Design BookMyShow (*New*)

Let's design an online ticketing system that sells movie tickets like BookMyShow.

Similar Services: bookmyshow.com, ticketmaster.com
Difficulty Level: Hard

1. What is an online movie ticket booking system?

A movie ticket booking system provides its customers the ability to purchase theatre seats online. It ticketing systems allow the customers to browse through movies currently being played and to book seats, anywhere and anytime.

2. Requirements and Goals of the System

Our online ticketing system should meet the following requirements:

LEARN TEACH

1. Our ticket booking service should be able to list down different cities where its affiliate cinemas are located.
2. Once the user selects the city, the service should display the movies released in that particular city.
3. Once the user selects the movie, the service should display the cinemas running that movie and its available shows.
4. The user should be able to select the show at a particular cinema and book their tickets.
5. The service should be able to show the user the seating arrangement of the cinema hall. The user should be able to select multiple seats according to their preference.
6. The user should be able to distinguish available seats from the booked ones.
7. Users should be able to put a hold on the seats for five minutes before they make a payment to finalize the booking.
8. The user should be able to wait if there is a chance that seats might become available - e.g. when holds by other users expire.
9. Waiting customers should be serviced fairly in a first come first serve manner.

Non-Functional Requirements:

1. The system would need to be highly concurrent. There will be multiple booking requests for the same seat at any particular point in time. The service should handle this gracefully and fairly.
2. The core thing of the service is ticket booking which means financial transactions. This means that the system should

LEARN TEACH

be ACID compliant.

3. Some Design Considerations

1. For simplicity, let's assume our service doesn't require user authentication.
2. The system will not handle partial ticket orders. Either user gets all the tickets they want, or they get nothing.
3. Fairness is mandatory for the system.
4. To stop system abuse, we can restrict users not to book more than ten seats.
5. We can assume that traffic would spike up exponentially around movie releases, and the system will still pretty fast. The system should be scalable, highly available to cope up with the surge in traffic.

4. Capacity Estimation

Traffic estimates: Let's assume that our service has 3 billion page views per month and sells 30 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 10 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 10 bytes. So, to store all the data about all shows of all cinemas of all cities for a day:

LEARN TEACH

To store five years of this data, we would need around 3.6PB.

5. System APIs

We can have SOAP or REST APIs to expose the functionality of our service. Following could be the definitions of the APIs to search movie shows and reserve seats.

```
SearchMovies(api_dev_key, keyword, city, lat_long, radius, start_datetime, end_datetime, postal_code,
includeplayOnlyNearby, results_per_page, sorting_order)
```

Parameters:
api_dev_key (string): The API developer key of a registered account. This will be used to, among other things, throttle users based on their allocated quota.
keyword (string): Keyword to search on.
city (string): City to filter movies by.
lat_long (string): Latitude and longitude to filter by **radius (number):** Radius of the area in which we want to search for events.
start_datetime (string): Filter movies with a start datetime after this datetime.

LEARN TEACH

includeplayOnlyNearby (Enum: "yes" or " not"): Yes, to include spell checks suggestions in the response.
results_per_page (number): How many results to return per page. Maximum is 30.
sorting_order (string): Sorting order of the search result. Some allowable values : `"name_asc", "name_dec", "date_asc", "date_dec", "distance_asc", "name,date_asc", "name,date_dec", "date,name_asc", "date,name_dec"`.

Returns: (JSON)
Here is a sample list of movies and their shows:

```
[
  {
    "MovieID": 1,
    "ShowID": 1,
    "Title": "Avatar",
    "Description": "Avatar",
    "Duration": "2 hours",
    "Summary": "Avatar",
    "Language": "English",
    "ReleaseDate": "9th Oct. 2014",
    "Country": USA,
    ...
  }
]
```

LEARN TEACH

```
[
  {
    "Cinema": {
      "Type": "Regular",
      "Price": 14.00,
      "Status": "Almost Full"
    },
    {
      "Type": "Premium",
      "Price": 24.00,
      "Status": "Available"
    }
  ],
  {
    "MovieID": 1,
    "ShowID": 2,
    "Title": "Avatar",
    "Description": "Avatar",
    "Duration": "2 hours",
    ...
```

LEARN TEACH

```
    "Country": USA,
    "StartTime": "10:00",
    "EndTime": "12:30",
    "Seats":
    [
      {
        "Type": "Regular",
        "Price": 14.00,
        "Status": "Full"
      },
      {
        "Type": "Premium",
        "Price": 24.00,
        "Status": "Almost Full"
      }
    ]
  },
  ...
]
```

LEARN TEACH

api_dev_key (string): same as above
session_id (string): User's session ID to track this reservation. Once the reservation time expires, user's reservation on the server will be removed using this ID.
movie_id (string): Movie to reserve.
show_id (string): Show to reserve.
seats_to_reserve (number): An array containing seat IDs to reserve.

