**Lower Level Design**

**Snake and Ladder:**

What are the requirements?

a.) How many players will be there? - 2

b.) Can there be more in the future?

c.) How many types of players? - Humans

d.) How many snakes and ladders? – User Request / Random

e.) Is size of the board configurable?

f.) How does a player win? The player crosses the cell

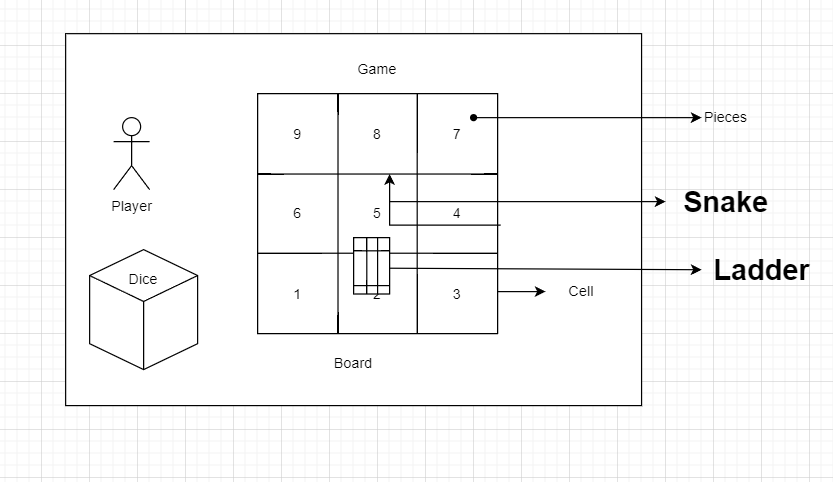
g.) can he cross with any number while at 99?

h.) how does the player start? – roll dice which is either 1 or 6

i.) Roll Dice – 6 faces

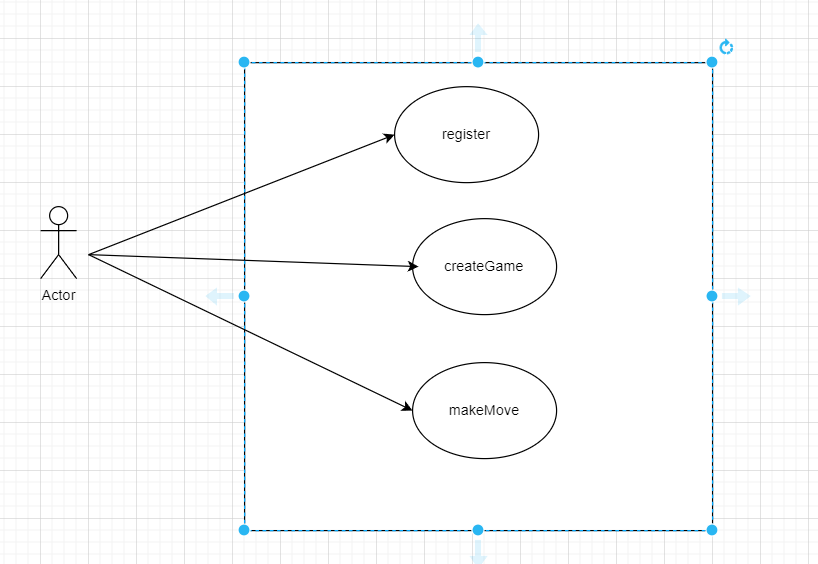
j.) One player is playing with multiple dices sum=3+6

**Entities :**  Game, Board, Cell, Player, Dice, Snake, Ladder,

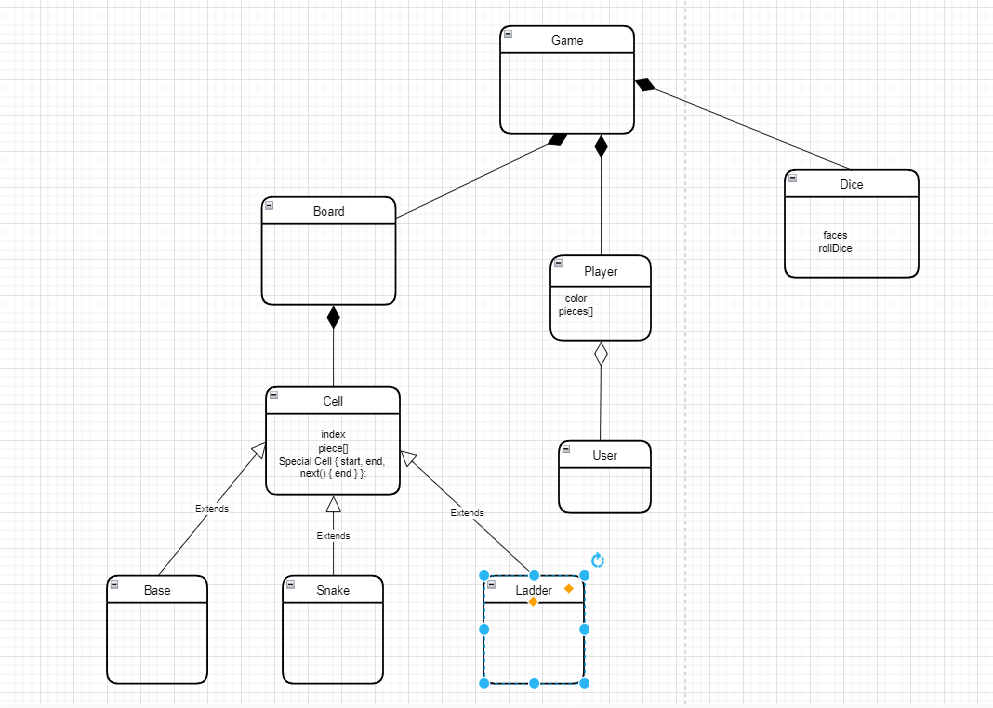


**Actors:** Player, Future (Admins)

**Use cases**:



**Class Diagram or ER Diagram:**



**Chess:**

What are the requirements?

a.) How many players will be there? - 2

b.) Can there be more in the future?

c.) How many types of players? - Humans

d.) What are the rules to followed?

e.) Is size of the board configurable?

f.) Can player cancel and rollback the moves?

