\WOOLF/

Applied Software Project Report

By Aditya Bindal

A Master's Project Report submitted to Scaler Neovarsity - Woolf in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

May, 2025

SCALER

Scaler Mentee Email ID: bindaladitya001@gmail.com

Thesis Supervisor: Naman Bhalla

Date of Submission: 30/05/2025

Certification

I confirm that I have overseen / reviewed this applied project and, in my judgment, it adheres to the appropriate standards of academic presentation. I believe it satisfactorily meets the criteria, in terms of both quality and breadth, to serve as an applied project report for the attainment of Master of Science in Computer Science degree. This applied project report has been submitted to Woolf and is deemed sufficient to fulfill the prerequisites for the Master of Science in Computer Science degree.

Naman Bhalla

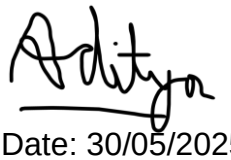
……

Project Guide / Supervisor

DECLARATION

I confirm that this project report, submitted to fulfill the requirements for the Master of Science in Computer Science degree, completed by me from 16th October 2023 to 30th April 2024 is the result of my own individual endeavor. The Project has been made on my own under the guidance of my supervisor with proper acknowledgement and without plagiarism. Any contributions from external sources or individuals, including the use of AI tools, are appropriately acknowledged through citation. By making this declaration, I acknowledge that any violation of this statement constitutes academic misconduct. I understand that such misconduct may lead to expulsion from the program and/or disqualification from receiving the degree.

Aditya Bindal



Date: 30/05/2025

ACKNOWLEDGMENT

I extend my heartfelt thanks to my mentor, Mr. Naman Bhalla, for his constant guidance and unwavering support throughout this assignment. His insights and encouragement played a crucial role in helping me stay focused and complete the task successfully.

I am also grateful to all the faculty members who contributed to my understanding of essential software development principles, without which this assignment would not have been possible.

Lastly, I sincerely appreciate the encouragement and support of my friends and family, whose belief in me kept me motivated during the entire process.

Table of Contents

List of Tables

List of Figures

Applied Software Project

Abstract

The online movie ticket booking system offers users a convenient platform to reserve seats for movie shows and access detailed information about films and theaters. Registered users can log into the system, browse available movies and theaters, and choose their preferred seats for a selected show. Upon selecting the desired seats, the system confirms availability and proceeds to the payment process. Users have the ability to manage their profiles, print their tickets, and receive booking confirmations via email notifications. Accessible from any internet-enabled device, the platform ensures a seamless movie ticket booking experience anytime, anywhere.

Project Description

The primary objective of the online movie booking system is to offer customers a seamless and automated method for purchasing cinema tickets. Once data is entered into the system's database, staff intervention for order handling becomes unnecessary, thereby reducing manual workload.

This system is designed to deliver comprehensive information about movies and theaters, enabling users to make informed decisions while booking their tickets.

The key goals of this system include:

- Providing real-time updates on seat availability and allowing users to select seats interactively.
- Building a scalable, user-friendly, and secure platform for online movie ticket reservations.
- Reducing the need for staff at physical ticket counters.
- Generating statistical insights from booking records to support operational decisions.

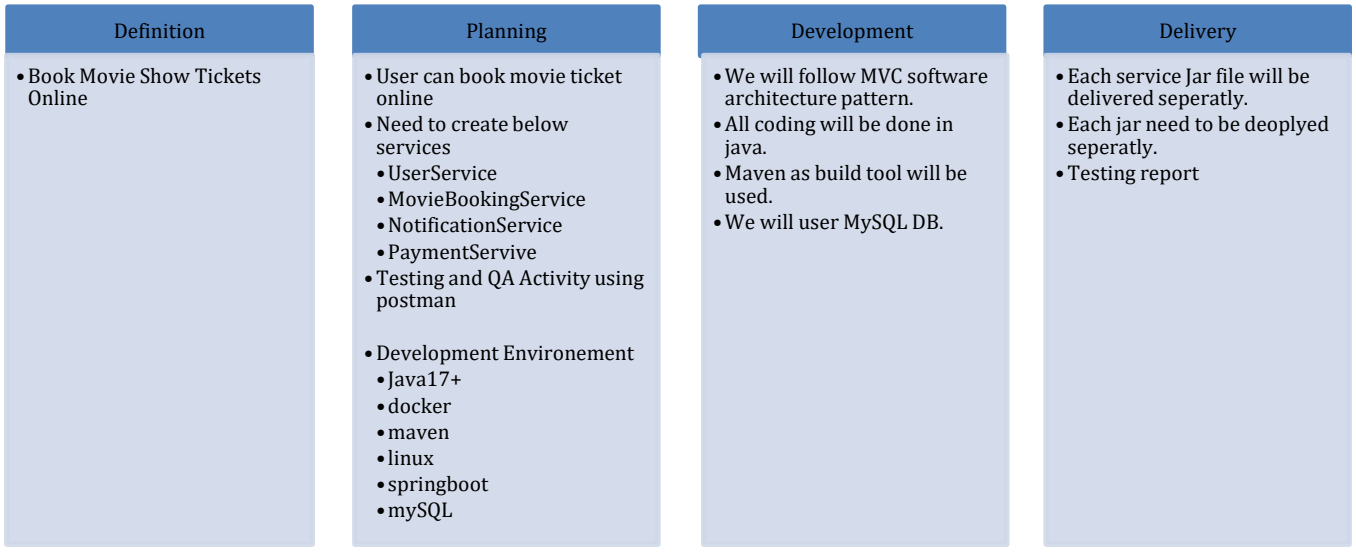


Figure 1.1: Project Development Process

Requirement Gathering

Functional Requirements

Requirement Description	Requirement in Detail
User Registration and Login	Users can create an account, log in, and manage their profiles.
Movie Listings	Display a list of available movies with details (title, description, ratings).
Showtimes and Seat Availability	Real-time information about movie schedules and seat availability.
Seat Selection	User can choose one or multiple seats
Booking and Payment	Users can book selected seats and pay online using multiple payment options.
E-ticket Generation	Upon successful booking, users receive an e-ticket
Booking History	Users can view past bookings, make cancellations, and view refunds.
Admin Dashboard	Manage movie listings, showtimes, bookings, and seat availability.
Booking notification	System should send notifications when a ticket is booked (with seat numbers, show timings, hall details etc.) or cancelled via email or text

