not possible

3.2.5 Payment Processing

3.2.5.1 Introduction

The system shall support multiple payment methods, including credit cards, PayPal and Bitcoin.

3.2.5.2 Inputs

- User-selected payment method
- Payment details

3.2.5.3 Processing

- Securely transmit payment data to third-party gateways
- Verify payment success before ticket issuance

3.2.5.4 Outputs

- Payment confirmation page
- Email and SMS receipt

3.2.5.5 Error Handling

- Display error messages for failed transactions
- Suggest alternative payment methods if needed

3.2.6 Promotions and Discounts

3.2.6.1 Introduction

The system shall provide discount options for students, military personnel and seniors.

3.2.6.2 Inputs

- User-selected discount type
- Proof of eligibility (if required)

3.2.6.3 Processing

- Validate discount eligibility and adjust final ticket price

3.2.6.4 Outputs

- Discount applied confirmation
- Updated payment amount

3.2.6.5 Error Handling

- Display error if discount eligibility is not verified

3.2.7 Customer Reviews and Ratings

3.2.7.1 Introduction

The system shall allow users to submit reviews and ratings for movies they have watched.

3.2.7.2 Inputs

- User-submitted review text and star rating
- Movie identification (linked to purchased ticket)

3.2.7.3 Processing

- Validate that the user has attended the movie before submitting a review
- Store and associate the review with the movie database

3.2.7.4 Outputs

- Display of aggregated ratings and user reviews on the movie detail page
- Confirmation message after submitting a review

3.2.7.5 Error Handling

- Prevent users from submitting multiple reviews for the same movie
- Display error if a review violates content policies

3.2.8 System wide Ticket Purchase Logging

3.2.8.1 Introduction

The system shall maintain systemwide logs of ticket purchases for auditing and analytics purposes.

3.2.8.2 Inputs

- User transaction details
- Purchase timestamp
- Payment method used

3.2.8.3 Processing

- Store transaction logs in a secure and centralized database
- Generate reports based on ticket sales trends and system activity

3.2.8.4 Outputs

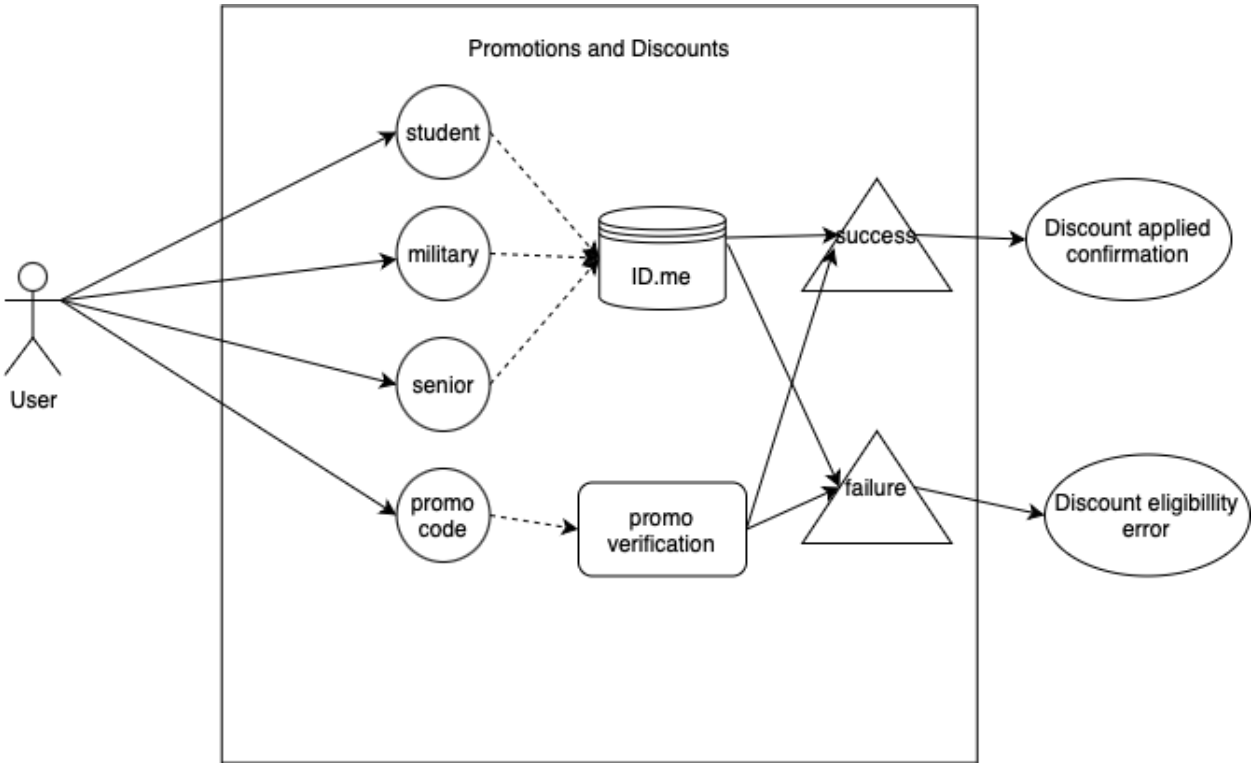
- Access logs for administrators for auditing purposes
- Sales reports for theater managers

3.2.8.5 Error Handling

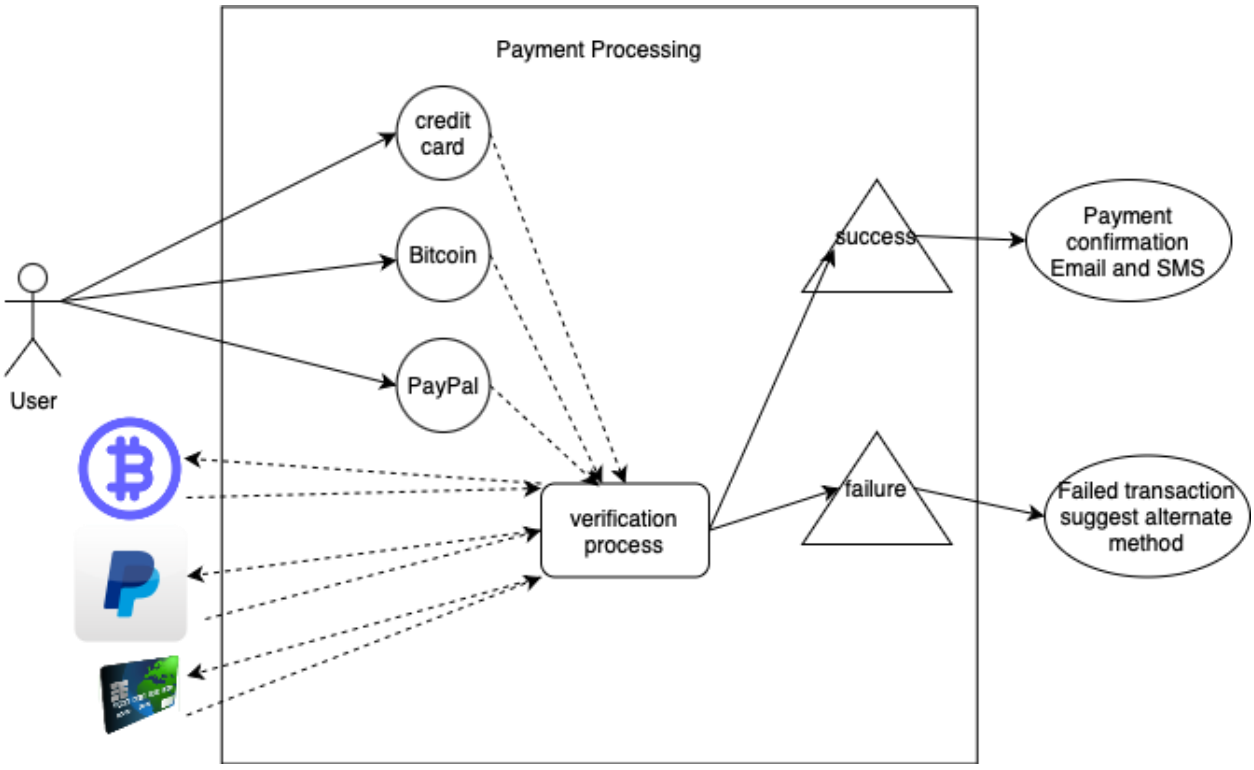
- Handle database connection failures by retrying the log storage process
- Prevent unauthorized access to purchase logs