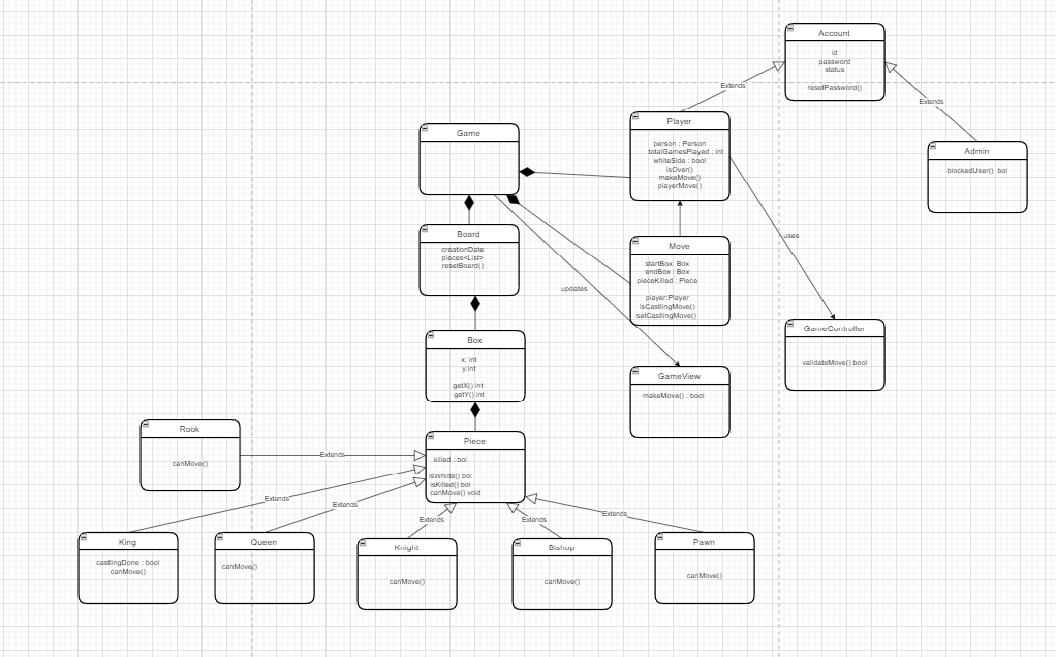
g.) Each side will start with 8 pawns, 2 rooks, 2 bishops, 2 knights, 1 queen, and 1 king

h.) What are the conditions to finish the game?

The game can finish either in a checkmate from one side, forfeit or stalemate (a draw), or resignation.

Actors: Player, Admin

Entities: Game, Board, Box, Piece, Player, Account, GameController, GameView



**Parking Lot:**

What are the requirements?

a.) Do we have multiple floors?

b.) Do we have multiple entry and exit points?

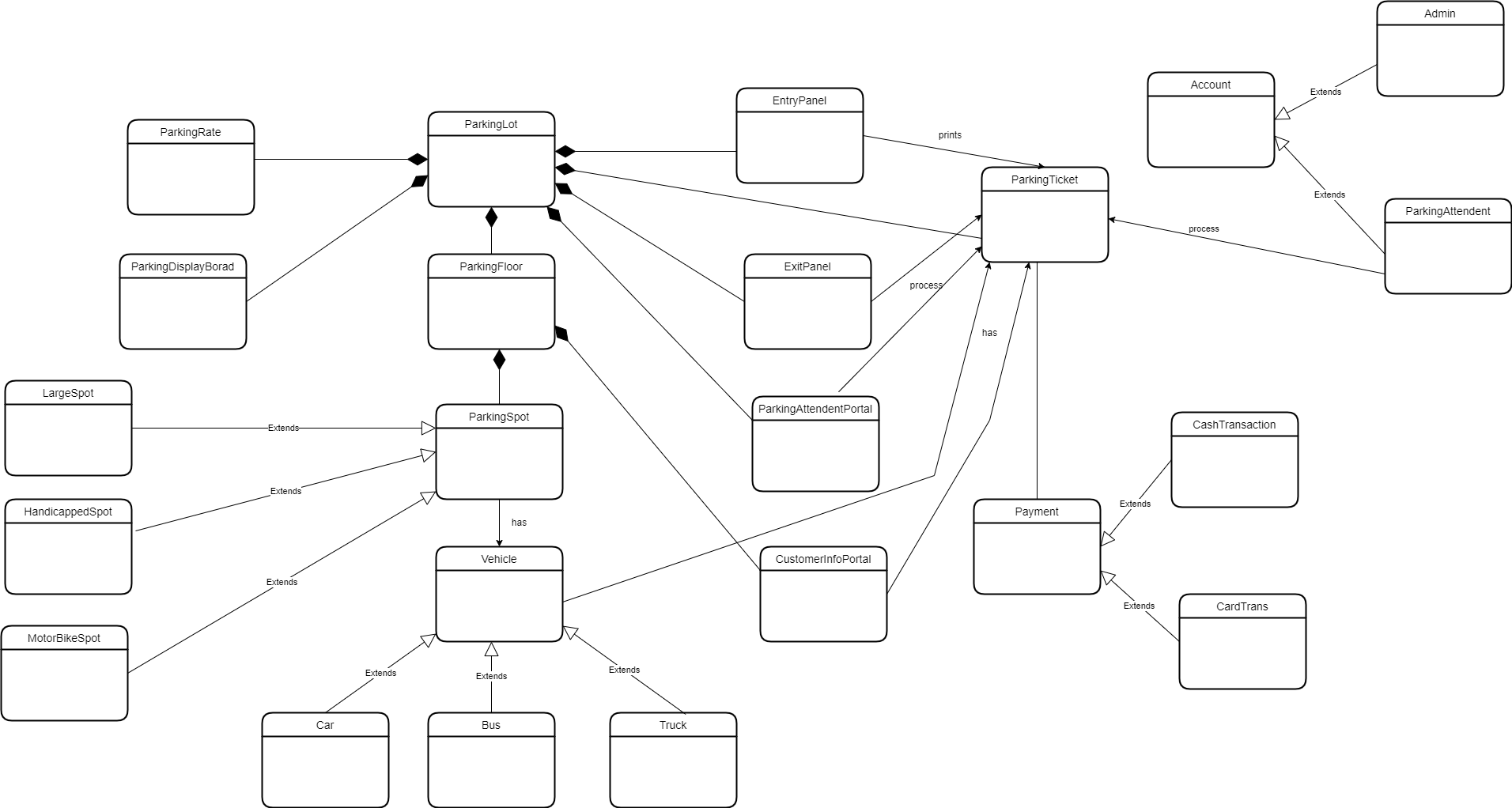
c.) Do we have display board?

d.) What are the payment options?

e.) What are the types of parking spots?