Table 1.1: Functional Requirements

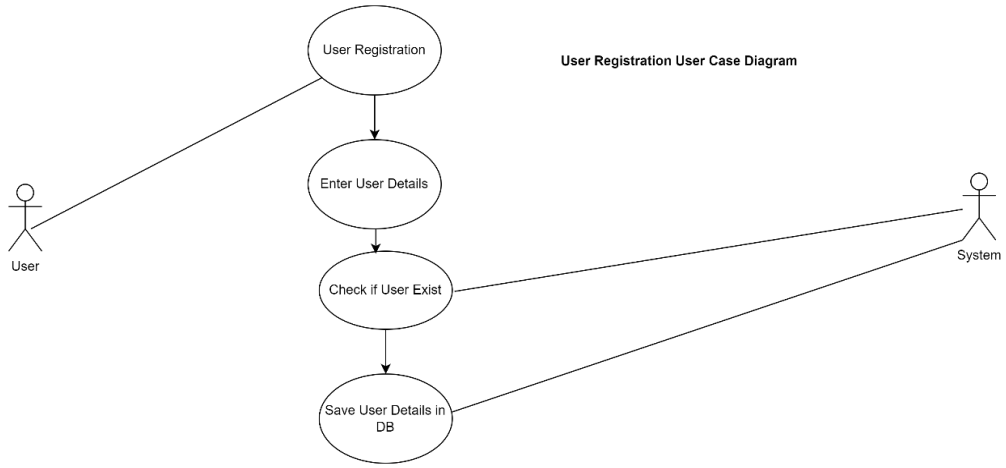
Non-Functional Requirements

Requirement Description	Requirement in Detail
Performance	Handle high traffic during peak times (e.g., weekends, movie releases).
Scalability	Ability to scale the system as demand grows
Availability	High availability to ensure the system is always operational

