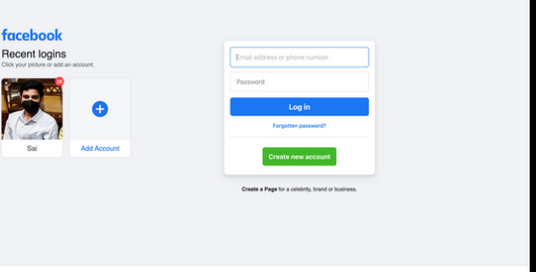
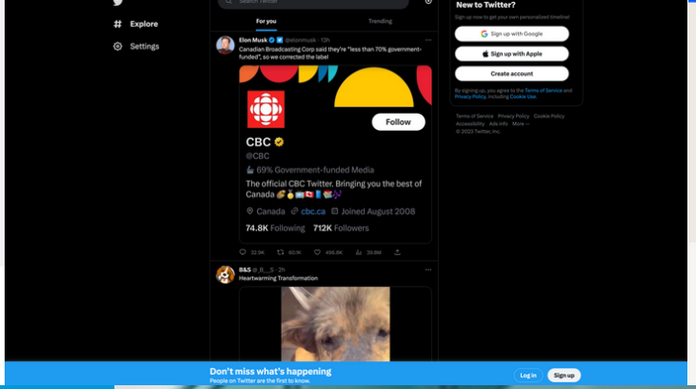
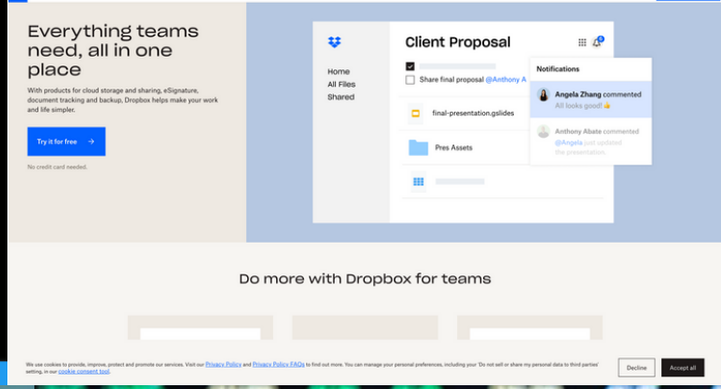
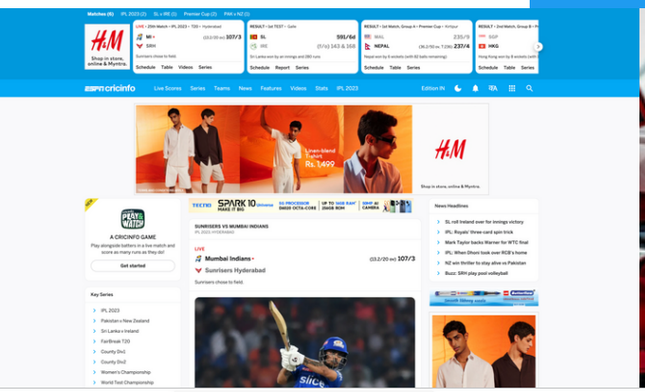
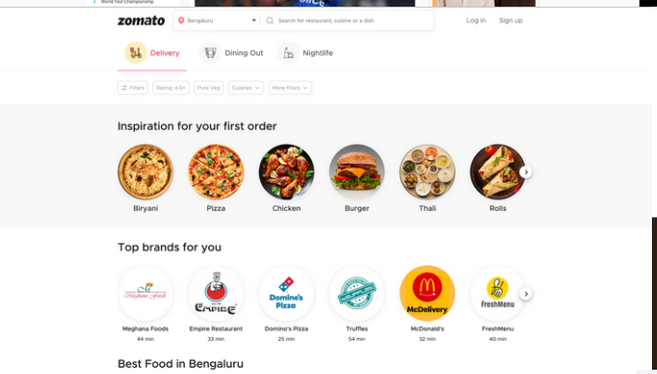
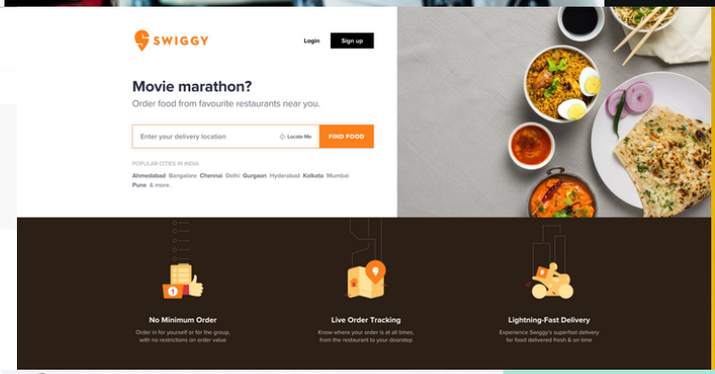