Actors: User, Admin

Entities : ParkingLot, ParkingFloor, ParkingSpot, Vehicle, Account, Parking Rate, Entry Panel, Exit Panel, Payment, ParkingTicket, CustomerInfoPortal, ParkingAttendentPortal, ParkingDisplayBoard



**Movie Ticket:**

**Parking Lot:**

What are the requirements?

a.) Each cinema can have multiple halls and each hall can run one movie show at a time.

b.) Each movie has multiple shows

c.) Customer should be able to search movies by their title, language, genre, release date and city name.

d.) Customer should be able to select multiple seats according to their preference

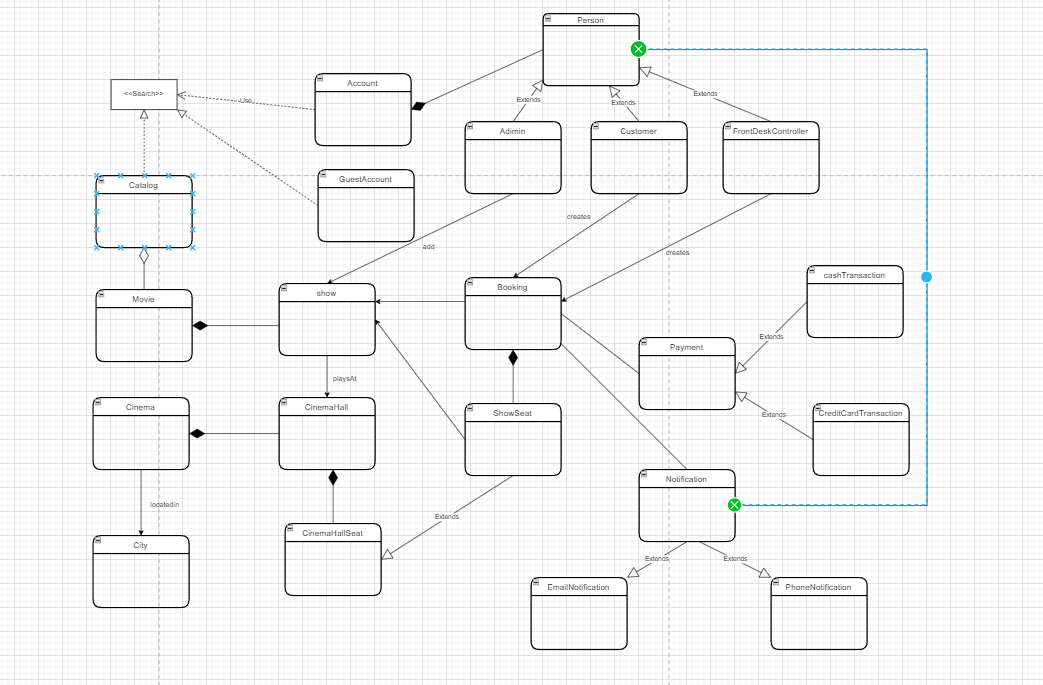
e.) They should be able to distinguish between available seat and booked ones

f.) System should able to send the notification

g.) Customer should be able to do the payment using cash or credit cards.

h.) Customer should be able to add a discount coupon to their payment

Entities : Account, Guest, City, Cinema, CinemaHall, Movie, Show, CinemaHallSeat, ShowSeat, Booking, Payment, Notification



SET TRRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION

{  
 SELECT \* FROM SHOWSEAT WHERE SHOWID=99 && SHOWSEATID IN (54,55,56) && isReserved=0;

}

COMMIT TRANSACTION;

import java.sql.DriverManager;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.sql.ResultSet;

public class Customer extends Person {

public boolean makeBooking(Booking booking) {

List<ShowSeat> seats = booking.getSeats();

Integer seatIds[] = new Integer[seats.size()];

int index = 0;

for(ShowSeat seat : seats) {

seatIds[index++] = seat.getShowSeatId();

}

Connection dbConnection = null;

try {

dbConnection = getDBConnection();

dbConnection.setAutoCommit(false);

// ‘Serializable’ is the highest isolation level and guarantees safety from

// Dirty, Nonrepeatable, and Phantoms reads

dbConnection.setTransactionIsolation(Connection.TRANSACTION\_SERIALIZABLE);

Statement st = dbConnection.createStatement();

String selectSQL = "Select \* From ShowSeat where ShowID=? && ShowSeatID in (?) && isReserved=0";

PreparedStatement preparedStatement = dbConnection.prepareStatement(selectSQL);

preparedStatement.setInt(1, booking.getShow().getShowId());

Array array = dbConnection.createArrayOf("INTEGER", seatIds);

preparedStatement.setArray(2, array);

ResultSet rs = preparedStatement.executeQuery();

// With TRANSACTION\_SERIALIZABLE all the read rows will have the write lock, so we can

// safely assume that no one else is modifying them.

if (rs.next()) {

rs.last(); // move to the last row, to calculate the row count

int rowCount = rs.getRow();

// check if we have expected number of rows, if not, this means another process is

// trying to process at least one of the same row, if that is the case we

// should not process this booking.

if(rowCount == seats.size()) {

// update ShowSeat table...

