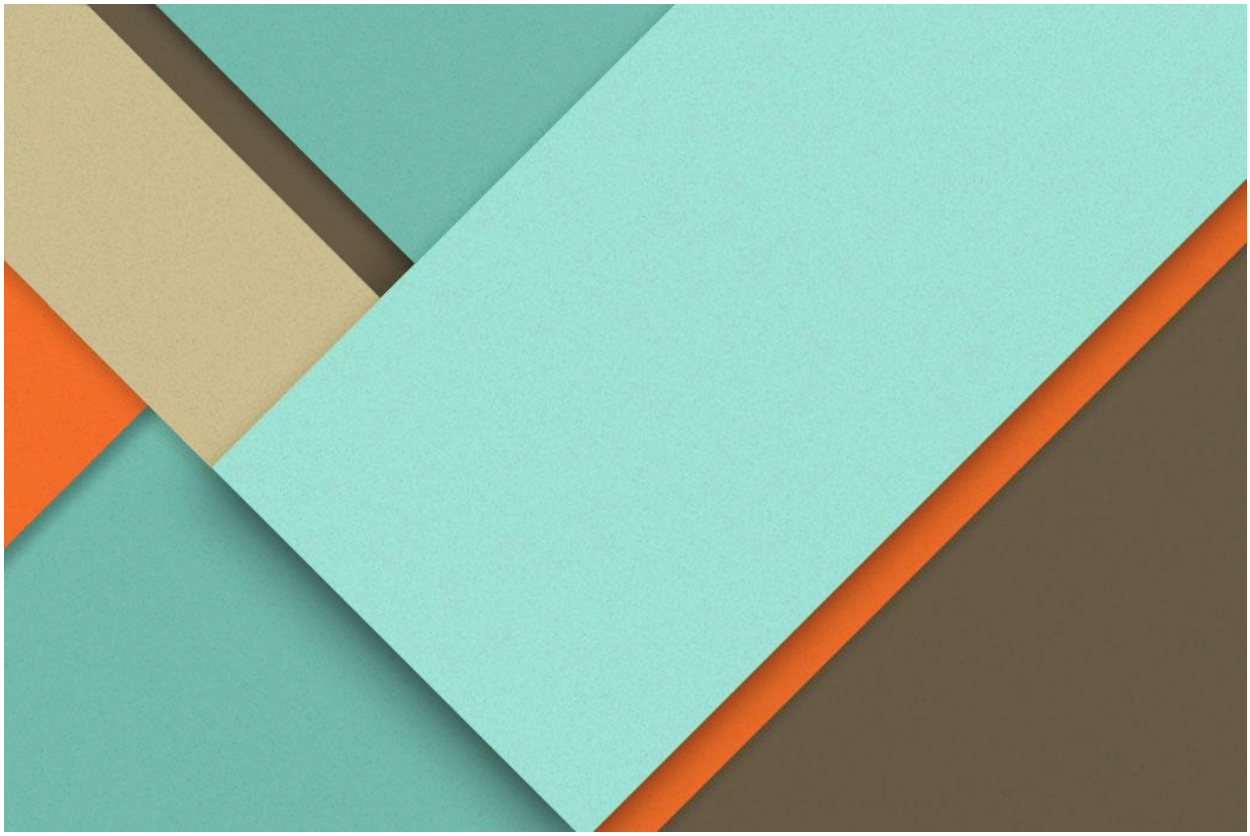


Online Movie Booking System

18.02.2025

Vivek Bhatnagar

Architectural Style

- Microservices Architecture:
    - Each functionality (e.g., user management, booking, payment, inventory(seat) management) is a separate service.
    - This allows scalability and flexibility while putting enhancement/bug fixes live.
    - Enabling independent deployment and testing where each microservice can be deployed as needed individually rather than making the whole system at risk with specific changes.
- Domain-Driven Design (DDD):
    - Focus on modeling the domain (theatres, movies, users, bookings, shows, seats) in alignment with the business requirements.

2. Key Modules and Services

Core Services

a. User Management Service:

   The user management service takes care of registration, login, and role management like Admin, Theatre Partner, and End Customer.

   It supports OAuth 2.0 for authentication and authorization.

b. Movie Management Service:

   Enables Theatres to upload movie schedules, manage shows, and update availability.

   Provides APIs for movie metadata such as title, duration, and genre.

c. Booking Service:

   Enables customers to check seat availability, make bookings, and send confirmations.

   Uses a strategy to avoid double bookings through locking mechanisms.

d. Payment Service:

   Works with payment gateways like Stripe and PayPal.

