Online Movie Ticket Booking System

System Design Document

Tarun Jain

6/14/2024

[Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.]

1. Contextualization

XYZ Company wants to build an online movie ticket booking platform that caters to both B2B (theatre partners) and B2C(end customers)

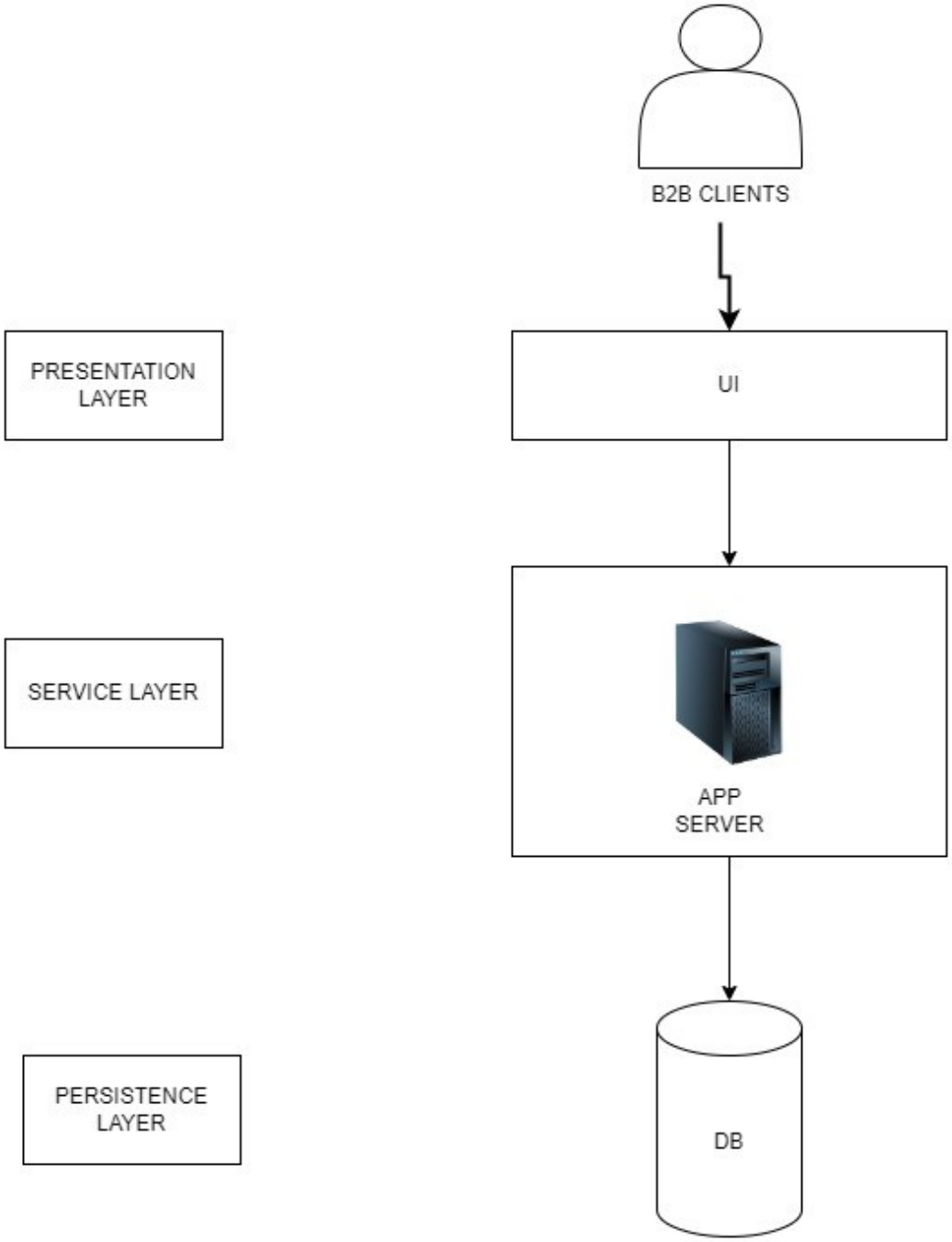
2. Key Goals

- Enable theatre partners to onboard their theatres over this platform and get access to a bigger customer base while going digital.
- Enable end customers to browse the platform to get access to movies across different cities, languages, and genres, as well as book tickets in advance with a seamless experience.

