|  |  |  |
| --- | --- | --- |
| **Online Movie Ticket Booking Platform Design Doc** | **Summary**  This document describes the technical design of an online movie ticket booking platform. The platform is a digital platform that caters to both B2B (theater partners) and B2C (end customers) clients. It allows customers to access the services of a theater, reserve seats, and purchase tickets. The platform also provides details such as the time a movie will be played, the seats that are available, movie previews, and more. | |
| **Author**: PRAKASH MISHRA  **Created: 2025-02-23** | **Status**: In Review **Updated**: |  |

# Overview

## Objective

Design a digital movie ticket booking platform that caters to both B2B (theater partners) and B2C (end customers) clients. It allows customers to access the services of a theater, reserve seats, and purchase tickets. The platform also provides details such as the time a movie will be played, the seats that are available, movie previews, and more.

## Goals:

Key goals it wants accomplished as part of its solution:

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

## Out of scope:

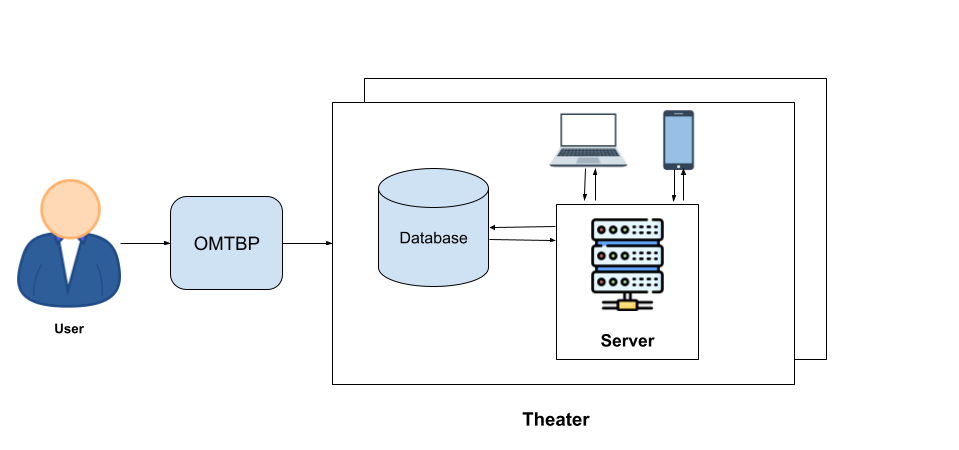
The following is out of scope for this solution:

* No UI required, Only Service Implementation needed

# Proposed Solution

Build a digital platform to allow users (theater partners and customers) to access the services of a theater, reserve seats, and purchase tickets. With the conditions below:

* **Theater Partner** - To on board their theaters over the platform and access a larger customer base while going digital
* **Customers** - To navigate the platform and access movies from different cities, languages, and genres, as well as book tickets in advance with smooth user experience.



