Technical Architecture Document
Title: Book My Cinema

Author: Mr. Umesh Singh

Version: V1.0

Date: 22 Sep 2024

Table of Contents

1. Introduction

The **Book My Cinema (BMC)** project is a micro services-based application designed to offer users a seamless experience when booking movie tickets, selecting theaters, managing shows, and processing payments. The system consists of various independent services, all orchestrated to provide a high-performing and scalable platform

1.1 Overview:

The Book My Cinema platform is designed to provide users with a seamless movie booking experience. The system is built using micro services architecture, ensuring scalability and independent deployment of each service. Below is a detailed explanation of each service and its role in the system.

1.2 Key Features:

- **User Authentication:** Secure login, registration, and authentication mechanisms.
- **Movie Listings:** Access to a catalog of movies with show timings and availability.
- **Theater Selection:** Users can select theaters based on location, preferences, and availability.
- **Ticket Booking:** Real-time ticket booking and seat selection.
- **Payment Gateway:** Secure, multi-option payment processing.
- **Notifications:** Email and SMS notifications for booking confirmations, cancellations, and promotions.

2. Details Architecture Overview

The BMC will be built using a micro services-based architecture to ensure modularity, scalability, and high availability. Each component will have its own responsibilities, and they will communicate with each other through well-defined APIs. Here's a breakdown of the architecture and the chosen technologies for each component.

2.1 High-Level Design Components

1. **API Gateway Server**
    - **Purpose**: Acts as the entry point for all client requests
    - **Responsibilities**:
        - ❖ Routing requests to the appropriate service.
        - ❖ Load balancing across multiple instances of micro services.
        - ❖ Rate limiting and request throttling.
        - ❖ User authentication and authorization in coordination with the Authentication Service.
        - ❖ Logging and monitoring incoming traffic.
    - **Tech Stack**: Spring Cloud Gateway, Java 17, Spring Boot.
2. **Load Balancer Server**
    - **Purpose**: Ensures even distribution of traffic across multiple instances of services.
    - **Responsibilities**:
        - ❖ Distribute incoming requests evenly to avoid overload on a single instance.
        - ❖ Improve system availability and fault tolerance.
        - ❖ Health checks of underlying services to remove unhealthy instances from the pool.
    - **Tech Stack**: Spring Cloud Eureka Load Balancer, Java 17, Spring Boot.
3. **Authentication Service**
    - **Purpose**: Manages user authentication and authorization.
    - **Responsibilities**:
        - ❖ User registration and login.
        - ❖ Token-based authentication (JWT).
        - ❖ Role-based access control for different system roles (admin, user).
        - ❖ Password encryption and verification.
    - **Tech Stack**: Spring Security, OAuth2, MySQL, Java 17, Spring Boot.
4. **Cache Service**
    - **Purpose**: Improves the performance of the platform by caching frequently accessed data**.**
    - **Responsibilities:**
        - ❖ Cache movie details, theater information, and seat availability to reduce database load.
        - ❖ Implement cache eviction policies (LRU, TTL) to ensure fresh data.
        - ❖ Provide distributed caching across microservices to maintain consistency.
    - Tech **Stack**: Java 17, Redis, Spring Boot.

5. **Log Stream Service**
    - **Purpose**: Improves the performance of the platform by caching frequently accessed data**.**
    - **Responsibilities:**
        - ❖ Aggregate logs from all microservices in real-time.
        - ❖ Stream logs to centralized logging systems like ELK (Elasticsearch, Logstash, Kibana) or AWS CloudWatch.
        - ❖ Enable log filtering and querying for faster debugging.
        - ❖ Support log retention and rotation policies.
    - Tech **Stack**: Java 17,Apache Kafka Stream, Spring Boot
6. **Config Service**
    - **Purpose**: Manages configurations across all services.accessed data**.**
    - **Responsibilities:**
        - ❖ Store and serve configurations in a centralized manner.
        - ❖ Handle environment-specific configurations (dev, test, prod).
        - ❖ Version control for configuration changes.
    - Tech **Stack**: Java 17, Spring Boot, Spring Cloud Config, Github
7. **Movie Service**
    - **Purpose**: Provides information about movies available for booking**.**
    - **Responsibilities:**
        - ❖ Manage movie data including title, genre, cast, duration, and rating.
        - ❖ Fetch movie listings from external sources or internal databases.
        - ❖ Serve movie metadata to other services like Show Service and Theater Service.
    - Tech **Stack**: Java 17, Spring Boot, MySQL
8. **Show Service**
    - **Purpose**: Handles scheduling and availability of movie shows**.**
    - **Responsibilities:**
        - ❖ Manage show timings, dates, and movie-to-theater mapping.
        - ❖ Fetch show availability for specific dates and times.
        - ❖ Integration with Theater Service to manage seating.
    - Tech **Stack**: Java 17, Spring Boot, Micro service, MySQL
9. **Theater Service**
    - **Purpose**: Manages theater details and seating arrangements**.**
    - **Responsibilities:**
        - ❖ Store and retrieve details of cinema halls and seating configurations.
        - ❖ Coordinate with Show Service for seat availability.
        - ❖ Handle seat selection during ticket booking.
    - Tech **Stack**: Java 17, Spring Boot, Micro service, MySQL.
10. **Ticket Service**
    - **Purpose**: Manages ticket generation and booking**.**
    - **Responsibilities:**
        - ❖ Handle booking requests and issue tickets.
        - ❖ Confirm seat availability before booking.

- ❖ Coordinate with Payment Service for payment confirmation before finalizing bookings.
- ❖ Generate and send ticket details to the Notification Service.

- Tech **Stack**: Java 17, Spring Boot, Micro service, MySQL, Kafka for event-driven

11. Review Service

- **Purpose**: Allows users to submit and view movie reviews**.**
- **Responsibilities:**
  - ❖ Store and retrieve user-submitted reviews and ratings.
  - ❖ Ensure moderation and filtering of reviews for abusive content.
  - ❖ Serve aggregated review scores to the Movie Service.
- Tech **Stack**: Java 17, Spring Boot, Micro service, MongoDB, OpenSearch

12. Payment Service

- **Purpose**: Manages payments for ticket bookings**.**
- **Responsibilities:**
  - ❖ Integration with third-party payment gateways (Tabapay, PPS Merchant, PayPal, etc.).
  - ❖ Handle payment authorization and capture.
  - ❖ Support multiple payment methods (credit/debit cards, wallets, net banking).
  - ❖ Generate payment invoices.
- Tech **Stack**: Java 17, Spring Boot, Micro service, MongoDB.