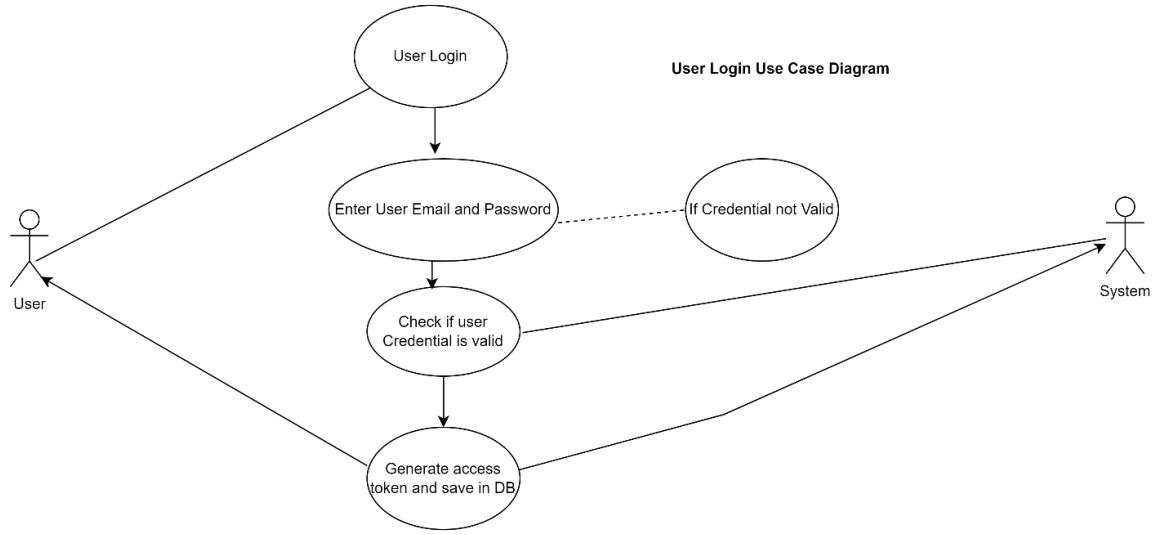
Table 1.2: Non-Functional Requirements

Use Case Diagrams

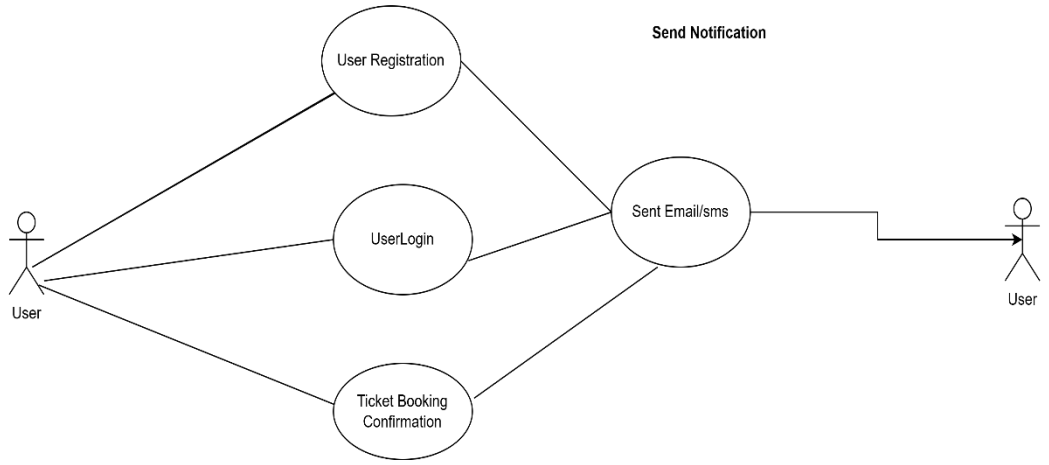
User Registration Use Case Diagram



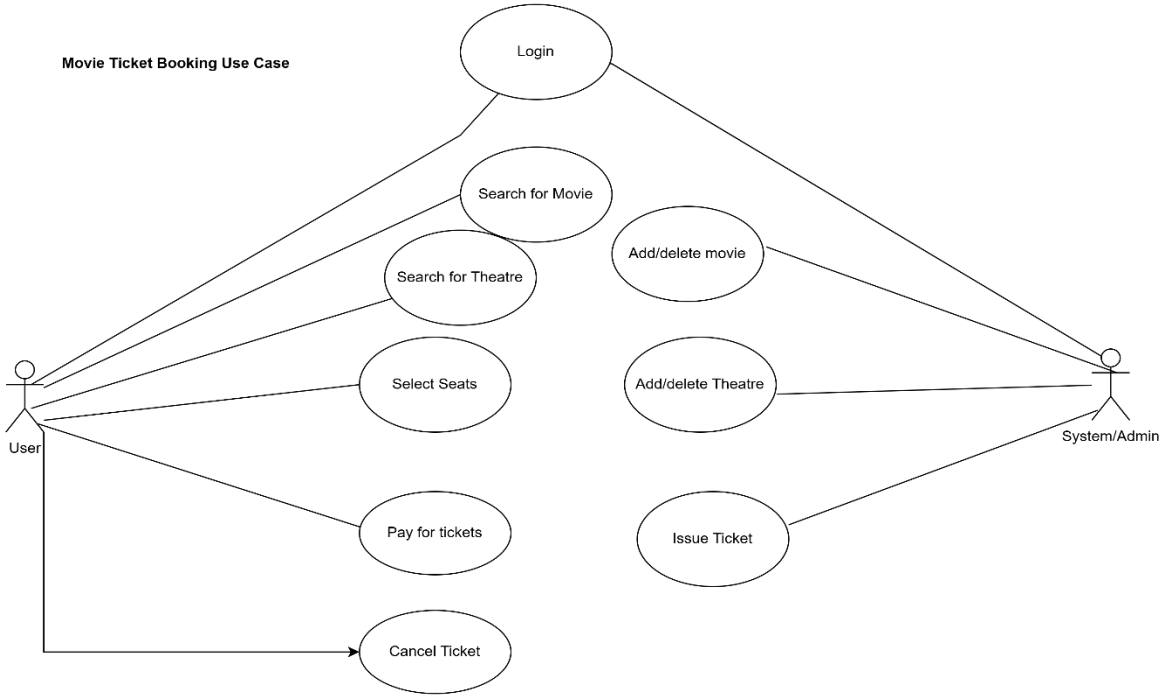
User Login User Case Diagram



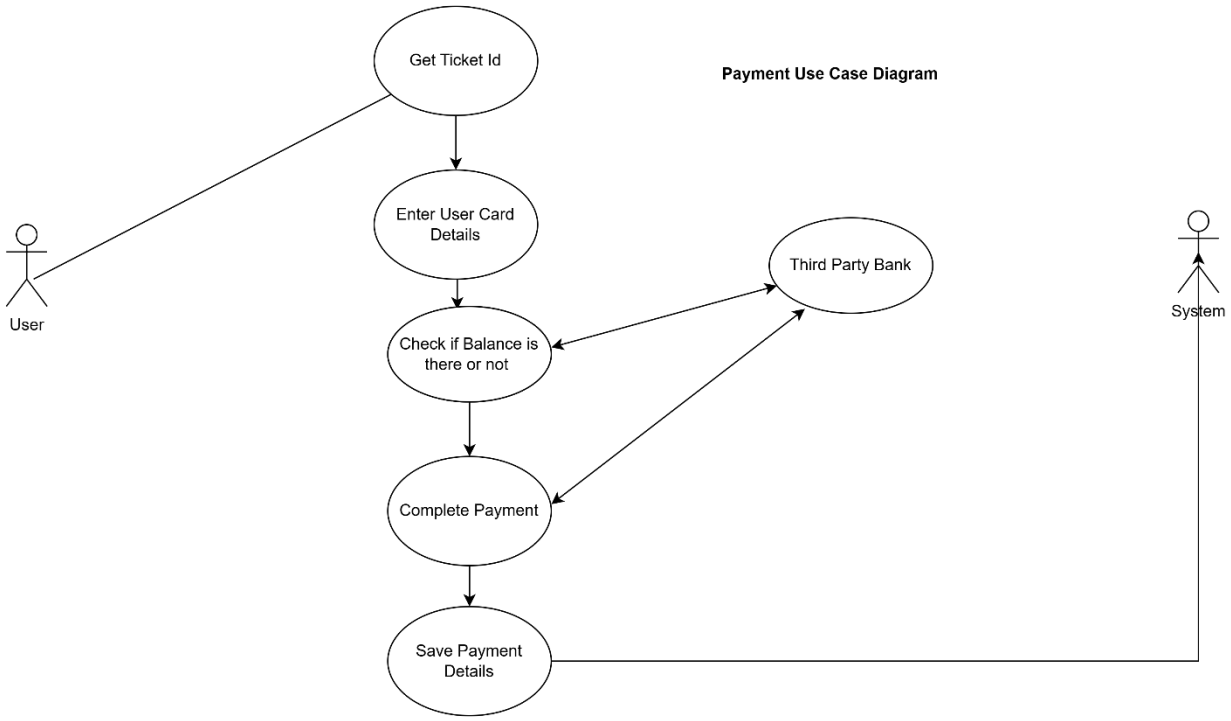
Notification Use Case Diagram



Ticket Booking User Case Diagram



Payment Service Use Case Diagram



Class Diagrams

Our system includes several core classes and actors that define its functionality and interactions.

Primary Actors in the System:

Admin: The administrator has privileges to manage the platform, including adding or removing movies and showtimes. The admin can also suspend user accounts if any suspicious behavior is detected.

User: End-users can browse available movies and shows, book or cancel tickets, and manage their bookings.

System: This component maintains records of all movies and show schedules. It also handles sending notifications to users regarding ticket confirmations, cancellations, or other updates.

Key Classes in the System:

Theatre: Represents the physical cinema venue. It includes attributes such as the theatre's name, address, and the list of halls it contains.

Theatre Hall: A cinema has several individual screening halls, each represented by this class.

Movie Show: Captures details of specific screenings, including associated movie data, hall assignment, and show timings (start and end).

Movie: This central class stores information about each film, including its title, genre, release date, user ratings, reviews, and comments.

