Cinema-View Ticketing System

Software Title: Cinema-View Ticketing System

Individual Member: Abdihakim Ahmed

System Description:

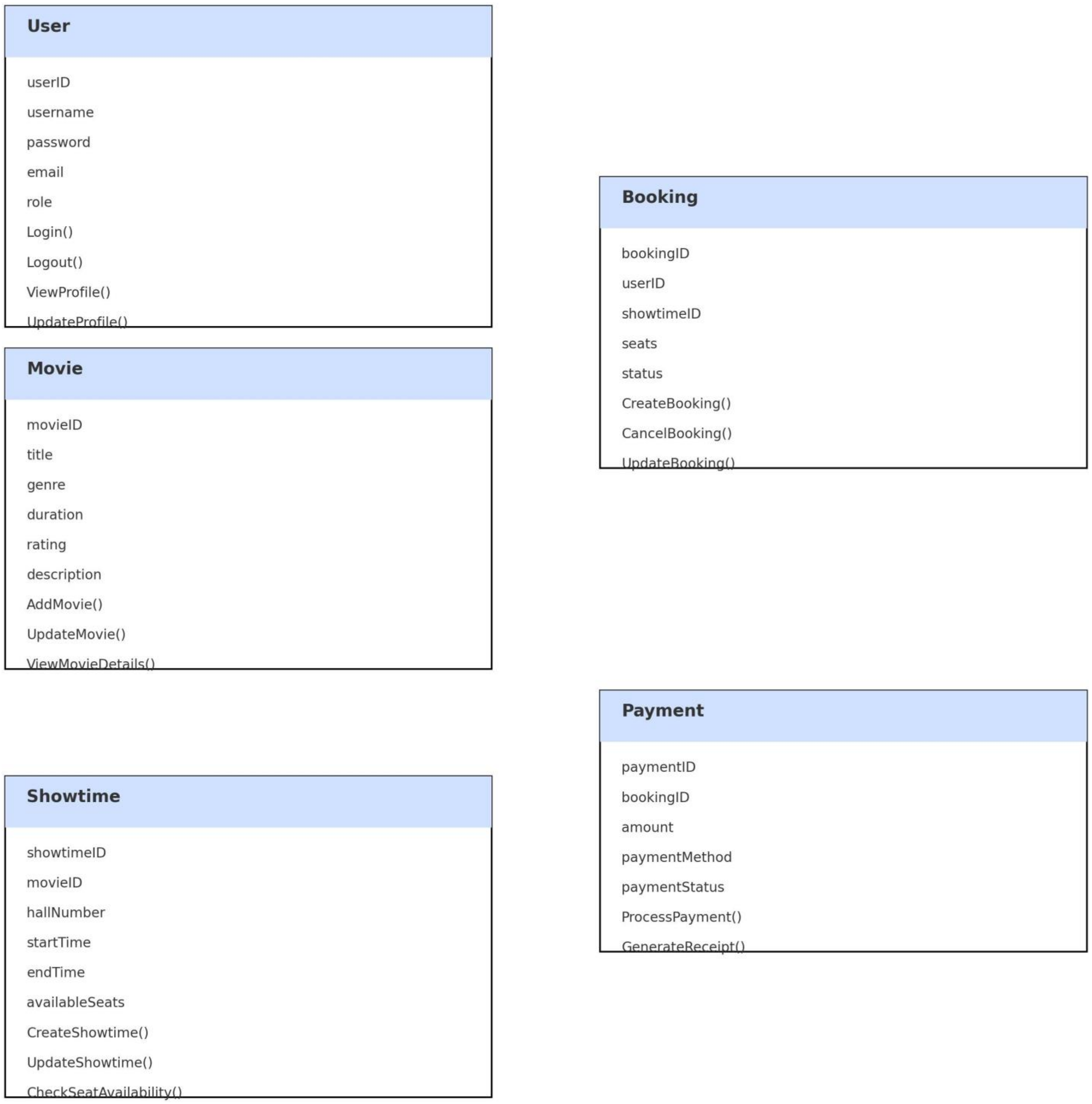
The Cinema-View Ticketing System is a web-based application for booking movie tickets, selecting showtimes and seats, and managing reservations through a responsive UI. It includes admin functionality for managing movies, showtimes, and theater seating.