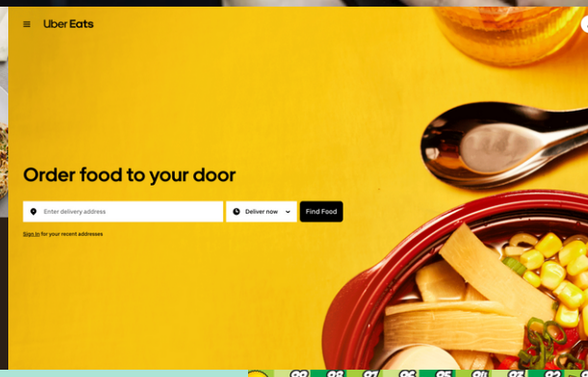


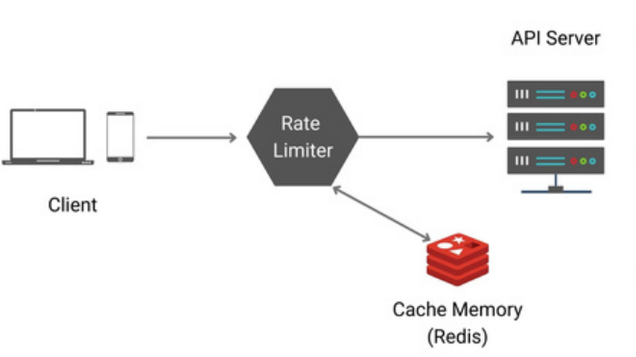


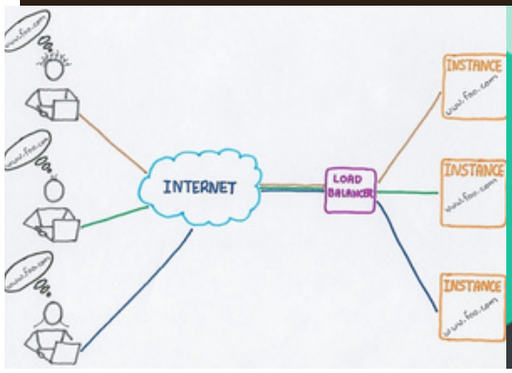












Design Facebook (A Social Network), Twitter, Dropbox, Cricinfo, Car Rental System, Online Stock Brokerage System, Architecture and Design Principles Behind the Swiggy's Delivery Partners app, Rate Limiter API, Splitwise, Snake & Ladder, Food delivery app - Zomato, Swiggy, UberEats, Load Balancing Algorithms, Eventual, Sequential Consistency, and Durability.

Design Facebook (A Social Network):