   Processes refunds, payment confirmation, and handles errors.

e. Notification Service:

   Delivers emails, SMS, or push notifications pertaining to booking confirmation, reminders, or cancellations.

f. Analytics Service:

   Gather booking data, theatre analytics, and customer data to understand preferences and trends.

Auxiliary Services

a. Theatre Partner Portal:

   Dashboard for theatre partners for schedule, seating, price management on the Web.

b. Customer Portal:

   Seating arrangements and payments can be made via the web or mobile application.

c. Admin Panel:

   Admins can control business activities, conflict resolution, and user or configuration management.

Technology Stack

Backend (Java)

Framework: Spring Boot (Microservices and REST APIs)

Persistence: Hibernate with JPA for ORM

Database:

Relational DB (e.g., MySQL or PostgreSQL) for transactional data.

NoSQL DB (e.g., DynamoDB or Redis) for caching or catalog data.

Message Queue: RabbitMQ or Apache Kafka for asynchronous communication (e.g., notifications, event processing).

API Gateway: AWS API Gateway

Authentication: Spring Security with OAuth2/JWT.

Frontend

Web (B2C and B2B portals): React.js or Angular.

Cloud and DevOps

Cloud: AWS.

Use S3 for static files (e.g., movie posters, other images for the project).

RDS for database.

Lambda for serverless event processing.

Containerization: Docker and Kubernetes for deployment.

CI/CD: Jenkins, GitHub

Payment Integration

Integrate with payment gateways using SDKs/APIs provided by e.g., Stripe, PayPal, or Razorpay.

4. Key Design Principles

Separation of Concerns:

Each microservice focuses on a single responsibility.

Scalability:

Use horizontal scaling for services like booking and payments.

Security:

Encrypt sensitive data (e.g., passwords, payment info) using HTTPS and encryption libraries, checksums etc.

High Availability:

Use load balancers and distributed databases.

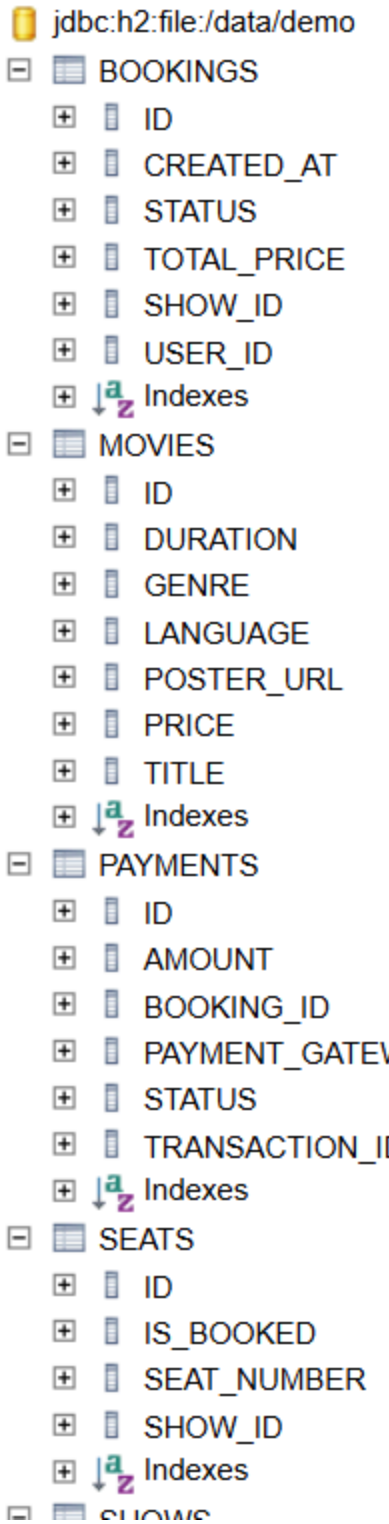
Resilience:

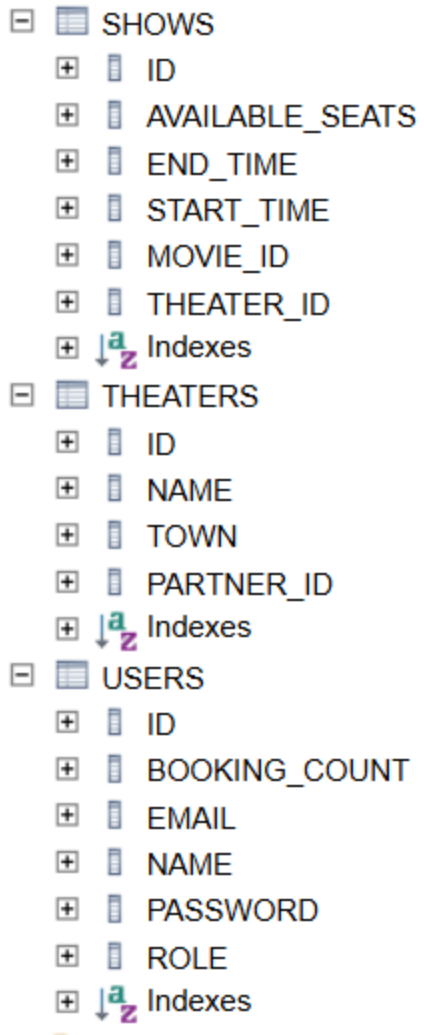
Implement Circuit Breakers (using tools like Resilience4j) to handle failures gracefully.

Versioning:

Version APIs to support backward compatibility.

5. Database Design


jdbc:h2:file:/data/demo
BOOKINGS
ID
CREATED_AT
STATUS
TOTAL_PRICE
SHOW_ID
USER_ID
Indexes
MOVIES
ID
DURATION
GENRE
LANGUAGE
POSTER_URL
PRICE
TITLE
Indexes
PAYMENTS
ID
AMOUNT
BOOKING_ID
PAYMENT_GATEWAY
STATUS
TRANSACTION_ID
Indexes
SEATS
ID
IS_BOOKED
SEAT_NUMBER
SHOW_ID
Indexes


SHOWS
ID
AVAILABLE_SEATS
END_TIME
START_TIME
MOVIE_ID
THEATER_ID
Indexes
THEATERS
ID
NAME
TOWN
PARTNER_ID
Indexes
USERS
ID
BOOKING_COUNT
EMAIL
NAME
PASSWORD
ROLE
Indexes

6. Deployment Strategy

Staging Environment: Test APIs, UI, and integrations here before production.

Blue-Green Deployment: Roll out updates without downtime.

Monitoring and Logging: Use tools like AWS Cloudwatch, AWS Synthetics for monitoring.

7. Example Workflow

Customer Journey:

Browse movies → Select theatre & time → Pick seats → Make payment → Receive confirmation.

Theatre Partner:

Upload schedule → Update seat layout → Monitor sales through the dashboard.

Non Functional Requirements

Performance , scaling, availability, Security, Maintainability, Reliability, Observability and Monitoring, Disaster Recovery, Compatibility, Cost Optimization, Localization, Data Retention etc are the aspects that we need to keep in mind in this design.

The system should handle at least X simultaneous user requests for seat availability and bookings without noticeable delays.

Response time for ticket booking or availability checking should not exceed 1 second under normal load conditions.

The system should scale to support peak loads during high-demand events (e.g., blockbuster movie releases).

We can use **Use Microservices**, Asynchronous Processing, Load Balancing, Caching, Read and Write Separation if needed more efficiency in the system, Cloud-based Autoscaling, Database Query Optimization, Thread Management, Optimize thread usage by using thread pools (e.g., in ExecutorService) and avoid creating too many threads for concurrent tasks.Serve static assets (images, CSS, JavaScript) via CDNs like Cloudflare or AWS CloudFront to reduce load on your servers.Use tools like API Gateway (AWS, Kong, Apigee) to throttle requests per user or IP.Set up alerts for critical metrics like CPU usage, memory consumption, and request latency.Use tools like Apache JMeter, Gatling, or k6 to simulate 10,000+ concurrent users and identify bottlenecks. Use Docker to deploy isolated, lightweight containers for services.Orchestrate containers with Kubernetes to manage scaling and ensure high availability.

Frontend: Static content served via CDN.

API Gateway: Routes traffic to microservices.

Microservices: Stateless services handling user, booking, payment, etc.

Cache Layer: Redis for frequently accessed data.

Database: PostgreSQL with read replicas, partitioning, and optimized queries.

Load Balancer: NGINX or cloud-based load balancer to distribute requests.

Autoscaling: Kubernetes with autoscaling policies.

Monitoring: Prometheus and Grafana to ensure performance meets SLAs.

All user data, including payment information, should be encrypted both in transit and at rest using encryption mechanism or checksum etc.

Using OAuth2 or JWT Tokens for secure user authentication and authorization.

Implement rate-limiting and captcha for critical APIs to prevent abuse and denial-of-service attacks.

