Problem Statement

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

Functional Requirements

There are two actors in our system: users who need to buy tickets for movies, and show owners who need to upload their information for the movie.

- Users should be able to search for movies
- Users should be able to choose their movies and seats for booking
    - The user experience should let the user cling on to the seat for say, 10 minutes.
- Users should be notified if seats have become available for a movie if previously it was full
- Users should be able to make purchases for the interested movie(s)
- Show owners should be able to add/update movies they are currently showcasing

Non-Functional Requirements

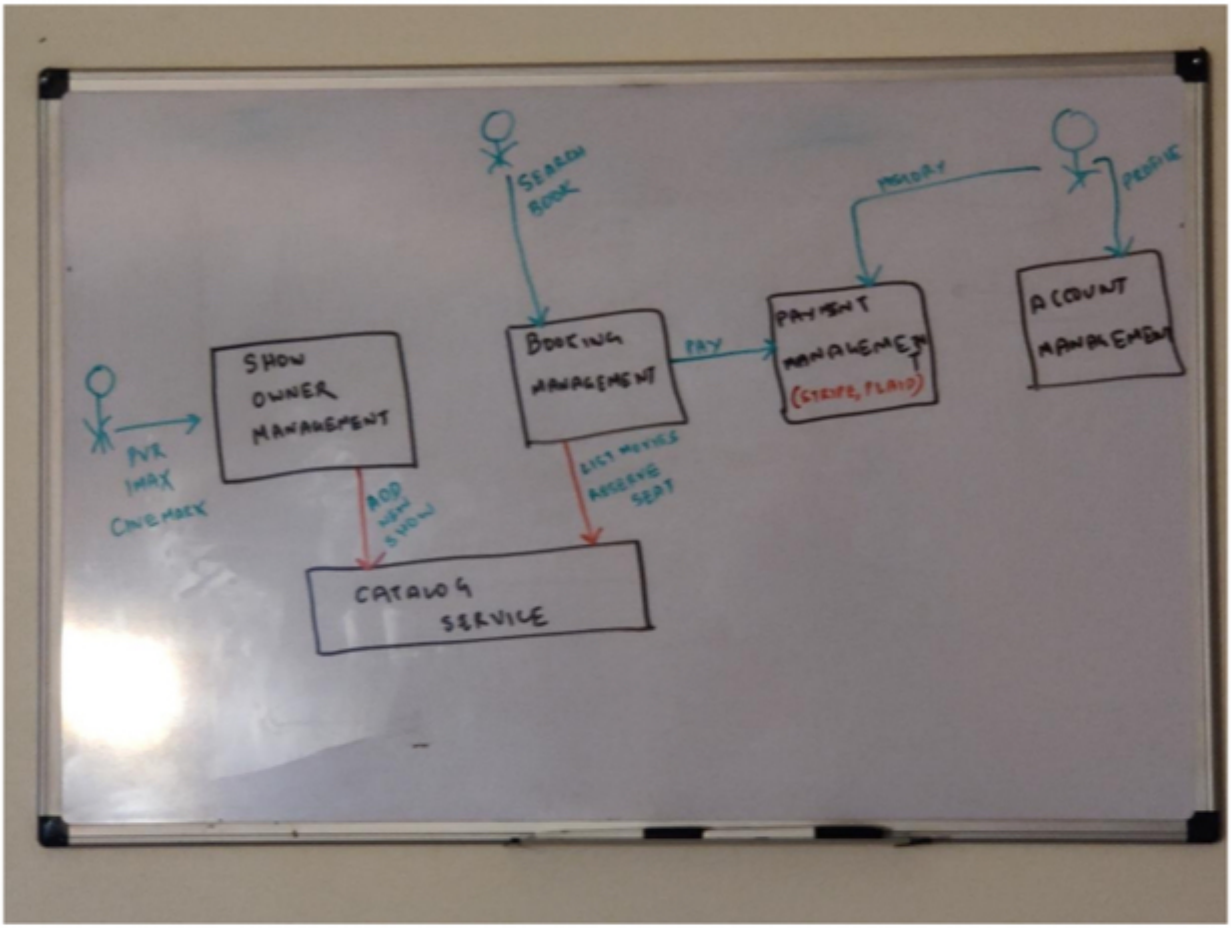
- We need a concurrent system where multiple users should not be able to book the same seat for an event
- Scalability (more on that later)
- Payment transactions should be secure

Microservices

Now that we have our use-cases defined, we should be good to come up with our data models and microservices! Let's define the microservices required for our system:

- Account Management Service: User registration, history
- Payments Management Service: for making payments, history of transactions etc.
- Booking Management Service: for reserving and booking seats
- Show Owner Service: Allows show owners to add information
- Catalog Service: Provides a service layer for booking management service and show owner service to get and update movie information respectively. In this way, we keep our show owner and booking management services stateless, and they rely on catalog service for any movie related data (state).