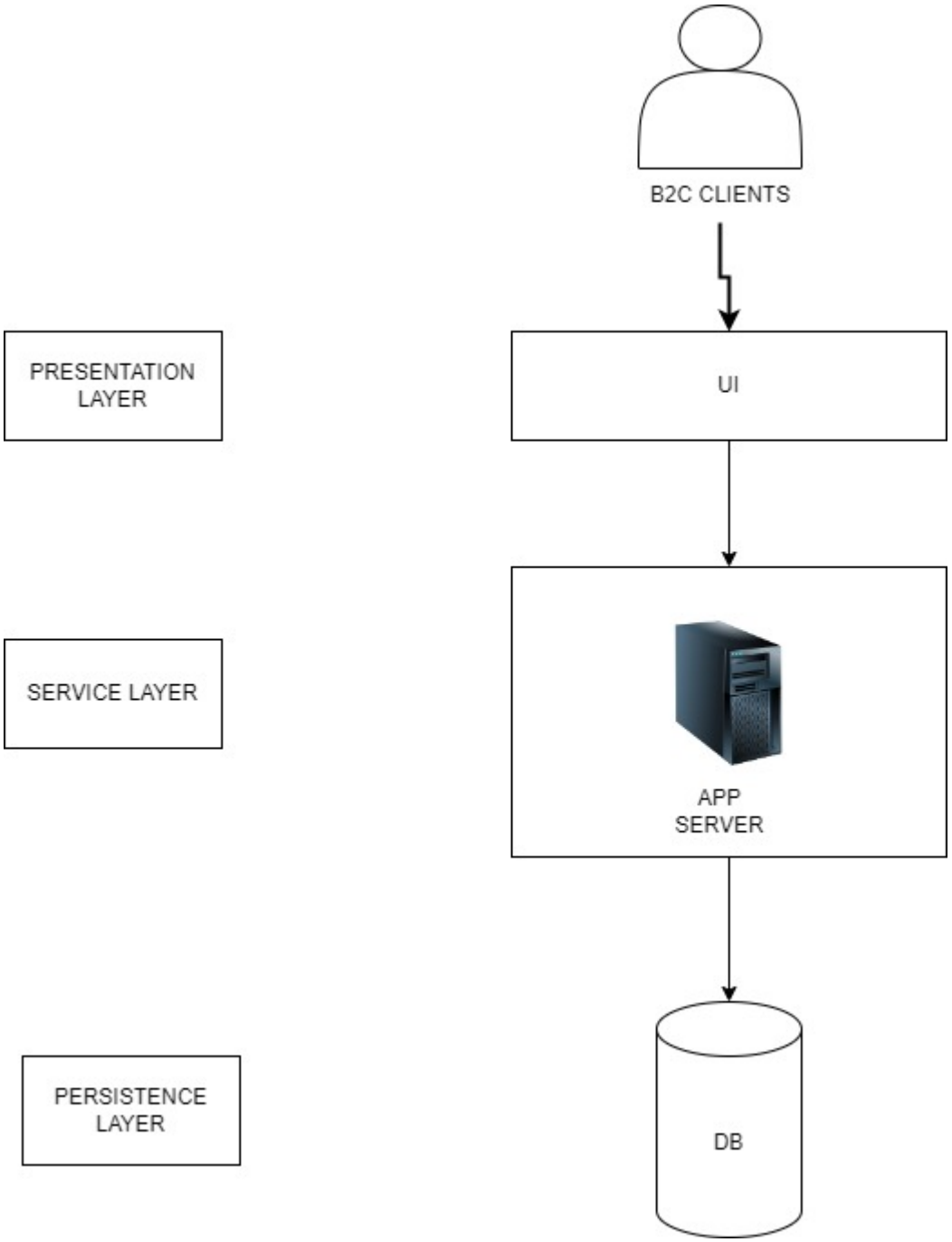
3. High Level Architecture

- Will be using 3-tier architecture with distinct layers
- Each layer will have a unique set of responsibilities, viz
    - Presentation
    - Service
    - Persistence