Software Architecture Overview

The system is built on a multi-tier MVC architecture:

- Frontend: React.js or Vue.js (HTML, CSS, JavaScript)
- Backend: Node.js with Express or Django
- Database: MySQL or PostgreSQL
- External APIs: Email/SMS for confirmations, Payment Gateway for transactions

UML Class Diagram



Class Descriptions

Class: User

Attributes:

- userID: int
- username: string
- password: string
- email: string
- role: string
- registrationDate: datetime

Operations:

- Login(username: string, password: string): boolean
- Logout(): void
- ViewProfile(userID: int): User
- UpdateProfile(userID: int, data: dict): boolean

Class: Movie

Attributes:

- movieID: int
- title: string

-genre: string
- duration: int
- rating: string
- description: string

Operations:

- AddMovie(data: dict): boolean
- UpdateMovie(movieID: int, data: dict): boolean
- ViewMovieDetails(movieID: int): Movie

Class: Showtime

Attributes:

- showtimeID: int
- movieID: int
- hallNumber: int
- startTime: datetime
- endTime: datetime
- availableSeats: int

Operations:

- CreateShowtime(data: dict): boolean
- UpdateShowtime(showtimeID: int, data: dict): boolean
- CheckSeatAvailability(showtimeID: int): list

Class: Booking

Attributes:

- bookingID: int
- userID: int

- showtimeID: int
- seats: list
- status: string

Operations:

- CreateBooking(data: dict): boolean
- CancelBooking(bookingID: int): boolean
- UpdateBooking(bookingID: int, data: dict): boolean

Class: Payment

Attributes:

- paymentID: int
- bookingID: int
- amount: float
- paymentMethod: string
- paymentStatus: string

Operations:

- ProcessPayment(data: dict): boolean
- GenerateReceipt(paymentID: int): string

Development Plan and Timeline

Partitioning of Tasks:

- Phase 1: Architecture and Setup (Weeks 1-2)
  - System Architecture, Database Design, UI/UX Design
- Phase 2: Frontend Development (Weeks 3-5)
  - UI Pages, Seat Selection Interface, Mobile Responsiveness
- Phase 3: Backend Development (Weeks 6-8)
  - Booking Logic, Admin Portal, User Authentication
- Phase 4: Testing & QA (Weeks 9-10)
  - Functional Testing, Security, Performance, Bug Fixes
- Phase 5: Deployment (Weeks 11-12)
  - Deployment on Cloud, Post-Deployment Testing, Admin Training

Team Member(Individual) Responsibilities

- Abdihakim Ahmed: Lead Backend Architecture, Database Schema, Booking Logic

1. Software Architecture Diagram

The *Cinema-View Ticketing System* employs a modular, multi-tiered architecture based on the Model-View-Controller (MVC) design pattern, which ensures separation of concerns, scalability, and maintainability. This architecture is composed of two layers: the frontend (presentation layer), backend (application logic), and data layer (database), which are seamless in integrating with external services like notification and payment gateways.

System Components and Delegations System Functions and Delegations

- Frontend Layer: It can be done in the React.js or Vue.js user interface. It will enable responding to the user and dynamicity on any device. It calls the backend using RESTful APIs exclusively, making it a loosely coupled system.
- Backend Layer: The backend is provided in the form of a Node.js server with Express or in Python-based implementations in the form of Django. The backend is tailored to special services:

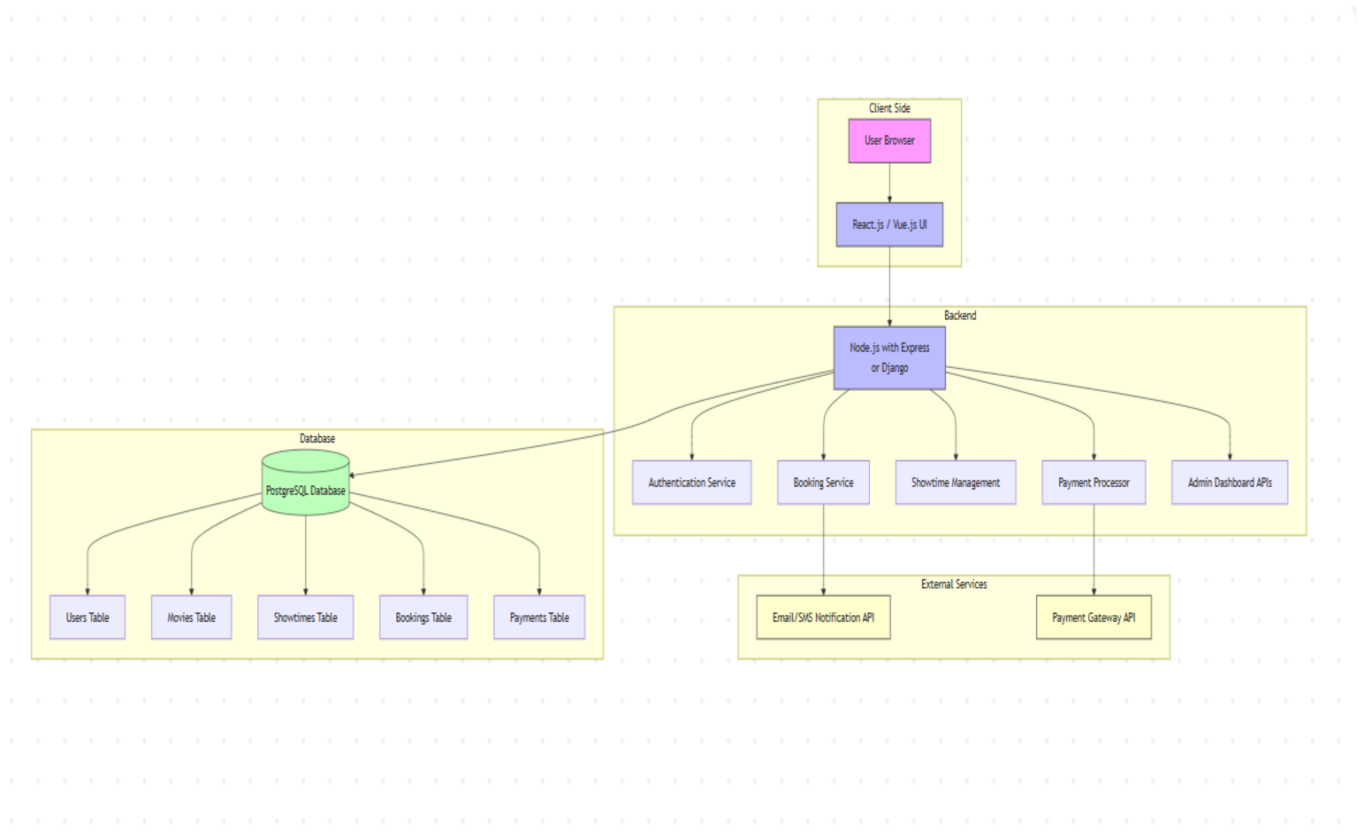
  1. Authentication Service for login, registration, and session management.

2. Booking Service for ticket reservations and seat allocations.
3. Showtime Management Service for movie schedules and seat availability.
4. Payment Processor for secure financial transactions.
5. Admin Portal APIs for managing content and system configuration.

- **Database Layer**: The backend exchanges with a PostgreSQL relational database with normalized users, movies, showtimes, bookings, and payment tables. The data control approach provides security for storage, referential integrity, and transaction concurrency.
- **External Integrations**:

1. Payment Gateway API is used to process customer payments securely.
2. Email/SMS Notification API is used to send booking confirmations and alerts.

Architecture Visualization



Design Justification

This architecture is designed for scalability, modularity, and security**.** Independence of components means that they can be developed and tested independently because of the separation of concerns. Business logic is dealt with in the backend, and all client requests are processed prior to communicating with the database. The database will enforce data integrity by implementing constraints and normalization. Supporting external APIs on payments and notifications provides extensibility and real-time responsiveness without straining internal services. This tiered system can easily be extended with future improvements, like analytics modules or machine learning-based recommendation systems.