3.3 Use Cases

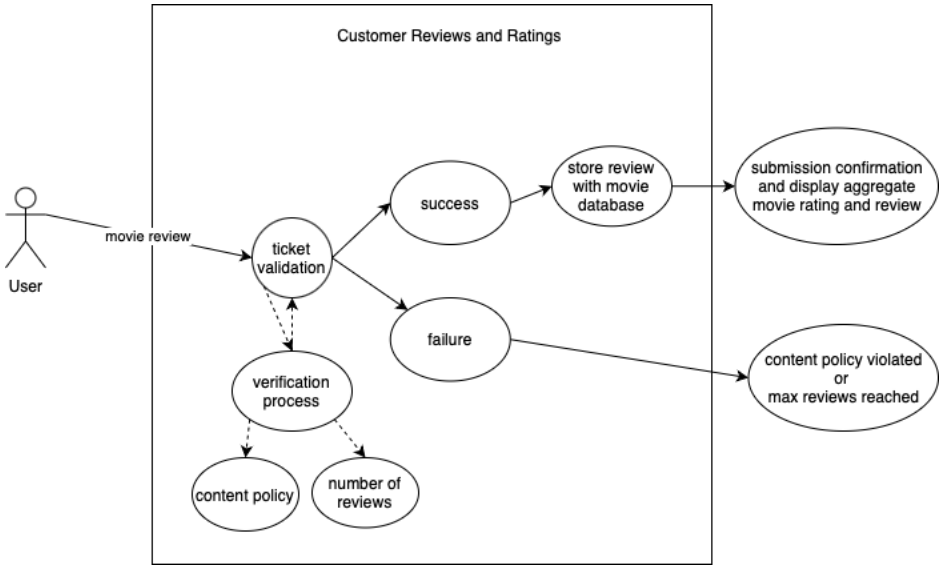
3.3.1 Use Case #1: Promotions and Discounts



3.3.2 Use Case #2: Payment Processing

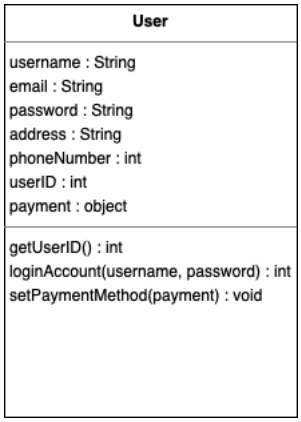


3.3.2 Use Case #3: Customer Reviews and Ratings

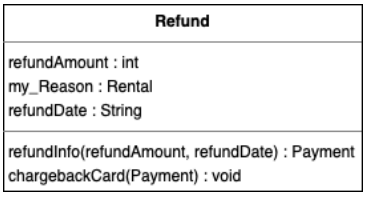


3.4 Classes / Objects

3.4.1 Class / Object #1: User



3.4.2 Class / Object #2: Refund



3.5 Non-Functional Requirements

3.5.1 Performance

- The system shall process **95% of transactions within 1 second** during peak hours.

- The system shall support **up to 100,000 concurrent users** without degradation in response time.
- The system shall handle **at least 1000 concurrent seat selections** for a single movie screening without performance bottlenecks.
- The system shall ensure **database queries execute within 500 milliseconds** under normal load.

3.5.2 Reliability

- The system shall have a **Mean Time Between Failures (MTBF) of at least 30 days**.
- The system shall provide automated failover mechanisms to minimize disruptions in case of server failures.
- Daily backups shall be performed, with **recovery point objectives (RPO) of 5 minutes** to prevent significant data loss.

3.5.3 Availability

- The system shall maintain **99.99% uptime**, allowing downtime of no more than **5 minutes per month**.
- The system shall provide real-time monitoring and alerting mechanisms for service disruptions.
- Redundant cloud-based servers shall ensure seamless operation in case of data center failures.

3.5.4 Security

- All data transactions shall be encrypted using **SSL/TLS (AES-256 encryption)**.
- The system shall support **two-factor authentication (2FA)** for administrator access.
- Only one device may be logged into a user account at a time to **prevent account sharing and fraud**.
- Tickets must be **unique and non-replicable**, utilizing NFT-based validation to prevent counterfeiting.
- The system shall store all passwords using **bcrypt hashing** with a minimum cost factor of 12.
- Security logs shall be stored and **audited weekly** for potential breaches.

3.5.5 Maintainability

- The system shall allow updates and patches to be applied **without downtime** using a **rolling update mechanism**.
- The system's **codebase shall follow modular architecture**, allowing individual components to be updated without affecting the whole system.
- The system shall provide **automated test coverage of at least 90%** for critical components.

3.5.6 Portability

- The system shall be accessible via **Google Chrome, Mozilla Firefox, Safari and Microsoft Edge**.
- The system shall run on **Windows, macOS and Linux** without requiring modifications.
- The system shall be deployable to **AWS, Azure and Google Cloud** with minimal configuration changes.

3.5.7 Usability

- The system shall provide an **intuitive and user-friendly interface**, allowing ticket purchases to be completed in **three clicks or fewer**.
- The system shall follow **WCAG 2.1 accessibility guidelines**, ensuring usability for users with disabilities.
- The ticket selection and checkout process shall take **less than 2 minutes** under normal conditions.
- The system shall provide **real-time feedback** (e.g., validation messages, loading indicators) to guide users through the purchase process.
- The interface shall be designed with a **consistent and modern UI**, maintaining visual clarity across all pages.
- The system shall offer a **Dark Mode** and customizable color schemes for better accessibility.
- The system shall support **voice commands** for searching and ticket selection.
- The system shall provide an **auto-seat selection feature** that suggests the best available seats if the user does not choose manually.

3.5.8 Performance Enhancements

- The system shall dynamically scale its **server resources** to handle traffic spikes and high demand.
- The system shall implement **content caching and preloading** for frequently accessed data to reduce loading times.
- The system shall support an **offline mode for theater kiosks**, allowing ticket purchases to be temporarily stored and synchronized when online connectivity is restored.

3.5.9 Security Enhancements

- The system shall detect and block **fraudulent activity**, such as mass ticket purchases from bots, using AI-based detection.
- The system shall prevent **double bookings** by confirming seat availability in real-time before finalizing transactions.
- The system shall automatically **lock accounts** after multiple failed login attempts and require identity verification.

3.5.10 Maintainability & Support

- The system shall include **self-healing mechanisms**, capable of restarting failed components without manual intervention.
- The system shall provide an **administrative dashboard** with real-time metrics, including server health, ticket sales and system errors.
- The system shall integrate a **live customer support chat** with both AI and human assistance options.
- The system shall provide an **intuitive and user-friendly interface**, allowing ticket purchases to be completed in **three clicks or fewer**.
- The system shall follow **WCAG 2.1 accessibility guidelines**, ensuring usability for users with disabilities.
- The ticket selection and checkout process shall take **less than 2 minutes** under normal conditions.
- The system shall provide **real-time feedback** (e.g., validation messages, loading indicators) to guide users through the purchase process.
- The interface shall be designed with a **consistent and modern UI**, maintaining visual clarity across all pages.
- The system shall be accessible via **Google Chrome, Mozilla Firefox, Safari and Microsoft Edge**.
- The system shall run on **Windows, macOS and Linux** without requiring modifications.
- The system shall be deployable to **AWS, Azure and Google Cloud** with minimal configuration changes.