// update Booking table ...

dbConnection.commit();

return true;

}

}

} catch (SQLException e) {

dbConnection.rollback();

System.out.println(e.getMessage());

}

return false;

}

}

**Redis – Ahead of file, RDB**

**Solid Design Principal:**

**1.) Single Responsibility Principle**

**// single responsibility principle - bad example**

interface IEmail {

public void setSender(String sender);

public void setReceiver(String receiver);

public void setContent(String content);

}

class Email implements IEmail {

public void setSender(String sender) {// set sender; }

public void setReceiver(String receiver) {// set receiver; }

public void setContent(String content) {// set content; }

}

**// single responsibility principle - good example**

interface IEmail {

public void setSender(String sender);

public void setReceiver(String receiver);

public void setContent(IContent content);

}

interface Content {

public String getAsString(); // used for serialization

}

class Email implements IEmail {

public void setSender(String sender) {// set sender; }

public void setReceiver(String receiver) {// set receiver; }

public void setContent(IContent content) {// set content; }

}

**2.) Open Closed Principle:**

**// Open-Close Principle - Bad example**

class GraphicEditor {

public void drawShape(Shape s) {

if (s.m\_type==1)

drawRectangle(s);

else if (s.m\_type==2)

drawCircle(s);

}

public void drawCircle(Circle r) {....}

public void drawRectangle(Rectangle r) {....}

}

class Shape { int m\_type; }

class Rectangle extends Shape { Rectangle() {

super.m\_type=1;}

}

class Circle extends Shape

{Circle() { super.m\_type=2; } }

**// Open-Close Principle - Good example**

class GraphicEditor {

public void drawShape(Shape s) {

s.draw(); } }

class Shape {

abstract void draw(); }

class Rectangle extends Shape {

public void draw() {

// draw the rectangle

}

}

**3.) Liskov’s Substitution Principle:**

Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

**//Violation of Likov's Substitution Principle**

class Rectangle

{

protected int m\_width; protected int m\_height;

public void setWidth(int width) { m\_width = width; }

public void setHeight(int height) { m\_height = height; }

public int getWidth() { return m\_width;}

public int getHeight() { return m\_height; }

public int getArea(){ return m\_width \* m\_height; }

}

class Square extends Rectangle {

public void setWidth(int width){

m\_width = width;

m\_height = width;}

public void setHeight(int height){

m\_width = height;

m\_height = height;

}}

class LspTest{

private static Rectangle getNewRectangle() { return new Square();}

public static void main (String args[]){

Rectangle r = LspTest.getNewRectangle();

r.setWidth(5); r.setHeight(10);

System.out.println(r.getArea());

}

}

**4.) Interface Segregation Principle:**

**// interface segregation principle - bad example**

interface IWorker {

public void work(); public void eat();

}

class Worker implements IWorker{

public void work() {

// ....working

}

public void eat() {

// ...... eating in launch break

}

}

class SuperWorker implements IWorker{

public void work() {

//.... working much more

}

public void eat() {

//.... eating in launch break

}

}

class Manager {

IWorker worker;

public void setWorker(IWorker w) {

worker=w;

}

public void manage() {

worker.work();

}

}

**// interface segregation principle - good example**

interface IWorker extends Feedable, Workable {}

interface IWorkable { public void work();}

interface IFeedable { public void eat(); }

class Worker implements IWorkable, IFeedable

{ public void work() { // ....working }

public void eat() {

//.... eating in launch break }

}

class Robot implements IWorkable{

public void work() { // ....working }

}

class SuperWorker implements IWorkable, IFeedable{

public void work() {

//.... working much more

}

public void eat() {

//.... eating in launch break

}

}

class Manager {

Workable worker;

public void setWorker(Workable w) {

worker=w; }

public void manage() {

worker.work();

}

}

**5.) Dependency Inversion Principle:**

**// Dependency Inversion Principle - Bad example**

class Worker {

public void work() { // ....working }

}

class Manager {

Worker worker;

public void setWorker(Worker w) {

worker = w;

}

public void manage() {

worker.work();

}}

class SuperWorker {

public void work() { }

}

**// Dependency Inversion Principle - Good example**

interface IWorker { public void work(); }

class Worker implements IWorker{

public void work() {

// ....working }

}

class SuperWorker implements IWorker{

public void work() {}

}

class Manager {

IWorker worker;

public void setWorker(IWorker w) {

worker = w;

}public void manage() {

worker.work();}

}

**HLD**

Choice of database: 1.) Structure of data 2.) Query Pattern 3.) Amount of scale

Caching – Redis

Whenever u want to store Images, Video – Amazon S3

Blob – Amazon S3+CDN

Text Search Engine – Elastic Search, Solar both of them built on top of something called Apache Lucene

Fuzzy Searching – Airport ,Airprot – Level 2

Metric Data – TimeSeries : OpenTSDB

Analytics of all the transaction – Hadoop

Ever Increasing data finite queries – Columnar - Cassandra (easy to deploy) , HBase – Uber drivers continue sending location