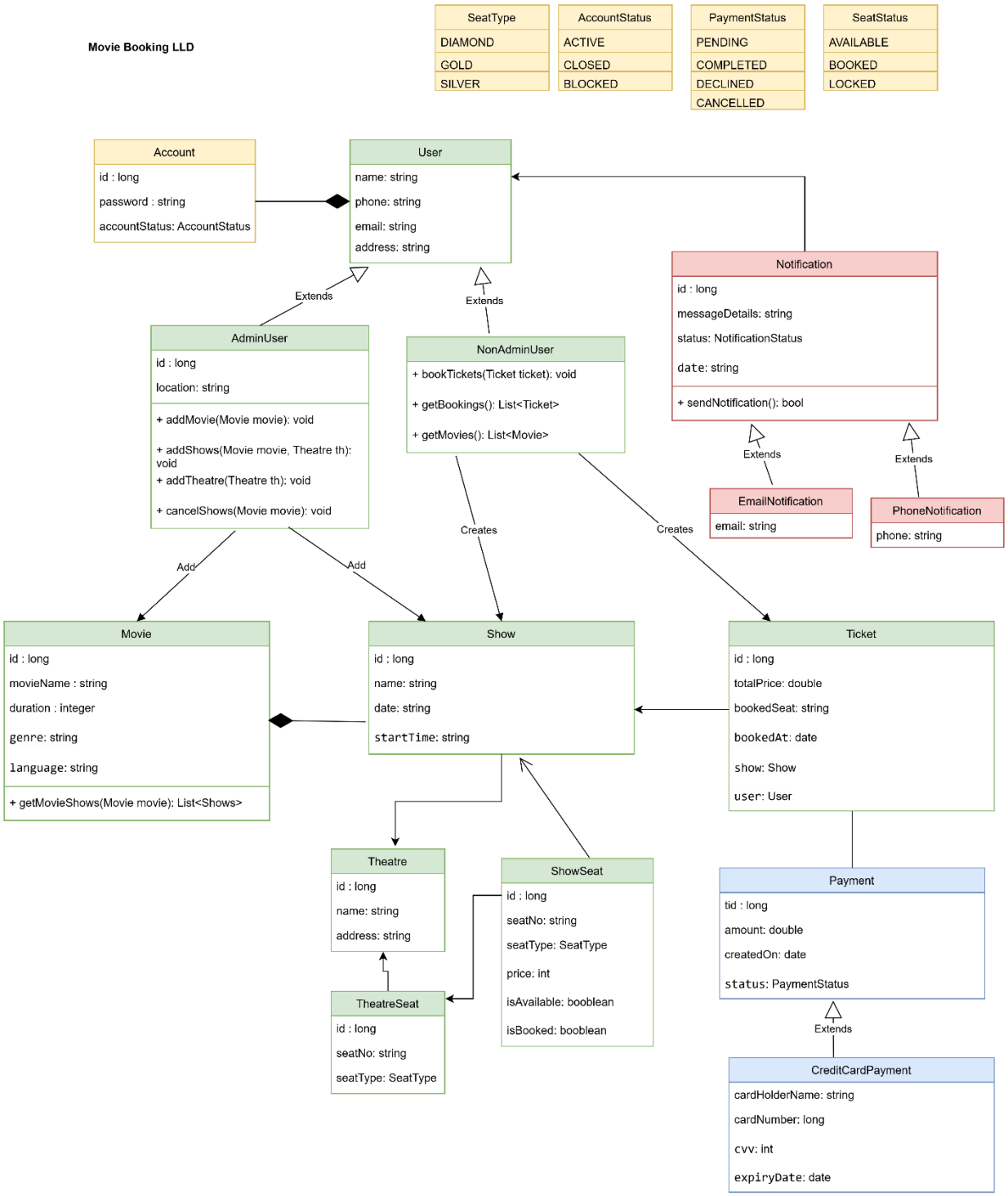
Theatre Hall Seat: Each screening hall has a seating arrangement, with seats categorized (e.g., Gold, Silver, Regular).

Show Seat: Since different shows run at various times in the same hall, this class tracks the booking status of each seat per show.

Ticket: Represents a user's booking. A ticket includes multiple seats and references to the movie, show details, and payment information.

Payment: Manages payment transactions, storing necessary payment-related data.

Notification: Responsible for delivering booking confirmations and updates to users via notifications.



Design description for BookMyMovie

System that encompasses its architecture, core components, key functionalities, and essential workflows. This breakdown includes frontend, backend, and database layers, as well as communication between services.

1. High-Level Architecture

The **BookMyMovie system** can be divided into three major layers:

Frontend (Client-Side)→Not in Scope

Web interface that allows users to browse movies, select theaters, book tickets, and make payments.

Backend (Server-Side):

Core services that manage data processing, handle business logic, interact with databases, and coordinate API calls.

Database:

Structured storage of user data, movie, and theater information, showtimes, seat allocations, and booking records.

Additionally, **external services** like **payment gateways** and **notification services** (for emails/SMS) are integrated into the backend.

2. System Components and Services

A. Frontend Layer

UI/UX:

A user-friendly, responsive web app interface.

Allows users to view available movies, showtimes, theaters, and seat selection in real-time.

Frontend Components(Not in Scope)

Browse Component: Displays lists of movies, including search and filtering options.

Theater/Show Component: Shows available theaters, screens, and showtimes for selected movies.

Seat Selection: Interactive seat layout for users to select specific seats.

Booking and Payment: Handles ticket booking and payment processing.

Technologies:

Web app (React, Vue, or Angular).

Communication with backend via RESTful APIs.

B. **Backend Layer**

The backend includes several services that operate independently.

User Service:

Manages user accounts, including registration, login, profile updates, and authentication (JWT-based).

Movie Service:

Stores and retrieves movie information, including title, genre, language, duration, ratings, and showtimes.

Provides search and filter functionalities.

Theater and Show Service:

Manages theater locations, screen details, seat arrangements, and available showtimes.

Responsible for fetching shows based on the selected movie, location, or time.

Ticket Booking Service:

Manages seat selection, booking creation, and booking status.

Prevents double-booking by locking selected seats until the transaction completes.

Integrates with the Payment Service to process payments and confirm booking.

Payment Service:

Connects to third-party payment providers (e.g., Stripe, PayPal).

Handles payment initiation, status tracking, and confirmation.

Generates payment receipts and updates booking status upon payment success.

Notification Service:

Manages email/SMS notifications for booking confirmations, reminders, and cancellations.

Uses third-party SMS/Email APIs to send messages.

C. Database Layer

Database schema design follows relational principles with some use of NoSQL for performance where necessary.

Relational Database:

Stores structured data on users, bookings, payments, and shows.

Tables for Users, Movies, Theaters, Shows, Seats, Bookings, and Payments.

3. Workflow and Sequence of Events

A. User Browses and Selects a Movie

User searches for a movie using filters (e.g., location, date, genre).

Frontend fetches movie details from the **Movie Service**.

Movie Service fetches showtimes from the **Theater and Show Service**.

The data is displayed to the user, showing available theaters and showtimes.

B. Seat Selection and Booking Workflow

The user selects a showtime and theater, proceeding to seat selection.

Show Service provides the current seat layout, indicating available and reserved seats.

User selects desired seats and proceeds to book.

Booking Service reserves seats temporarily to prevent double-booking.

C. Payment and Booking Confirmation

User initiates payment.

The **Payment Service** processes the transaction with a third-party payment provider.

On payment success, **Ticket Booking Service** finalizes the booking and updates seat availability.

Notification Service sends a confirmation email/SMS to the user.

D. Error Handling

If payment fails, **Ticket Booking Service** releases reserved seats, and the user is notified.

If seat selection expires (no payment), the system reopens seats for others to book.

4. Technology Stack

Backend:

Core Framework: Spring Boot (Java).

Authentication: JWT

Database:

Primary DB: MySQL

External Integrations:

Payments: Stripe etc.

Notifications: Twilio

DevOps:

Deployment: Docker, Kubernetes, CI/CD pipelines.

Cloud Hosting: AWS

5. Non-Functional Requirements

Scalability: System should handle high traffic during peak times, such as movie releases or weekends.

Performance: Low latency for browsing, seat selection, and booking.

Reliability: Ensure no seat double-booking through transactions and seat-locking mechanisms.

Security: Protect user data with encryption, secure payment channels, and access controls.

Availability: 99.9% uptime, ensuring booking is available anytime.

Database Schema Design

To design the **Low-Level Design (LLD) for BookMyMovie**, let us break down the requirements and create a detailed **database schema**. BookMyMovie is a movie ticket booking system that allows users to browse theaters, movies, and showtimes, and then book tickets accordingly. For our design, we will focus on core entities like User, Movie, Theater, Show, TicektBooking, Payment and Notification.