3.5.11 Supportability

3.5.11.1 Configuration Management Tool

- The system's source code shall be maintained in a **Git-based version control system (GitHub/GitLab/Bitbucket)**.
- Every change must go through **automated CI/CD pipelines** to ensure stability before deployment.

3.5.12 On-line User Documentation and Help System

- The system shall include a **self-service help center** with FAQs, troubleshooting guides and video tutorials.
- The system shall provide **context-sensitive help**, allowing users to access relevant information directly from the UI.

3.5.13 Licensing Requirements

- The system shall comply with all **regional data protection laws (GDPR, CCPA, etc.)**.
- Any third-party software components shall be **open-source or properly licensed**.

3.5.14 Legal, Copyright and Other Notices

- The system shall display **copyright and trademark notices** on all pages where applicable.
- All user-generated content (reviews, feedback) shall be moderated according to **community guidelines**.

3.5.15 Applicable Standards

- The system shall adhere to **IEEE 830-1998 SRS guidelines**.
- The system shall comply with **PCI-DSS requirements** for handling payment transactions.

3.6 Inverse Requirements

This section outlines features and functionalities that the Movie Theater Ticketing-System will **not** support, to clarify system limitations and avoid misunderstandings.

- The system will not support paperless-only ticketing.
  Customers can choose between **digital and printed tickets**; there is no requirement for exclusive digital usage.

- The system will not support telephone-based ticket purchases.
  All ticket transactions must be completed through the **web application or theater kiosks**.

- The system will not allow unlimited seat reservations without payment.
  Reserved seats will be automatically released after **5 minutes of inactivity** if payment is not completed.

- The system will not support ticket purchases without user authentication.
  Customers must be logged in via **a registered account or OAuth authentication (Google, Facebook, etc.)**.

- The system will not process refunds online.
  Refunds will only be handled **in-person by theater support staff**.

- The system will not provide group discounts.
  Only **student, military and senior discounts** will be available; there are no discounts for group bookings.

- The system will not allow seat modifications after purchase.
  Customers must book a new ticket if they wish to change their seat after completing the transaction.

- The system will not allow ticket transfers to other users.
  Tickets are tied to the purchaser and cannot be reassigned to another person.

3.7 Design Constraints

This section defines the design constraints for the Movie Theater Ticketing-System, ensuring compliance with industry standards, performance expectations and security requirements.

3.7.1 Standard Development Tools

- The system shall be developed using **React.js for frontend** and **FastAPI/Node.js for backend**.
- The database shall use **PostgreSQL or MySQL**, ensuring **ACID compliance** and scalability.
- The system shall follow a **Microservices architecture** for flexibility and modularity.
- **Docker containers** shall be used for all services to facilitate **deployment and scalability**.
- **Code quality shall be enforced** using ESLint for JavaScript and PEP8 for Python.
- The system shall include **automated CI/CD pipelines** for integration, testing and deployment.

3.7.2 Web-Based Product

- The system shall be **entirely web-based**, with no requirement for standalone mobile or desktop applications.
- The web client must be **fully responsive**, supporting **mobile, tablet and desktop** devices.
- The web application shall **load within 3 seconds** on standard broadband connections.
- The system shall be optimized for **low-bandwidth environments**, ensuring functionality on **3G networks**.
- **Progressive Web App (PWA) support** shall be implemented for **offline access** in theater kiosks.

3.7.3 Security and Compliance

- The system shall adhere to **PCI-DSS standards** for secure payment processing.
- All data transmissions shall be **encrypted using TLS 1.2+ and AES-256 encryption**.
- No sensitive user data shall be stored in browser cookies or client-side storage.
- The system must be compliant with **GDPR and CCPA data privacy regulations**.
- **Multi-factor authentication (MFA)** shall be required for administrator logins.
- Access logs shall be retained for **six months** for auditing and security reviews.

3.7.4 Scalability and Infrastructure

- The system shall be deployed on **cloud infrastructure (AWS, Azure or Google Cloud)**.
- It must support **Auto-Scaling** to handle high traffic loads during peak hours.
- A **load balancer** shall distribute user traffic across multiple servers.
- **Content Delivery Networks (CDN)** shall be used to optimize asset loading speeds.

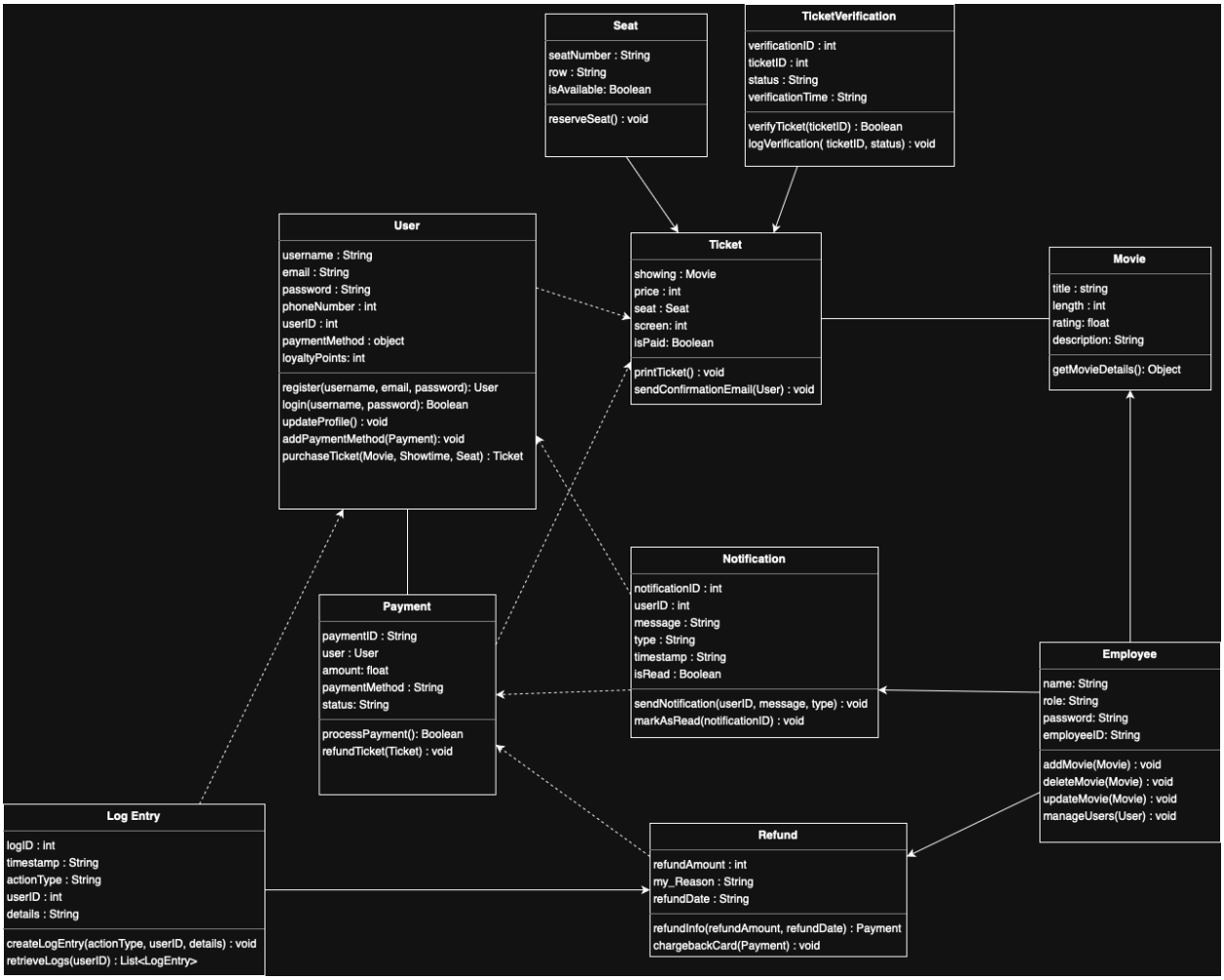
3.7.5 Hardware Compatibility

- The system must support **theater kiosks** equipped with **touchscreens and barcode scanners**.
- Ticket validation must be compatible with **QR-code scanning devices**.
- The system shall function efficiently on **theater box office terminals with limited processing power**.

