Key Goals

1. Enable Theatre Partners:

* Allow theatre partners to onboard their theatres over this platform and access a larger customer base while going digital.

1. Enable End Customers:

* Provide end customers with the ability to browse movies across different cities, languages, and genres.
* Allow customers to book tickets in advance seamlessly.

Functional Requirements

Theatre Partners(B2B)

1. Onboard a new theatre.

* Add/Update/Delete details of the number of halls in the theatre.
* Add/Update/Delete seating arrangements for different halls.
* Add/Update/Delete show details.

1. Integration with existing theatre IT systems to get updates on shows/movies/bookings.

B2C (End Customers)

1. Display a list of cities with registered theatres.
2. Allow browsing of movies currently running in selected cities.
3. Provide options to select movie and languages.
4. Display theatres running the selected movie along with show times.
5. Enable seat selection and booking, with a maximum of 10 tickets at a time.
6. Display seat layout with availability status.
7. Block seats for 10 minutes before payment.
8. Facilitate payment and booking finalization.
9. Send booking success notifications via SMS and Email.
10. Show booking details and history on the platform.

Non-Functional Requirements (NFR)

* Low Latency: Ensure browsing and booking features are seamless without delays.
* High Availability: Guarantee system availability for a consistent customer experience.
* High Concurrency: Handle multiple users booking seats simultaneously gracefully.
* Fault Tolerance: Maintain operations despite failures.
* ACID Compliance: Ensure booking and payment transactions adhere to ACID properties.
* Scalability: Scale the system during high demand and popular releases.

Out of Scope

* Detailed flow for cache and Elasticsearch integration.
* Systems Analytics with AI/ML.

Assumptions

* 50 million visits per month.
* 10 million tickets sold per month.
* Number of cities: 700.
* Number of theatres per city: 5.
* Number of seats per theatre: 700.
* Number of shows per day: 4.

Estimations

1. Infrastructure Estimation for Production System

Ensure that each microservice has a minimum of 1 core and 2 GB of memory for efficient performance.

Use horizontal pod autoscaling to scale the number of pods based on CPU and memory utilization.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Service | Number of Pods | Min CPU Cores | Max CPU Cores | Min Memory (GB) | Max Memory (GB) |
| Theatre Service | 3 | 1 | 5 | 2 | 8 |
| Movie Service | 3 | 1 | 5 | 2 | 8 |
| Show Service | 3 | 1 | 5 | 2 | 8 |
| User Service | 3 | 1 | 5 | 2 | 8 |
| Booking Service | 3 | 1 | 5 | 2 | 8 |
| Payment Service | 3 | 1 | 5 | 2 | 8 |
| Notification Service | 3 | 1 | 5 | 2 | 8 |
| Admin Service | 3 | 1 | 5 | 2 | 8 |
| Redis Cache | 1 | 1 | 1 | 2 | 2 |
| Elasticsearch | 3 | 1 | 5 | 2 | 8 |
| Logstash | 1 | 1 | 1 | 2 | 2 |
| Prometheus | 1 | 1 | 1 | 2 | 2 |
| Grafana | 1 | 1 | 1 | 2 | 2 |

AWS EKS Cluster:

* Nodes: 10 (assuming each node has 4 cores and 16 GB memory)
* Total CPU Cores: 40
* Total Memory: 160 GB

Database Size and Plan on AWS RDS PostgreSQL:

* Bookings per month: 10 million
* Estimated size per booking record: 0.5 KB
* Monthly data: 10 million \* 0.5 KB = 5 GB
* Annual data: 5 GB \* 12 = 60 GB
* 5 years data: 60 GB \* 5 = 300 GB

Plan:

* Instance Type: db.m5. large (2 vCPUs, 8 GB RAM)
* Storage: 300 GB Provisioned IOPS (SSD)
* Backup Retention Period: 7 days
* Multi-AZ Deployment: Enabled

1. Work Planning Estimation

The development effort is split across multiple developers working on different microservices in parallel.