Core Functional Requirements

User Management: Users should be able to create accounts and log in.

Movie Information: Users can view movies, including showtimes, descriptions, ratings, etc.

Theater Information: Theaters and their locations need to be accessible.

Seat Selection and Booking: Users can select specific seats and book them.

Payment Handling: Payment processing for ticket booking.

Notifications: Notify users via SMS/email on booking confirmation, reminders, etc.

Entities and Database Schema

1. **User Table**

Stores information about users, including their contact details.

```
CREATE TABLE Users (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255),
    email VARCHAR(255) UNIQUE,
    phone_number VARCHAR(15),
    createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

2. Movie Table

Stores information about movies, including title, genre, and duration.

```
CREATE TABLE Movies (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    movie_name VARCHAR(255),
    genre VARCHAR(50),
    duration INT,
    release_date DATE,
    language VARCHAR(50));
```

3. Theatre Table

Stores information about theatres, including location details.

```
CREATE TABLE Theatres (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255),
    address TEXT,
    city VARCHAR(50),
    state VARCHAR(50),
    zipcode VARCHAR(10));
```

4. Show Table

Stores information about shows for each movie on a specific screen at a specific time.

```
CREATE TABLE Shows (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    movie_id BIGINT,
   name VARCHAR(50),
    theatre_id BIGINT,
    date DATE,
    start_time TIME,
    FOREIGN KEY (movie_id) REFERENCES Movies(movie_id),
    FOREIGN KEY (theatre_id) REFERENCES Theatre(theatre _id));
```

5. Seat Table

Represents individual seats within each screen, specifying seat type and other details.

```
CREATE TABLE Seats (
```

```
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    show_id BIGINT,
    seat_no VARCHAR(10),
    seat_type ENUM('REGULAR', 'PREMIUM', 'VIP') DEFAULT 'REGULAR',
    price int,
    is_available bit,
    is_food_booked bit,
    FOREIGN KEY (show_id) REFERENCES shows(show_id));
```

6. Ticket Booking Table

Stores booking details for each user, linked to the show and seats.

```
CREATE TABLE Tickets (
   id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT,
    show_id BIGINT,
    total_price DECIMAL(10, 2),
    status ENUM('CONFIRMED', 'PENDING', 'CANCELLED') DEFAULT 'PENDING',
    booked_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    booked_seat VARCHAR(255)
    FOREIGN KEY (user_id) REFERENCES Users(user_id),
    FOREIGN KEY (show_id) REFERENCES Shows(show_id));
```

7. TheatreSeat Table

```
CREATE TABLE TheatreSeat (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
```

```
    seat_no VARCHAR(30),
    seat_type ('DIAMOND','GOLD','SILVER'),
    FOREIGN KEY (theatre_id) REFERENCES Theatre(theatre_id));
```

8. Payment Table

Stores payment details for each booking.

```
CREATE TABLE Payments (
    tid BIGINT PRIMARY KEY AUTO_INCREMENT,
    ticket_id BIGINT,
    amount DECIMAL(10, 2),
    created_on TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('SUCCESS', 'FAILED', 'PENDING', 'DECLINED') DEFAULT 'PENDING',
    FOREIGN KEY (ticket _id) REFERENCES Bookings(ticket _id));
```

9. Notifications Table

Stores notifications sent to users.

```
CREATE TABLE Notifications (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT,
    message_details TEXT,
    created_on TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('SENT', 'PENDING', 'FAILED') DEFAULT 'PENDING',
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
```

);

Additional Indexes and Optimizations

Indexes on Users.email, Movies.name, and Theaters.show_id for faster searches.

Relationships

One-to-Many: A User can have multiple Bookings.

One-to-Many: A Theatre can have multiple Shows.

Many-to-One: Each Show can show multiple Shows.

Feature Development Process

The **Ticket booking** feature involves selecting seats, confirming the booking, and processing payments. Here is how this is typically handled through an agile, iterative process:

A. Requirement Gathering and Analysis

Functional Requirements: Define features like movie selection, seat availability, booking confirmation, and payment processing.

Non-Functional Requirements: Ensure that the system can handle peak loads, prevent double bookings, and offer a seamless, fast user experience.

B. System Design and Workflow Creation

Define the **workflow** of the booking process, covering seat selection, temporary seat reservation, payment processing, and final booking confirmation.

Create detailed **low-level design (LLD)**, including data flow diagrams and state diagrams to handle different booking states.

C. Implementation Planning

Break down the functionality into **microservices** or modules, including:

Seat Selection Service

Booking Service

Payment Service

Notification Service

D. **Development Phases**

Use **agile sprints** for iterative development:

Sprint 1: Implement seat selection and temporary reservation.

Sprint 2: Integrate payment processing.

Sprint 3: Add booking confirmation and notification.

At each stage, perform **unit testing** and **integration testing** for seamless function.

E. **User Testing and Feedback**

Conduct **user acceptance testing (UAT)** to ensure usability and handle edge cases like session timeouts, seat unavailability, and payment failures.

2. Implementation Strategy

The **Book a Ticket** feature's implementation is divided into several parts:

A. Seat Selection and Reservation

Real-Time Seat Availability: The seat selection service provides users with a real-time layout of available and reserved seats.

Temporary Seat Locking: Seats are locked for a short time (e.g., 10 minutes) once selected. This prevents other users from booking the same seats.

B. **Booking Confirmation and Payment**

Create Booking Record: Once seats are selected, a provisional booking record is created with a PENDING status.

Payment Integration: Payment details are handled securely by integrating with third-party payment providers. Payment status is checked before moving forward.

Confirm or Release Seats: If payment is successful, the booking status changes to CONFIRMED, and the seat lock becomes permanent. In case of payment failure, the system releases the seats.

C. Notification

Email and SMS Confirmation: Upon successful booking, confirmation notifications are sent to the user.

Reminders: Notifications are sent closer to the showtime as reminders.

3. Performance Optimization and Metrics Optimization

A. Load Balancing and Scaling

Horizontal Scaling:

Use **horizontal scaling** for services like seat selection, booking, and payment to handle high traffic volumes.

Set up multiple instances of services to ensure high availability and distributed load handling.

Auto-Scaling:

Enable **auto-scaling** for backend servers during peak hours, such as weekends or special movie releases.

B. **Payment Process Optimization**

Asynchronous Processing:

Payments are handled asynchronously, with a callback from the payment provider updating booking status. This minimizes user wait time and improves system responsiveness.

Notifications and Confirmations:

Track success rates for notifications (email/SMS) sent to users, monitoring delivery failures.

Mainly used API Request Payload

1. **User Signup Payload**