## Requirements

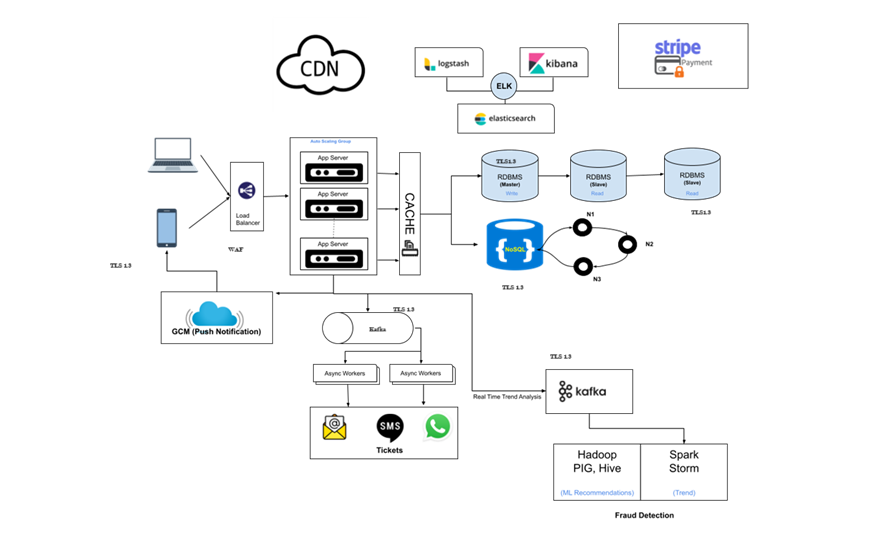
### Functional

* B2C users can browse theaters currently running the show (movie selected) in the town, including show timing by a chosen date.
* Book movie tickets by selecting a theater, timing, and preferred seats for the day.
* The users should be able to select from among a list of cities to display a list of movies showing in the area.
* Atomic transactions — only one customer can book a particular seat for a show.
* Financial transactions must be secure.
* Booking platform offers discounts (offers, deals, etc.) on bookings meeting a certain criterion.
* Theaters (B2B users) can create, update, and delete shows for the day.
* Booking and cancellation for users.
* Theaters can allocate seat inventory and update them for the show.

### Non-Functional

* The system should be highly scalable, available, reliable, consistent and fault tolerant.
* Integrate with theaters having existing IT systems and new theaters and
* localization(movies).
* How will you scale multiple cities, countries and guarantee platform availability of 99.99%?
* Movie metadata — description, actors’ info, etc.
* Personalization — user history-based suggestions.
* Log management — to debug application errors.
* Monitoring system — errors, alerts, request count, visualization tools etc.
* How do you monetize the platform?

## High Level System Design



### Architecture Principles and Guidelines

|  |  |  |
| --- | --- | --- |
| **Key Points** | **Principle** | **Description Use Case** |
| Scaling | Each service needs to replicate, typically either Vertically or Horizontally. | The most frequently used functionality is the Search functionality for the platform. The service needs to replicate either Vertically or Horizontally as the quantum of search requests increases. Kubernetes provides a great way to replicate services easily using a Replication Controller. |
| Service discovery | Multiple services might be collaborating to provide an application’s functionality. | Booking service needs data from Movie, Screen, Theatre etc.. Related services to initiate the ticket booking for the user. This requires the API gateway to identify and obtain the necessary handle . The API Gateway uses the Service Registry to perform service discovery. Netflix Eureka provides a good option for Service discovery. |
| Resiliency | High Availability and Disaster recovery | For HA and DR (High Availability and Disaster recovery) it's very important for services to automatically take corrective action and ensure the user experience is not impacted. For example, Reselience4J provides an implementation of Circuit Breaker pattern to deal with software resiliency. |
| Consistency | To maintain consistency when performing transactions | Use SQL where transactions should follow consistency and transaction level ACID properties need to be taken care of. Like ticket booking, updating the status from available to blocked to booked since it requires db transactions etc. |
| Availability | To maintain high availability meeting 99.999% uptime SLAs. | Use NoSQL while performing searches and mainly on data retrieval wherever High Availability needs to be achieved. Casandra is recommended. |
| Elastic Search | Seamless search for small and big data sets (Analytics) | Excellent choice to perform fast search for large datasets with high availability requirements. As the search load increases, this ElasticSearch container can be autoscaled. |
| Kafka with GCM | Queuing of Events for Notifications and Delivery | Booking and Notifications flows async operations which could be handled via queues. While booking tickets, there is a need to notify external systems. Queue is populated with appropriate messages/events along with metadata and the receiver service picks up the events and delivers them to the External system. On Payment confirmation, there is a need to deliver SMS/Email to customers and as well notify back to the external theater systems. |
| Service monitoring | Centralized logging easily identifies server problems or applications. | Zipkin to gather timing data needed to troubleshoot latency problems in microservice, ELK (Elastic Search, LogStash, Kibana) for searching, analyzing, and visualizing log data in real-time.  Allows you to identify any issues spanning multiple servers by correlating all the logs within a specified time frame. An open source tool which collects and stores logs, Logstash indexes the logs, while Kibana , a web interface, is used to view and search the logs already indexed. |

### Some Design Considerations

* The system will not handle partial ticket orders. Either user gets all the tickets they want, or they get nothing.
* Fairness is mandatory for the system.
* We can assume that traffic would spike on popular /much-awaited movie releases, and the seats fill up pretty fast.
* The system should be scalable, highly available to cope up with the surge in traffic.