Data Types Queries – TShirt – Size, Color, --- DocumentDB - MangoDB, CouchDB

Adding And Removing Server - Consistent Hashing Comes into picture

H(r1) -> m1%n

H (10) 🡪 3/4 = 3

RDBMS Scale MySQL – indexing, caching, replication

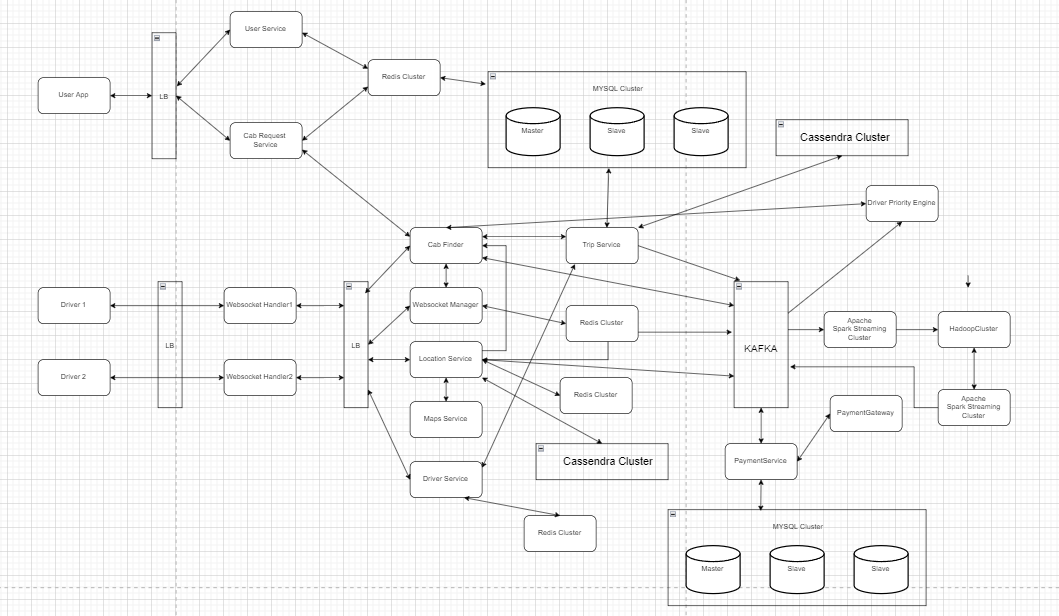
**1.) System Design UBER And OLA:**

**Functional Requirements:**

* See Cabs
* ETA & Approx. Price
* Book a Cab
* Location Tracking

**Non-Functional Requirements:**

* Platform should be global
* Low Latency
* High Available
* High Consistence
* Scale 100M 1 Month – 14M Rides a day



**2.) System Design URL Shortener:**

**Functional Requirements:**

* Get Short URL
* Redirect to long URL

**Non-Functional Requirements:**

* Very High Available (Consistency not on priority)
* Very Low Latency

**APIS:**

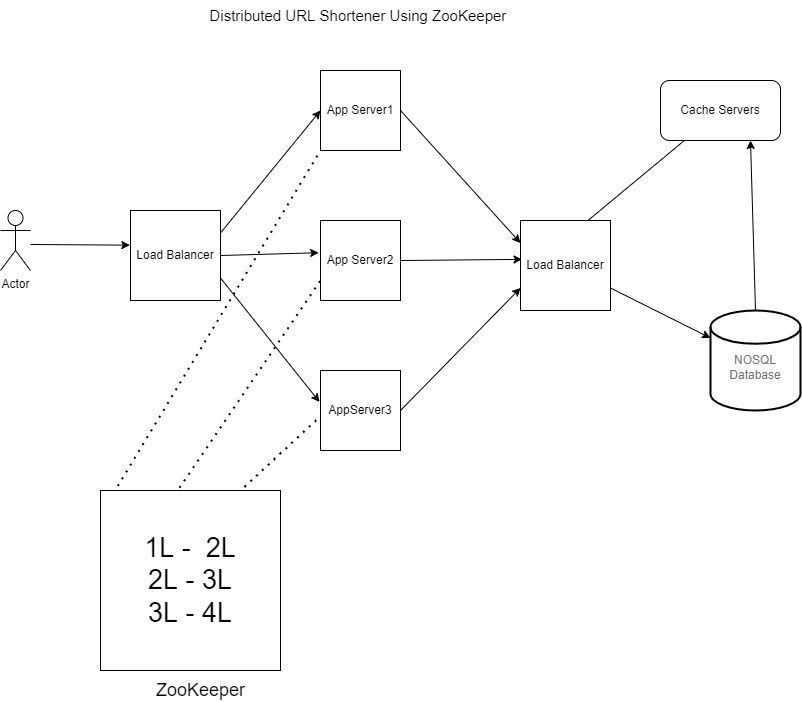
* createTinyUrl(Long url)
* getLongUrl(Short Url)

**Size of Database Required:**

* Each URL mapping size – 2kb
* Number of users per month 10 M – 10M\*2KB = 20GB per month – 1TB 5 years

BASE 62 – A-Z, a-z, 0-9

MD5 Hash



**3.) Designing an API Rate Limiter:**

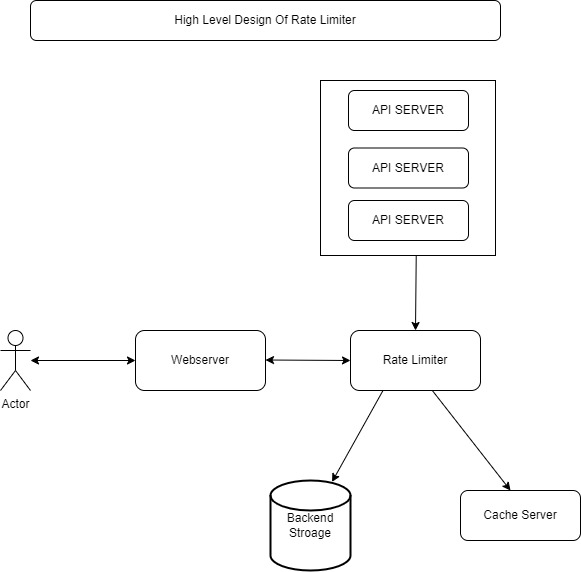
**How to implement rate limiter: Most common algorithms are:**

* Token Bucket – 5 ball per min for unique user fetchToken, updateTheToken

Will cause race around condition on distributed environment if 2 request are coming from

Different app severs both are trying to update the time

* Leaky Bucket (FIFO)– Put tasks in queue if queue is full return too many requests return after sometime – (Effective and easy to implement)
* Fixed Window Counter– similar to token bucket – time range is fixed for ex – 5:00 to 05:59 - 5 request – Problems (Clustering around boundary, race condition in case of distributed systems)
* Sliding Window: Better the fixed window: it will check 5 request is previous window if it’s there then it will not accept any new request;



**4.) System Design Instagram:**

**Functional Requirements:**

* Upload Photo/Create Post
* View Photo/Post
* Search Users
* User Feed Display
* Comment and like a post

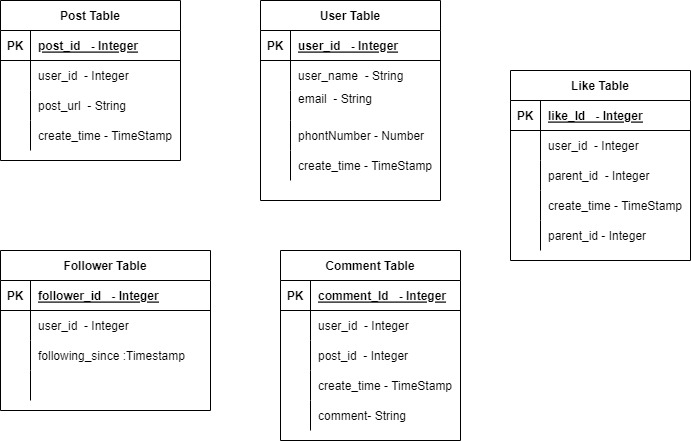
**Non-Functional Requirements:**

* High Available (Consistency not on priority)
* Reliable System hence partition tolerant - photo should not be lost
* Low latency while user feed service
* Read Heavy