```
curl --location 'http://moviebooking:8083/auth/signup' \
--header 'Content-Type: application/json' \
--data-raw '{
    "email":"bindala1@gmail.com",
     "password":"Aabb@3344",
    "name": "Aditya",
    "phoneNumber": "9538965425"
}
```

2. **User Login Payload**

```
curl --location 'http://moviebooking:8000/auth/login' \
--header 'Content-Type: application/json' \
--data-raw '{
     "email":"bindaladi1@gmail.com",
     "password":"Aabb@3344"
}
```

3. **Get All Movies**

```
curl --location 'http://moviebooking:8081/movie/movies' \
--data ''
```

4. **Book Ticket**

```
curl --location 'http://moviebooking:8081/ticket/bookTicket' \
--header 'Content-Type: application/json' \
--data '{
    "showId":1,
    "userId":452,
    "seatsToBook":["1A","1B","1C"]
}
```

5. **Get Ticket**

```
curl --location 'http://moviebooking:8081/ticket/1' \
--data ''
```

6. **Complete Payment**

```
curl --location 'http://moviebooking:8085/payment/1' \
--header 'Content-Type: application/json' \
--data '{
    "cardNumber":4857946924752484,
    "cardHolder": "ADITYA BINDAL",
    "cvv" : 343,
    "expiryDate": "10-11-25"}'
```

Deployment Flow

Deploying the BookMyMovie system on AWS involves setting up infrastructure that supports scalability, high availability, and fault tolerance. AWS provides numerous services for compute, storage, networking, database management, and monitoring that can be configured to meet the needs of a high-traffic, distributed ticket booking platform. Here is a breakdown of the deployment strategy and the AWS services used for each component.

1. AWS Architecture Overview

The system's AWS deployment would follow a microservices-based architecture across multiple availability zones for resilience. Each service (e.g., user management, booking, payment, notification etc.) would be hosted independently, allowing for easy scaling and fault isolation. The core AWS services utilized include EC2, ECS/EKS, RDS, S3, CloudFront, Lambda, and API Gateway.

2. **Deployment Architecture Components**

A. Compute Layer

1. Elastic Load Balancer (ELB)

Application Load Balancer (ALB) distributes incoming traffic across multiple instances of each microservice.

ALB is configured to route traffic based on URL paths (e.g., /movies, /booking) to the appropriate backend services.

2. Compute Services

Amazon ECS/EKS (Elastic Container Service / Elastic Kubernetes Service)

- Microservices are containerized using Docker and deployed on ECS or EKS clusters.
- ECS Fargate mode enables serverless compute for containers, so we do not manage the underlying infrastructure.
- Alternatively, EC2 Auto Scaling groups can be used if instances are required instead of containers.

AWS Lambda

- Used for serverless, asynchronous tasks (e.g., notifications, analytics processing).
- Allows for scalable, event-driven processing of lightweight tasks like sending confirmation emails or SMS.

3. Auto Scaling

- Auto Scaling Groups adjust the number of EC2 instances or ECS tasks based on load.
- Scaling policies can be configured for traffic peaks (e.g., weekends, new movie releases) to ensure seamless availability.

B. Database Layer

Amazon RDS (Relational Database Service)

- RDS MySQL is used for transactional data storage for key entities like users, bookings, theaters, screens, and shows.
- Multi-AZ deployment ensures high availability by replicating data across multiple availability zones.
- Read replicas are used to distribute read traffic and reduce load on the primary database.

C. Storage Layer

1. Amazon S3

- Used to store static assets such as movie posters, user profile images, and booking receipts.
- Configured with S3 Lifecycle Policies for cost management (e.g., moving older assets to cheaper storage classes).

2. CloudFront (Content Delivery Network)

- Distributes static content globally and caches it closer to users, reducing load times for images and other assets.
- S3 is configured as the origin for CloudFront, allowing for global distribution of media assets.

D. API Management

1. API Gateway

- Serves as a managed gateway for accessing backend services.
- API Gateway routes requests to appropriate microservices (e.g., movie service, booking service) and enforces security policies.
- Integrates with Lambda functions for serverless APIs and can throttle requests to protect backend systems.

2. Authentication and Authorization

- AWS Cognito manages user registration, login, and JWT token-based authentication.
- Cognito integrates with API Gateway to ensure secure access to services.

E. Payment Processing

1. Integration with External Payment Provider

- Payments are handled via a third-party payment gateway (e.g., Stripe) to ensure PCI compliance.
- AWS Secrets Manager securely stores and manages API keys for these integrations.

F. Notifications

- Notification service used to send email or SMS notifications for booking confirmations, payment status, and reminders.

G. Monitoring and Logging

1. CloudWatch

- Monitors logs, performance metrics, and system health, alerting the operations team for any anomalies.

- CloudWatch Alarms trigger scaling actions based on metrics (CPU usage, latency) and notify the team if thresholds are breached.

2. AWS X-Ray:

- Provides distributed tracing to analyze and debug requests as they pass through various microservices, helping to identify bottlenecks in the system.

AWS CloudTrail:

- Tracks API activity and changes to the infrastructure, ensuring auditability and security compliance.

Deployment Strategy

A. CI/CD Pipeline

- CodePipeline and CodeBuild provide a continuous integration and deployment pipeline.

- CodePipeline automates code changes from development to production, running tests, builds, and deployments.

- CodeDeploy deploys updates to ECS or EC2 instances without downtime, using blue/green deployment or rolling updates.

Performance Optimization and Scaling

1. **Auto Scaling Policies**

    - Configure auto-scaling based on CloudWatch metrics to automatically adjust the number of EC2 instances or ECS tasks.

2. **Database Scaling**

    - Read replicas in RDS handle read-heavy workloads.

3. **Content Delivery**

    - CloudFront caches and delivers static and media content globally, reducing load on origin servers.

4. **Security Scaling**

    - WAF (Web Application Firewall) integrated with CloudFront to protect against DDoS attacks and malicious requests.

    - IAM roles and policies manage fine-grained access controls for services and users within AWS.

Step-by-Step Deployment Process

Step 1: Set Up the VPC

1. **Create a VPC**

    - Define a CIDR range (e.g., 10.0.0.0/16) for the VPC.

2. **Create Subnets**

    - Set up multiple **public** and **private subnets** across **two or more availability zones** (for high availability).

    - Public subnets for resources that need internet access (e.g., load balancers).

- Private subnets for backend resources (e.g., application servers, databases).

3. **Add an Internet Gateway**

    - Attach an **Internet Gateway** (IGW) to the VPC to allow public subnets to access the internet.

4. **Create NAT Gateway**

    - Deploy a **NAT Gateway** in a public subnet for internet access for instances in private subnets (e.g., for updates, logging).

5. **Set Up Route Tables**

    - Create separate route tables for public and private subnets.
    - Public route tables should include a route to the Internet Gateway.
    - Private route tables should include a route to the NAT Gateway.

Step 2: Set Up Compute Resources for Services

1. **Choose a Containerization Platform**

    - **ECS (Elastic Container Service)** with Fargate or **EKS (Elastic Kubernetes Service)** for managing microservices.