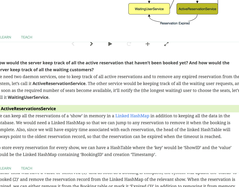
Returns: (JSON)
Returns the status of the reservation, which would be one of the following: 1) "Reservation Successful" 2) "Reservation Failed - Show Full," 3) "Reservation Failed - Retry, as other users are holding reserved seats".

6. Database Design

Here are a few observations about the data we are going to store:

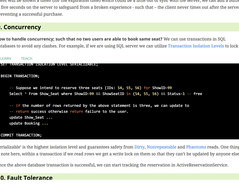
1. Each City can have multiple Cinemas.
2. Each Cinema will have multiple halls.
3. Each Movie will have many Shows, and each Show will have multiple Bookings.

LEARN TEACH



7. High Level Design

At a high level, our web servers will manage Users' sessions, and application servers will handle all the ticket management, storing data in the databases and work with cache servers to process reservations.



8. Detailed Component Design

First, let's try to build our service assuming if it is being served from a single server.

Ticket Booking Workflow: Following would be a typical ticket booking workflow:

LEARN TEACH

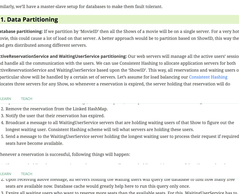
1. User searches for a movie.
2. User selects a movie.
3. User is shown the available shows of the movie.
4. User selects a show.
5. User selects the number of seats to be reserved.
6. If the required number of seats are available, the user is shown a map of the theater to select seats. If not, the user is taken to 'step' of below.

LEARN TEACH

- Show is full, the user is shown the error message.
- The seats user wants to reserve are no longer available, but there are other seats available, so the user is taken back to the theater map to choose different seats.
- There are no seats available to reserve, but all the seats are not booked yet as there are some seats that other users are holding in reservation pool and have not booked yet. The user will be taken to a waiting page where they can wait until required seats get freed from the reservation pool. This waiting could result in following options:
  - If the required number of seats become available, the user is taken to the theater map page where they can choose the seats.
  - While waiting if all seats get booked, or there are fewer seats in the reservation pool than the user intend to book, the user is shown the error message.
  - User cancels the waiting and is taken back to the movie search page.
  - At maximum, a user can wait one hour, after that user's session gets expired and the user is taken back to the movie search page.

7. If seats are reserved successfully, the user has five minutes to pay for the reservation. After payment, booking is marked complete. If the user is not able to pay within five minutes, all their reserved seats are freed to become available to other users.

LEARN TEACH



LEARN TEACH

How would the server keep track of all the active reservation that haven't been booked yet? And how would the server keep track of all the waiting customers?

We need two daemon services, one to keep track of all active reservations and to remove any expired reservation from the system, let's call it **ActiveReservationService**. The other service would be keeping track of the waiting user requests, and as soon as the required number of seats become available, it'll notify the the longest waiting user to choose the seats, let's call it **WaitingUserService**.

a. ActiveReservationService

We can keep all the reservations of a 'show' in memory in a Linked HashMap in addition to keeping all the data in the database. We would need a Linked HashMap so that we can jump to any reservation to remove it when the booking is complete. Also, since we will have expiry time associated with each reservation, the head of the linked HashTable will always point to the oldest reservation record, so that the reservation can be removed when the timeout is reached.

To store every reservation for every show, we can have a HashTable where the 'key' would be 'ShowID' and the 'value' would be the Linked HashMap containing 'BookingID' and creation 'Timestamp'.

LEARN TEACH

Whenever a new booking request comes in, we will insert a new record into this HashTable and a corresponding entry in the Linked HashMap. When booking is complete, we will remove the reservation record from the Linked HashMap of the relevant show. When the reservation is expired, we can either remove it from the Booking table or mark it 'Expired' CH' in addition to removing it from memory.