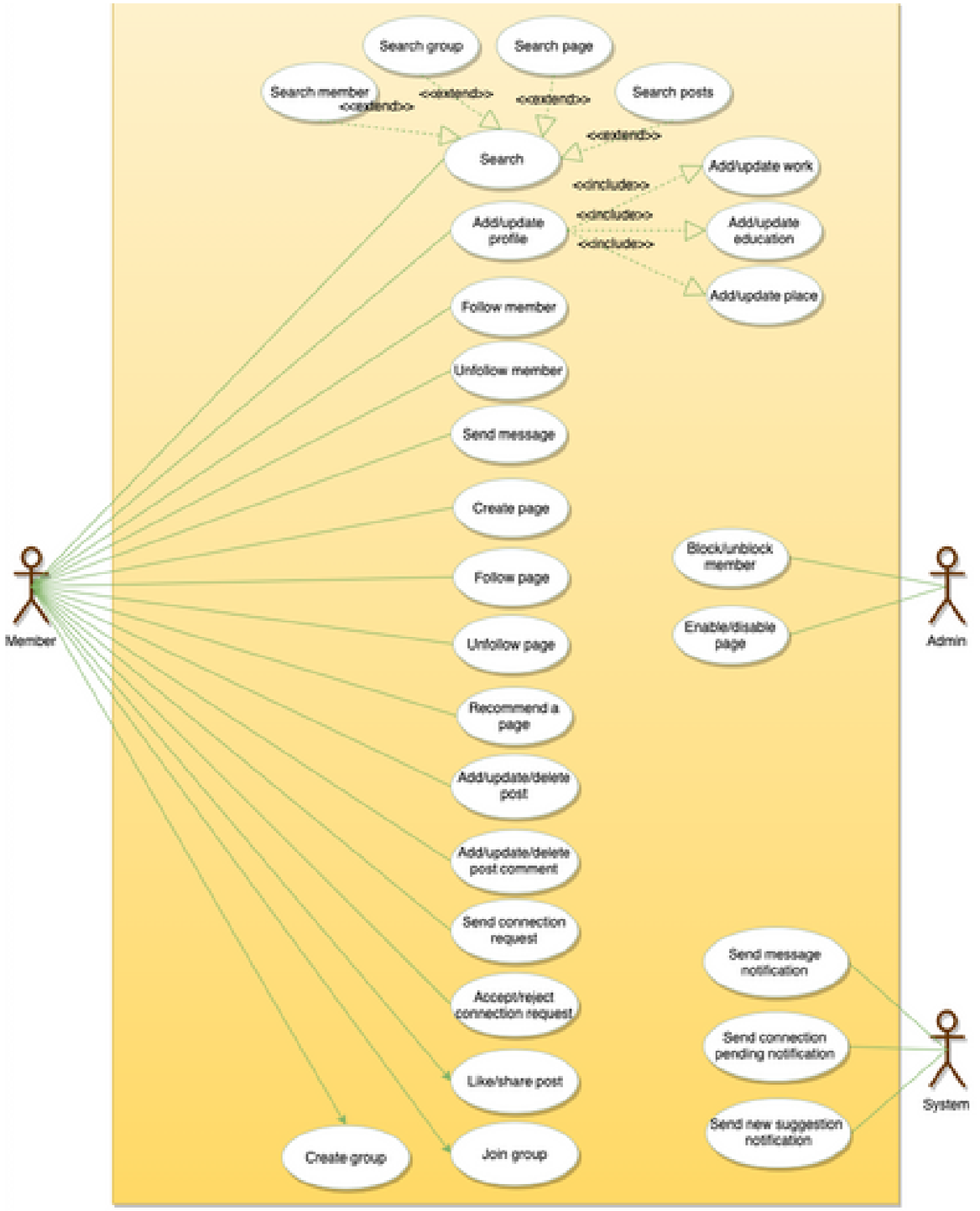
Description:

Facebook is an online social networking service where users can connect with other users to post and read messages. Users access Facebook through their website interface or mobile apps.

Requirements:

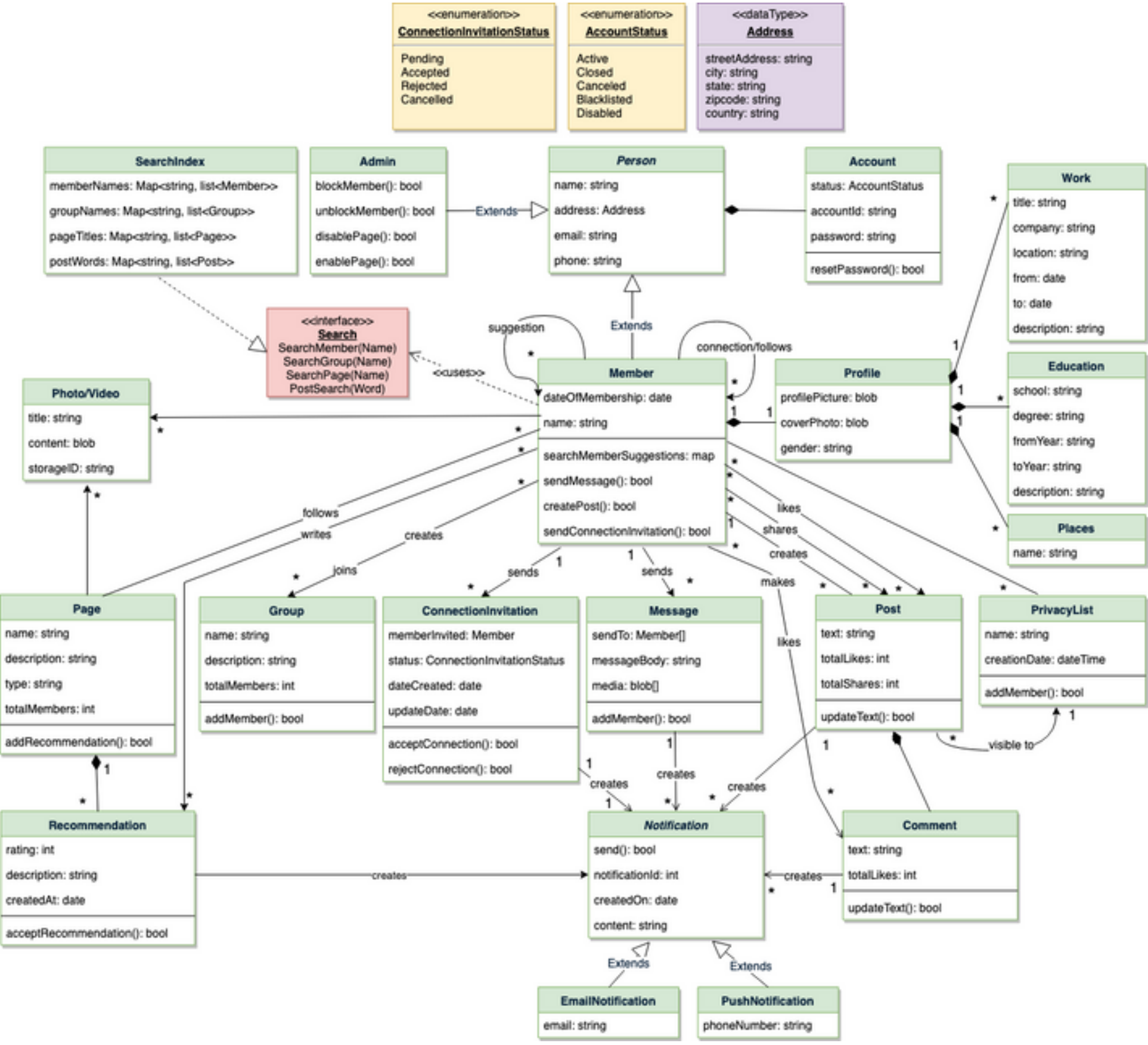
- Each member should be able to add information about their basic profile, work experience, education, etc.
- Any user of our system should be able to search other members, groups or pages by their name.
- Members should be able to send and accept/reject friend requests from other members.
- Members should be able to follow other members without becoming their friend.
- Members should be able to create groups and pages, as well as join already created groups, and follow pages.
- Members should be able to create new posts to share with their friends.
- Members should be able to add comments to posts, as well as like or share a post or comment.
- Members should be able to create privacy lists containing their friends. Members can link any post with a privacy list to make the post visible only to the members of that list.
- Any member should be able to send messages to other members.

- Any member should be able to add a recommendation for any page.
- The system should send a notification to a member whenever there is a new message or friend request or comment on their post.
- Members should be able to search through posts for a word.