Testing includes both manual and automated testing to ensure high-quality deliverables.

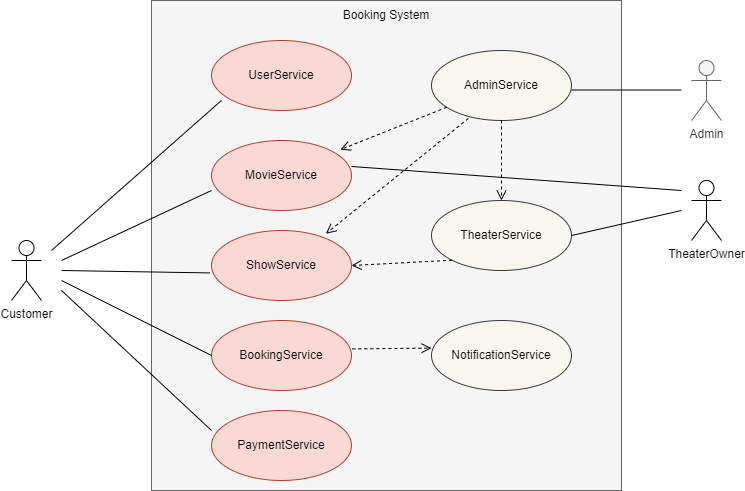
CI/CD pipelines are set up for automated build, test, and deployment processes.

|  |  |  |
| --- | --- | --- |
| **Task** | **Estimated Hours** | **Assigned To** |
| Project Planning | 40 | Project Manager |
| Requirement Gathering & Analysis | 60 | Business Analyst |
| Architecture Design | 80 | Solution Architect |
| Database Design | 40 | DBA |
| Microservices Development | 400 | Developers |
| - Theatre Service | 50 | Developer |
| - Movie Service | 50 | Developer |
| - Show Service | 50 | Developer |
| - User Service | 50 | Developer |
| - Booking Service | 50 | Developer |
| - Payment Service | 50 | Developer |
| - Notification Service | 50 | Developer |
| - Admin Service | 50 | Developer |
| API Development | 120 | Developers |
| Frontend Development (Web, Mobile) | 200 | Developers |
| Integration with Redis | 40 | Developer |
| Integration with Elasticsearch | 40 | Developer |
| Integration with Payment Gateway | 40 | Developer |
| Security Implementation | 60 | Security Engineer |
| CI/CD Pipeline Setup | 60 | DevOps Engineer |
| Testing (Unit, Integration, System) | 160 | QA Team |
| Performance Testing | 80 | QA Team |
| User Acceptance Testing | 80 | QA Team |
| Deployment & Go-Live | 40 | DevOps Engineer |
| Documentation | 40 | Technical Writer |
| Training & Knowledge Transfer | 40 | Project Manager |
| Project Management | 80 | Project Manager |

Total Estimated Hours: 1700 hours

Use Case

Following is ticket booking system’s use case diagram for three actors Customer, Theatre owner and Admin of the booking system.



* Platform Admin

Add/Modify/Delete shows and movies.

Search for movies based on various criteria.

* Customer

Search for the latest release movies.

Search theatres and shows running in a city.

Filter shows based on language and format.

Create/Modify/View/Cancel bookings.