13. Notification Service

- **Purpose**: Sends notifications to users regarding bookings and other activities**.**
- **Responsibilities:**
  - ❖ Notify users about booking confirmation, cancellations, and reminders for upcoming shows.
  - ❖ Support for SMS, email, and push notifications.
  - ❖ Integration with external notification services AWS SNS.
- Tech **Stack**: Java 17, Spring Boot, AWS SDK, Micro service, AWS SNS.

2.2 Technology Selection

Backend Technologies:

1. **Java 17:**
   Long-term support (LTS) release offering enhanced performance, security, and modern language features.
2. **Spring Boot 3.2.8**
   Provides a streamlined approach for building micro services with minimal configuration, embedded servers (like Tomcat), and ease of integration with other Spring modules.

Database:

5. **MySQL:**
   Relational database that ensures data consistency and ACID compliance, making it suitable for transactional data (e.g., ticket bookings).
6. **MongoDB:**
   NoSQL database optimized for unstructured data, ideal for managing user reviews, movie data, or any semi-structured information that scales horizontally.

Caching:

7. **Redis Cache:**
   An in-memory data structure store that can improve response times for frequently accessed data, reducing database load and enabling faster retrieval.

Security:

8. **Keycloak:**
   Open-source identity and access management solution for adding authentication, authorization, and user management, which simplifies security for the entire platform.
9. **Snyk:**
   Snyk is a security platform designed to help developers find, prioritize, and fix vulnerabilities in their applications, dependencies, containers, and infrastructure

Monitoring & Observability:

9. **Grafana:**
   Provides real-time visualization and monitoring of system metrics, allowing the team to track performance, troubleshoot issues, and ensure high availability.
10. **Telemetry (OpenTelemetry):**
    Open standard for collecting distributed traces, logs, and metrics, providing end-to-end visibility and debugging for distributed applications.
11. **Tempo:**
    A distributed tracing backend for Grafana, which helps in analyzing performance and bottlenecks in micro service architectures.

Containerization & Orchestration:

12. **Docker:**
    Provides lightweight containers to package the application with all dependencies, ensuring consistency across different environments (local, staging, production).
13. **Kubernetes:**
    Powerful container orchestration platform, allowing you to scale services, manage deployments, and handle failover effectively across clusters.
14. **Helm Charts:**
    Helm simplifies Kubernetes application deployment by managing configuration files and templates, making it easier to deploy, upgrade, or rollback microservices.

Build & Version Control:

15. **Maven:**
    Provides dependency management, a build lifecycle, and easy integration with CI/CD pipelines for automated deployments.
16. **GitHub:**
    Provides version control, collaboration, and integration with CI/CD workflows, ensuring smooth and efficient team collaboration.

Messaging & Notifications:

17. **AWS SNS:**
    A fully managed pub/sub messaging service that can send notifications to various downstream systems like user notifications, SMS, or emails, ensuring scalability and reliability.
18. **SMTP Server:**
    Allows sending automated emails (e.g., booking confirmations, notifications) to users, a key feature for any booking system.

Streaming:

19. **Apache Kafka Stream:**
    Real-time data streaming between services for events like booking, payment processing, and notifications. Ensures scalability and reliability in handling large volumes of events, essential for streaming ticket data and user interactions.

Cloud Services:

20. **AWS ECR (Elastic Container Registry):**
    Managed Docker container registry that simplifies the storage, management, and deployment of container images, integrating seamlessly with AWS services.
21. **AWS EKS (Elastic Kubernetes Service):**
    Managed Kubernetes service that automates cluster management tasks such as upgrades and scaling, while providing seamless integration with AWS services.

2.3 Design Pattern

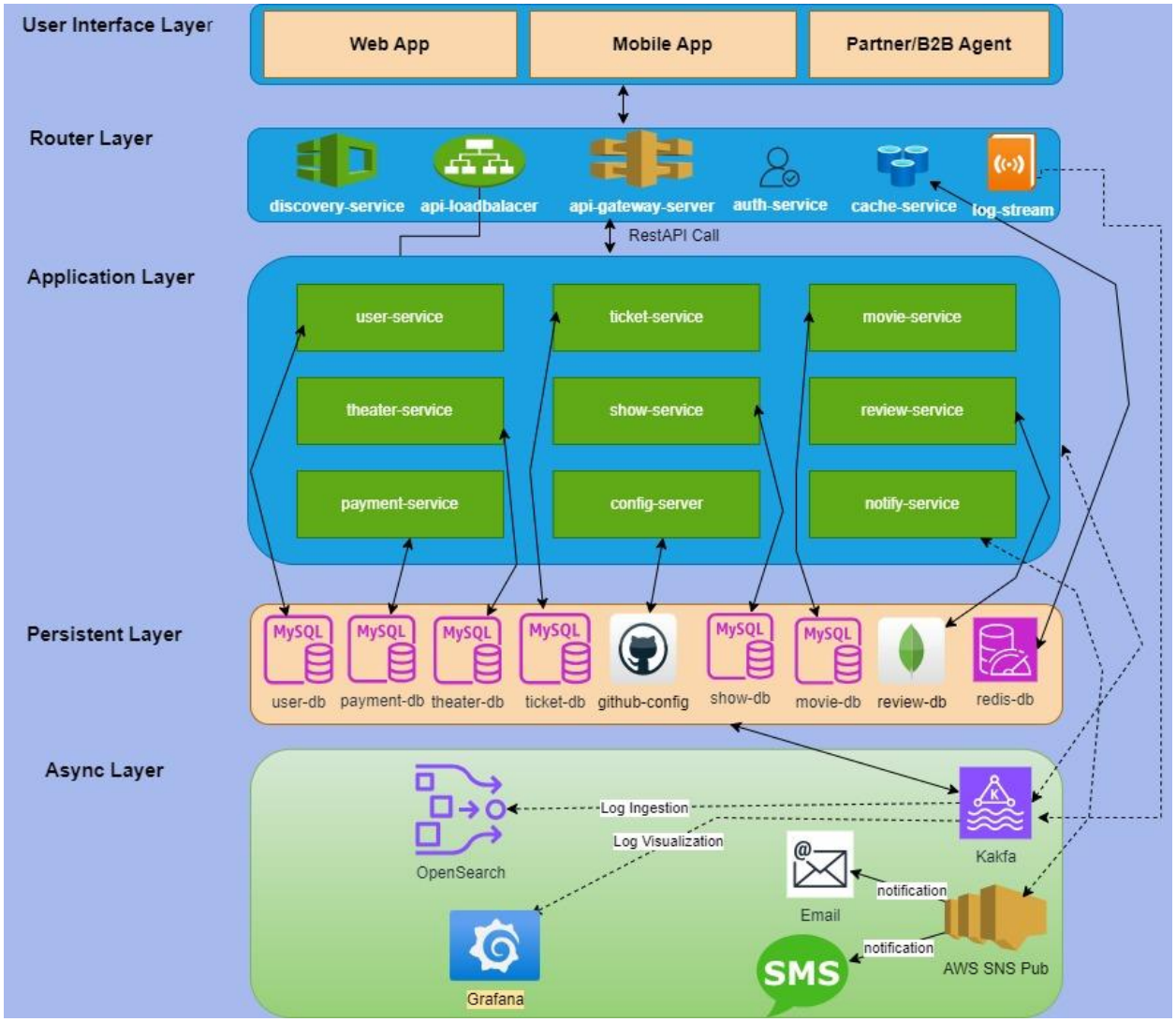
For a **Book My Cinema** project using Java and micro services, several design patterns can be employed to ensure scalability, flexibility, and maintainability. Given the nature of the project, a combination of patterns will be suitable for different aspects of the system. Here are the best-suited design patterns:

1. **Micro services Architecture Pattern:**
   - ❖ It enables you to decompose the system into independent services like Movie Service, Show Service, Ticket Service, Payment Service, etc.
   - ❖ Each service can be developed, deployed, and scaled independently.
   - ❖ Promotes separation of concerns, making it easier to maintain and scale the individual components.
2. **API Gateway Pattern:**
   - ❖ Acts as a single entry point to all the services (Movie Service, Theater Service, etc.).
   - ❖ Handles request routing, composition, and sometimes authentication and rate limiting.

❖ Centralizes requests from users or external clients, simplifying access management.

3. **Service Discovery Pattern (Infrastructure Pattern):**
   - ❖ Micro services in a distributed system need to locate each other.
   - ❖ A service registry (e.g., Eureka, Consul) can dynamically register instances of services.
   - ❖ The system will auto-discover available services without hardcoding URLs.
4. **Circuit Breaker Pattern (Resilience Pattern)**
   - ❖ Helps in handling faults gracefully when calling remote services like Payment Service or Notification Service.
   - ❖ Prevents cascading failures by stopping retries to a failing service and providing fallback mechanisms.
   - ❖ Tools like **Resilience4j**
5. **Event-Driven Pattern (Asynchronous Communication):**
   - ❖ The system can be decoupled using events for certain actions, such as sending notifications after a booking or updating inventory after a transaction.
   - ❖ Allows different services to react asynchronously to events via a message broker (like Apache Kafka or AWS SNS).
6. **Saga Pattern (for distributed transactions):**
   - ❖ In microservices, when a transaction spans across multiple services (like Payment Service, Ticket Service), Saga can ensure consistency through either a series of compensating actions or forward recovery.
   - ❖ Suitable for handling distributed transactions across different services without locking resources.
7. **Observer Pattern (for real-time notifications)**
   - ❖ Multiple services need to respond to an event (like a ticket purchase), the observer pattern can be used to notify all relevant services (e.g., notification service, analytics, etc.).

2.4 Architecture Diagram



2.5 Database Schema

MySQL DB Schema

1. CREATE TABLE `movies` (
   `id` int NOT NULL AUTO_INCREMENT,
   `duration` double NOT NULL,
   `genre`
   enum('ACTION','ANIMATED','ANIME','COMEDY','CRIMINAL','DOCUMENTARY','HORROR','ROMANTIC','SCIFICTION','THRILLER') DEFAULT NULL,
   `language`
   enum('BHOJPURI','ENGLISH','HINDI','KANNADA','MALLAYALAM','MARATHI','PUNJABI','TAMIL','TELUGU') DEFAULT NULL,
   `movie_name` varchar(255) NOT NULL,
   `rating` double DEFAULT NULL,
   `release_date` datetime(6) DEFAULT NULL,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4
   COLLATE=utf8mb4_0900_ai_ci;

   CREATE TABLE `show_seats` (
   `id` int NOT NULL AUTO_INCREMENT,
   `is_available` bit(1) NOT NULL,
   `is_food_attached` bit(1) NOT NULL,
   `price` int NOT NULL,
   `seat_no` varchar(255) DEFAULT NULL,
   `seat_type` enum('CLASSIC','PREMIUM') DEFAULT NULL,
   `show_id` int DEFAULT NULL,
   PRIMARY KEY (`id`),
   KEY `FKldtrq74q8syptlbgqag9cw9w1` (`show_id`),
   CONSTRAINT `FKldtrq74q8syptlbgqag9cw9w1` FOREIGN KEY (`show_id`) REFERENCES `shows`
   (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

2. CREATE TABLE `shows` (
   `id` int NOT NULL AUTO_INCREMENT,
   `date` datetime(6) DEFAULT NULL,
   `time` time(6) DEFAULT NULL,
   `movie_id` int DEFAULT NULL,
   `theater_id` int DEFAULT NULL,
   PRIMARY KEY (`id`),
   KEY `FKqdpwhiv5r3lx844pct0eudapk` (`movie_id`),
   KEY `FKq1hxb5a1cutihabqq6lo5ccud` (`theater_id`),
   CONSTRAINT `FKq1hxb5a1cutihabqq6lo5ccud` FOREIGN KEY (`theater_id`) REFERENCES
   `theater` (`id`),
   CONSTRAINT `FKqdpwhiv5r3lx844pct0eudapk` FOREIGN KEY (`movie_id`) REFERENCES
   `movies` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

3. CREATE TABLE `theater` (
   `id` int NOT NULL AUTO_INCREMENT,
   `location` varchar(255) DEFAULT NULL,
   `name` varchar(255) DEFAULT NULL,
   PRIMARY KEY (`id`),
   UNIQUE KEY `UK_71lt5dyvgngmkh75o0rx6myo3` (`location`)

