Design BookMyShow / TicketMaster Application

Practiced on - 5th feb 2025
Time Taken - 90 Mins (1 Hour 30 mins)

Qs for the interviewer

1. For what all events this application will be used? - Movie, concert, sports events, workshop, and other events
2. Who will be the end users? system or people
3. What all capabilities we will be giving to users as user actions?

Requirements

Functional:
1. User should be able to search an event on the platform
2.User should be able to select a seat for an event
3.User should be able to make the payment and reserve a seat for an event
4. Application will show event details (location, time, for, trailer) and show the seat map

Non Functional:
1. Consistency should be string for seat booking - no duplicate booking for a seat
2. Availability should be high for application - search, see events details, rating, review
3. Latency for seat booking should be low

Entity

1. User
2. Ticket
3. Event
4. Booking

API Design

Flow for user
1. user opens the application (website / mobile application)
2. user search for an event (movie)
3. user click on book seat
4. user select the seat
5. user make the payment
6. user gets the ticket

REQUEST > GET HTTPS/search?event=""&dateFrom=""&dateTo=""&location=""
THINK?? Should pass parameters via query param or body or the API request??
RESPONSE > JSON response for list of events

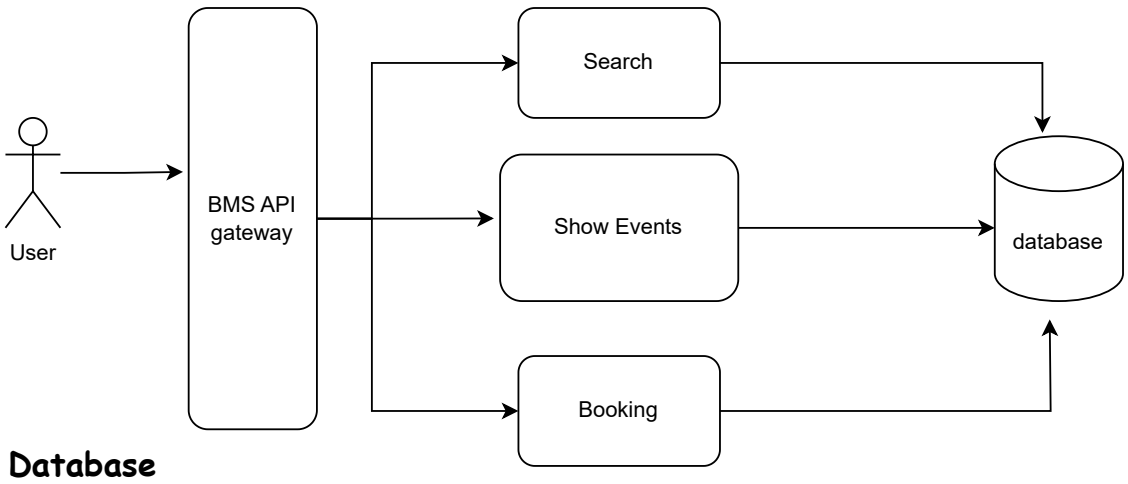
REQUEST > GET HTTPS/event="event_id"
RESPONSE > JSON response with all details about the clicked event

REQUEST > GET HTTPS/book
    body{event details}
RESPONSE > show seat-map

REQUEST > GET HTTPS/book/seat
    body{seat details}
RESPONSE > redirect to checkout / payments page > GET HTTP / payments
    body {event and seat details}

Post making payment > redirect back to HTTPS/ticket
RESPONSE with have {event seat and payment details}

High Level Design - 1st draft



Database

1. As transaction is needed in the system - it will be better if we use relational database for booking flow. RDBM will give strong Consistency
2. RDBM can have tables for [User] [Events] [Show] [Location] [Venue] [Hall] [Seat Map] [Booking] [Ticket]
3. New data in these table will be minimal!

1. For event details like videos, photos, rating, user reviews we can have No SQL, as all these are unstructured data, consistency not impo
   and all these data like rating is been updated fast and might need to scale well.