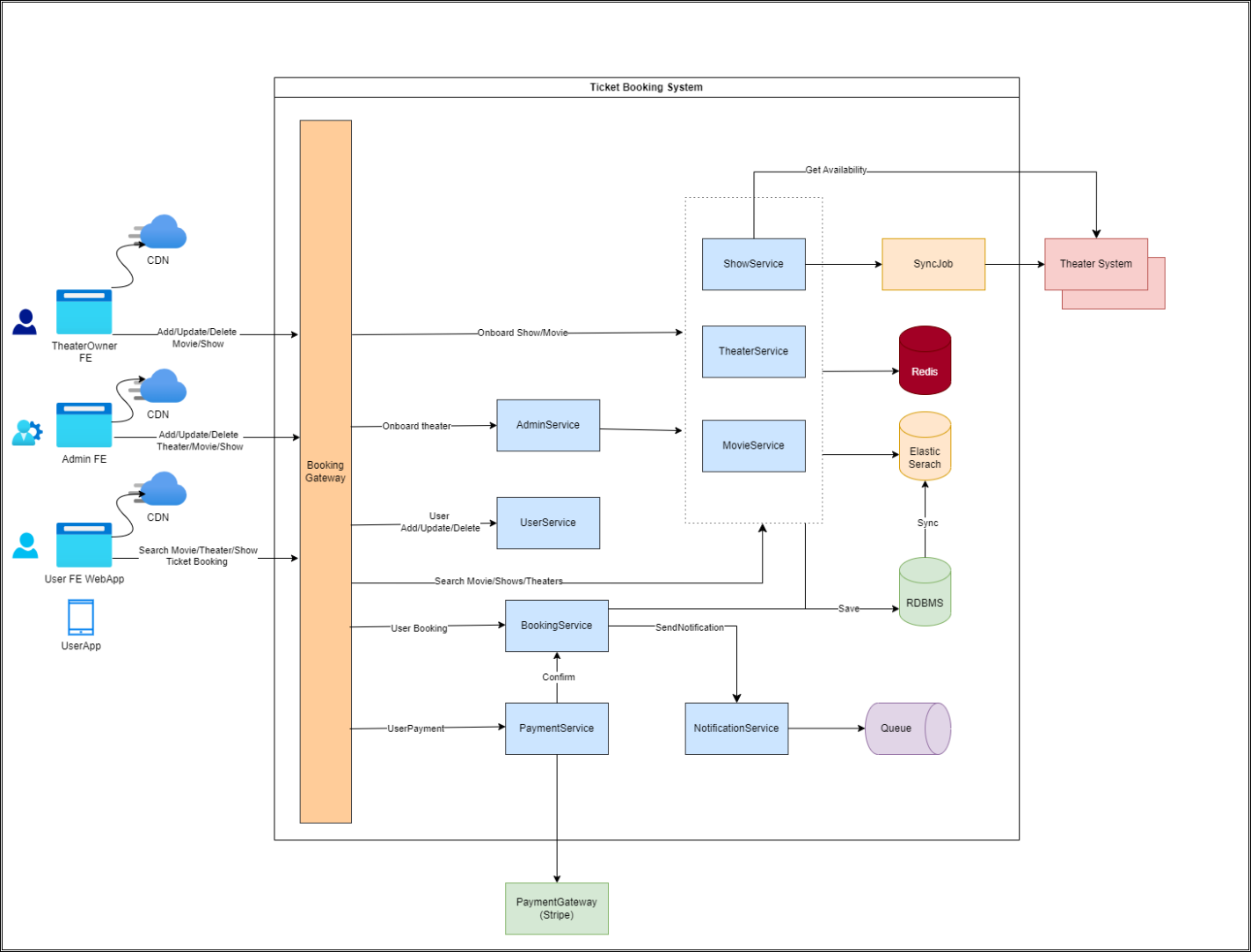
Reserve a seat and make payments.

* Theatre Owners

Create/Modify/Delete shows of the movies running in theatres.

High Level System Architecture

Below is ticket booking system’s Architectural diagram.



Frontend UI Components:

1. Web Client
2. Mobile Client
3. Admin Frontend
4. Theatre User Frontend

Backend Microservices:

1. Theatre Service
2. Movie Service
3. Search Service
4. Show Service
5. Booking Service
6. Payment Service
7. Sync Service
8. Integration Service

Other Components:

1. Ticket Service Gateway
2. CDN
3. Redis Cache
4. Elasticsearch
5. Notification Queue (AWS SNS)
6. ELK Log Aggregation Queue
7. Prometheus and Grafana for monitoring
8. Technology Selection
9. Backend microservices: Spring Boot
10. Database: PostgreSQL on AWS RDS
11. Deployment: AWS EKS
12. Monitoring: Prometheus and Grafana
13. Logging: ELK stack
14. Caching: Redis
15. Search: Elasticsearch
16. Message Queues: AWS SNS, SQS
17. CDN: AWS CloudFront