Ensure PCI DSS compliance for secure payment processing.

The codebase should follow clean coding practices and be modular to facilitate easy updates and feature additions.

Automated tests (unit, integration, and end-to-end) should cover at least 80% of the code.

Microservices should log detailed information for debugging and monitoring purposes.

Data consistency should be ensured, particularly during concurrent booking scenarios to prevent overbooking.

Implement retries and fallback mechanisms for failed payment transactions.

Use monitoring tools like ELK Stack to track system performance and detect anomalies.

Alerts should be configured for critical system metrics, such as response times, error rates, and service downtimes etc.

Regular backups should be performed to ensure data can be restored in case of failures.

Implement a disaster recovery plan to restore operations within 15 minutes in case of a major failure.

Compatibility

The system should be compatible across web browsers (Chrome, Firefox, Safari, etc.) and mobile platforms (Android, iOS).

Payment gateways should support multiple methods, including credit/debit cards, UPI, and wallets.

Optimize infrastructure usage (e.g., via container orchestration with Kubernetes) to minimize costs during off-peak hours.

Use serverless services (like AWS Lambda) for non-critical background tasks.

Data Retention and Archiving

Support multiple languages and currencies to cater to a diverse user base across different regions.

—----------

Describe transactional scenarios and design decisions to address the same.

Payment failures due to (external) third party integration - Need to track payment transaction using different transaction statuses like Initiated, In-Progress, Done, Failed etc.

Retry mechanism can be added

o Integrate with theatres having existing IT system and new theatres and localization(movies)

Canonical objects mapping based approach can help here by which system will be able to integrate with external schemas with minimum changes at the service end.

How will you scale to multiple cities, countries and guarantee platform availability of 99.99%? -

Multi-Region Deployment - Deploying the application in multiple regions using cloud AWS.

Deploying the platform in active-active mode across regions using load balancers and traffic routing. Ensuring that traffic is routed to the nearest region using AWS Route 53.

Partition data by region to reduce latency for local queries (e.g., US users interact with the US database).

Using Content Delivery Network (CDN) such as AWS CloudFront to cache and serve static assets (e.g., images, posters, CSS, JS) close to users worldwide.

This reduces load on the application and improves speed.

Redundancy

- Infrastructure Redundancy:
  - Use multiple availability zones (AZs) within each region.
  - Deploy redundant instances of services in each AZ.
- Data Redundancy:
  - Use multi-region replication for databases (e.g., Aurora, DynamoDB).
  - Store backups in multiple regions with frequent snapshots.

o Integration with payment gateways

We can choose based on our need from providers like Stripe, RazorPay, PayPal, paytm etc.

o How do you monetize platform?

There are various ways by which we can monetize this system like displaying advertisements, charging for premium features with a subscription model, selling virtual goods or services, offering affiliate marketing partnerships, charging for direct access to content

o How to protect against OWASP top 10 threats.

We can follow the standard processes and designs while making applications so that we adhere to them. Noted following we can discuss them -

Broken Access Control, Cryptographic Failures, Injection, Insecure Design, Security Misconfiguration, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, Security Logging and Monitoring Failures, Server-Side Request Forgery

Discuss hosting solution and sizing (Cloud / Hybrid/ Multi cloud)- Any

Choosing any of the hosting methods depends on the system requirements and includes many factors to be considered before deciding on any solution. Following are some points which help to decide the needed solution

Cloud Hosting:

Pros:

High scalability, Cost-effective-Pay only for what you use. Flexibility: Access applications and data from anywhere. Disaster recovery: Redundancy across multiple data centers.

Cons:

Potential security concerns: Data is stored on third-party servers.

Vendor lock-in: Can be difficult to switch providers.

Hybrid Cloud Hosting:

Pros:

Optimal balance: Store sensitive data on-premises while using public cloud for scalable needs.

Cost optimization: Leverage the most cost-effective cloud services for different workloads.

Data control: Maintain control over critical data while benefiting from cloud flexibility.

Cons:

Complexity: Requires careful management of both public and private cloud environments.

Integration challenges: Ensuring seamless data transfer between clouds.

Multi-Cloud Hosting:

Pros:

Best of breed: Choose the best features from different cloud providers based on specific needs.

Vendor independence: Avoid reliance on a single cloud provider.

Competitive pricing: Leverage pricing options across different clouds.

Cons:

Management complexity: Requires expertise to manage multiple cloud environments.

Potential for increased costs: Improper management can lead to higher cloud bills.