ActiveReservationService will also work with the external financial service to process user payments. Whenever a booking is completed, or a reservation gets expired, WaitingUserService will get a signal, so that any waiting customer can be served.

```
LinkedHashMap<BookingID, Timestamp> LinkedHashMap = new LinkedHashMap();
HashTable<ShowID, LinkedHashMap> ActiveReservationsOfAllShows = new HashTable();
ActiveReservationsOfAllShows.Add(1, LinkedHashMap);
ActiveReservationsOfAllShows.Add(2, LinkedHashMap);
```

Ex: ShowID : LinkedHashMap: BookingID, Timestamp }
1(0 : { (1, 1:00:00:00:000), (2, 1:00:01:07:000), (3, 1:00:01:00:000) }

ActiveReservationsService keeping track of all active reservations

LEARN TEACH

b. WaitingUserService

For a popular show, we will have many users waiting to book tickets, as soon as some seats become available, we can't just let the first user in the queue to book the seats, as that would be unfair to other users. So we need to maintain a fair queue for each show. We can use a Linked HashMap to store all the waiting users for every show. The 'key' would be 'ShowID, and the 'value' would be a Linked HashMap containing 'UserID' and their wait-start-time.

LEARN TEACH

Clients can use Long Polling for keeping themselves updated for their reservation status. Whenever seats become available, the server can use this request to notify the user.

Reservation Expiration

On the server, ActiveReservationsService keeps track of expiry (based on reservation time) of active reservations. As the client will be shown a timer (for the expiration time) which could be a little out of sync with the server, we can add a buffer of five seconds on the server to safeguard from a broken experience - such that - the client never times out after the server, preventing a successful purchase.

9. Concurrency

How to handle concurrency; such that no two users are able to book same seat? We can use transactions in SQL databases to avoid any clashes. For example, if we are using SQL server we can utilize Transaction Isolation Levels to lock

LEARN TEACH

```
BEGIN TRANSACTION;

  -- Suppose we intend to reserve three seats (IDs: 54, 55, 56) for ShowID 99
  Select * From Show_Seat where ShowID=99 && ShowSeatID IN (54, 55, 56) && Status 0 - Free

  -- If the number of rows returned by the above statement is three, we can update to
  -- return success otherwise return failure to the user.
  update Show_Seat ...
  update Booking ...

COMMIT TRANSACTION;
```

Serializable is the highest isolation level and guarantees safety from Dirty, Nonrepeatable and Phantom reads. One thing to note here, within a transaction if we read rows we get a write lock on them so that they can't be updated by anyone else.

Once the above database transaction is successful, we can start tracking the reservation in ActiveReservationService.

10. Fault Tolerance

LEARN TEACH

Whenever ActiveReservationsService crashes, we can read all the active reservations from the 'Booking' table. Remember that we keep the 'Status' column as 'Reserved (1)' until a reservation gets booked. Another option is to have master-slave configuration so that when master crashes slave can take over. Since we are not storing the waiting users in the database, so when WaitingUserService crashes, we don't have any means to recover that data unless we have a master-slave setup.

Similarly, we'll have a master-slave setup for databases to make them fault tolerant.

11. Data Partitioning

Database partitioning: If we partition by 'MovieID' then all the shows of a movie will be on a single server. For a very hot movie, this could cause a lot of load on that server. A better approach would be to partition based on ShowID, this way the load gets distributed among different servers.

ActiveReservationsService and WaitingUserService partitioning: Our web servers will manage all the active users' sessions and handle all the communication with the users. We can use Consistent Hashing to allocate application servers for both ActiveReservationsService and WaitingUserService based upon the 'ShowID'. This way, all reservations and waiting users of a particular show will be handled by a certain set of servers. Let's assume for load balancing our Consistent Hashing allocates three servers for any Show, so whenever a reservation is expired, the server holding that reservation will do

LEARN TEACH

1. Remove the reservation from the Linked HashMap.
2. Notify the user that their reservation has expired.
3. Broadcast a message to all WaitingUserService servers that are holding waiting users of that Show to figure out the longest waiting user. Consistent Hashing scheme will tell what servers are holding these users.
4. Send a message to the WaitingUserService server holding the longest waiting user to process their request if required seats have become available.

LEARN TEACH

Whenever a reservation is successful, following things will happen:

LEARN TEACH

1. Close receiving service messages, so service running on waiting users will query our database to see how many free seats are available now. Database cache would greatly help here to run this query only once.
2. Expire all waiting users who want to reserve more seats than the available seats. For this, WaitingUserService has to iterate through the Linked HashMap of all the waiting users.