Detailed Component Description

* Redis Cache: Cache important and frequently accessed data like show details for theatres and cities.
* Elastic Search: Use for faster search results for movies and theatres information.
* Backend Services
* Theatre Service: Manage theatre-related operations.
* Movie Service: Manage movie-related operations.
* Show Service: Manage show-related operations.
* Booking Service: Manage ticket booking operations.
* Payment Service: Integrate with third-party payment gateways.
* Notification Service: Send notifications via SMS/Email.

Database Design

Microservices are using RDBMS PostgreSQL. Each microservice will be having separate DB or schema and only one microservice which is Theatre service having 2 tables with entity relationship hence below ERD depicts the same.

Theatre Service: 2 tables (Theatre, Show)

Movie Service: 1 table (Movie)

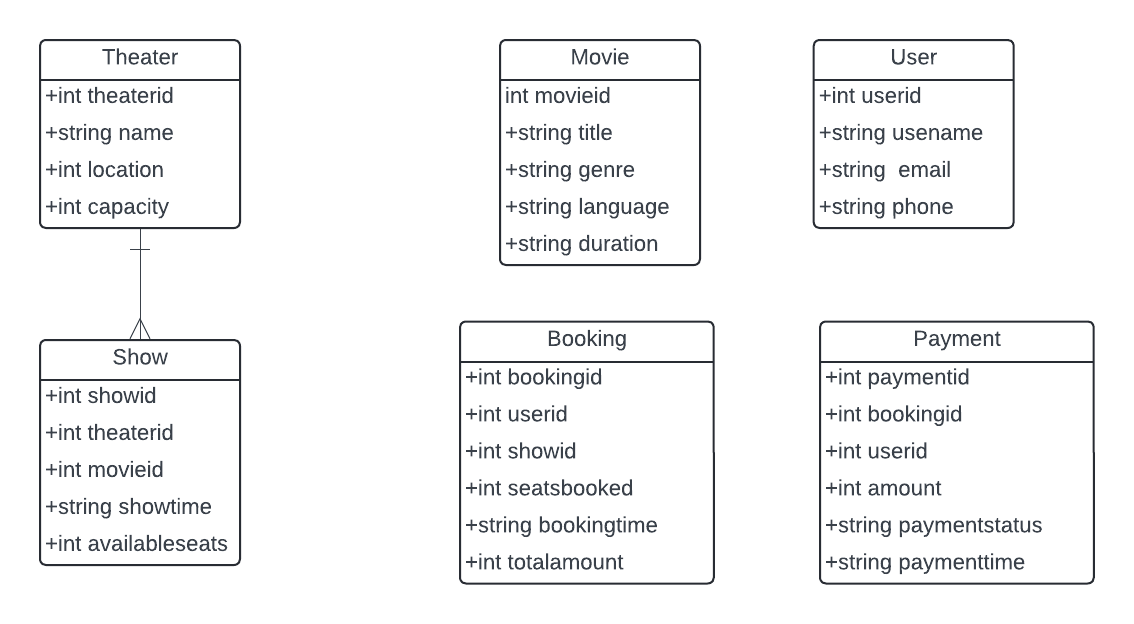
Show Service: 1 table (Show)

User Service: 1 table (User)

Booking Service: 1 table (Booking)

Payment Service: 1 table (Payment)

Notification Service: 1 table (Notification)