3.7.6 Legal and Licensing Compliance

- The system shall comply with **regional taxation regulations**, ensuring correct ticket price calculations.
- Any third-party integrations shall comply with **open-source licensing policies**.
- **Copyright and trademark notices** must be displayed on all relevant pages.
- The system shall provide a **term of service and privacy policy agreement** for users before account registration.

3.7.7 UML Diagram



3.7.8 Description of Classes, Attributes and Methods/Operations

3.7.8.1 User Class

Definition: Represents a registered customer using the system. This class is connected to the **Ticket** and **Payment** classes. It also interacts with the **Refund** class when ticket cancellations or refunds are requested.

Attributes:
- username: String - Unique identifier for the user.
- email: String - Email address for communication.
- password: String - Encrypted password for authentication.
- phoneNumber: String - Contact number for notifications.
- paymentMethod: Object - Stores payment details.
- loyaltyPoints: int - Tracks user's reward points.

Methods:
- register(username, email, password): User - Creates a new user account.
- login(username, password): Boolean - Authenticates user login.
- updateProfile(): void - Updates user profile.
- addPaymentMethod(Payment): void - Stores new payment method.
- purchaseTicket(Movie, Seat): Ticket - Reserves a seat and completes payment.

3.7.8.2 Employee Class

Definition: Represents theater employees who manage operations. This class is connected to the **Movie** classes, allowing employees to manage content. It also interacts with **Refund** for customer service support.

Attributes:
- employeeID: String - Unique employee identifier.
- password: String - Encrypted password for authentication.
- name: String - Employee's full name.
- role: String - Role within the theater (Admin, Manager, Staff).

Methods:
- addMovie(Movie): void - Adds a new movie to the schedule.
- updateMovie(Movie): void - Modifies movie schedules.
- deleteMovie(Movie) : void - Deletes a movie from the schedule.
- manageUsers(User): void – Manages customer accounts.

3.7.8.3 Movie Class

Definition: Represents movies available for viewing. This class is connected to **Ticket** class, linking each movie to scheduled screenings and ticket purchases.

Attributes:
- title: String - Movie title.
- length: int - Duration of the movie in minutes.
- rating: float - IMDB/Rotten Tomatoes rating.
- description: String - Movie synopsis.

Methods:
- getMovieDetails(): Object Fetches detailed movie info.

3.7.8.4 Ticket Class

Definition: Represents a purchased ticket. This class is connected to **User** and **Payment**, ensuring that purchases are linked to a specific movie session.

Attributes:
- ticketID: String - Unique identifier for the ticket.
- seat: Seat - Assigned seat.
- screen : int - screen where movie is shown.
- price: float - Cost of the ticket.
- isPaid: Boolean - Payment status.

Methods:
- printTicket(): void - prints ticket for validation.
- sendConfirmationEmail(User): void - Sends ticket receipt.

3.7.8.5 Payment Class

Definition: Handles financial transactions. This class is connected to **User**, **Ticket** and **Refund**, managing payments and handling refunds if needed.

Attributes:
- paymentID: String - Unique payment identifier.
- user: User - Associated user.
- amount: float - Transaction amount.
- paymentMethod: String - Credit Card, PayPal, Bitcoin.
- status: String - Pending, Completed, Failed.

Methods:
- processPayment(): Boolean - Confirms payment.
- refundTicket(Ticket): void - Issues refund if applicable.

3.7.8.6 Seat Class

Definition: Represents an individual seat in a theater.

Attributes:
- seatNumber: String - Identifier of the seat.
- row: String - Row in which the seat is located.
- isAvailable: Boolean - Availability status.

Methods:
- reserveSeat(): void – Marks the seat as taken.

3.7.8.7 Refund Class

Definition: The Refund class handles ticket refunds for users who cancel or modify their reservations before the allowed deadline. It ensures that refund requests are processed correctly and linked to the appropriate payment transactions. The Refund class is directly connected to the **Payment** class to process financial transactions.

Attributes:
- refundAmount: int – The total amount refunded to the user.
- my_Reason: Stringi – The reason for the refund, usually linked to a canceled or modified ticket.
- refundDate: String – The date when the refund was processed.

Methods:
- refundInfo(refundAmount: int, refundDate: String) : Payment - Retrieves details of a specific refund, including the amount and date.
- chargebackCard(payment: Payment) : void - Processes the chargeback and returns the refunded amount to the original payment method.

3.7.8.8 LogEntry Class

Definition: The LogEntry Class represents a system log entry that tracks important actions performed by users and employees. The LogEntry class is connected to the User, Employee, and Payment classes to log account actions, ticket purchases, refunds, and other significant activities.

Attributes:
- logID: int – Unique identifier for each log entry.
- timestamp: String – The exact time when the action occurred.
- actionType: String – Describes the type of action (e.g., "Ticket Purchase", "Refund Request", "Payment Failure").
- userID: int – The ID of the user or employee who performed the action.
- details: String – Additional information regarding the action.

Methods:
- createLogEntry(actionType, userID, details): void – Creates a new log entry and stores it in the system.
- retrieveLogs(userID): List<LogEntry> – Retrieves all log entries associated with a specific user for auditing and debugging.

3.7.8.9 Notification Class

Definition: Handles user notifications for ticket purchases, refunds, and reminders. The Notification class is linked to the User and Refund classes to send alerts regarding transactions, upcoming movies, and refund confirmations.

Attributes:
- notificationID: int – Unique identifier for each notification.
- userID: int – The ID of the user receiving the notification.
- message: String – The content of the notification.
- type: String – Type of notification (e.g., "Ticket Confirmation", "Refund Approved", "Showtime Reminder").
- timestamp: String – Time when the notification was sent.
- isRead: Boolean – Indicates whether the user has read the notification.

Methods:
- sendNotification(userID, message, type): void – Sends a notification to the specified user.
- markAsRead(notificationID): void – Marks a notification as read.