Let's draw the high-level diagram from what we already know:



Flow of Requests:

- Owners such as Cinemark or PVR can add/update shows for movies through show owner management
    - Show owner management simply contacts the catalog service to persist the new information
    - Catalog service does the meat of data mutations and additions. We will discuss this service in depth later.
- Customers can search for movies
    - It is okay to pluck out search to a different service as well; but in the end that will also call catalog service, so we have bundled them together for simplicity. Also, as we will see later, “search” here is different from Google’s search that relies on inverted index. In our case, catalog service’s APIs will handle the search from the data stores which contain the movies and its related information.
    - Search requests go to catalog service which returns the response based on the query (we will come up with data models and APIs for this in a later section)
    - The search can be “incremental”; such as the user selecting city, “then” movie, and “then” time.

        - For our purposes, we don't care as for us the API design will simply add more "filters".

- Customers will select the appropriate show
    - At this point, there are no more search operations; the UI/client will take the response of the final selected show and present the information
    - UI/client will make another query to us to ask for the seats available for the show
- Customers can book seats:
    - We ask our catalog service to "reserve" the seat for the customer.
    - The catalog service takes care of the timeout/expiration of reservations
- Customers confirm their purchase:
    - We hand-off the purchase to the payment management service with a transaction no or reference ID
    - Note that we have marked the diagram as "Stripe, Plaid" -> these are just FinTech companies that can be used to delegate payments. Again, this is not the problem we want to solve.

Account Management Service

- This is a fairly solved problem, so won't be discussed in depth.
- This service will also be used by catalog service to fetch user information
- Simple key-value workload required for which a row-oriented database will suffice.

Table: User

- **Primary Key:** UserID
- **Columns:** Name, Email, Password, Contact
- A caching layer on top of the database will help for "noisy" or frequent users.
- For "guests", which can be argued as a valid use-case for booking movies, we store everything but their username/password (marked as "anonymous").

Catalog Service

This microservice forms the crux of our problem to serve low latency requests for users.

One observation from our system is that it consists of highly structured data, i.e., contains various entities to serve a customer request. A city might have multiple cinema halls, who in turn might be broadcasting multiple movies with multiple shows. Moreover, each show can have various kinds of seats and prices. In such an environment, we value data integrity and want to keep each transaction secure. A "transaction" in our case can take various forms: reserving a seat *only if it's free*, making a payment *transactionally (credit and debit)*, *accurately* update the *available seats* for a movie hall, and so on. Keeping these considerations in mind, we choose a relational database for our design to keep things normalized, ACID compliant, and correct. This blog has some key pointers for choosing between SQL vs NoSQL.

Before getting into multiple tables, let's assume we had to store everything in 1 row. In this way, we can identify the tables from a gigantic block (also termed as normalization).

User	Movie	City	Cinema	Branch	Show	Hall	Seats	Credit Card
Alice	Batman	Seattle	Cinemark	Downtown	1 PM	4A	1,2,3,4	xxxx
Bob	Batman	Seattle	Cinemark	SLU	1 PM	4A	10,11	xxxx
Satoshi	Bahubali	Redmond	Cinemark	Downtown	5 PM	1B	18,19	xxxx

As we can see above, we have already duplicated Batman, Seattle, and Cinemark multiple times in three rows. Through normalization techniques, or in this case by visual inspection itself, we can pluck out the various columns into its own separate table. Note that this is also required because individual columns above themselves have a lot of information to store, and we don't want to keep duplicating data for all of them. I'll let you brainstorm normalization from here further.

Once we normalize our database, i.e., remove redundancy and enhance integrity (by adding foreign and primary keys), we can come up with the below list of entities or tables required for our system. The below tables have been inspired from existing sources.

Our system's catalog contains multiple movies.

Table: Movie
- **Primary Key:** MovieID
- **Columns:** Title, Description, Duration, Language, Date, Genre

Multiple movies maybe shown in multiple locations

Table: Location
- **Primary Key:** LocationID
- **Columns:** City, Latitude, Longitude, State, Zip

Each location can have multiple movie theatres

Table: MovieTheatre
- **Primary Key:** TheatreID
- **Foreign Key(s):** LocationID (Location)
- **Columns:** Name

Each theatre can have multiple halls that showcases movies

Table: MovieTheatreHall
- **Primary Key:** MovieTheatreHalliD
- **Foreign Key(s):** TheatreID (MovieTheatre)

- **Columns:** Name, Seats

Each theatre hall contains of seats

Table: MovieTheatreHallSeat
- **Primary Key:** MovieTheatreHallSeatID
- **Foreign Key(s):** TheatreID (MovieTheatre)
- **Columns:** Name, SeatLabel, SeatType

Now every movie hall will showcase a "Show".