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

4.
```
CREATE TABLE `theater_seats` (
  `id` int NOT NULL AUTO_INCREMENT,
  `seat_no` varchar(255) DEFAULT NULL,
  `seat_type` enum('CLASSIC','PREMIUM') DEFAULT NULL,
  `theater_id` int DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FKe461xicbrhr1l20149loqt7qd` (`theater_id`),
  CONSTRAINT `FKe461xicbrhr1l20149loqt7qd` FOREIGN KEY (`theater_id`) REFERENCES `theater`
(`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

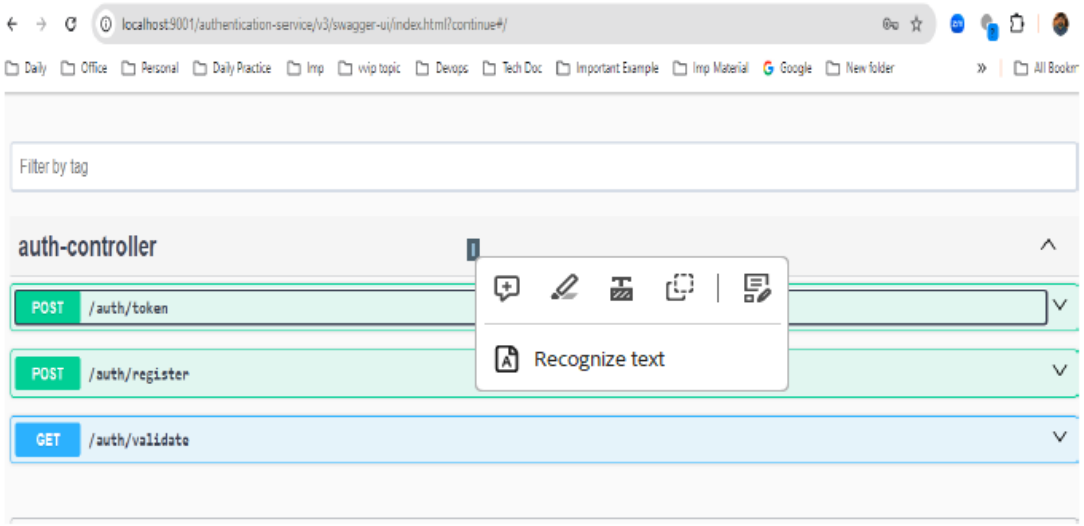
5.
```
CREATE TABLE `tickets` (
  `id` int NOT NULL AUTO_INCREMENT,
  `booked_at` datetime(6) DEFAULT NULL,
  `booked_seats` varchar(255) DEFAULT NULL,
  `total_tickets_price` int NOT NULL,
  `show_id` int DEFAULT NULL,
  `user_id` int DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FKosj8dc2tn2tcidsfimopidq13` (`show_id`),
  KEY `FK4eqsebpimnjen0q46ja6fl2hl` (`user_id`),
  CONSTRAINT `FK4eqsebpimnjen0q46ja6fl2hl` FOREIGN KEY (`user_id`) REFERENCES `users`
(`id`),
  CONSTRAINT `FKosj8dc2tn2tcidsfimopidq13` FOREIGN KEY (`show_id`) REFERENCES `shows`
(`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