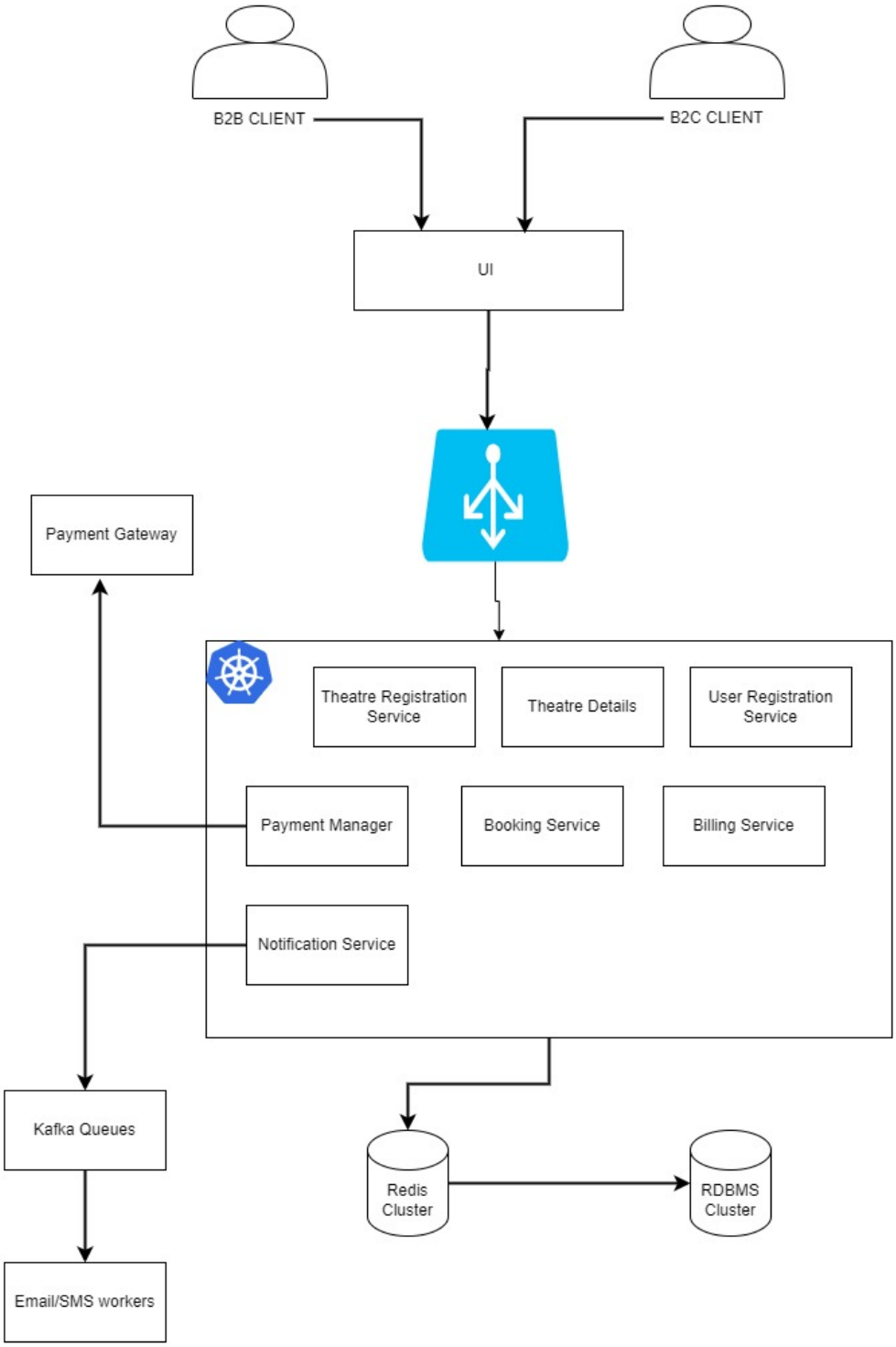
3.1. Theatre Management System



3.2. Ticket Booking System



4. Low Level Architecture


B2B CLIENT
B2C CLIENT
UI
Payment Gateway
Theatre Registration Service
Theatre Details
User Registration Service
Payment Manager
Booking Service
Billing Service
Notification Service
Kafka Queues
Redis Cluster
RDBMS Cluster
Email/SMS workers