Table: Show
- **Primary Key:** ShowID
- **Foreign Key(s):** MovieThreatreHallID (MovieTheatreHall), MovieID (Movie)
- **Columns:** StartTime, EndTime, Date

Each show will have a booking

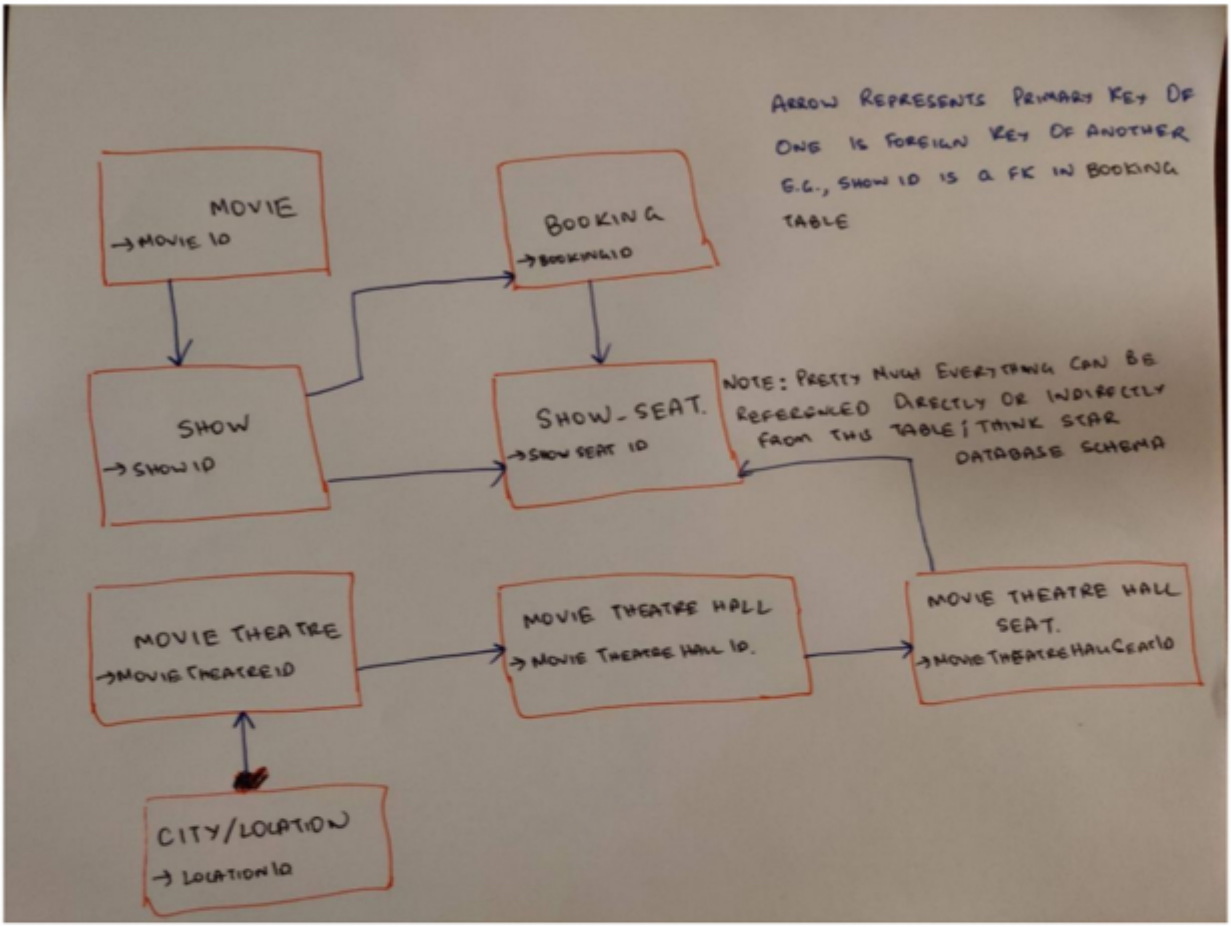
Table: Booking
- **Primary Key:** BookingID
- **Foreign Key(s):** ShowID
- **Columns:** UserID**,** SeatCount, Status, Timestamp

Note that userID doesn't have to be a foreign key. We can always retrieve the user details from session tokens and through the account management service.