Search

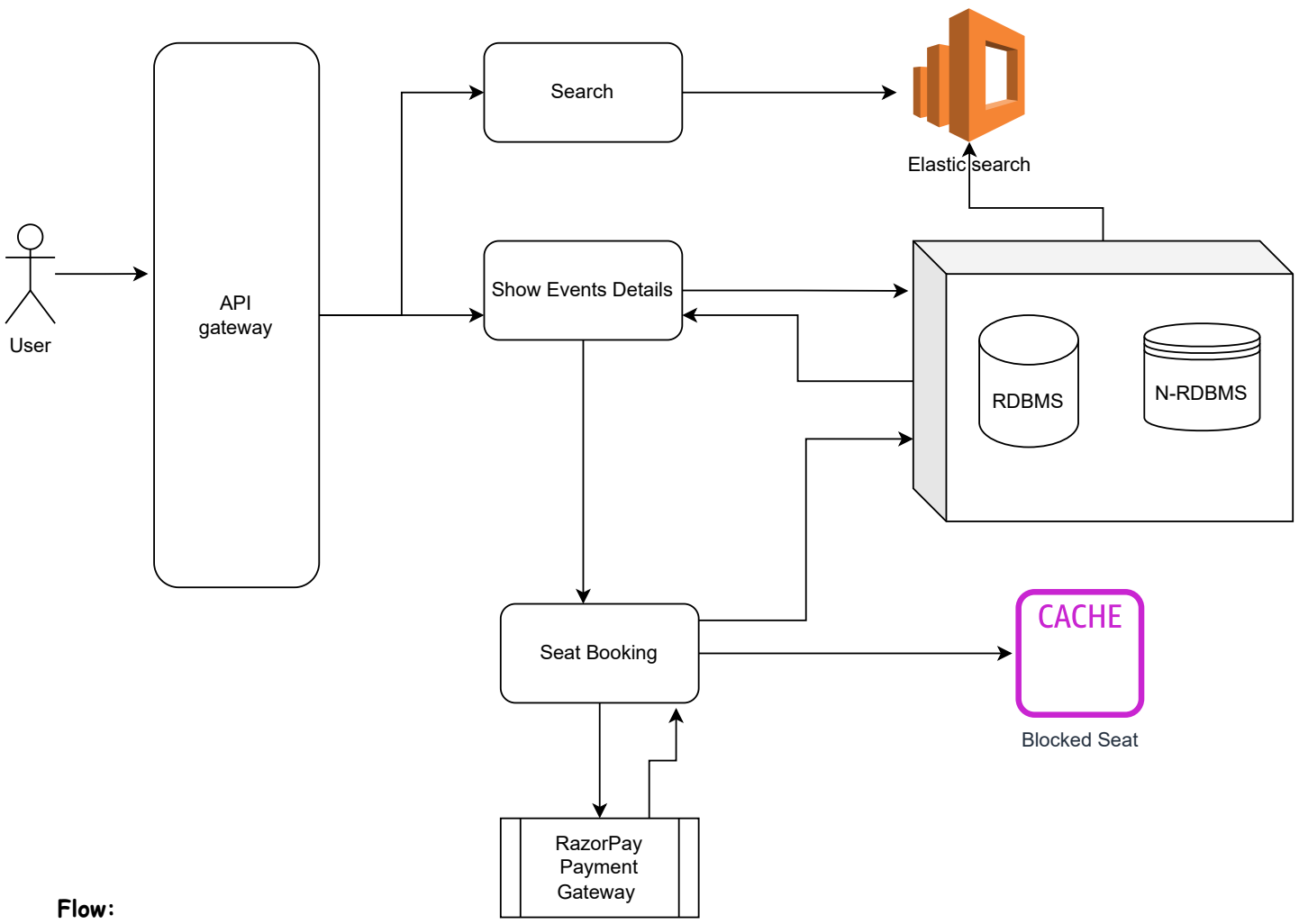
Elastic search can be used for search capabilities

Seat Booking

Seat booking will be a 2 phase work
1. Seat is been selected via the user - at this moment the seat will be in blocked state and we will wait for the user to make the payement
2. System will wait for 10 mins for the user to make the payment and in this meantime the seat will be kept blocked.
3. Seat will be reserved if user make the payment under 10 mins
4. 3rd party payment gateway like raisepay can be used here

To keep the seat blocked we can have a cache in which all the seat that are currently been blocked will be kept.

High Level System Design - Draft 2



Flow:

1. User land on the homepage of the application
2. User search for a movie
3. User see the movie details
4. User clicked on "BOOK NOW"
5. User shown the seat map
    5.1. Seat map is bought from the database - GOT FROM DATABASE
    5.2 Seat with status reserved are blacked out and can't be selected by the user - GOT FROM DATABASE
    5.3 Seats that are blocked state will able be blacked out and can't be selected by the user - GOT FROM BLOCKED SEAT CACHE
6. User select a valid un reserved seat and clicked "MAKE PAYMENT"
7. Selected seat is put in blocked seat cache with TTL of 10 mins - return an booking id
8. User is redirected for payments with booking id
9. If system got payment successful under 10 min - booking id conferment / booking id expired
10. In database the seat is marked as reserved and seat is removed from blocked seat cache

