I. Designing Ticketmaster

1. requirements and goals of the system
    a. functional requirements
        i. List other cities where affiliated cinemas are located
        ii. Select a city to display movies released in that city
        iii. When you select a movie, it displays the movie theater and the available performance time.
        iv. Select a performance at a specific cinema and book a ticket
        v. Shows the seating arrangement at the cinema and selects multiple seats
        vi. Reserved / available seat classification
        vii. Must be able to secure a seat for 5 minutes prior to payment to complete the reservation
        viii. Waiting customers should be served on a first-come, first-served basis.
    b. non-functional requirements
        i. High concurrent-must be handled fairly and elegantly even if there are multiple requests for the same seat
        ii. Ticket Reservation = Financial Transaction: System must be secure and database ACID compliant
2. design considerations
    a. User authentication is not required
    b. Partial ticket orders are not processed (ex) get all tickets or get nothing
    c. Fairness is essential
    d. To prevent system abuse, reservations cannot be made for more than 10 seats at a time
    e. The system needs to be designed for scalability and requires high availability (availablity)-to cope with the surge of traffic

3. capacity estimation
    a. assume
        i. views : 3B views/month
        ii. sells : 10M tickets/month
        iii. cities : 500
        iv. average each city has 10 cinemas
        v. average each cinema has 2000 seats
        vi. shows : 2 shows/day
        vii. seat booking size : 50 byte (ID, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc)
        viii. movie info size : 50 byte
    b. estimation
        i. 500 (cities) * 10 (cinemas) * 2000 (seats) * 2 (shows) * (50 + 50) byte = 2GB / day
        ii. 3.5 TB / 5years
4. system apis - soap, rest api
    a. SearchMovie (api_dev_key, keyword, city, lat_long, radius, start_datetime, end_datetime, postal_code, includeSpellcheck, results_per_page, sorting_order)
        i. keyword: Keyword to search
        ii. lat_long (string): Latitude and longitude to search
        iii. start_datetime (string): Start date time
        iv. end_datetime (string): End date time
        v. includeSpellcheck (enum: ‘yes’ or ‘no’): Include spelling suggestions in the response
        vi. results_per_page (number): Number of results to return per page
        vii. sorting_order (string): Sort order of search results
        viii. return (json)
    b. ReserveSeats (api_dev_key, session_id, movie_id, show_id, seats_to_reserve [])
        i. session_id (string): User's session ID to track the reservation, if the reservation time (5 minutes)

expires, the user's reservation is removed from the server through session_id

ii. show_id (string): Show reservation

iii. seats_to_reserve (number): Array containing the sin ID to reserve

iv. return (json)

- Reservation success
- Failure to Book-Show All
- Failed reservation-retry, reservation already held by another user