5. Technical Stack

5.1. Presentation Layer- UI

We can use any of the JS frameworks. I am proposing to use ReactJS. ReactJS has become one of the most popular libraries for building user interfaces, especially single-page applications (SPAs). Here are some key advantages of using ReactJS:

1. **Component-Based Architecture**:
   - **Modularity**: ReactJS allows developers to build encapsulated components that manage their own state, making it easier to manage and reuse code.
   - **Maintainability**: Components can be easily tested and maintained, improving the overall maintainability of the codebase.
2. **Virtual DOM**:
   - **Performance**: ReactJS uses a virtual DOM to efficiently update and render components. This improves performance by minimizing direct manipulation of the actual DOM, which can be slow.
   - **Efficient Updates**: Changes in the UI are first applied to the virtual DOM, and then a diffing algorithm determines the minimal number of updates needed to synchronize with the actual DOM.
3. **Unidirectional Data Flow**:
   - **Predictability**: ReactJS enforces a unidirectional data flow, making the application's state more predictable and easier to debug.
   - **State Management**: Libraries like Redux or Context API can be used to manage the global state, further enhancing the predictability and maintainability of the application.
4. **JSX**:
   - **Readable Syntax**: JSX, a syntax extension for JavaScript, allows developers to write HTML-like code within JavaScript. This makes the code more readable and easier to understand.
   - **Template and Logic Integration**: JSX allows for the seamless integration of HTML and JavaScript, enabling developers to write cleaner and more expressive UI code.
5. **Strong Community Support**:
   - **Ecosystem**: ReactJS has a large and active community, providing a wealth of resources, tutorials, and third-party libraries that can accelerate development.
   - **Job Market**: With its popularity, there is a high demand for ReactJS developers, making it a valuable skill in the job market.
6. **React Native**:
   - **Cross-Platform Development**: ReactJS knowledge can be transferred to React Native, allowing developers to build mobile applications for iOS and Android using the same principles and often the same codebase.
7. **SEO-Friendly**:
   - **Server-Side Rendering (SSR)**: ReactJS can be rendered on the server side using frameworks like Next.js, improving SEO and load times for web applications.

8. **Flexibility**:
    - **Integration**: ReactJS can be easily integrated with other libraries or frameworks, such as Redux for state management, or with backend frameworks like Node.js.
    - **Scalability**: React's component-based architecture makes it scalable, as components can be developed, tested, and deployed independently.
9. **Developer Tools**:
    - **Debugging**: React Developer Tools, available as a browser extension, provide powerful capabilities for inspecting React component hierarchies and debugging applications.
10. **Backward Compatibility**:
    - **Stable API**: ReactJS maintains a relatively stable API with backward compatibility, reducing the risk of major changes breaking existing applications.

Overall, ReactJS offers a robust set of features and a supportive ecosystem that enables developers to build efficient, maintainable, and scalable user interfaces.

5.2. Service Layer

For implementing service layer we have two choices NodeJS and JAVA with SpringBoot. I recommendation is SpringBoot as per the following:

Spring Boot and Node.js each have their strengths and are suited to different types of projects. Here are some reasons why Spring Boot might be considered better than Node.js for certain scenarios:

1. **Enterprise-level Applications**:
    - **Spring Boot**: It is often preferred for enterprise-level applications due to its robust, comprehensive ecosystem and strong support for building complex, large-scale applications.
    - **Node.js**: While capable, it may not have as extensive a set of tools and libraries for enterprise use cases.
2. **Performance and Scalability**:
    - **Spring Boot**: Known for its performance and ability to handle high-throughput transactions, especially when dealing with complex, CPU-intensive tasks.
    - **Node.js**: It excels in handling I/O-bound operations and can be more efficient for real-time applications but may struggle with CPU-intensive tasks due to its single-threaded nature.
3. **Mature Ecosystem and Tools**:
    - **Spring Boot**: Comes with a mature ecosystem with powerful tools for building, deploying, and maintaining applications, such as Spring Cloud for microservices and Spring Security for security.
    - **Node.js**: Has a rapidly growing ecosystem but is still catching up in terms of maturity and enterprise-level tools.
4. **Strong Typing and Compile-time Checking**:

- **Spring Boot**: Java, the primary language used with Spring Boot, provides strong typing and compile-time checking, which can help catch errors early in the development process.
- **Node.js**: JavaScript is dynamically typed, which can lead to runtime errors that might be avoided with a strongly-typed language. (Though using TypeScript with Node.js can mitigate this to some extent.)