    - Alternatively, use **EC2 Auto Scaling** if containers are not used.

2. **Define and Deploy Microservices**

    - Deploy each microservice (e.g., movie service, notification service, payment service) as a separate container in ECS/EKS.
    - Set up **Auto Scaling** for containers or EC2 instances based on load metrics.

3. **Configure an Application Load Balancer (ALB)**

- Use **ALB** to route incoming traffic to the appropriate service.
- Configure ALB with **path-based routing** (e.g., /movies, /bookings) for different microservices.
- Attach ALB to public subnets.

Step 3: Set Up Databases

1. **Deploy Relational Database (RDS)**
    - Use **Amazon RDS MySQL** for data storage.
    - Enable **Multi-AZ deployment** for high availability.
    - Use **read replicas** to distribute read traffic and reduce load on the primary database.

Step 4: Implement Storage and Content Delivery

1. **Set Up Amazon S3**:
    - Store static content like images (e.g., movie posters), user profile pictures, and booking receipts in **S3**.
    - Use **S3 Lifecycle Policies** to manage storage costs.
2. **Use CloudFront for Content Delivery**:
    - Set up **CloudFront CDN** to cache and deliver static content globally, improving latency for users accessing images and static files.

Step 5: Configure API Gateway and Authentication

1. **Set Up API Gateway**
    - Configure **API Gateway** as a managed entry point for client requests.
    - Integrate API Gateway with backend services (e.g., ECS/EKS or Lambda functions for lightweight operations).

2. **Set Up User Authentication**

    - Use **AWS Cognito** for user authentication and management (supports OAuth2, JWT).
    - Integrate Cognito with API Gateway for secure API access.

Step 6: Implement Payment Processing

1. **Integrate with Payment Gateway**

    - Configure connections with a third-party payment provider.
    - Use **Secrets Manager** to securely store API keys and other credentials for payment providers.

2. **Enable Asynchronous Payment Confirmation**:

    - Use **SQS (Simple Queue Service)** to handle payment events (e.g., pending, success).
    - Configure Lambda functions to process events from the payment provider and update booking status.

Step 7: Set Up Monitoring and Logging

1. **Configure Amazon CloudWatch**:

    - Enable **CloudWatch Metrics** to monitor key parameters (e.g., CPU, memory, request latency).
    - Use **CloudWatch Alarms** to trigger notifications for abnormal metrics or errors.

2. **Enable Logging**:

    - Set up **CloudWatch Logs** to capture logs from ECS tasks, EC2 instances, and other resources.

- Use **AWS X-Ray** for distributed tracing across microservices to diagnose and troubleshoot issues.

3. **VPC Flow Logs**

    - Enable **VPC Flow Logs** to capture information about IP traffic within the VPC for monitoring and security auditing.

Step 8: Set Up CI/CD Pipeline for Automated Deployment

1. **Define a CI/CD Pipeline**:

    - Use **CodePipeline** for continuous integration and continuous delivery.

    - Integrate **CodeBuild** for automated testing and building Docker images.

    - Deploy Docker images to **ECR (Elastic Container Registry)**, the managed Docker repository.

2. **Automated Deployment**:

    - Use **CodeDeploy** or native ECS/EKS rolling updates for zero-downtime deployments.

    - Set up blue/green deployment or rolling updates to deploy new code without impacting users.

Step 9: Implement Security Controls

1. **Configure Security Groups and Network ACLs**

    - Define **Security Groups** to control inbound and outbound traffic to EC2 instances, databases, and load balancers.

    - Use **Network ACLs** for additional subnet-level security.

2. **Set Up Web Application Firewall (WAF)**:

    - Use **AWS WAF** with **CloudFront** or **API Gateway** to protect against DDoS attacks and malicious requests.

3. **Use IAM for Access Control**:
    - Define **IAM roles and policies** to manage access to resources.
    - Apply least privilege principles to all users and services.

Step 10: Scaling and Performance Optimization

1. **Set Up Auto Scaling**:
    - Enable **Auto Scaling** for EC2 instances or ECS tasks based on metrics such as CPU, memory usage, or request count.
2. **Enable Read Replicas and Database Scaling**:
    - Use **RDS Read Replicas** to offload read-heavy queries.
    - Configure DynamoDB with **auto-scaling** for high-demand tables.
3. **Optimize Caching**:
    - Use **ElastiCache** to cache frequently accessed data and reduce load on primary databases.
4. **Tune API Gateway Caching**:
    - Enable **API Gateway caching** for infrequently changing responses (e.g., showtimes, seat maps).

Step 11: Testing and Validation

1. **Testing Across Environments**:
    - Set up multiple environments (e.g., development, staging, production).
    - Use automated and manual testing to validate deployments in staging before pushing to production.

Step 12: Go Live and Monitor

1. **Gradual Rollout (Blue/Green Deployment)**:

    - Gradually roll out the application in production, monitoring for issues and rolling back if needed.

2. **Continuous Monitoring**:

    - Use CloudWatch, VPC Flow Logs, and X-Ray to continuously monitor application performance and network health.

    - Configure alarms and dashboards to track key metrics and alert the operations team for any issues.

Tools and Technologies Used

1. Java

- Java is a popular, object-oriented programming language that is used to create a variety of applications, including mobile apps, enterprise software, and server-side technologies

2. MySQL

- MySQL is an open-source relational database management system (RDBMS) that stores and manages data. It is known for its reliability, performance, scalability, and ease of use

3. Spring Boot

- Spring Boot is an open-source tool that helps developers create web applications and microservices using Java:

- A Java framework that uses the Spring Framework to simplify the development of Java-based applications

- Spring Boot provides a platform for developers to quickly get started with a production-grade Spring application. It uses an opinionated approach to configuration and autoconfiguration to minimize the amount of configuration and setup required

- Spring Boot can help developers create REST APIs with minimal configuration and without the need for complex XML configurations. It also simplifies the creation of the Model component, which corresponds to all the data-related logic

4. AWS and Digital Ocean Cloud

- A cloud platform is a combination of hardware, software, and operating systems that provides cloud computing services to customers.

- Cloud platforms create a virtual pool of shared resources for data storage, compute, and network services.

- Customers can access these resources as needed, and only pay for what they use.

5. RabbitMQ

- RabbitMQ is a free, open-source message broker that facilitates communication between different parts of an application.

- RabbitMQ acts as a mediator between message producers and consumers, enabling asynchronous messaging and decoupling the sender and receiver. It supports a variety of messaging patterns, including publish/subscribe (Pub/Sub), request/reply, and point-to-point communication.

- RabbitMQ uses queues to store messages. A queue is like a post box in RabbitMQ, and many producers can send messages to one queue. Many consumers can also try to receive data from one queue.

- RabbitMQ is a popular platform for building real-time applications. For example, a web application can use RabbitMQ to process a job in the background while the user continues to use the application.