2. Data Management Strategy

The *Cinema-View Ticketing System* relies heavily on persistent and sensitive data, such as user credentials, booking records, payment details, and real-time seat availability. As such, the system uses a powerful and relationship-oriented data management design courtesy of PostgreSQL, a high-reliability and scalable SQL-based database management system. PostgreSQL was chosen because it supports ACID-compliant transactions, referential integrity, and complex queries, which are vital to consistency, particularly in the case of multiple ticket bookings and seat validation in a real-time scenario.

Database Architecture and Database Design Choices

Its database schema is organized in terms of five major entities:

- Users - entails personal information, login and role information, role identification keywords and the date of registration.
- Movies- films are named, categorised, timed, and rated metadata.
- Showtimes - associates movies with particular times and halls and verifies the available number of seats in them.
- Bookings -establish connections between users and showtimes and select and book seat statuses.
- Payments- logs- this is a set of logs used to see the method of payment, the amount of payment and the status.

Normalization to the Third Normal Form (3NF) is carried out on the database, redundancy is eliminated, and the data is reliable. The reason behind such a design choice is to optimize transaction operations access patterns. As an example, the Bookings table would have foreign keys referencing both Users and Showtimes tables. A stringent linkage between Data would be maintained, with a user being able only to book a shown time and with each booking being linked to an actual user session and vice versa.

Data Partitioning, Access, and Performance

Horizontal sharding has been applied logically in the Bookings table to constrain performance ideally, particularly in large transactions. Also, administrative dashboards have been developed with materialised views that allow real-time access to analytical results such as the most popular movies, overall revenues per day, and seat occupancy rates. Any data transaction has to pass through a backend API layer, which mediates it and enforces integrity and security by offering validation and sanitisation.

Security and Compliance

Given the sensitivity of stored data, the system incorporates several security mechanisms:

- Passwords are encrypted using bcrypt hashing algorithms.
- All database access is performed through parameterized queries to prevent SQL injection attacks.
- Role-Based Access Control (RBAC) is enforced at the application layer to limit administrative privileges.
- Data-in-transit is protected using SSL/TLS encryption between the backend server and database.

Alternatives and Trade-Offs

NoSQL solutions like MongoDB that plausibly represent scalable solutions with document-based flexibility have also been investigated. However, it has been simplified that these solutions are not adequate for such a system because of its requirements for strict consistency and transactional control. Likewise, SQLite was examined and ruled out due to its drawbacks in supporting multiple submissions in multi-user applications. Adoption of PostgreSQL was a trade-off regarding the database's relational competence. However, the database might fail to scale horizontally compared to NoSQL options. Nonetheless, its maturity, comprehensive documentation, and ability to support transactional guarantees make it a suitable option for the Cinema-View Ticketing System.

Cinema-View Ticketing System - Test Cases

Test ID	Level	Feature/Function	Test Input	Test Data	Expected Output	Resul
U1	Unit	CheckSeatAvailability()	Available seats before b	ooking 5 seats	5 seats	Pass
U2	Unit	ProcessPayment()	Valid payment method a	nd amount Valid card, $20	Success	Pass
F1	Functional	BookTicket()	Select movie, showtime,	and seats Movie A, 7PM, A1-A3	Booking confirmed	Pass
F2	Functional	CancelTicket()	Valid booking ID before	showtime Booking #1234	Booking canceled	Pass
S1	System	Full Booking Flow	Login > Select > Book >	Pay Valid user, seats A1-A2	Ticket confirmed	Pass
S2	System	Concurrent Booking	100 users book same se	at Seat A1	Only one booking succeeds	Pass
U3	Unit	Login()	Correct username and p	assword user123 / pass	Login success	Pass
F3	Functional	AddMovie()	Admin adds new movie	Title: Inception	Movie listed	Pass
S3	System	Booking History	User checks booking log	User ID 44	Shows past bookings	Pass
S4	System	Invalid Cancellation	Cancel after showtime	Booking #5555	Cancellation denied	Pass