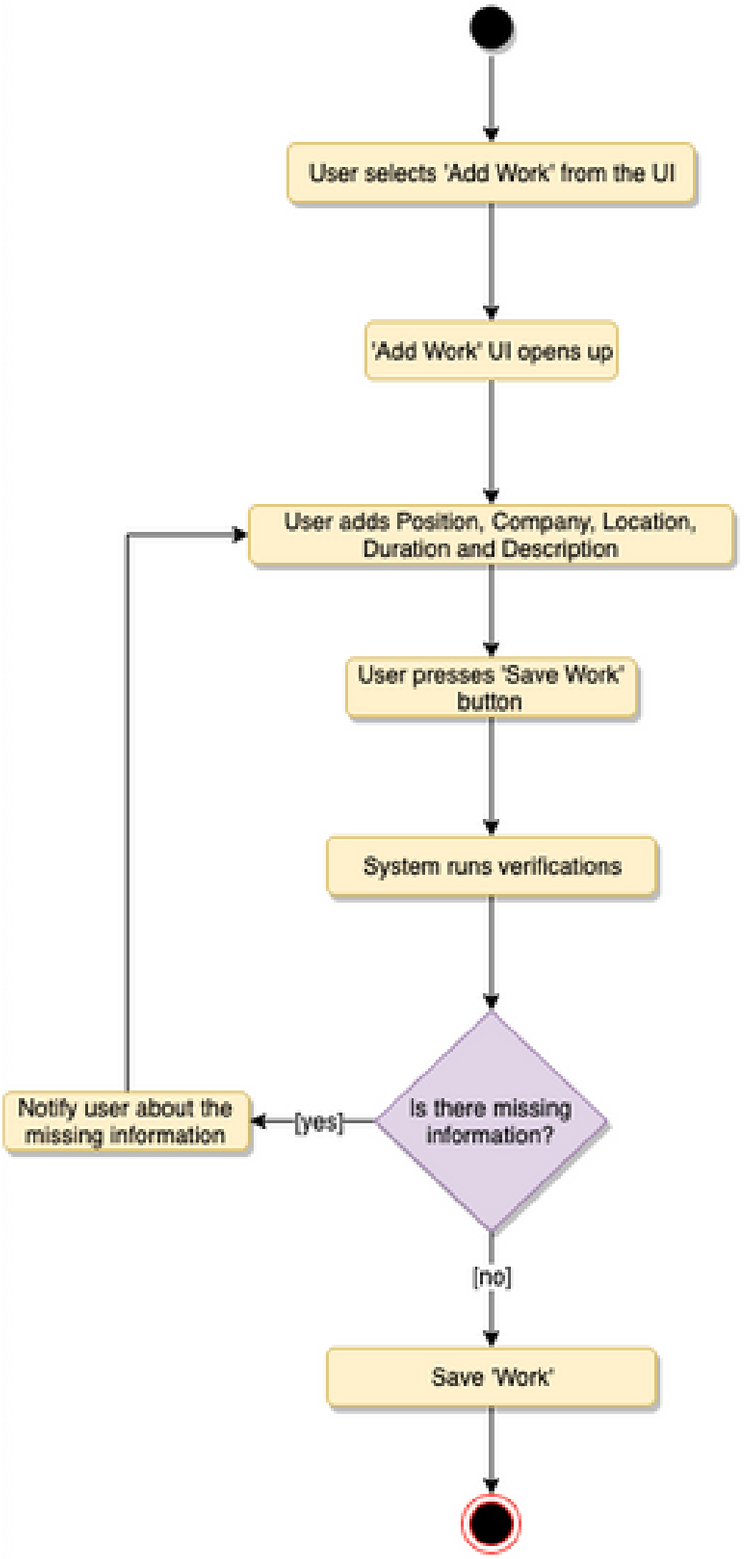
Class Diagram:

- Member: This will be the main component of our system. Each member will have a profile which includes their Work Experiences, Education, etc. Members will be connected to other members and they can follow other members and pages. Members will also have suggestions to send friend requests to other members.
- Search: Our system will support searching for other members, groups and pages by their names, and through posts for any word.
- Message: Members can send messages to other members with text, photos, and videos.
- Post: Members can create posts containing text and media, as well as like and share a post.
- Comment: Members can add comments to posts as well as like any comment.
- Group: Members can create and join groups.
- PrivacyList: Members can create privacy lists containing their friends. Members can link any post with a privacy list, to make the post visible only to the members of that list.
- Page: Members can create pages that other members can follow, and share messages there.
- Notification: This class will take care of sending notifications to members. The system will be able to send a push notification or an email.