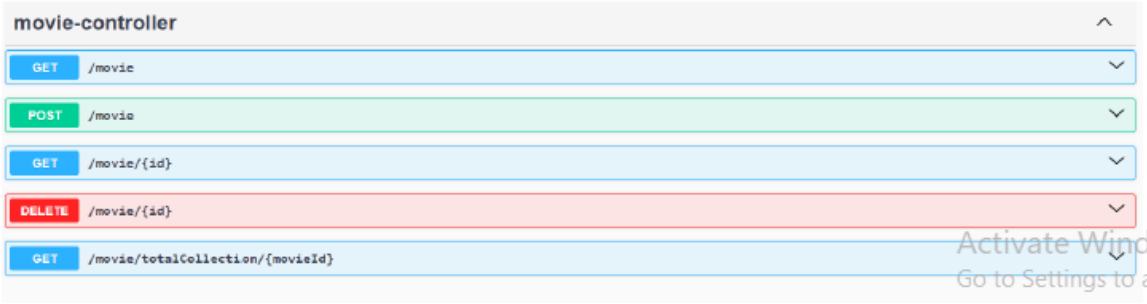
6.
```
CREATE TABLE `user_credential` (
  `id` int NOT NULL AUTO_INCREMENT,
  `email` varchar(255) DEFAULT NULL,
  `name` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

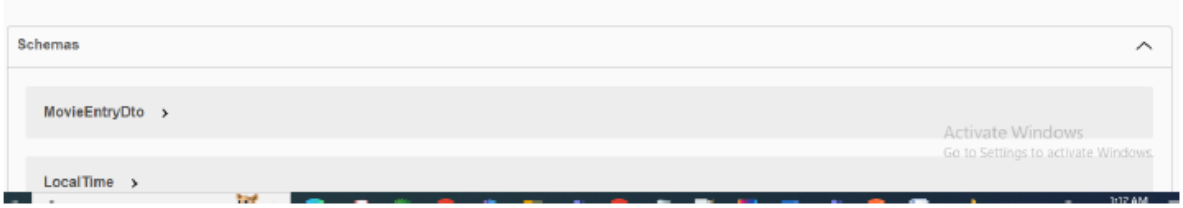
MongoDB (Review Collection)

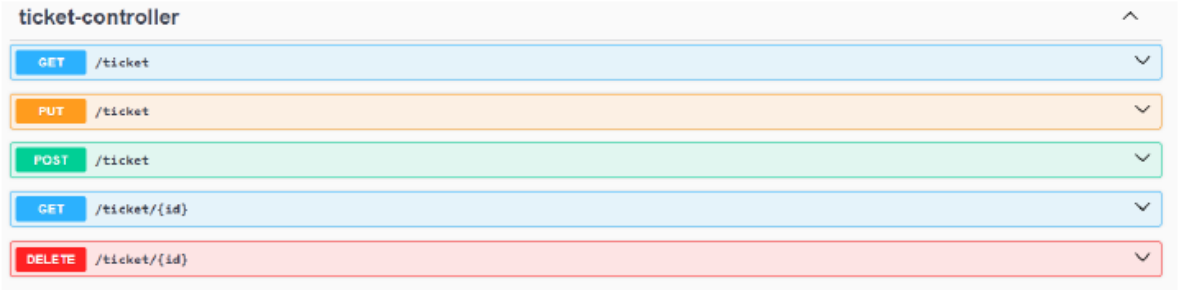
```
{
  "_id": ObjectId("..."),  // Unique ID for the review
  "movieId": ObjectId("..."),  // Reference to the movie being reviewed
  "userId": ObjectId("..."),  // Reference to the user who posted the review
  "reviewText": "Great movie! Very engaging and action-packed.",  // The actual review
  "rating": 4.5,  // Rating out of 5
  "reviewDate": review Date,  // Date when the review was posted
  "likes": 120,  // Number of likes/upvotes the review has received
  "dislikes": 5,  // Number of dislikes/downvotes the review has received
  "comments": [
    {
      "commentId": ObjectId("..."),  // Unique ID for each comment on the review
      "userId": ObjectId("..."),  // User who commented
      "commentText": "I agree with this review!",  // Comment text
      "commentDate": dateAndTime  // Date of the comment
    }
  ],
  "flagged": false,  // If the review is flagged for inappropriate content
  "verifiedPurchase": true  // Indicates if the review is from a verified ticket purchase
}
```

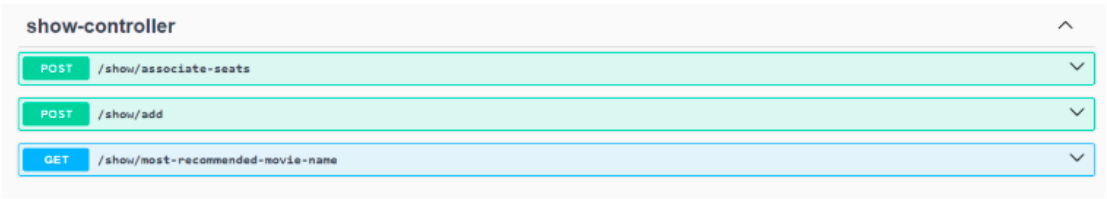
2.6 API Endpoints (OpenAPI Screenshot)











Payment Service API Endpoint:

1. **Process a Payment**
   POST /api/v1/payment
2. **Retrieve Payment Status**
   GET /api/v1/payment/{paymentId}
3. **Cancel a Payment**
   POST /api/v1/payment/{paymentId}/cancel
4. **Issue a Refund**
   POST /api/v1/payment/{paymentId}/refund
5. **Get Payment History for a Use**
   GET /api/v1/payment/user/{userId}

Review Service API Endpoint

1. Create a Review
POST /api/v1/reviews
2. Get Review by ID
GET /api/v1/reviews/{reviewId}
3. Get Reviews for a Movie
GET /api/v1/reviews/movie/{movieId}
4. Update a Review
PUT /api/v1/reviews/{reviewId}
5. Delete a Review
DELETE /api/v1/reviews/{reviewId}
6. Like/Dislike a Review
POST /api/v1/reviews/{reviewId}/like

2.7 Scaling Strategy

Horizontal Scaling:

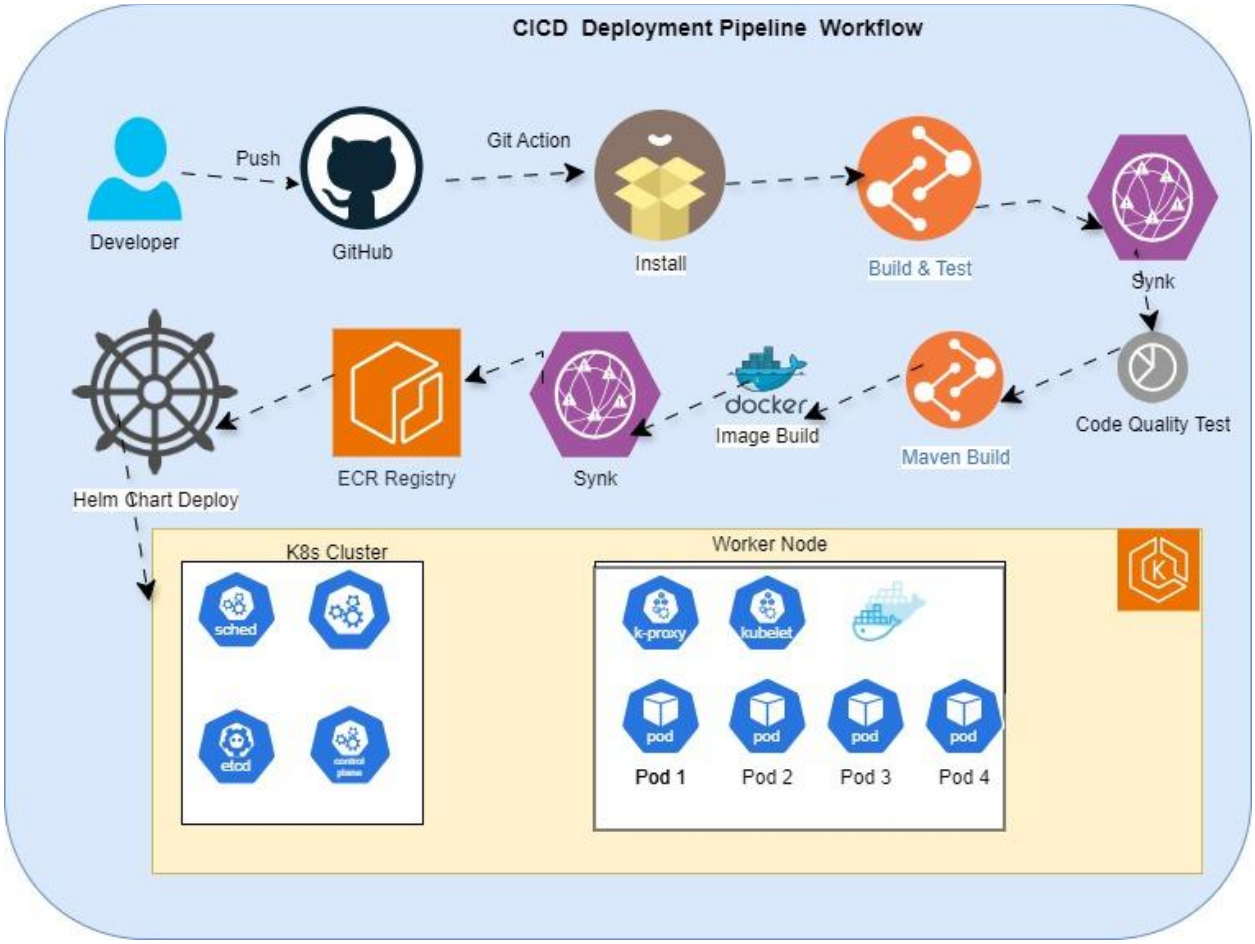
- Load Balancing: Use a load balancer (e.g., Spring Cloud Load Balancer) to distribute incoming requests among multiple instances of each microservice.
- Container Orchestration: Use Kubernetes or AWS EKS to manage scaling and deployment of containerized microservices. Configure Horizontal Pod Autoscaling based on metrics (e.g., CPU usage).

Caching: Use Redis to cache responses for frequently accessed resources (e.g., movie details, user reviews).

Database Sharding:

- MongoDB Sharding: Distribute reviews and other collections across multiple MongoDB instances, ensuring each shard can be scaled independently.
- MySQL Partitioning: For relational data, use partitioning to improve query performance and management.

2.8 CI/CD Deployment Pipeline Workflow



3. Functional Requirement

3.1 Movie Management

FR1.1: Add Movie

- **Description**: Admin users can add new movies to the system.
- **Input**: Movie details (title, genre, description, duration, release date, poster).
- **Output**: Confirmation of movie creation.

FR1.2: Edit Movie

- **Description**: Admin users can edit details of existing movies.
- **Input**: Movie ID and updated details.
- **Output**: Confirmation of movie update.

FR1.3: Remove Movie

- **Description**: Admin users can remove movies from the system.
- **Input**: Movie ID.
- **Output**: Confirmation of movie deletion.

FR1.4: View Movies

- **Description**: Users can view the list of all available movies.
- **Output**: List of movies with details (title, genre, description).

MovieController.java