- RabbitMQ supports several open standard protocols, including AMQP, MQTT, and STOMP. It also provides features like message acknowledgments, persistence, and routing to ensure message delivery and reliability.

- RabbitMQ can be deployed on cloud environments, on-premises, or on your local machine.

6. Twilio

- Twilio is a cloud communications platform that helps developers build communication features into their applications. Twilio offers a variety of products and services, including:

- **MessagingX**: Send and receive SMS, MMS, and OTT messages

- **Programmable Voice**: Integrate calling functionalities like PSTN, SIP, and VoIP.

- **Video**: Build scalable, secure, real-time video and HD audio applications

- **Live**: Create applications that host live virtual events with interactive stream able content

- **Customer Data Platform (CDP):** Acquire customers and keep them engaged with personalized journeys

- **User Authentication & Identity:** Drive secure logins and reduce fraud

- **Twilio Flex:** Offer proactive support and personalized recommendations

Conclusion

The *BookMyMovie* project offered an enriching opportunity to apply core concepts of web development and software engineering in a real-world scenario. Throughout the development process, several key technical areas were explored and implemented effectively:

1. **Backend Development**

- Designed and built RESTful APIs to manage server-side operations and ensure seamless communication between frontend and backend.
- Followed standardized practices for API structuring and response formatting.
- Implemented secure user authentication, registration, and session management using technologies such as JWT (JSON Web Tokens).
- Managed request-response cycles with efficient logging, validation, and error handling mechanisms.

2. **Database Management**

- Designed relational database schemas, defined relationships between tables, and optimized query performance.
- Performed core CRUD (Create, Read, Update, Delete) operations, essential for ticket booking and cancellation features.

3. **DevOps and Deployment**

- **Version Control**: Utilized Git and GitHub for source code management, team collaboration, and version tracking.
- **CI/CD**: Implemented Continuous Integration and Deployment pipelines using tools like GitHub Actions or Jenkins to streamline code testing and deployment.
- **Cloud Hosting**: Deployed the application on cloud platforms like AWS or DigitalOcean for reliable and scalable access.
- **Dockerization**: Used Docker to containerize the application, ensuring consistent deployment across various environments.

4. **Payment Integration**

- Integrated secure payment gateways (e.g., Stripe) to facilitate online transactions for movie ticket bookings.

5. **Testing and Debugging**

- Conducted unit and integration testing using frameworks like Jest, Mocha, or Chai to validate the functionality of individual components and the system as a whole.
- Implemented robust error detection and handling strategies to maintain a smooth and user-friendly experience.

Practical Applications

All above technologies are used by companies like LinkedIn, Netflix, and Walmart, Facebook, Instagram, google map, many mobile applications etc.

Limitations

1. **RESTful APIs**

    - **Limitations**: REST APIs can be rigid and may require multiple requests to fetch related data, leading to slower performance.

    - **Cost Implications**: Over fetching or under fetching data can increase server load and, consequently, operational costs.

    - **Suggestions for Improvement**: Consider using GraphQL, which allows clients to request only the data they need, reducing network load. Alternatively, gRPC can be used for high-performance microservices.

2. **Database Management**

    - Limitations: SQL databases struggle with scalability for large datasets, while NoSQL databases can lack ACID (Atomicity, Consistency, Isolation, Durability) compliance, potentially compromising data integrity.

    - Cost Implications: Scaling databases can be costly, as SQL databases require vertical scaling and NoSQL often demands horizontal scaling, both of which add costs.

    - Suggestions for Improvement: Consider a hybrid approach (using both SQL and NoSQL) to optimize data storage and retrieval based on data type. For example, MongoDB for user activity logs and PostgreSQL for transactions. Use cloud-hosted databases with managed scaling options to minimize operational complexity and cost.

3. **Version Control (Git & GitHub)**

    - **Limitations**: Version control systems don’t inherently prevent accidental merges or unintended overwrites, which can cause deployment issues.

    - **Cost Implications**: Private repositories and CI/CD integration may incur costs in large teams, especially on GitHub or other enterprise-grade platforms.

- **Suggestions for Improvement**: Implement automated checks and code review processes to prevent unintended code changes. Use alternatives like GitLab, which offers CI/CD with fewer additional costs in its free tier.

4. **CI/CD Pipelines**

   - **Limitations**: Initial setup for CI/CD can be complex, and debugging build failures across distributed systems can be challenging.

   - **Cost Implications**: CI/CD services can increase costs, especially when scaling to multiple environments or requiring high availability for automated pipelines.

   - **Suggestions for Improvement**: Use open-source CI/CD solutions like Jenkins for more customization and cost savings. Also, schedule pipelines to run only when needed (e.g., on key branches or tagged commits) to reduce costs.

5. **Cloud Platforms (AWS)**

   - **Limitations**: While cloud platforms provide scalability, they can lead to unpredictable costs, especially with high traffic or resource-intensive operations.

   - **Cost Implications**: Cloud services often charge based on usage, which can significantly increase operational costs as traffic grows.

   - **Suggestions for Improvement**: Implement resource monitoring and use serverless or containerized options to reduce costs by only paying for what you use. Platforms like DigitalOcean may offer more affordable solutions for smaller-scale projects.

6. **Payment Integration**

   - **Limitations**: Payment gateways charge transaction fees and may not offer full customization options for checkout experiences.

   - **Cost Implications**: Payment processing fees can add up, especially for high-volume platforms, impacting profit margins.

   - **Suggestions for Improvement**: Explore payment providers with volume discounts or flexible pricing structures. For businesses with higher volumes, integrating directly with bank APIs can sometimes reduce fees.

7. **Testing and Debugging**

- **Limitations**: Writing comprehensive tests can be time-consuming, especially as the codebase grows. Also, not all edge cases can be tested effectively in controlled environments.

- **Cost Implications**: The time required to write and maintain tests increases development costs, though it may save on debugging costs in the long run.

- **Suggestions for Improvement**: Prioritize critical areas for testing to reduce time spent on testing non-essential parts. Consider automated testing tools that allow for script reuse and integration with CI/CD pipelines.

References

https://www.scaler.com/academy/mentee-dashboard/todos

https://www.google.com/search?

https://www.geeksforgeeks.org/design-movie-ticket-booking-system-like-bookmyshow/

https://medium.com/@choudharynishantplawat/java-popcorn-build-your-own-bookmyshow-in-a-day-86ee4a7b2527

https://spring.io/projects

https://www.baeldung.com/spring-boot

https://www.interviewbit.com/blog/spring-boot-architecture/