### Capacity Estimation

#### Traffic estimates:

* Let’s assume that our service has 3 billion page views per month and sells 10 million tickets a month.

#### Storage estimates:

* Let’s assume that we have 500 cities and on average each city has ten cinemas.
* If there are 2000 seats in each cinema and on average, there are two shows every day.
* Let’s assume each seat booking needs 50 bytes (IDs, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc.) to store in the database.
* We would also need to store information about movies and cinemas, let’s assume it’ll take 50 bytes.
* Total storage in a day for all shows of all cinemas of all cities: 500 cities\*10 cinemas\*2000 seats\*2 shows\*(50+50) bytes = 2GB / day.
* To store 5 years of this data, we would need around 3.6PB.

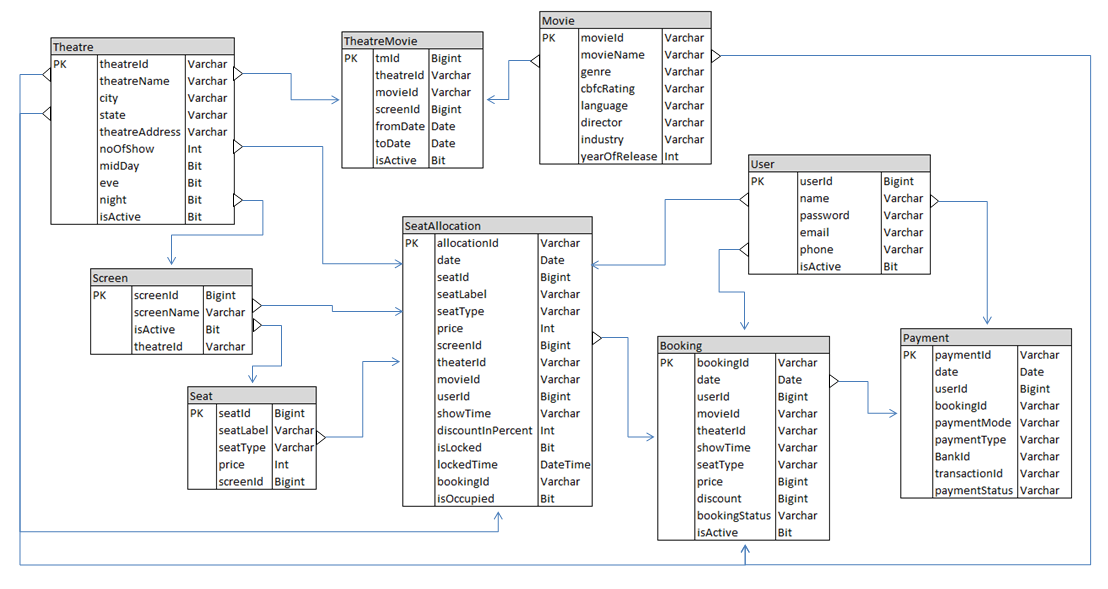
### Concurrency

* How to handle concurrency; such that no two users are able to book the same seat ?
* We can use transactions in SQL databases to avoid any clashes.
* Further we can utilize Transaction Isolation Levels to lock the rows before we can update them.

### Fault Tolerance

* The circuit breaker pattern is a reliable solution for improving the stability and resilience of integration applications.
* It handles Cascading failures and Builds a fault-tolerant and resilient system that can survive when key services are unavailable or have high latency

## Database Design



# Services

## User Service

### Purpose

* + - To manage user information and provide functionality to generate bearer tokens to authenticate users.