```
package com.bmc.controller;

import com.bmc.mapper.request.MovieEntryDto;
import com.bmc.model.Movie;
import com.bmc.service.MovieService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
public class MovieController {
    @Autowired
    MovieService movieService;

    @PostMapping("/movie")
    public ResponseEntity<String> addMovie(@RequestBody MovieEntryDto
movieEntryDto){
     try {
         String result= movieService.addMovie(movieEntryDto);
         return new ResponseEntity<>(result,HttpStatus.CREATED);
     }catch (Exception e){
          return new
ResponseEntity<>(e.getMessage(),HttpStatus.BAD_REQUEST);
     }
    }
    @GetMapping("/movie/totalCollection/{movieId}")
    public ResponseEntity<Long> totalCollection(@PathVariable Integer
movieId) {
        try {
            Long result = movieService.totalCollection(movieId);
```

```
            return new ResponseEntity<>(result,HttpStatus.CREATED);
        } catch (Exception e) {
            return new ResponseEntity<>(null,HttpStatus.BAD_REQUEST);
        }
    }
    @GetMapping("/movie")
    public List<Movie> getAllMovies()
    {
        return movieService.getAllMovies();
    }
    //creating a get mapping that retrieves the detail of a specific movie
    @GetMapping("/movie/{id}")
    public Movie getMovie(@PathVariable("id") int id)
    {
        return movieService.getMovieById(id);
    }
    //creating a delete mapping that deletes a specified movie
    @DeleteMapping("/movie/{id}")
    public void deleteBook(@PathVariable("id") int id)
    {
        movieService.delete(id);
    }
    //creating post mapping that post the movie detail in the database
    //creating put mapping that updates the movie detail
}
```

3.2 Theater Management

FR2.1: Add Theater

- **Description**: Admin users can add new theaters to the system.
- **Input**: Theater details (name, location, number of screens).
- **Output**: Confirmation of theater creation.

FR2.2: Allocate Seats

- **Description**: Admin users can allocate seats for shows in a theater.
- **Input**: Theater ID and seat configuration (number of seats, layout).
- **Output**: Confirmation of seat allocation.

FR2.3: Edit Theater

- **Description**: Admin users can edit details of existing theaters.
- **Input**: Theater ID and updated details.
- **Output**: Confirmation of theater update.

FR2.4: Remove Theater

- **Description**: Admin users can remove theaters from the system.
- **Input**: Theater ID.
- **Output**: Confirmation of theater deletion.

FR2.5: View Theaters

- **Description**: Users can view a list of available theaters.
- **Output**: List of theaters with details (name, location, screens).

Code :: TheaterController.java

```
package com.bmc.controller;

import com.bmc.mapper.request.TheaterEntryDto;
import com.bmc.mapper.request.TheaterSeatsEntryDto;
import com.bmc.service.TheaterService;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/theater")
public class TheaterController {

    @Autowired
    TheaterService theaterService;

    @PostMapping("/add")
    public ResponseEntity<String> addTheater(@RequestBody TheaterEntryDto
theaterEntryDto){
    try {
        String result= theaterService.addTheater(theaterEntryDto);
        return new ResponseEntity<>(result, HttpStatus.CREATED);
    }catch (Exception e) {
        return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
      }
    }

    @PostMapping("/addTheaterSeats")
    public  ResponseEntity<String> addTheaterSeats(@RequestBody
TheaterSeatsEntryDto entryDto){

        try{
            String result=theaterService.addTheaterSeats(entryDto);
            return new ResponseEntity<>(result,HttpStatus.CREATED);
        }catch (Exception e){
             return new
ResponseEntity<>(e.getMessage(),HttpStatus.BAD_REQUEST);
        }
    }}
```

3.3 User Registration

FR3.1: User Registration

- **Description**: Users can create an account in the system.
- **Input**: User details (name, email, password, phone number).
- **Output**: Confirmation of account creation and a verification email.

FR3.2: User Login

- **Description**: Users can log in to their accounts.
- **Input**: Email and password.
- **Output**: Confirmation of successful login or error message.

FR3.3: Manage Profile

- **Description**: Users can update their profile information.
- **Input**: Updated user details.
- **Output**: Confirmation of profile update.

Codebase: AuthController.java

```
package com.bmc.controller;

import java.util.List;
import java.util.stream.Collectors;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.validation.ObjectError;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

import com.cts.userauthservice.dto.ForgotPasswordRequest;
import com.cts.userauthservice.dto.LoginRequest;
import com.cts.userauthservice.dto.LoginResponse;
import com.cts.userauthservice.dto.RegistrationRequest;
import com.cts.userauthservice.dto.Response;
import com.cts.userauthservice.dto.UpdatePasswordRequest;
import com.cts.userauthservice.dto.ValidationDto;
import com.cts.userauthservice.service.AuthService;

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.Parameter;
import io.swagger.v3.oas.annotations.media.Content;
import io.swagger.v3.oas.annotations.media.Schema;
import io.swagger.v3.oas.annotations.responses.ApiResponses;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import io.swagger.v3.oas.annotations.tags.Tag;

@Tag(name = "Authentication", description = "the user auth API")
@RestController
@CrossOrigin
public class AuthController {
    @Autowired
    private AuthService authService;