Design Deep Dive

Consistency - No duplicate Tickets

There are 3 state of a seat
1. free - PRIMARY DB - meaning seat is free for booking - will be shown to user to select
2. blocked - SEAT BLOCK CACHE - meaning seat is blocked by another user, TTL is 10 mins and current user can't select this seat
3. reserved - PRIMARY DB - meaning seat is reserved, can't se selected by current user

High Availability & Low Latency for system

To increase availability we can do the followings:
1. Add CDN in front of user - we will keep media on CDN itself - this will reduce latency
2. To increase availably we can have multiple instance of our services [Search] [Show Events] [Booking]
    2.1. We can keep multiple instances behind a load balancer for equal load
    2.2. Each service can horizontally scale independently
3. Primary DB can have a cache for table [User] [Events] [Show] [Location] [Venue] [Hall] [Booking] [Ticket]
    3.1 Can have read replica for DB, as WRITE are quite low compared to READ
4. Few (top 10)reviews for each events can be cached

[THINK: If we increase instance of Booking service we have race condition with the SEAT BLOCK CACHE - How do we keep distributed instance with single cache instance / distributed cache instances]

What in case to tone of user request for a concert

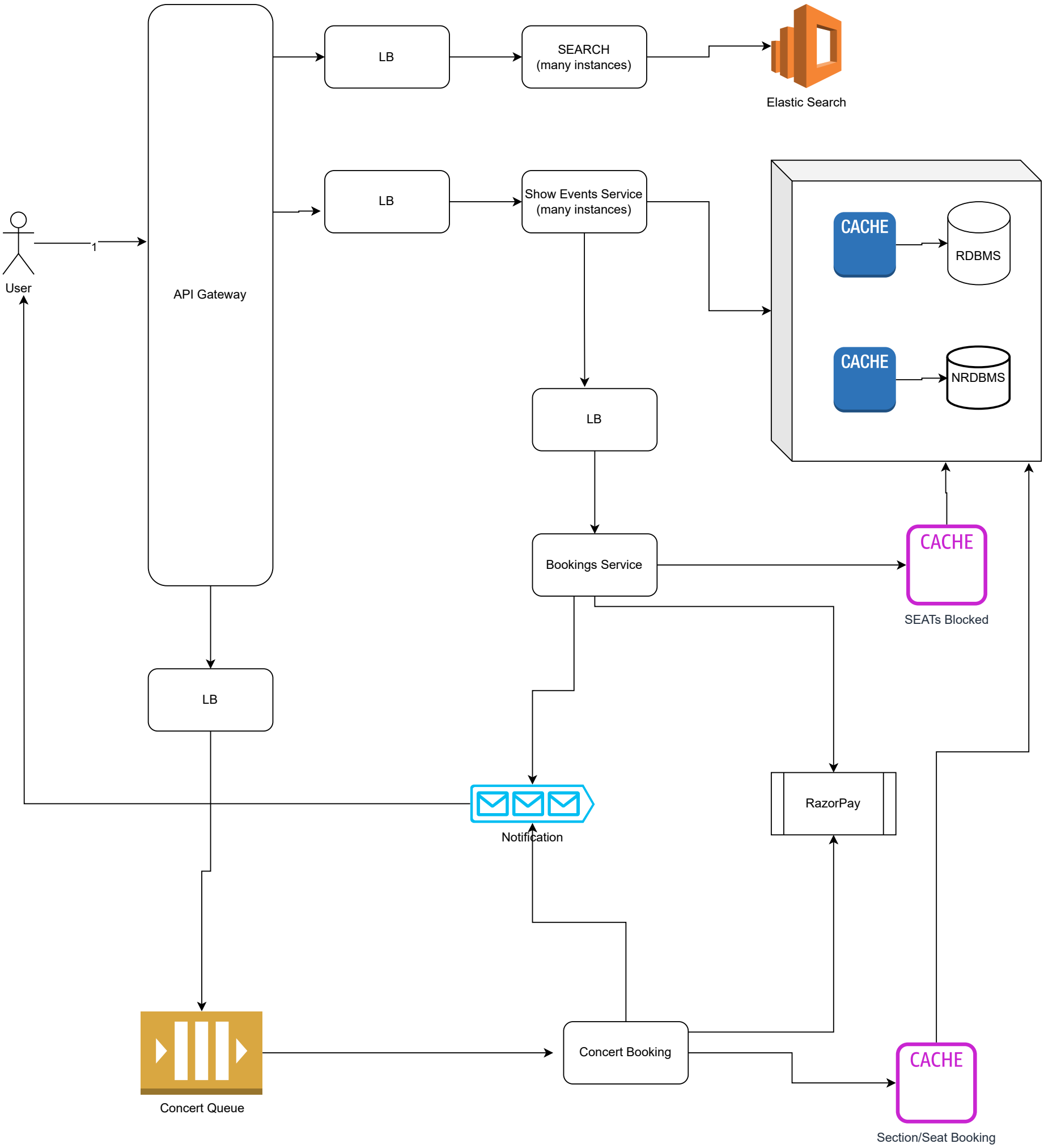
1. Incase of concert's we can introduce a diff flow all together that involved Queue
2. If all seats are already in BLOCKED / RESERVED state we can show user that they are in queue
3. If all seats are RESERVED we can show user that booking is already full