Capacity Estimation:

* 100M users - 1M active users - Each Image or post size 1MB
* 1M\*1MB - 975GB perday
* Per year - 975\*365 = 350TB
* sharding will be on post\_id

**Database Schema:**



User Service: getUserDetails, postUserDetails, updateUserDetails

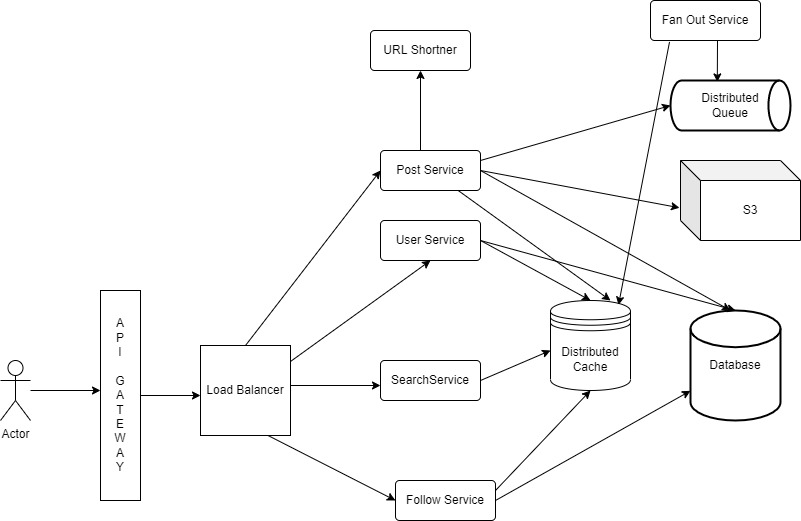
Follow Service: followUser, getListOfUserFollowedByAUser

Post Service: uploadPost, getPost, getUserFeed

Comment And Like Service: addCommentToThePost, getCommentForThePost, addLikeToThePost, getNumberOfLikesForThePost, getListOfUsersWhoLikeThePost

**Promise Based Cache:**

* Cache miss happen for same object for multiple request
* Rather than all request going to db and fetching result and setting in DB
* 1st request creates a promise object in cache
* Subsequent request will cache miss will check if a promise is present in a cache
* If promise is present, all subsequent request will wait for the promise to get completed