Sizing your cloud solution:

Traffic patterns:

Analyze peak usage times and expected traffic growth to determine required server capacity.

Data storage needs:

Consider the amount of data you need to store and its expected growth rate.

Application requirements:

Assess the processing power and memory needed for your applications.

Cost considerations:

Choose a pricing model that aligns with your budget and usage patterns (pay-as-you-go, reserved instances).

o Discuss release management across cities, languages etc

We can develop the system in a modular architecture where city/region-specific features are isolated into separate modules or services.

For example:

Pricing logic for different countries.

Localization for cities and languages.

Payment methods specific to regions (e.g., PayPal, Alipay, UPI).

We can also use feature toggles to enable or disable features for specific

cities or countries.like Unleash, or custom feature toggle systems.

Multi-Language and Multi-Currency Support

Translation Management

We can use translation management tools like Lokalise, Phrase, or Transifex.

Ensure all static and dynamic content is localized into supported languages.

For dynamic content (e.g., movie titles), store translations in a database.

Maintain fallback languages for cities or countries where a full translation is unavailable.

4.2 Currency Handling

Integrate currency conversion APIs (e.g., OpenExchangeRates, XE) to handle real-time currency updates.

Validate pricing models for currency-specific nuances (e.g., rounding rules).

4.3 Region-Specific Releases

Bundle city/country/language-specific changes in separate branches for testing and rollout.

CI/CD Pipeline for Regional Releases

Build a multi-branch CI/CD pipeline:

Mainline branch for core features.

Regional branches for city-specific configurations.

Use automated deployment tools like Jenkins, GitHub Actions, or GitLab CI/CD to:

Build, test, and deploy changes for specific regions.

Ensure that regional changes don't impact global stability.

Blue-Green or Canary Deployment

Blue-Green Deployment:

Maintain two environments (Blue: current version, Green: new version).

Redirect traffic to the new environment (Green) for a specific city or region.

Canary Deployment:

Deploy new changes to a small percentage of users in selected cities.

Gradually expand as confidence grows.

Regional Testing and Validation

6.1 Automated Testing

Use test automation frameworks like Selenium or Cypress to simulate user behavior for different:

Cities (e.g., bookings in New York vs. Tokyo).

Languages (e.g., RTL languages like Arabic).

Devices (e.g., mobile vs. desktop).

6.2 Localization Testing

Validate translations, date/time formats, and number formats.

Test for language-specific layout issues (e.g., long strings breaking UI).

6.3 Load Testing

Perform load testing for region-specific traffic spikes (e.g., festival movie releases in specific countries).

Use tools like JMeter or Locust to simulate city-wide or country-wide traffic.

6.4 A/B Testing

Test different features or layouts in different regions to determine what works best.

o Provide details on monitoring solution and Discuss overall KPIs

Latency, error rates, and inter-service communication times

Response times, error rates (4xx, 5xx status codes), and throughput (requests per second).

Bookings: Number of successful bookings per minute/hour.

Revenue: Payments processed, failed payments, and revenue trends.

User Activity: Active users, session lengths, and abandoned transactions.

Search Performance: Average search times and results returned.

Payment Fraud: Track anomalies in payment data.

Data Security: Ensure encryption is working and sensitive data is not exposed.

Break down metrics per region or city

o Create a high-level project plan and estimates breakup.

Effort Estimation (Team Size: 5 Developers)			
Phase	**Duration**	**Effort (Person-Weeks)**	**Notes**
Planning and Analysis	**2 weeks**	**10**	**Includes requirement gathering, scoping, and technical discussions.**
Architecture Design	**3 weeks**	**15**	**Covers system design, database schema, and API contracts.**
User Management Service	**2 weeks**	**10**	**User registration, login, authentication (JWT), and role-based access.**
Theater Onboarding	**2 weeks**	**10**	**CRUD operations for theater details, partner onboarding process.**
Movie & Show Management	**3 weeks**	**15**	**Movie listings, scheduling shows, seat mapping, and regional preferences.**
Booking Service	**2 weeks**	**10**	**Booking tickets, availability checks, cancellation policies.**
Payment Integration	**3 weeks**	**15**	**Payment gateway integration, refunds, and transaction tracking.**
Testing (All Services)	**4 weeks**	**20**	**Unit, integration, performance, and security testing.**
Deployment	**2 weeks**	**10**	**Cloud deployment, CI/CD setup,**

			and go-live support.
Post-Deployment	**Ongoing**	**1-2 weeks per iteration**	**Bug fixes, new features, and performance optimization.**