API endpoints

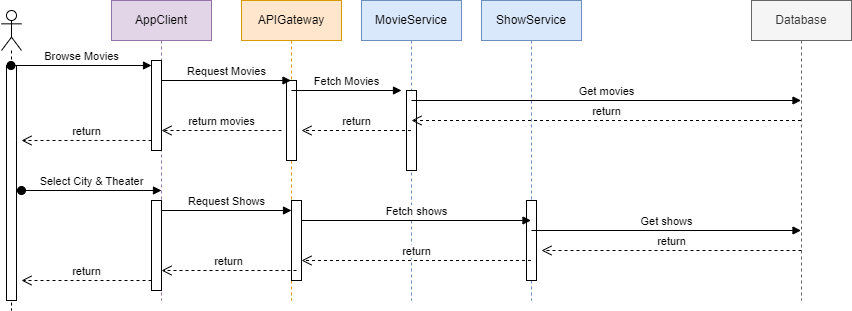
API details can be accessed using Swagger/Open API UI:

http://localhost:8080/swagger-ui/index.html

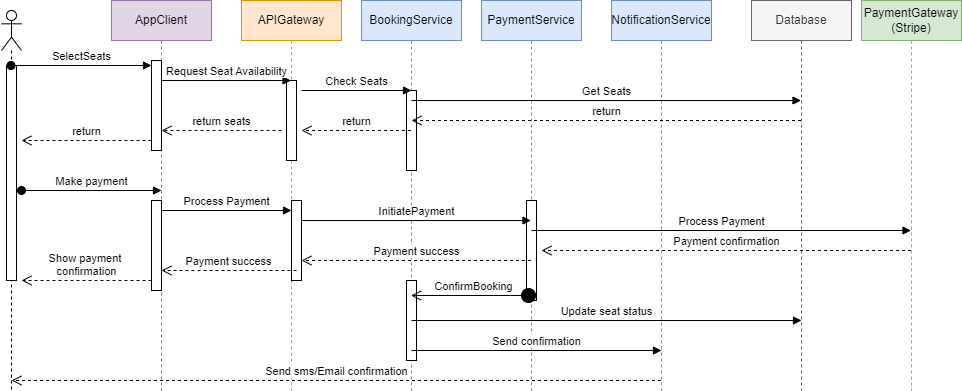
|  |  |  |
| --- | --- | --- |
| **Microservice** | **Endpoint** | **HTTP Method** |
| Theatre Service | /theatres | GET, POST |
|  | /theatres/{id} | GET, PUT, DELETE |
|  | /theatres/{id}/shows | GET, POST |
| Movie Service | /movies | GET, POST, PUT, DELETE |
|  | /movies/{id} | GET, PUT, DELETE |
| Show Service | /shows | GET, POST |
|  | /shows/{id} | GET, PUT, DELETE |
|  | /shows/{id}/availability | GET, PUT |
| User Service | /users | GET, POST, PUT, DELETE |
|  | /users/{id} | GET, PUT, DELETE |
|  | /auth/login | POST |
|  | /auth/register | POST |
| Booking Service | /bookings | GET, POST |
|  | /bookings/{id} | GET, PUT, DELETE |
|  | /bookings/user/{userId} | GET |
|  | /bookings/show/{showId} | GET |
| Payment Service | /payments | GET, POST |
|  | /payments/{id} | GET, PUT, DELETE |
|  | /payments/booking/{bookingId} | GET |
| Notification Service | /notifications | GET, POST |
|  | /notifications/{id} | GET, PUT, DELETE |
|  | /notifications/user/{userId} | GET |
|  |  |  |

Runtime View.

Below is ticket booking system’s runtime view in sequence diagram for user’s movie browsing and city and theatre selection.



Below is ticket booking system’s runtime view in sequence diagram for user’s seat selection and booking.



Concurrent Seat Booking Scenario

1. User U1 selects a preferred seat and proceeds to book.
2. Seat is locked in the Locked\_Seats table.
3. User U2 sees the seat as unavailable.
4. If U1 fails to complete payment within 10 minutes, the seat lock is released.

Other Considerations from Future Perspective

* Monetization: Charge commissions, service charges, and cancellation fees.
* Integration with Payment Gateways: Use third-party solutions like Razor pay.
* Scaling: Scale to multiple cities and countries, ensuring compliance with local policies.
* Logging and Monitoring: Use ELK stack and Grafana for continuous monitoring.
* KPIs: Track user behaviour, system performance, and resource utilization.