### Features

* + - Implemented password encoder (BCryptPasswordEncoder) for data in rest security to safely store user credentials
    - Implemented JWT to generate JWT token
    - Spring Doc implemented for API Doc generation
    - Global exception handling

## Movie Service

### Purpose

* + - To manage movie information based on different languages, genres, industries, directors etc.

### Features

* + - Uses NoSQL database (MongoDB) as persistence storage to store movie meta informations
    - Spring Doc implemented for API Doc generation
    - Global exception handling

## Theater Service

### Purpose

* + - To manage theater and its inventory information like theater details, location, screens, seating arrangements etc.

### Features

* + - Uses NoSQL database (MongoDB) as persistence storage to store movie meta informations
    - Spring Doc implemented for API Doc generation
    - Global exception handling

## Booking Service

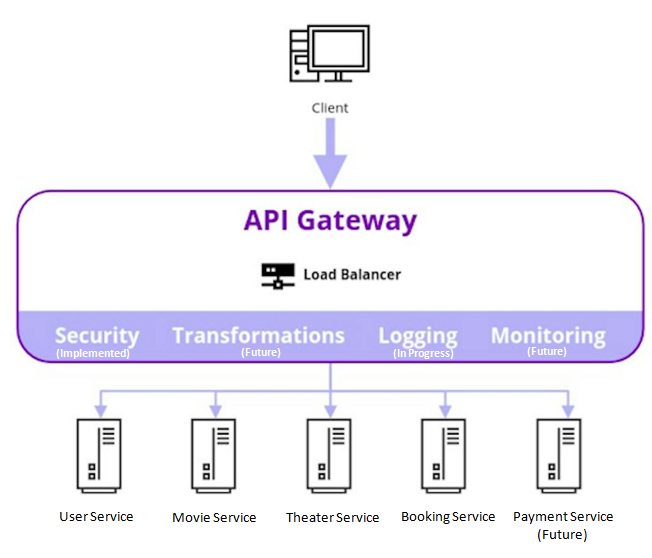
### Purpose

* + - Enables digital movie ticket bookings. Users can manage theater and its inventory information like theater details, location, screens, seating arrangements etc.

### Features

* + - Concurrency management
      * Concurrency has been achieved using the “***LockModeType.PESSIMISTIC\_WRITE***” type of lock to lock the database while the transaction is happening.
      * Here we use an instance of ***ReentrantLock()*** which returns a Lock object lock which is used with the ***tryLock()*** method.
      * Using ***lock.tryLock()*** function we make sure that if there are multiple threads executing concurrently, and one thread has already acquired the lock the remaining thread requests return an Internal Server Error exception.
    - Circuit Breaker Implementation
      * To detect failures and encapsulates the logic to prevent a failure from constantly recurring, during maintenance, external service down or unexpected system failures
    - Configurable offer management
      * 50% discount on the third ticket
      * Tickets booked for the afternoon show get a 20% discount
    - Spring Doc implemented for API Doc generation
    - Global exception handling

## Gateway Service



### Purpose

* + - For API management across different microservices. It acts as a reverse proxy to accept all application programming interface (API) calls, aggregate the various services required to fulfill them, and return the appropriate result.
    - Front interface for transforming, logging, maintaining, monitoring, and securing API calls

### Features

* + - Implemented Spring security for authentication, authorization, and protection against common attacks
    - Authenticating request payload through authorization token
    - Centralised Spring Doc implemented for API Doc generation
    - Global exception handling

## Registry Service

### Purpose

* + - Service registry enables client-side load-balancing and decouples service providers from consumers without the need for DNS

### Features

* + - Implemented Eureka Service Registry for load-balancing and decouples service providers from consumers without the need for DNS

# Future Implementations

### Service Replication using Kubernetes/GCP/EKS:

* + The service needs to replicate either Vertically or Horizontally as the quantum of search requests increases. Kubernetes provides a great way to replicate services easily using a Replication Controller.

### Distributed Transaction Management:

* + Distributed Transaction Management implementation using SAGA coreography and CAP theorem

### Payment Interface:

* + Payment Interface would be added for booking each movie ticket in their regional currency.