**Database Types:**

RDBMS

NOSQL DB – Document – Columnar – Key Value – Graph

**5.) System Design Facebook Messenger/WhatsApp/Share Chat:**

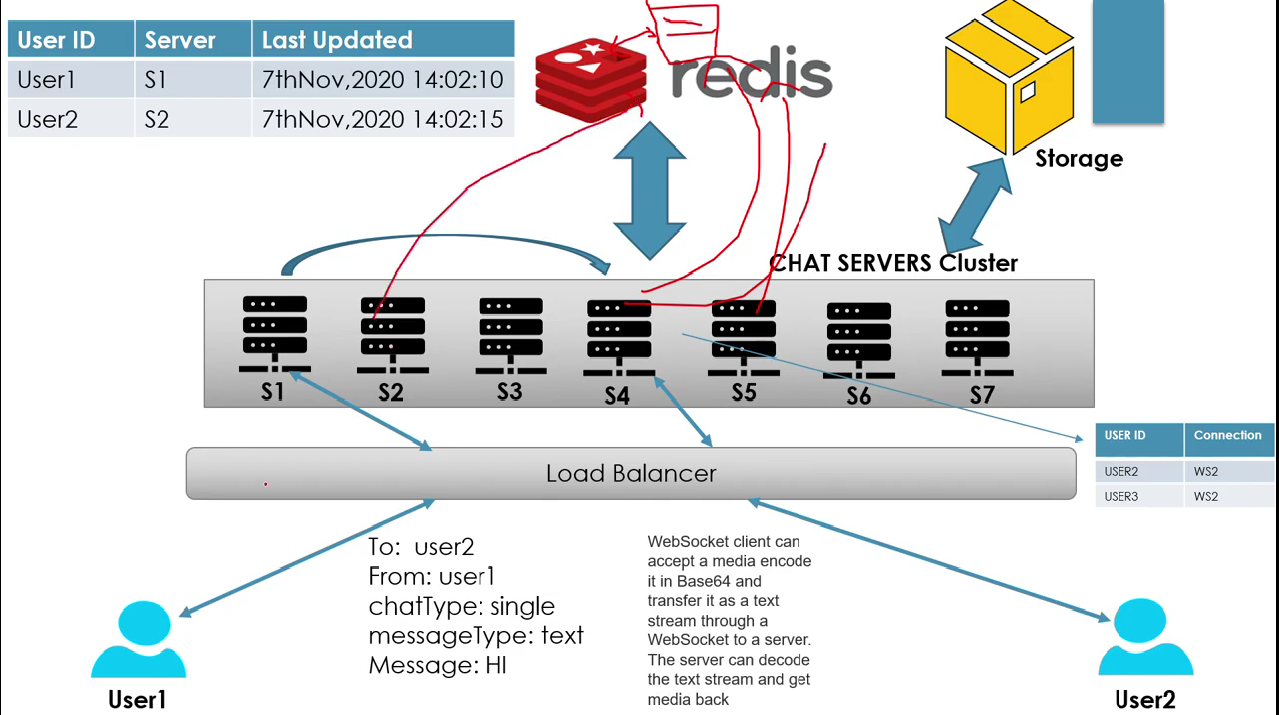
**Functional Requirements:**

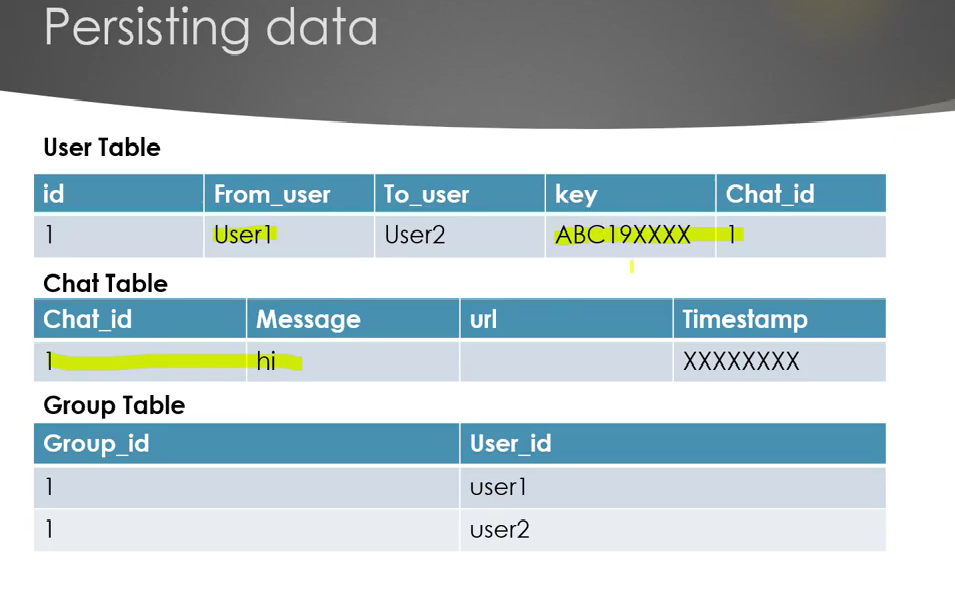
* One to One Message
* Sent/Delivered/Seen
* Last Online
* Media Message (Picture/Audio/Video)
* Persistent chat required or not?)

**Non-Functional Requirements:**

* Real Time Experience
* Consistent
* Scalable
* High Availability







**6.) System Twitter:**

Twitter – Tweet, TimeLine (Home, User, Search), Trends

Write: 600 Tweets/Sec

Read: 600000 Tweets/Sec

Read Heavy, Eventual Consistency, Storage

User – 1 to Many – Tweets

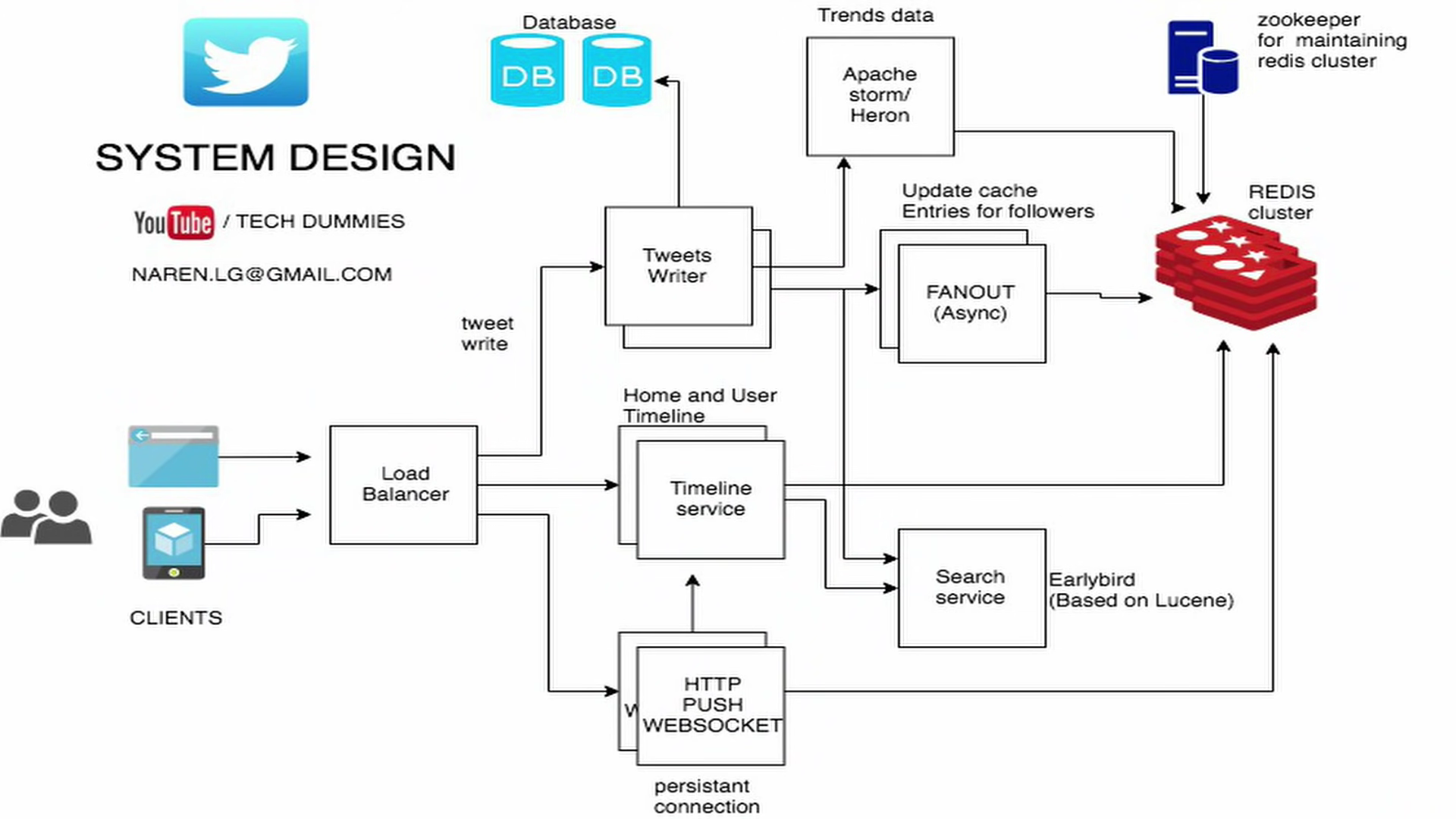
User – 1 to Many – Followers

Redis - <User-Id> -tweets: [1,2,3,4,5];

Fanout service – It’s responsible to update the tweet into all the followers’ timelines home timeline or user time line