    @Operation(summary = "Register new user", description = "Create a
new user account with provided credentials.")
    @ApiResponses(value = {
          @ApiResponse(responseCode = "201", description = "Successful
registration", content = @Content(schema = @Schema(implementation =
Response.class))),
          @ApiResponse(responseCode = "409", description = "Email
already exists", content = @Content(schema = @Schema(implementation =
Response.class))) })
    @PostMapping("/register")
```

```
    public ResponseEntity<Response> register(
            @Parameter(description = "Registration request containing
firstname, lastname, email, password and secret-question", required =
true) @RequestBody @Valid RegistrationRequest registrationRequest,
            BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return errorResponse(bindingResult.getAllErrors());
        }
        authService.register(registrationRequest);
        return new ResponseEntity<Response>(
                Response.builder().status("success").message("User
resgistration successful").build(),
                HttpStatus.CREATED);
    }

    @Operation(summary = "Login user", description = "Authenticate user
and generate a JWT token for accessing protected resources.")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Successful
login", content = @Content(schema = @Schema(implementation =
LoginResponse.class))),
            @ApiResponse(responseCode = "400", description = "Bad
credentials", content = @Content(schema = @Schema(implementation =
Response.class))) })
    @PostMapping("/login")
    public ResponseEntity<?> login(
            @Parameter(description = "Login request containing email and
password", required = true) @RequestBody @Valid LoginRequest
loginRequest,
            BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return errorResponse(bindingResult.getAllErrors());
        }
        LoginResponse response = authService.login(loginRequest);
        return new ResponseEntity<LoginResponse>(response,
HttpStatus.OK);
    }

    @Operation(summary = "Forgot Password", description = "Reset user
password after checking the answer to the security question.")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Password
reset successful", content = @Content(schema = @Schema(implementation =
Response.class))),
            @ApiResponse(responseCode = "400", description = "Bad
request", content = @Content(schema = @Schema(implementation =
Response.class))) })
    @PutMapping("/{userid}/forgot")
    public ResponseEntity<Response> forgotPassword(
            @Parameter(description = "Id of the user", required = true)
@PathVariable String userid,
            @Parameter(description = "Forgot password request containing
new password, security question and answer") @RequestBody @Valid
ForgotPasswordRequest passwordChangeRequest,
            BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return errorResponse(bindingResult.getAllErrors());
```

```
        }
        authService.forgotPassword(userid, passwordChangeRequest);
        return new ResponseEntity<Response>(
                Response.builder().status("success").message("Your
password recovered successfully").build(),
                HttpStatus.OK);
    }

    @Operation(summary = "Update Password", description = "Update the
current password of the logged in user.")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Password
update successful", content = @Content(schema = @Schema(implementation
= Response.class))),
            @ApiResponse(responseCode = "400", description = "Bad
request", content = @Content(schema = @Schema(implementation =
Response.class))) })
    @PutMapping("/updatepassword")
    public ResponseEntity<Response> updatePassword(
            @Parameter(description = "JWT token to authenticate the
requester", required = true) @RequestHeader(name = "Authorization")
String jwtToken,
            @Parameter(description = "Update password request containing
new password, security question and answer") @RequestBody @Valid
UpdatePasswordRequest passwordChangeRequest,
            BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return errorResponse(bindingResult.getAllErrors());
        }
        authService.updatePassword(jwtToken, passwordChangeRequest);
        return new ResponseEntity<Response>(
                Response.builder().status("success").message("Your
password updated successfully").build(),
                HttpStatus.OK);
    }

    @Operation(summary = "Validate JWT Token", description = "Validate
the provided JWT token and check if it is still valid or expired.")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Valid JWT
token", content = @Content(schema = @Schema(implementation =
ValidationDto.class))),
            @ApiResponse(responseCode = "401", description = "Invalid
token", content = @Content(schema = @Schema(implementation =
ValidationDto.class))),
            @ApiResponse(responseCode = "400", description = "Bad
request", content = @Content(schema = @Schema(implementation =
Response.class))) })
    @GetMapping("/validate")
    public ResponseEntity<ValidationDto> validateAuthToken(
            @Parameter(description = "JWT token to validate", required =
true) @RequestHeader(name = "Authorization") String jwtToken) {
        ValidationDto response =
authService.validateAuthToken(jwtToken);
        if (response.isStatus())
            return
ResponseEntity.ok(authService.validateAuthToken(jwtToken));
```

```
        else
            return new ResponseEntity<>(response,
HttpStatus.UNAUTHORIZED);
    }
    private ResponseEntity<Response> errorResponse(List<ObjectError>
errors) {
        String errorMsg =
errors.stream().map(ObjectError::getDefaultMessage).collect(Collectors.
joining(". "));
        return new
ResponseEntity<>(Response.builder().status("error").message(errorMsg).b
uild(),
                HttpStatus.BAD_REQUEST);
    }
}
```

3.4 Ticket Booking

FR4.1: Browse Movies

- **Description**: Users can browse available movies.
- **Output**: List of movies with show times.

FR4.2: Select Event

- **Description**: Users can select a desired movie event to book tickets.
- **Input**: Movie ID and showtime.
- **Output**: Show details and available seats.

FR4.3: Book Tickets

- **Description**: Users can book tickets for the selected event.
- **Input**: User ID, show ID, number of tickets.
- **Output**: Confirmation of ticket booking and booking details.

Codebase : TicketController.java

```
package com.bmc.controller;
import com.bmc.mapper.request.TicketRequestDto;
import com.bmc.mapper.response.TicketResponseDto;
import com.bmc.model.Ticket;
import com.bmc.service.TicketService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
public class TicketController {
    @Autowired
    private TicketService ticketService;
    @PostMapping("/ticket")
    public ResponseEntity<TicketResponseDto> bookTicket(@RequestBody
TicketRequestDto ticketRequestDto){
        try{
            TicketResponseDto response =
ticketService.bookTicket(ticketRequestDto);
            response.setResponseMessage("Ticket booked successfully");
            return new ResponseEntity<>(response, HttpStatus.OK);
        }catch (Exception e){
```