5. **Comprehensive Documentation and Community Support**:
   - **Spring Boot**: Backed by a large community and comprehensive documentation, making it easier to find solutions and support for complex problems.
   - **Node.js**: Has a vibrant community and growing documentation but may not be as comprehensive for enterprise-level challenges.
6. **Security**:
   - **Spring Boot**: Built-in support for security best practices and tools such as Spring Security, making it easier to implement robust security measures.
   - **Node.js**: While there are security modules available, the ecosystem is more fragmented, and security best practices can be harder to enforce consistently.
7. **Integration with Java Ecosystem**:
   - **Spring Boot**: Seamlessly integrates with other Java-based technologies, frameworks, and libraries, benefiting from the extensive Java ecosystem.
   - **Node.js**: Although versatile, it may require more effort to integrate with non-JavaScript-based systems.

Ultimately, the choice between Spring Boot and Node.js depends on the specific needs and constraints of the project. Spring Boot is often favored for enterprise-level applications requiring robust tools, performance, and security, while Node.js is preferred for lightweight, real-time applications and rapid development cycles.

5.3. Database Layer

For our application Polyglot Persistence Approach is the best solution:

1. **MySQL/PostgreSQL for Core Transactions**:
   - Use an RDBMS to handle core transactional data, such as user accounts, payment processing, and booking records. This ensures data integrity and strong consistency.
2. **MongoDB for Flexible Data and Content Management**:
   - Use MongoDB for managing event details, user reviews, and other semi-structured data that may change frequently and require flexibility.
3. **Cassandra for Log Management and Analytics**:
   - Use Cassandra to handle logs of user interactions and event attendance records, providing high availability and scalability for large-scale data.
4. **Redis for Caching and Real-Time Data**:
   - Use Redis for caching frequently accessed data, such as event listings and seat availability, to reduce load on the primary databases and improve response times.

Conclusion

For BookMyShow, a polyglot persistence approach leveraging both RDBMS and NoSQL databases is likely the best solution. This approach allows you to take advantage of the strengths of each type of database, ensuring scalability, performance, data integrity, and flexibility to handle the diverse and demanding requirements of a high-traffic, real-time ticketing platform.

6. API Details

1. Browse theatres currently running the show (movie selected) in the town, including show timing by a chosen date.

Endpoint: `/api/v1/theatres`
Method: `GET`
Query Parameters: • `movieId` (required): The ID of the selected movie. • `town` (required): The name or ID of the town. • `date` (optional): The date for which to retrieve show timings. If not specified, defaults to the current date.
Example Request: GET /api/v1/theatres?movieId=12345&town=Springfield&date=2024-06-15
Example Response { "movieId": "12345", "town": "Springfield", "date": "2024-06-15", "theatres": [{ "theatreId": "67890", "theatreName": "Springfield Cinema", "address": "123 Main Street, Springfield", "showtimes": ["10:00 AM", "1:00 PM", "4:00 PM", "7:00 PM", "10:00 PM"] }, { "theatreId": "54321", "theatreName": "Grand Theatre", "address": "456 Elm Street, Springfield", "showtimes": ["11:00 AM", "2:00 PM", "5:00 PM", "8:00 PM", "11:00 PM"] }] }

2. Booking platform offers in selected cities and theatres: 50% discount on the third ticket

Endpoint: `/api/v1/offers`
Method: `GET`

Query Parameters:

- `city` (required): The name or ID of the city.
- `theatreId` (optional): The ID of the theater. If not provided, offers for all theaters in the city will be returned.
- `numTickets` (optional): The number of tickets being purchased, used to calculate the discount.