Extras

1. We can keep queue to send diff kind of notification post seat booking is confirmed
2. Ticket cancelation WF

3. Reminder WF

Final Design Diagram



Design Review by ChatGPT

What's Well Done?

1. **Clear Functional & Non-Functional Requirements**
    - You've defined the core functionalities well (search, seat selection, booking, payments).
    - Non-functional requirements like consistency, availability, and latency considerations are well thought out.
2. **Good Choice of Databases**
    - **RDBMS for bookings** to maintain strong consistency is a great choice.
    - **NoSQL for event details** (photos, reviews, ratings) is a smart approach for scalability and fast retrieval.
    - Use of **ElasticSearch** for search improves efficiency.
3. **Seat Booking Process is Well Designed**
    - The **three seat states (free, blocked, reserved)** are clearly defined.
    - Using a **cache (with TTL of 10 min) for blocked seats** is a good way to handle concurrency issues.
    - Race condition handling is addressed in scaling the booking service.

4. **High Availability & Scalability Considerations**
    - **CDN for media** to reduce latency.
    - **Load balancing and horizontal scaling** for services like Search, Show Events, and Booking.
    - **Read replicas for the database** to handle higher read traffic efficiently.
5. **Handling of High-Traffic Scenarios (Concerts)**
    - The **queue mechanism** for high-demand events ensures a better user experience.
    - Showing users their **queue position** and only allowing booking when a seat is free is a great approach.

What Can Be Improved?

1. Handling Race Conditions in Distributed Caching

- You mentioned:

  > "If we increase instances of Booking service, we have a race condition with the SEAT BLOCK CACHE."

- **Possible Solution:**
    - Instead of a single cache instance, use a **distributed caching system** like **Redis with Redlock** (a distributed locking mechanism).
    - This ensures that no two users can block the same seat simultaneously in a multi-node environment.
    - Another option is using **Optimistic Locking** in the database, where the seat selection requires checking the last modified timestamp before confirming.

2. Payment and Seat Expiry Edge Cases

- You assume that a blocked seat will be automatically unblocked after **10 minutes if no payment is received.**
- **Potential Problem:**
    - What happens if a user completes payment at exactly **9 minutes 59 seconds**, but the system delays processing?
    - Does the seat get unblocked while payment is being processed?
- **Improvement:**
    - Introduce a **grace period** where the seat remains in "pending payment confirmation" for a few more seconds before releasing it.

3. Search API Design Improvement

- Currently, you have:

  GET HTTPS/search?event=""&dateFrom=""&dateTo=""&location=""

- **Considerations:**
    - Searching for movies/events should support **filters like genre, rating, language, venue capacity, etc.**
    - Use **POST request for complex searches** (instead of GET) where body parameters can be used.
    - Implement **pagination** for large event listings.

4. Load Balancer Strategy for High Availability

- You mentioned:

  > "Each service can horizontally scale independently."

- **Improvement:**
    - Use **sticky sessions** (session affinity) for seat blocking requests to avoid conflicts.
    - **Service Discovery** (like Consul or Kubernetes) can dynamically allocate traffic between services.

5. Ticket Cancellation & Refund Flow Missing Details

- You briefly mentioned a **ticket cancellation workflow** but didn't elaborate.
- **Considerations:**
    - How will refunds be processed?
    - Will canceled tickets **go back to available seats** immediately?
    - Should there be **cancellation fees** or refund rules based on the event type?

Final Verdict

✅ **Strong system design overall** with a well-thought-out architecture.
✅ **Good choice of technologies** (RDBMS, NoSQL, Redis Cache, ElasticSearch).
✅ **Scalability and availability considerations** are solid.
⚠️ **Some edge cases and race conditions need better handling.**
⚠️ **Queue mechanism could be further refined for high-demand events.**

Next Steps

1. Address race conditions in caching (use Redlock or Optimistic Locking).
2. Improve handling of edge cases in payment and seat expiry.
3. Refine search API for better filtering and pagination.
4. Improve load balancing and service discovery for better failover handling.
5. Flesh out the ticket cancellation and refund policies.