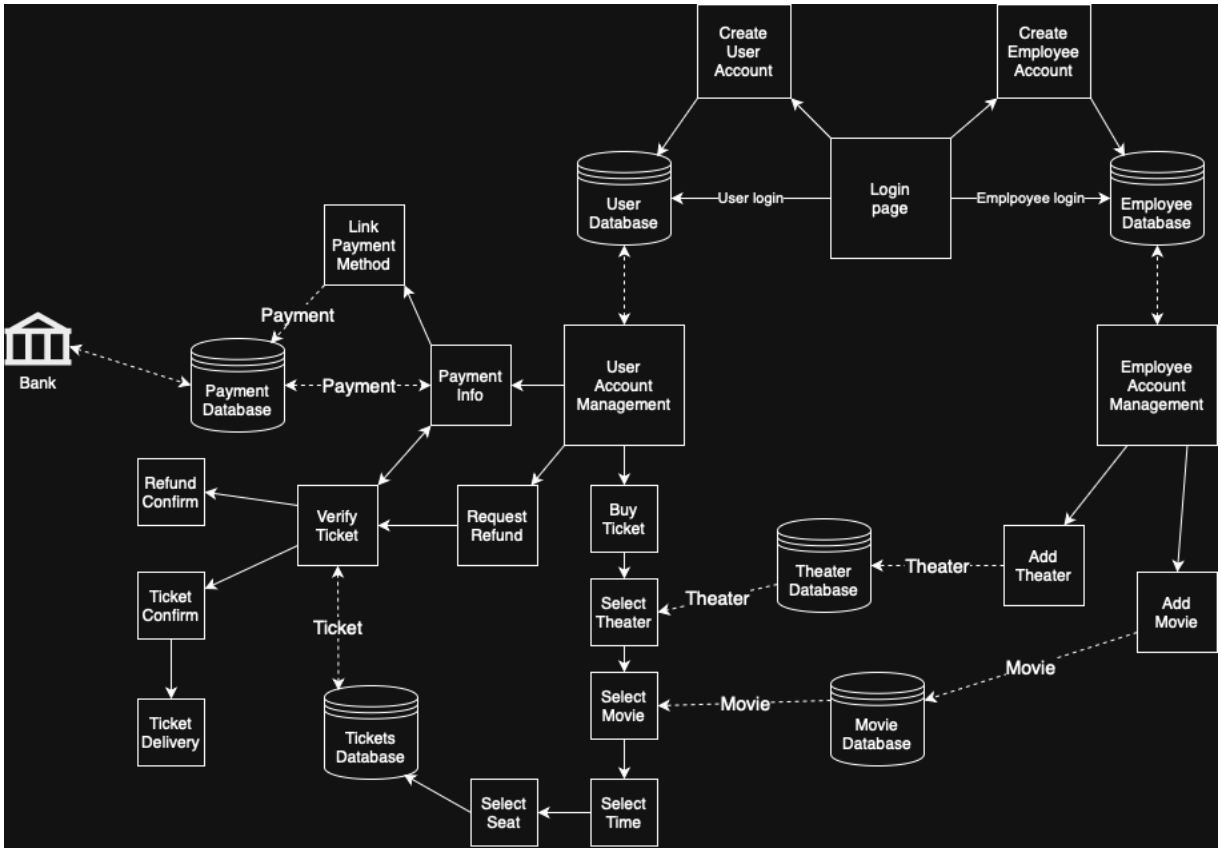
3.7.8.10 TicketVerification Class

Definition: Verifies the validity of purchased tickets when customers arrive at the theater. The TicketVerification class is associated with the Ticket class and ensures that only legitimate tickets are used.

Attributes:
- verificationID: int – Unique identifier for each ticket verification attempt.
- ticketID: int – The ID of the ticket being verified.
- status: String – Indicates whether the ticket is valid, expired, or already used.
- verificationTime: String – Timestamp of when the ticket was verified.

Methods:
- verifyTicket(ticketID): Boolean – Checks if the ticket is valid for entry.
- logVerification(ticketID, status): void – Logs the verification attempt for tracking purposes.

3.7.9 Software Architecture Diagram



3.7.9.1 Description and Key Components of Software Architecture Diagram

1. **User Account Management**
   - Allows users to create accounts, log in and manage their personal information.
   - Stores user credentials and preferences in the User Database.
   - Supports authentication via OAuth 2.0 (Google, Facebook) for secure login.

2. **Movie and Theater Management**
   - The **Movie Database** contains all movie details, including showtimes, durations and ratings.
   - The **Theater Database** stores theater locations, available screens and seating arrangements.
   - Employees can manage movie schedules, add or remove movies and update theater details via the **Employee Account Management System**.

3. **Ticket Purchasing and Management**
   - Users can **browse movies**, **select theaters**, **choose showtimes** and **reserve seats** in real-time.
   - The **Tickets Database** maintains the availability of seats to prevent overbooking.

- Digital tickets are issued with unique **QR codes**, stored in the user's profile and can be printed or scanned at entry points.

4. **Payment and Refund Processing**
   - The system integrates with external payment gateways such as **Stripe, PayPal and Bitcoin transactions** for secure payments.
   - Payment information is securely stored in the **Payment Database**, linked to the user's profile.
   - Refund requests can be initiated if users meet the eligibility criteria (e.g. show cancellation or valid refund reason).
   - The **Bank API** processes refunds and transaction verifications.
   - The **Refund System** ensures that only authorized refunds are processed, preventing fraud.

5. **Ticket Verification and Entry Management**
   - A **Verification System** cross-checks purchased tickets against the **Tickets Database**.
   - Digital and printed tickets are validated via **QR Code Scanners** at the theater entrance.
   - If a ticket is marked as already used, the system denies access to prevent duplication.

6. **Scalability and Performance Optimization**
   - Built using a **microservices architecture** to ensure modularity and easy maintenance.
   - Uses **load balancing and caching mechanisms** (e.g. Redis) to handle high traffic demands.
   - The architecture is designed to support **up to 100,000 concurrent users**.

7. **Customer Support and Feedback System**
   - Users can contact **customer support** via an in-app chat or email system.
   - After purchasing tickets, users can submit **movie reviews and ratings** pulled from **Rotten Tomatoes and IMDB APIs**.
   - A **feedback system** allows customers to rate their experience using a simple smiley-face selection.

8. **Administrative and Employee Functions**
   - Employees can access an **Admin Dashboard** to manage movies, theaters and user issues.
   - Administrators can **override transactions**, adjust movie schedules and provide manual refunds.
   - Employees can **monitor ticket sales**, view analytics and track system activity for **business insights**.

3.7.10 Development Plan and Timeline

Elias
- UML Class Diagram
- Software Architecture Diagram

Tibelya
- Description of UML Class Diagram
- Explanation of Class Attributes and Operations

Cecilia
- Description of Software Architecture Diagram
- Explanation of SAD and Refinement and Documentation

PARTITIONING OF TASKS

3/01 - 3 hours
- Brainstorming session: Elias, Tibelya, Cecilia

3/02 - 3/04 (3 Days)
- Creation of UML Class Diagram and Software Architecture Diagram: Elias

3/05
- Writing the description of UML Class Diagram: Tibelya

3/05
- Writing the description of Software Architecture Diagram: Cecilia

3.8 Test Plan

3.8.0 Link to GitHub-Testcases

https://github.com/Ediaz047/CS250-Group-4/blob/5847bdd36d8138acec9eb9e7cbb28499602869a7/testCases_Group4.xlsx

3.8.1. Overview of Testing Strategy

The test plan aims to verify and validate the functionality, performance, and security of the Movie Theater Ticketing System. Our testing approach ensures that critical user workflows operate as expected, potential failure points are identified, and the system meets its functional and non-functional requirements.

The main focus of our testing strategy includes:
- **User Authentication**: Ensuring secure login and account management.
- **Ticket Purchasing**: Verifying seat selection, pricing, discounts, and payment processing.
- **Refund and Cancellation Process**: Ensuring smooth refund requests and transaction reversals.
- **Concurrency Handling**: Testing system performance under heavy load.

- **Security Measures**: Validating CAPTCHA protection and fraud detection.

3.8.2. Test Categories and Coverage

To ensure full test coverage, we categorize our tests into **unit tests, functional tests, and system tests**.

3.8.2.1 Login_01 - User Authentication

This test ensures that users can successfully log into the system using valid credentials while handling incorrect login attempts.

Unit Testing:

- **User.loginAccount(username, password): Boolean**

Test 1:

- **Input:** ("john_doe", "CorrectPassword123")
- **Expected Output:** Authentication successful

Test 2:

- **Input:** ("john_doe", "WrongPassword")
- **Expected Output:** "Invalid credentials. Please try again."