Example Request

GET /api/v1/offers?city=Springfield&theatreId=67890&numTickets=3

Example Response

{ "city": "Springfield", "theatreId": "67890", "offers": [{ "offerId": "123", "description": "50% discount on the third ticket", "conditions": { "minTickets": 3, "discountType": "percentage", "discountValue": 50, "applicableTo": "third ticket" }, "calculatedDiscount": 5.00, "totalPriceBeforeDiscount": 30.00, "totalPriceAfterDiscount": 25.00 }] }

3. Booking platform offers in selected cities and theatres: Tickets booked for the afternoon show get a 20% discount

Endpoint: `/api/v1/offers`

Method: `GET`

Query Parameters:

- `city` (required): The name or ID of the city.
- `theatreId` (optional): The ID of the theater. If not provided, offers for all theaters in the city will be returned.
- `showTime` (required): The time of the show. This will determine if the show is an afternoon show.
- `numTickets` (optional): The number of tickets being purchased, used to calculate the discount.

Example Request

GET /api/v1/offers?city=Springfield&theatreId=67890&showTime=14:00&numTickets=3

Example Response

{ "city": "Springfield", "theatreId": "67890", "offers": [{ "offerId": "abc123", "description": "20% discount on afternoon shows", "conditions": { "timeRange": { "start": "12:00", "end": "17:00" }, "discountType": "percentage", "discountValue": 20 }, "calculatedDiscount": 12.00, "totalPriceBeforeDiscount": 60.00, "totalPriceAfterDiscount": 48.00 }] }

4. Book movie tickets by selecting a theatre, timing, and preferred seats for the day

Endpoint: `/api/v1/bookings`

Method: `POST`

Request Body:

- `theatreId` (required): The ID of the theater.
- `showTime` (required): The time of the show.
- `date` (required): The date of the show.
- `seats` (required): An array of seat identifiers.
- `userId` (required): The ID of the user making the booking.

Example Request

```
{ "theatreId": "67890", "showTime": "14:00", "date": "2024-06-15", "seats": ["A1", "A2", "A3"], "userId": "user123" }
```

Example Response

```
{ "bookingId": "booking789", "theatreId": "67890", "showTime": "14:00", "date": "2024-06-15", "seats": ["A1", "A2", "A3"], "userId": "user123", "totalPrice": 60.00, "status": "confirmed" }
```

5. User Management API

- **User Registration and Authentication**: Endpoints for user signup, login, logout, password reset, and social media authentication (OAuth).
- **User Profile Management**: Endpoints for viewing and updating user profile details, including preferences and payment methods.
- POST /api/v1/users/register
- POST /api/v1/users/login
- GET /api/v1/users/profile
- PUT /api/v1/users/profile

6. Event Management API

- **Event Listings**: Endpoints to fetch a list of upcoming events, movies, shows, etc., along with their details.
- **Event Details**: Endpoint to fetch detailed information about a specific event, including synopsis, cast, venue details, and showtimes.

- **Event Recommendations**: Endpoints for personalized event recommendations based on user preferences and behavior.
- GET /api/v1/events
- GET /api/v1/events/{eventId}
- GET /api/v1/events/recommendations

7. Booking and Ticketing API

- **Seat Availability**: Endpoint to check the real-time availability of seats for a specific event/show.
- **Booking Tickets**: Endpoint to reserve and purchase tickets, including handling seat selection and payment processing.
- **Booking Management**: Endpoints to view, modify, or cancel bookings and manage booking history.
- GET /api/v1/events/{eventId}/seats
- POST /api/v1/bookings
- GET /api/v1/bookings/{bookingId}
- DELETE /api/v1/bookings/{bookingId}

8. Payment Gateway API

- **Payment Processing**: Integration with payment gateways for processing payments securely.
- **Refunds and Cancellations**: Endpoints to handle refunds for cancelled bookings.
- **Transaction History**: Endpoints for viewing past transactions and payment statuses.
- POST /api/v1/payments
- POST /api/v1/payments/refund
- GET /api/v1/payments/history

9. Notification API

- **Email Notifications**: Sending booking confirmations, reminders, and promotional emails.
- **SMS Notifications**: Sending SMS confirmations and updates.
- **Push Notifications**: Sending real-time push notifications for reminders, updates, and promotions.
- POST /api/v1/notifications/email
- POST /api/v1/notifications/sms
- POST /api/v1/notifications/push

10. Search API

- **Search Events**: Endpoint for searching events based on various filters like date, location, genre, and keyword.
- **Search Venues**: Endpoint for searching venues and getting details about them.
- **Search Movies**: Endpoint for searching movies by title, genre, release date, etc.

- GET /api/v1/search/events
- GET /api/v1/search/venues
- GET /api/v1/search/movies