Even though we identified seats for a particular movie theatre's hall above, we need a separate entity, which we call **ShowSeat,** and it identifies the seats booked for a unique show, in a unique movie hall, and in a unique location. Note the words "unique" being mentioned multiple times. These are the constraints we will add to our unique "seat" with the help of foreign keys. We will maintain the state of seats for each show as well. The different "states" each seat can take will be Available, Reserved, and Expired. Expirations will help us let users cling onto seats for say 10 minutes. Don't confuse this entity with **MovieTheatreHallSeat,** which is meant only to identify the seats for a particular theatre's hall. That hall seat will be a foreign key in **ShowSeat.**

Table: ShowSeat
- **Primary Key:** ShowSeatID
- **ForeignKeys:** MovieTheatreSeatID **(MovieTheatreSeat),** BookingId (Booking), ShowID (Show)
- **Columns:** Status (enum: Available, Reserved, Expired)

Note that with the help of foreign keys above, we can also obtain any information from all the other tables. In a STAR relational schema, **ShowSeat** becomes our facts table where any dimensions can be retrieved through the foreign keys in this fact table.



Now that we have our data models setup, we can design our critical APIs for our users. Our request and response take JSON formats.

Users can search for movies:

- Response searchMovies(Request):
    - Request:
        - Keyword: String
        - Filters: List[Filter]
    - Response:
        - Movies: List[Movie]
    - Movie:
        - Details from movie, shows, location etc. (think Joins)
    - Filter:
        - Location: Location
        - Time: timestamp

        - .....

Users can reserve seats:

- Response reserveSeats(Request):
    - Request:
        - Movie: Movie
        - Show: Show
        - ShowSeat: ShowSeat
    - Response:
        - BookingID: String

Users can complete reservation by making payments. Alternately, the catalog service itself can redirect to payments service by calling their provide API. This API is managed by Payments microservice that is spoken about in a later section.

Show owners can upload show information:

- Response uploadNewShow(Request):
    - Request:
        - Show: Show
        - Movie: Movie
        - MovieTheatre: MovieTheatre
        - MovieTheatreHall: MovieTheatreHall
        - Location: Location
    - Response
        - Follow Linux philosophy; be silent on success and scream (exceptions) on failures ☺

A sample transaction to handle the booking can look like:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

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;
```

Traffic and Storage

- **Traffic:** Let's assume we have 1 billion views per month, and we sell 10 million tickets per month. That comes to around 400 views/second, and 5 bookings/second. If one view takes 200ms to execute, and let's say a transaction takes 500ms, then through our formula 30,000/t, we will need around 150 servers. We will also add a cache that will be useful particularly for viewing the movies and shows.

- **Storage:** Let's assume each show booking needs around 50 bytes of data. For other information such as movies and theatres, let's assume another 50 bytes. For 100 cities, 10 theatres each city, 1000 seats each theatre, and 5 daily shows will take up approximately:

  100 bytes * 100 cities * 10 theatres * 1000 seats * 5 shows ~ 0.5 gb/day

How do we partition our data?

MovieID has a low cardinality, and is a candidate for hotspots. Shows have a relatively high cardinality, and partitioning our data on ShowID is a viable approach.

How do we manage expired reservations and users who were sent back?

Since we maintain the state of reserved seats, and associate it with their timestamp, we can have an asynchronous job that periodically looks at reserved seats and expires them if the timestamp has crossed 10 minutes. We can then notify users who wanted to book seats but were sent back due to unavailability. Notification system is outside the scope of this document.

If our data grows and the job becomes slow, we will need to use API parallelization techniques to speed up the job.

Payments Service

- This is a fairly solved problem, so won't be discussed in depth; some key pointers:

Table: Payments

- **Primary Key: PaymentID**
- **Columns:** BookingID, Amount, Coupon, CC Number

API for clients/booking service to use:

Response completeBooking(Request):
    - Request:
        - BookingID: String
    - Response:
        - TransactionID: String

Note that bookingID is generated by the catalog service, so this field doesn't need to be a foreign key of the bookings table in the catalog service. The payments service will make sure that adding entries for a bookingID is idempodent.

- The only key thing here is to remember that payments should use ACID compliant databases and be made using transactions.
- Note that we have marked the diagram as "Stripe, Plaid" -> these are just FinTech companies that can be used to delegate payments. Again, this is not the problem we want to solve.

Show Owner Management Service

- Nothing concrete here; just a thought process of two different actors in a system
- Uses uploadNewShow API provided by Catalog Service to persis new shows
- Keeps details about show owners in a separate table call **ShowOwners**

Booking Management Service

- Nothing concrete here; just a thought process of two different actors in a system
- Uses reserveSeats and completeBooking APIs provided by other services