Integration Testing:

- **Session.createSession(User): void**

Test 1:

- **Input:** Valid user credentials
- **Expected Output:** New session created

Test 2:

- **Input:** Invalid credentials
- **Expected Output:** No session created, login prompt displayed

System Testing:

- **LoginPage.authenticate(User): Boolean**

Test 1:

- **Input:** ("john_doe", "CorrectPassword123")
- **Expected Output:** User redirected to homepage

Test 2:

- **Input:** ("john_doe", "WrongPassword")
- **Expected Output:** Error message: "Invalid login details"

3.8.2.2 SeatSelection_02 - Seat Reservation

This test ensures users can select and reserve seats in a theater while preventing double booking.

Unit Testing:
- **Seat.reserveSeat(): void**

Test 1:
- **Input:** (Seat #15, Row B)
- **Expected Output:** Seat successfully reserved

Test 2:
- **Input:** (Already reserved seat)
- **Expected Output:** "Seat unavailable. Please choose another."

Integration Testing:
- **TicketingSystem.validateSeatSelection(User, Seat): Boolean**

Test 1:
- **Input:** (User selects an available seat)
- **Expected Output:** Seat confirmation displayed

Test 2:
- **Input:** (User selects an occupied seat)
- **Expected Output:** Error message: "Seat is taken."

System Testing:
- **BookingPage.confirmSeatSelection(User, Seat): Boolean**

Test 1:
- **Input:** Seat selection and payment confirmation
- **Expected Output:** "Booking complete."

Test 2:
- **Input:** Payment failure
- **Expected Output:** "Payment unsuccessful. Seat not reserved."

3.8.2.3 Payment_03 - Ticket Payment Processing

Tests whether the payment system processes transactions correctly.

Unit Testing:
- **Payment.processPayment(User, amount, method): Boolean**

Test 1:
- **Input:** (Valid credit card, $20)
- **Expected Output:** "Payment successful."

Test 2:
- **Input:** (Expired credit card)
- **Expected Output:** "Payment declined."

Integration Testing:
- **Checkout.processTransaction(Payment): Boolean**

Test 1:
- **Input:** (Valid payment details)
- **Expected Output:** Transaction recorded in system

Test 2:
- **Input:** (Invalid payment details)
- **Expected Output:** Payment failure message

System Testing:
- **PaymentGateway.processExternalTransaction(Payment): Boolean**

Test 1:
- **Input:** (User makes a valid purchase)
- **Expected Output:** "Ticket purchase confirmed."

Test 2:
- **Input:** (User enters an invalid payment method)
- **Expected Output:** "Transaction failed."

3.8.2.4 Refund_04 - Ticket Refund Handling

Tests the refund functionality of the system.

Unit Testing:
- **Refund.processRefund(Ticket, Reason): Boolean**

Test 1:
- **Input:** (Valid ticket, "Event canceled")
- **Expected Output:** "Refund issued."

Test 2:
- **Input:** (Expired refund window)
- **Expected Output:** "Refund request denied."

Integration Testing:
- **Transaction.updateTransactionStatus(Refund): Boolean**

Test 1:
- **Input:** Valid refund request
- **Expected Output:** Status changed to "Refunded"

Test 2:
- **Input:** Invalid refund request
- **Expected Output:** Status remains "Completed"

System Testing:
- **CustomerSupport.handleRefundRequest(User, Ticket): Boolean**

Test 1:
- **Input:** Valid refund request
- **Expected Output:** Customer receives refund confirmation

Test 2:
- **Input:** Invalid refund request
- **Expected Output:** Customer receives refund denial notice

3.8.2.5 LoadTest_05 - System Performance Under Load

Ensures that the system handles a high number of simultaneous users.

Unit Testing:
- **System.checkResponseTime(): float**

Test 1:
- **Input:** (100 concurrent users)
- **Expected Output:** Response time < 2 sec

Test 2:
- **Input:** (10,000 concurrent users)
- **Expected Output:** Response time < 5 sec

Integration Testing:
- **Server.loadBalancer(): Boolean**

Test 1:
- **Input:** (Peak usage scenario)
- **Expected Output:** No server crash

Test 2:
- **Input:** (Mass ticket purchases)
- **Expected Output:** System slows but remains functional

System Testing:
- **StressTest.simulateHighTraffic(): Boolean**

Test 1:
- **Input:** (10,000 users logging in simultaneously)
- **Expected Output:** "System operational"

Test 2:
- **Input:** (100,000 concurrent purchases)
- **Expected Output:** "System under load, but responsive."

3.8.2..6 Database_06 - Data Integrity Testing

Ensures the database stores and retrieves information correctly.

Unit Testing:
- **Database.storeRecord(User): Boolean**

Test 1:
- **Input:** (New user registration)
- **Expected Output:** "User stored successfully."

Test 2:
- **Input:** (Duplicate user entry)
- **Expected Output:** "Error: User already exists."

Integration Testing:
- **Database.fetchRecord(Ticket): Boolean**

Test 1:

- **Input:** (Valid ticket ID)
- **Expected Output:** Ticket details displayed

Test 2:

- **Input:** (Non-existent ticket ID)
- **Expected Output:** "Ticket not found."

System Testing:

- **BackupSystem.restoreDatabase(): Boolean**

Test 1:

- **Input:** (Full system restore)
- **Expected Output:** All records correctly retrieved

Test 2:

- **Input:** (Partial restore)
- **Expected Output:** Some records missing, error displayed

3.8.2.7. SessionTimeout_07 - Automatic Logout Due to Inactivity

This test ensures that users are logged out after a period of inactivity to enhance security and free up system resources.

Unit Testing:

- **Session.checkActivityTimeout(): Boolean**

Test 1:

- **Input:** (User inactive for 4 minutes)
- **Expected Output:** Session remains active

Test 2:

- **Input:** (User inactive for 6 minutes)
- **Expected Output:** "Session expired. Please log in again."

Integration Testing:

- **SessionManager.terminateInactiveSession(Session): void**

Test 1:

- **Input:** (Session timeout trigger)
- **Expected Output:** User is logged out, redirected to login page

Test 2:

- **Input:** (Session timeout bypass attempt)
- **Expected Output:** System prevents unauthorized session persistence

System Testing:

- **SecurityPolicy.enforceSessionTimeout(): Boolean**

Test 1:

- **Input:** (Multiple users inactive beyond timeout threshold)
- **Expected Output:** All inactive users logged out

Test 2:

- **Input:** (User attempts actions after timeout)
- **Expected Output:** "Session expired. Please log in again."

3.8.2.8 Discounts_08 - Validation of Ticket Discounts

This test verifies that applicable discounts (student, military, senior) are correctly applied during checkout.

Unit Testing:
- **Discount.applyDiscount(User, Ticket): float**

Test 1:
- **Input:** (Student discount on $15 ticket)
- **Expected Output:** "$12.75 applied (15% discount)"

Test 2:
- **Input:** (Invalid discount code)
- **Expected Output:** "Invalid discount applied"

Integration Testing:
- **PaymentSystem.validateDiscount(Payment, Discount): Boolean**

Test 1:
- **Input:** (Valid student discount)
- **Expected Output:** Discounted price reflected in checkout

Test 2:
- **Input:** (Military discount applied incorrectly)
- **Expected Output:** Error message: "Invalid discount eligibility."

System Testing:
- **BookingSystem.processFinalPrice(Ticket, Discount): Boolean**

Test 1:
- **Input:** (Multiple discount applications)
- **Expected Output:** Only one discount applied, correct total displayed

Test 2:
- **Input:** (Senior discount on non-eligible user)
- **Expected Output:** "User does not qualify for discount."

3.8.2.9 EmailConfirmation_09 - Ticket Purchase Email Notification

This test checks whether users receive email confirmations after booking a ticket.