5. database design

a. consideration

i. Each city has multiple theaters

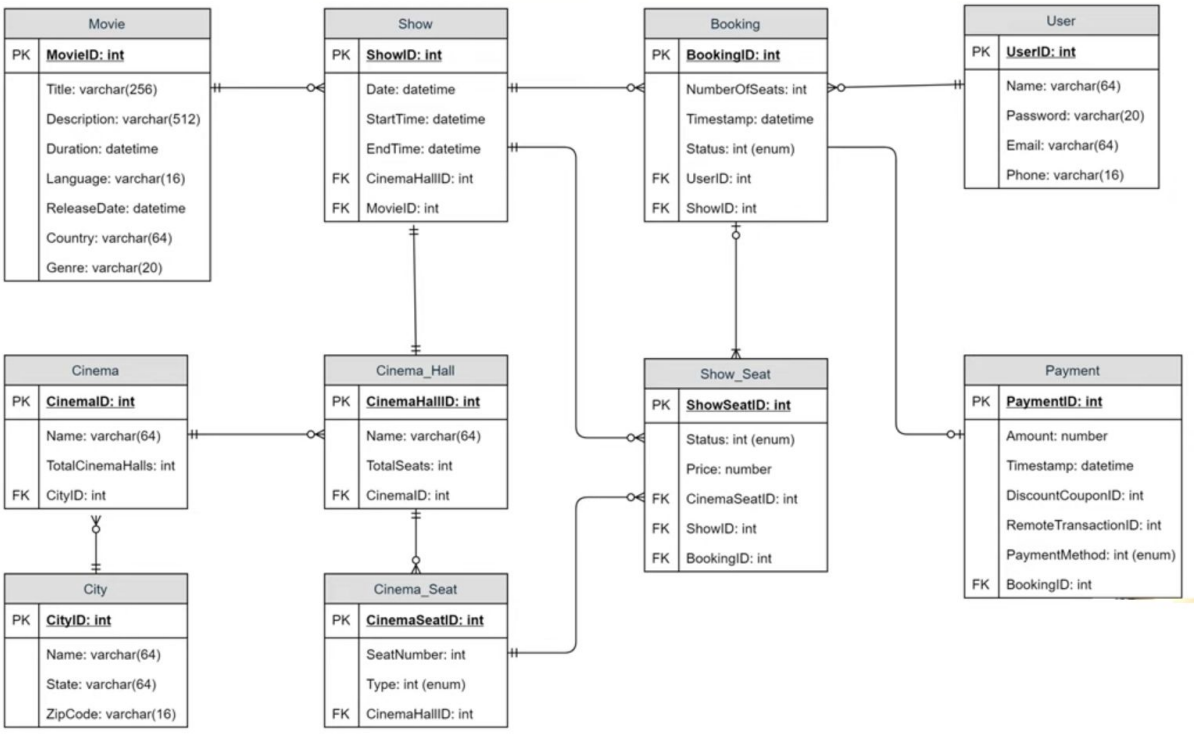
ii. Each cinema has multiple halls

iii. Each movie has multiple shows

iv. Each show has multiple seats

v. User can make multiple reservations

b. ERD Reference



6. high level design
    a. web server: User session management
    b. applicaton server: ticket management
    c. db server: data storage
    d. cache server: Improved performance (response speed)

e.

7. component design-Assuming that the service is provided by a single server
    a. ticket reservation workflow
        i. Movie Search
        ii. Movie Selection
        iii. Time selection
        iv. Select number of seats
        v. If there are selected seats-> Show theater map
        vi. If there are no magnets selected
            - Manseok (marked as error to the user)
            - User no longer wants to reserve, but there is another show, so return to the theater map
            - Go to the standby page
                ✓ Go to the theater map page when the number of seats required is available
                ✓ Error message if all seats are reserved while waiting or if there are fewer seats in the reservation pool

- ✓ Cancel waiting and go to movie search page
- ✓ User sessions expire and can return to the movie search page and wait up to an hour

vii. If the seat is successfully reserved, the user is given 5 minutes to pay. After 5 minutes, the seat is transferred to another user

b. How does the server track all active reservations that haven't been reserved yet, and how to track all waits?
   i. ActiveReservationService
      - Track all active reservations and remove expired reservations from the system
        - ✓ Requires HashMap data structure
        - ✓ HashMap head always points to the oldest reservation record-> reservation expires when timeout is reached
        - ✓ key : ShowID
        - ✓ value: HashMap with BookingID and reservation start time
      - Process user payments in cooperation with external financial services

- Whenever a reservation is completed / expired, WaitingUsersService is signaled to provide service to waiting customers.

ii. WaitingUserService
    - Track all pending user requests and notify the longest waiting waiter to select a seat as soon as the required number of seats is available
    - HashTable
        - ✓ Save all waiting users
        - ✓ key : CinemaID
        - ✓ value: UserID, HashMap with waiting start time
iii. Client can use Long Polling to keep up to date information on reservation status-Server notifies the user of this request when seat is available
iv. Reservation expiration
    - ActiveReservationsService on Server tracks the expiration of active reservations (based on reservation time)
    - Timer (during expiration time) that client and server may not be synchronized is displayed.

8. concurrency
    a. Handles concurrency so that two users cannot reserve the same seat
        i. Using SQL DB Transaction
        ii. When using SQL-SERVER, you can lock and update rows using Transaction Isolation Levels
        iii. When a row is read within a transaction, a write lock on the row is acquired so that others cannot update it
        iv. If transaction is successful, reservation tracking can be started in ActiveReservationService

9. fault tolerance
    a. What if ActiveReservationsService or WaitingUsersService crashes?
        i. Every active reservation can be read from the 'Reservation' table whenever the ARS crashes
            - Status remains reserved (1) until the reservation is completed
        ii. Master-slave configuration: slave can take over when master crashes
            - There is no means to recover data unless there is a master-slave setting when WUS crashes because the waiting user is not stored in the db.
    b. Set the master-slave db to have fault tolerance in the db
10. data partitioning
    a. database partitioning
        i. Partitioning by MovieID ensures that all shows in the movie are on a single server
        ii. In case of hot movie, a lot of load is imposed on the server.
        iii. A better way is to partition based on ShowID-load is distributed to other servers
    b. ActiveReservationService and WaitingUserService partitioning
        i. web server: manage sessions of all active users, and handle all communication with users
        ii. Assign application server to both ARS and WUS based on ShowID using Consistent hashing
        iii. All reserved and standby users of a specific show are processed by a specific server set
        iv. workflow-Suppose you assign 3 servers to show for LB of consistent hashing
            - Update DB to remove reservation or mark it as expired
            - Update the seat status in the 'Show_Seat' table

- Remove reservation from Linked HashMap
- To identify the user who has been waiting the longest, a message is broadcast to all WUS servers that have a standby user at the show.
- Sends a message to the WUS server to handle the request with the oldest standby user if the required seat is available

v. The following happens each time you make a successful reservation:
    - The server holding the reservation sends a message to all servers waiting for the user in the show, causing the server to expire all waiting users requiring more seats than are available.
    - Upon receiving the above message, all servers with waiting users query the DB to find the number of seats currently available-DB cache is very helpful in executing this query once.
    - Expire all waiting users who want to reserve more seats than available seats-WUS iterates and processes HashMaps of all waiting users