Http Push Socket – Handle millions of connections for mobile devices

Database – MySQL , Cassandra – For some kind of analytics



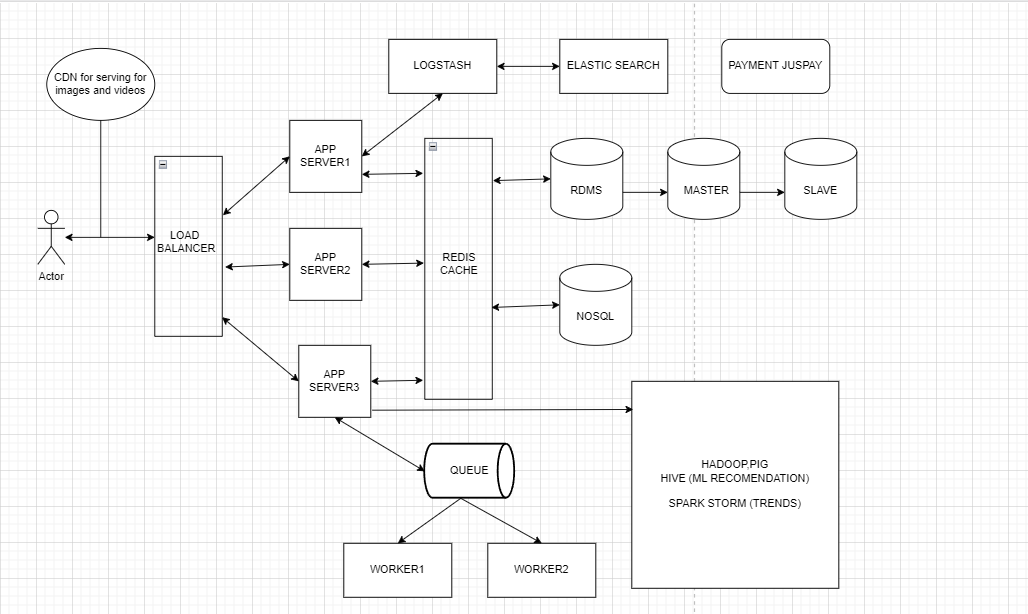
**7.) Design Book My Show:**

**Functional Requirements:**

* One to One Message
* Sent/Delivered/Seen
* Last Online
* Media Message (Picture/Audio/Video)
* Persistent chat required or not?)

**Non-Functional Requirements:**

* Real Time Experience
* Consistent
* Scalable
* High Availability



**8.) Designing Facebook’s Newsfeed**

**Functional requirements:**

1. Newsfeed will be generated based on the posts from the people, pages, and groups that a user follows.
2. A user may have many friends and follow a large number of pages/groups.
3. Feeds may contain images, videos, or just text.
4. Our service should support appending new posts as they arrive to the newsfeed for all active users.

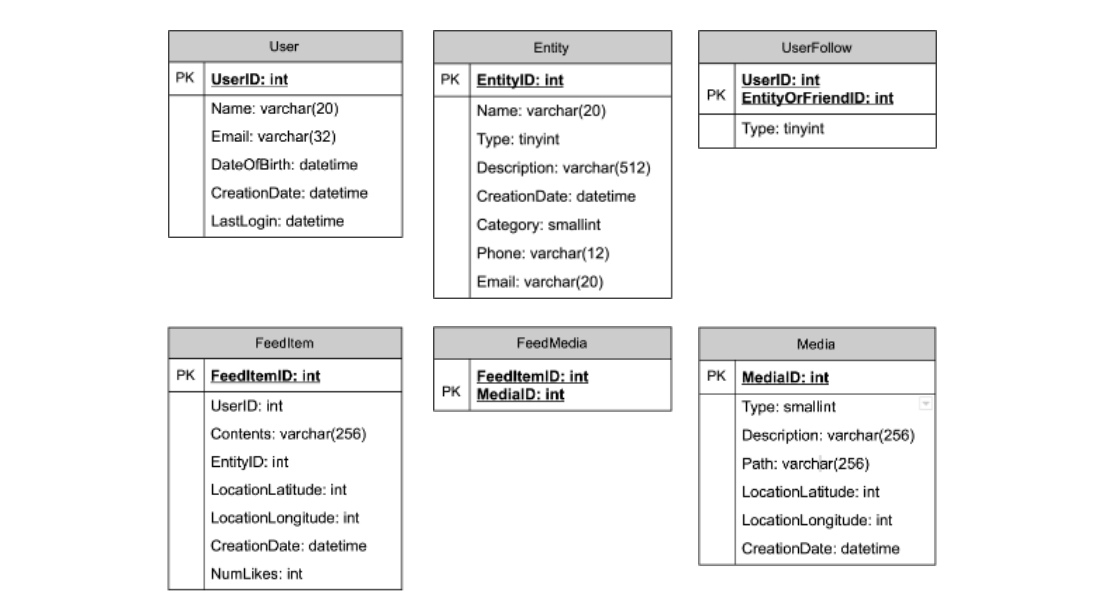
**Non-functional requirements:**

1. Our system should be able to generate any user’s newsfeed in real-time - maximum latency seen by the end user would be 2s.
2. A post shouldn’t take more than 5s to make it to a user’s feed assuming a new newsfeed request comes in.

System APIs:

getUserFeed(api\_dev\_key, user\_id, since\_id, count, max\_id, exclude\_replies)

Database Design:



**9.) Designing Youtube or Netflix:**

**Functional Requirements:**

* Users should be able to upload videos.
* Users should be able to share and view videos.
* Users should be able to perform searches based on video titles.
* Our services should be able to record stats of videos, e.g., likes/dislikes, total number of views, etc.
* Users should be able to add and view comments on videos.

**Non-Functional Requirements:**

* The system should be highly reliable, any video uploaded should not be lost.
* The system should be highly available. Consistency can take a hit (in the interest of availability); if a user doesn’t see a video for a while, it should be fine.
* Users should have a real-time experience while watching videos and should not feel any lag.

**System APIs:**

uploadVideo(api\_dev\_key, video\_title, video\_description, tags[], category\_id, default\_language, recording\_details, video\_contents)

searchVideo(api\_dev\_key, search\_query, user\_location, maximum\_videos\_to\_return, page\_token)

streamVideo(api\_dev\_key, video\_id, offset, codec, resolution)