Unit Testing:
- **EmailService.sendConfirmationEmail(User, Ticket): Boolean**

Test 1:
- **Input:** (Valid email, successful ticket purchase)
- **Expected Output:** "Confirmation email sent successfully."

Test 2:
- **Input:** (Invalid email format)
- **Expected Output:** "Failed to send email."

Integration Testing:
- **NotificationSystem.processEmailQueue(): Boolean**

Test 1:
- **Input:** (Large volume of confirmation emails)
- **Expected Output:** Emails sent without delay

Test 2:
- **Input:** (Email server failure)
- **Expected Output:** "Email service temporarily unavailable."

System Testing:
- **UserAccount.retrieveEmailConfirmation(): Boolean**

Test 1:
- **Input:** (User checks inbox after booking)
- **Expected Output:** Email with correct ticket details received

Test 2:
- **Input:** (User requests re-send confirmation email)
- **Expected Output:** "Email successfully re-sent."

3.8.2.10 MultiDeviceLogin_10 - Handling of Concurrent Logins

This test verifies that a single user account cannot be logged in from multiple devices at the same time.

Unit Testing:
- **Session.verifySingleActiveSession(User): Boolean**

Test 1:
- **Input:** (User logs in on a second device)
- **Expected Output:** First session is logged out

Test 2:
- **Input:** (User logs in from multiple devices at the same time)
- **Expected Output:** "You are already logged in from another device."

Integration Testing:
- **AuthenticationSystem.handleMultiSession(User): Boolean**

Test 1:
- **Input:** (User logs in on desktop while active on mobile)
- **Expected Output:** One session remains active, the other is logged out

Test 2:
- **Input:** (User attempts login bypass)
- **Expected Output:** System prevents duplicate session

System Testing:
- **SecurityPolicy.enforceSessionLimit(User): Boolean**

Test 1:
- **Input:** (High volume of users attempting simultaneous logins)

- **Expected Output:** Only one session per user allowed

Test 2:

- **Input:** (Admin allows multi-session override)
- **Expected Output:** Special permissions granted for authorized accounts

3.8.3 Test Approach & Execution

Tests will be executed using a combination of **manual testing and automated test scripts**:

- **Unit Tests**: Automated using Jest for frontend logic and PyTest for backend validation.
- **Functional Tests**: Semi-automated using Selenium for UI validation and Postman for API tests.
- **System Tests**: Conducted using load-testing tools such as Apache JMeter to simulate concurrent users.

Test execution phases:

1. **Development Testing** (Before deployment, local environment)
2. **Pre-Production Testing** (Testing on a staging environment)
3. **Final Validation** (Before system release)

3.8.4 Error Scenarios & Failure Handling

We will verify the system's resilience to failures by testing the following edge cases:

- **Invalid Inputs**: Submitting empty or incorrect data fields.
- **Concurrency Conflicts**: Two users attempting to book the same seat simultaneously.
- **Payment Failures**: Processing a transaction with insufficient funds or expired credit cards.
- **System Overload**: Handling a spike in users without system crashes.
- **Unauthorized Access**: Preventing non-admins from modifying ticket prices or availability.

3.8.5 Conclusion

This test plan provides a structured and comprehensive approach to verifying the Movie Theater Ticketing System. By covering unit, functional, and system-level tests, we ensure robust validation of all key functionalities. Additionally, failure handling tests strengthen the system's reliability against unexpected scenarios. The execution of these tests will provide confidence in the stability and performance of the final product.

3.9 Data Management Strategy

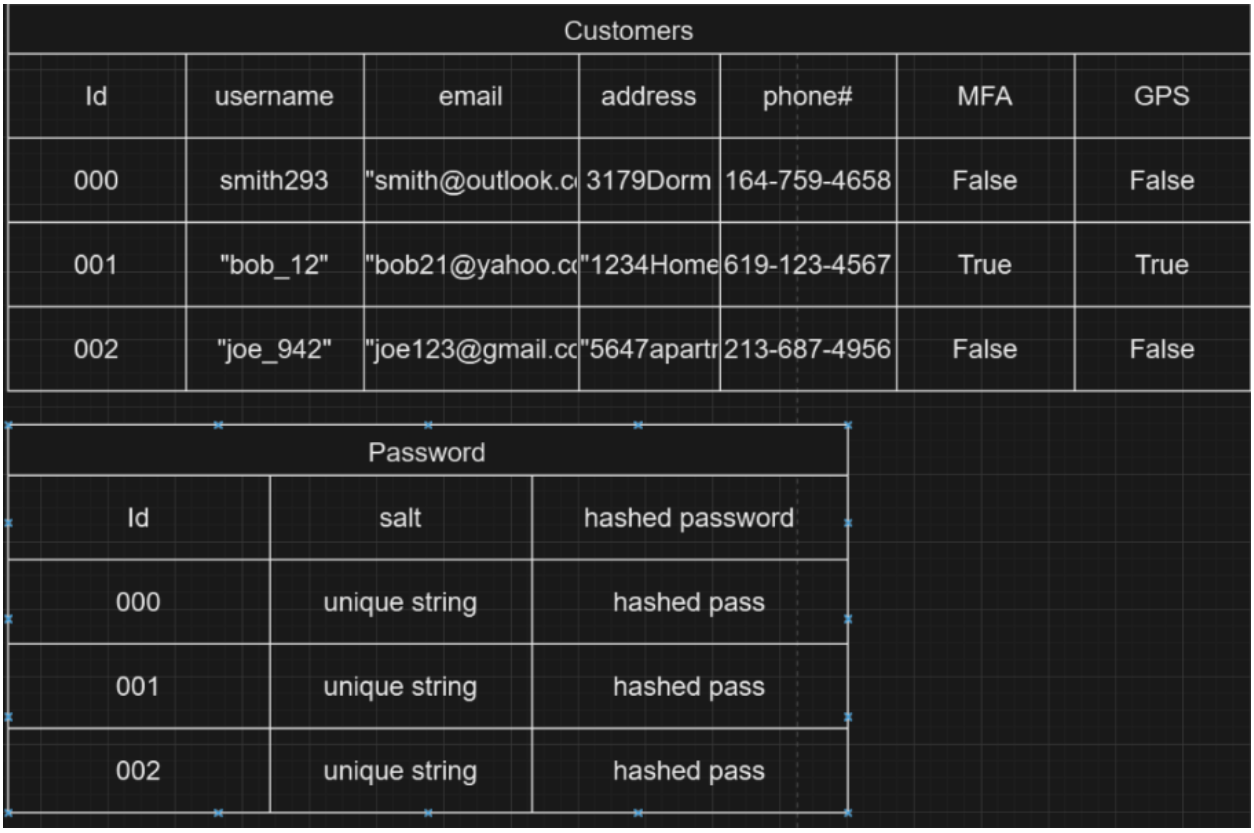
3.9.1 Overview

To support the real-time and security-sensitive nature of the **TriTicketing Movie Theater Ticketing System**, we designed a centralized relational database strategy using **PostgreSQL**. All persistent data is organized logically across normalized tables, supporting high data integrity, fast access for real-time operations, and clean integration across system components like ticket booking, showtimes, payments, and users.

The following section presents each major data domain, the reasons for choosing a relational model, what kind of data is stored, and potential alternatives considered.

Software architecture diagram in section 3.7.9. was updated to show an added step in user flow by including the selection of a time slot after selecting a movie and before selecting a seat. The movie and theater databases were also swapped to reflect the new steps in the flow in which they will be coming into use. All databases that had been included in the previous version of the architecture diagram have remained the same; these include a database for: User, Employee, Payment, Theater, Movie, and Tickets.