```
            TicketResponseDto ticketResponseDto = new TicketResponseDto();
            ticketResponseDto.setResponseMessage(e.getMessage());
            return new
ResponseEntity<>(ticketResponseDto,HttpStatus.BAD_REQUEST);
        }
    }
    @GetMapping("/ticket")
    public List<Ticket> getAllBooks()
    {
        return ticketService.getAllTickets();
    }
    //creating a get mapping that retrieves the detail of a specific ticket
    @GetMapping("/ticket/{id}")
    public Ticket getTickets(@PathVariable("id") int id)
    {
        return ticketService.getTicketById(id);
    }
    //creating a delete mapping that deletes a specified ticket
    @DeleteMapping("/ticket/{id}")
    public void deleteBook(@PathVariable("id") int id)
    {
        ticketService.delete(id);
    }
    //creating put mapping that updates the ticket detail
    @PutMapping("/ticket")
    public Ticket update(@RequestBody Ticket ticket)
    {
        ticketService.update(ticket);
        return ticket;
    }
}
```

4. Non-Functional Requirement

4.1 Transactional Scenarios and Design Decisions

Transactional Scenarios

- **Ticket Booking**: We are ensuring atomicity during ticket booking, where the process includes seat selection, payment processing, and confirmation. If any part fails, roll back all changes.
- **Movie Management**: Implement transactions for adding, editing, or deleting movie entries, ensuring consistency across all services.

Design Decisions

- **Saga Pattern**: Use the Saga pattern for managing distributed transactions, coordinating multiple microservices (e.g., Ticket Service, Payment Service).
- **Event Sourcing**: Maintain a log of changes in a dedicated event store to recover from failures and provide an audit trail.

4.2 . Integration with Theaters

Existing IT Systems

- **API-First Approach**: Develop RESTful APIs to facilitate integration with existing theater management systems.
- **Data Mapping**: Implement data transformation processes to map data formats between the *Book My Cinema* platform and the existing IT systems.

New Theaters and Localization

- **Localization Services**: Use a localization service to manage movies and shows in various languages, adapting content to meet regional preferences.
- **Dynamic Configuration**: Implement dynamic configurations for different theaters to adjust business logic and offerings based on locality.

4.3 Scalability and Availability

Scalability to Multiple Cities and Countries

- Design the system using microservices, allowing independent scaling of components based on demand.
- Use databases with global replication capabilities to serve data close to users in different geographical locations.

Guarantee Platform Availability (99.99%)

- **Load Balancing**: Implement load balancers to distribute traffic evenly across instances.
- **Auto-Scaling**: Utilize auto-scaling groups in cloud environments to automatically adjust resources based on traffic.
- **Multi-Region Deployment**: Deploy the application across multiple regions with failover strategies to ensure continuity in case of regional outages.
- **Monitoring and Alerts**: Implement comprehensive monitoring solutions to detect issues in real-time and trigger automated recovery actions.

4.4 Integration with Payment Gateways

Payment Gateway Integration

- **Payment APIs**: Integrate with multiple payment gateways(Tabapay, PayPal) to provide flexibility and redundancy.
- **Transaction Handling**: Implement callback/webhook mechanisms for asynchronous transaction handling and payment confirmation.

4.5 Monetization of the Platform

- **Ticket Sales**: Generate revenue from ticket sales, with dynamic pricing based on demand.
- **Advertising**: Integrate targeted advertising in the application for additional revenue streams.

- **Partnerships**: Collaborate with theaters for revenue-sharing models on ticket sales and promotional events.

4.6 Protection Against OWASP Top 10 Threats

- **Regular Security Audits**: Conduct frequent audits and vulnerability assessments to identify and remediate potential threats.
- **Secure Coding Practices**: Implement secure coding standards and conduct code reviews to minimize the risk of vulnerabilities.
- **Web Application Firewalls (WAF)**: Deploy WAFs to filter and monitor HTTP traffic to protect against attacks like SQL injection and XSS.

4.7 Incident Response Plan: Develop a robust incident response plan to quickly address any security breaches or vulnerabilities with Synk tools.

4.8 Compliance

- **GDPR Compliance**: Ensure user data handling complies with GDPR regulations, including user consent, data access, and the right to be forgotten.
- **PCI DSS Compliance**: Adhere to PCI DSS standards for handling credit card information and secure payment processing.
- **Local Regulations**: Stay informed about and comply with local regulations concerning data protection and consumer rights in each operational region.

5. Dimensioning and Calculations

5.1 Assumptions

- User Growth Rate: 5,000 new users registered per month.
- Movie Releases: 30 new movies added each month.
- Showtimes: Each movie has an average of 5 show times daily across 10 theatres.
- Ticket Sales: Average of 150 tickets sold per show with a 75% occupancy rate.
- Data Retention Period: Data is retained for 5 years.
- Average Size of Data:
    - User: 2 KB
    - Movie: 10 KB
    - Show: 2 KB
    - Ticket: 2 KB

5.2. Data Storage Calculation

User Data

- **Monthly Growth:** 5,000 users
- **Annual Growth:** 5,000 users×12 months=60,000
- **Storage Requirement:** Total User Storage=60,000 users×2 KB=120,000 KB=120 MB

Movie Data

- **Monthly Additions:** 30 movies
- **Annual Additions:** 30 movies×12 months=360 movies
- **Storage:** Total Movie Storage=360 movies×10 KB=3,600 KB=3.6 MB

Show Data

- **Total Shows per Month:** 30 movies×5 shows/movie=150 shows
- **Annual Storage Requirement** Total Show Storage=150 shows/month×12 months×2 KB=3,600 KB =3.6 MB

Ticket Data

- **Total Tickets per Month:** 150 shows×150 tickets/show=22,500 tickets
- **Annual Storage Requirement:**
  Total Ticket Storage=22,500 tickets/month×12 months×2 KB=540,000 KB=540 MB

5.3 Traffic and Latency Considerations

Assumptions

- **Peak Traffic:** 1,000 concurrent users during peak times.
- **Average Requests per User:** 3 requests/session.
- **Average Session Duration:** 10 minutes.

Traffic Calculation

1. **Total Requests per Minute:**
- **Users per minute:**1,000 users×3 requests=3,000 requests
- **Requests per minute (for 10-minute session):**Requests per minute=3,000 requests10 minutes=300 requests/minute
2. **Data Transfer Rate:**
- **If each request returns an average of 2 KB:**Total Data per Minute=300 requests/minute×2 KB=600 KB/minute

Latency Considerations

1. **API Response Time Target:** Aim for under 200 ms for API calls.
2. **Database Query Response Time:** Aim for under 100 ms for read operations.

Total Expected Latency

1. **Network Latency:** Assume a round-trip time of 50 ms.
2. **Processing Time:** Assume processing takes 100 ms.
3. **Total Latency Calculation:**

Total Latency=Network Latency+Processing Time=50 ms+100 ms=150 ms\text