3.9.2 Database Overview by Domain



Customers

Id	username	email	address	phone#	MFA	GPS
000	smith293	"smith@outlook.c	3179Dorm	164-759-4658	False	False
001	"bob_12"	"bob21@yahoo.c	"1234Home	619-123-4567	True	True
002	"joe_942"	"joe123@gmail.c	"5647apartr	213-687-4956	False	False

Password

Id	salt	hashed password
000	unique string	hashed pass
001	unique string	hashed pass
002	unique string	hashed pass

1. Users Database

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `userID`, `name`, `email`, `hashedPassword`, `phoneNumber`, `loyaltyPoints`, MFA, GPS
- **Data Types**: Integer, Varchar, Boolean
- **Justification**: User data is highly structured and relational; SQL supports strong consistency, relationships (e.g., tickets, payments), and authentication flows.
- **Additional Details**: The Customer SQL table will be storing username, emails, address, phone number, and whatever else is needed to keep the customer's account. Username, email, address, phone number will all be strings and MFA and GPS will be boolean values. This is a table with set columns that will not change, hence why it makes more sense to choose SQL over NoSQL. SQL will also allow us to scale vertically by adding more customers and fits our needs because we won't need to add any more fields.

1a. Passwords Table

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `userID`, `hashedPassword`, `salt`, `pepper`
- **Justification**: The Password database will be a SQL table that will hold a hash password and use salt and pepper which will keep the passwords from being unhashed by hackers. Salt will add a unique string to the password allowing the password to become tougher to crack. We wouldn't want to keep the password out in the open where anyone can see it. There is no tradeoff in this database compared to NoSQL simply because we care more about our user's password security which SQL provides us with.

2. Employees Database

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `employeeID`, `name`, `role`, `email`, `accessLevel`
- **Justification**: The employee table includes structured, well-defined fields. Role-based access control (RBAC) and auditability require consistent and queryable structures, making SQL ideal.
- **Additional Details**: This table helps us differentiate between admin and regular employees by storing access levels. Since roles and permissions are fixed per employee type, SQL is best for managing these consistent structures. Vertical scaling is supported as new employees are added.The employee table includes structured, well-defined fields. Role-based access control (RBAC) and auditability require consistent and queryable structures, making SQL ideal.

Employees				
ID	name	zip code	hours	title
9543532	EWynell	92173	10:00 - 18:00	salesperson
9412863	EWinters	91911	6:00 - 13:00	supervisor
9437153	JSunderland	17927	8:30 -15:50	salesperson
9738262	TStark	92101	13:00 - 20:00	salesperson
9453284	JValentine	10130	9:00 - 17:00	manager

3. Theaters Database

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `theaterID`, `name`, `city`, `state`, `capacity`
- **Justification**: Theater data rarely changes and must link reliably to showtimes. SQL provides strong integrity for location-based operations.
- **Additional Details**: Each theater has a unique capacity and location. Using SQL ensures that every theater is stored with precise, consistent metadata that supports scheduling and reporting.Theater data rarely changes and must link reliably to showtimes. SQL provides strong integrity for location-based operations.

4. Movies Database

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `movieID`, `title`, `duration`, `genre`, `rating`, `releaseDate`
- **Justification**: Movie metadata is structured and used for listings, recommendations, and rating. SQL supports flexible querying and joins with showtimes and reviews.
- **Additional Details**: Since movie data (title, duration, etc.) is fixed per record, a relational table helps avoid inconsistencies. We also ensure referential integrity when linking to showtimes and reviews.Movie metadata is structured and used for listings, recommendations, and rating. SQL supports flexible querying and joins with showtimes and reviews.

5. Tickets Database

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `ticketID`, `userID`, `showtimeID`, `seatNumber`, `price`, `purchaseTimestamp`
- **Justification**: Tickets must link to users, seats, and showtimes with transactional safety. SQL ensures each booking is processed reliably and prevents duplication.
- **Additional Details**: Each ticket is a unique record that must be tied to a specific user and showtime. SQL helps us enforce constraints that ensure no double-bookings and accurate

seat tracking.Tickets must link to users, seats, and showtimes with transactional safety. SQL ensures each booking is processed reliably and prevents duplication.

6. Payments Database

- **Technology**: PostgreSQL (SQL)
- **Stored Data**: `paymentID`, `ticketID`, `userID`, `amount`, `method`, `status`, `timestamp`
- **Justification**: SQL ensures transaction consistency and traceability of financial records. ACID compliance and rollback support are essential for payment processing.
- **Additional Details**: Each payment is permanently linked to a ticket and user. Using SQL allows us to log all transactions securely, detect fraud attempts, and generate detailed financial reports.SQL ensures transaction consistency and traceability of financial records. ACID compliance and rollback support are essential for payment processing.

Payment Info			
Payment ID	amount	date	method
001	$90	11-03-2024	Credit
002	$78	02-25-2024	Debit
003	$122	05-03-2024	Transfer
004	$67	12-23-2024	Credit
005	$104	09-04-2024	Cash

3.9.3 Technology Rationale

We selected PostgreSQL due to its:

- Full **ACID compliance**, enabling consistent and safe transactions (e.g., ticket bookings, payments)
- Mature support for **foreign keys**, indexing, and constraints
- Compatibility with security best practices (RBAC, encryption)

Indexes are applied on key fields such as `userID`, `ticketID`, `showtimeID`, and `seatNumber` to improve lookup performance during heavy load.

Concurrency is handled using transaction locks with **two-phase locking (2PL)** during seat reservation and ticket confirmation.

3.9.4 Security and Privacy Considerations

- **Passwords** are hashed using bcrypt and stored in a dedicated table
- All sensitive data is encrypted at rest and in transit (via TLS)

- **Role-based access control (RBAC)** ensures limited privileges across users, staff, and admins
- All data access is logged for compliance and traceability
- The system is designed to follow **GDPR-like data privacy principles**, ensuring user consent and control over personal information

3.9.5 Monitoring and Reliability

- System metrics and database queries are monitored using **Prometheus and Grafana**
- Logs and queries are tracked for performance bottlenecks
- **Automated backups** are taken daily with support for **point-in-time recovery (PITR)**

3.9.6 Performance Expectations

- The system is designed to handle **500+ concurrent seat reservations per minute** during peak hours (e.g. blockbuster openings)
- Read-heavy operations (e.g., browsing showtimes) are optimized using caching (e.g., Redis or Memcached)

3.9.7 Logical Requirements for Database Usage

Yes, the system uses a structured PostgreSQL database for all persistent data. The following logical requirements apply:

- **Data Formats**: All data is stored in strongly typed columns (e.g., integers, strings, timestamps, booleans). Timestamps are stored in UTC format. Identifiers use UUIDs or auto-incremented keys depending on the entity.
- **Storage Capabilities**: The database must support millions of records for tickets, users, and showtimes over time. Partitioning and indexing are used to ensure sustainable query performance.
- **Data Retention**:
    - User activity logs and tickets are stored for at least 12 months for audit and support.
    - Payments are retained according to financial policy and regional compliance requirements.
- **Data Integrity**:
    - Enforced through foreign keys, constraints, and ACID-compliant transactions.
    - Updates or deletions cascade appropriately to prevent orphaned or inconsistent records.
- **Scalability**: The schema supports horizontal scaling via read replicas and caching layers.
- **Compliance**: The design complies with GDPR-like policies, ensuring data minimization, consent control, and right-to-be-forgotten mechanisms.

3.9.8 Summary

The TriTicketing data management design prioritizes security, consistency, performance, and future scalability. SQL offers ideal support for highly relational, transactional workflows like ticketing, payments, and seat assignment. Features like RBAC, PITR, GDPR-alignment, and real-time monitoring make the solution production-ready and well-grounded in both academic and real-world best practices.

3.10.1 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 2