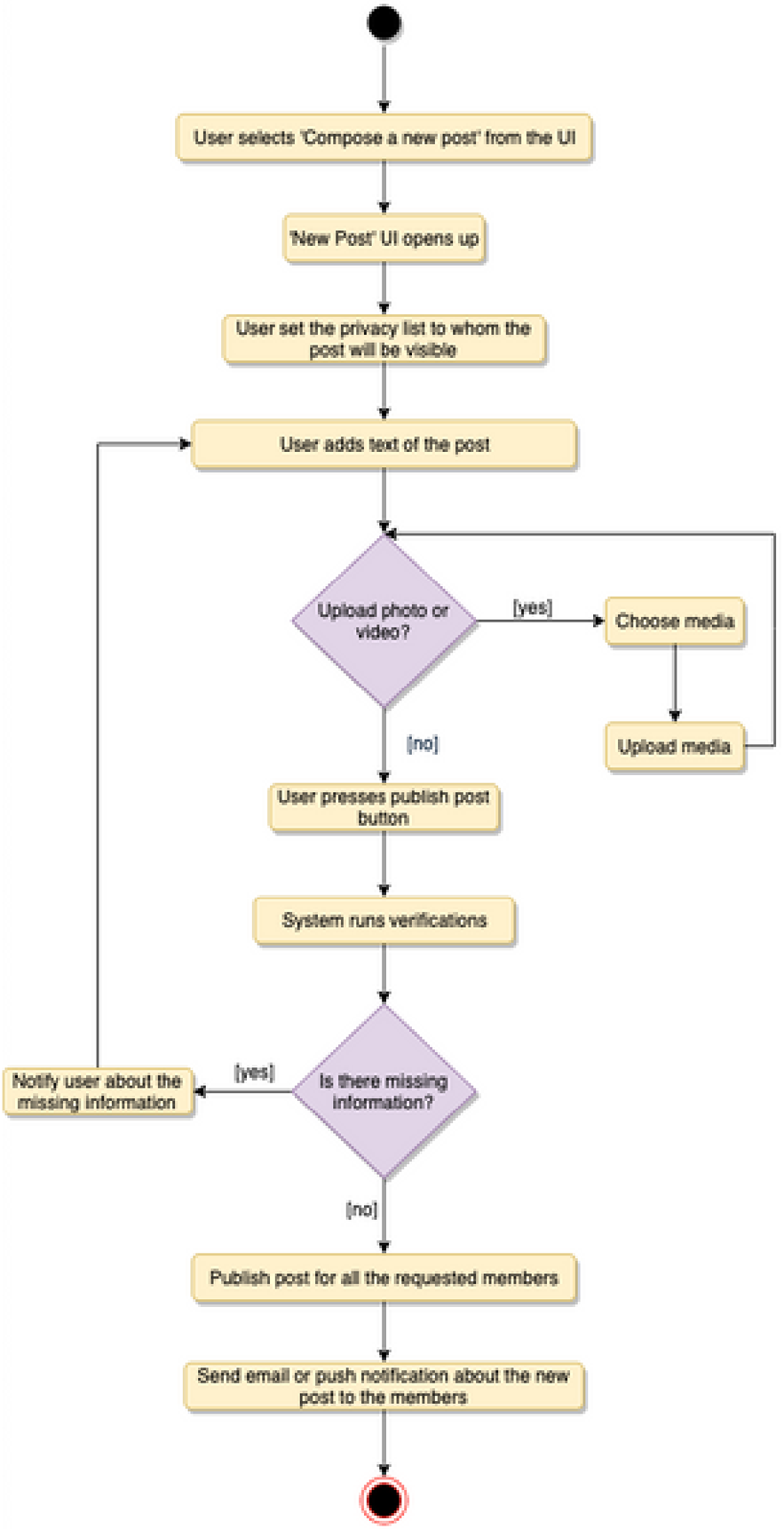

<<enumeration>>
ConnectionInvitationStatus
Pending
Accepted
Rejected
Cancelled
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
Disabled
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
SearchIndex
memberNames: Map<string, list<Member>>
groupNames: Map<string, list<Group>>
pageTitles: Map<string, list<Page>>
postWords: Map<string, list<Post>>
Admin
blockMember(): bool
unblockMember(): bool
disablePage(): bool
enablePage(): bool
Extends
Person
name: string
address: Address
email: string
phone: string
Account
status: AccountStatus
accountId: string
password: string
resetPassword(): bool
Work
title: string
company: string
location: string
from: date
to: date
description: string
<<interface>>
Search
SearchMember(Name)
SearchGroup(Name)
SearchPage(Name)
PostSearch(Word)
<<uses>>
suggestion
Extends
connection/follows
Member
dateOfMembership: date
name: string
searchMemberSuggestions: map
sendMessage(): bool
createPost(): bool
sendConnectionInvitation(): bool
Profile
profilePicture: blob
coverPhoto: blob
gender: string
Education
school: string
degree: string
fromYear: string
toYear: string
description: string
Places
name: string
Photo/Video
title: string
content: blob
storageID: string
follows
writes
creates
likes
shares
creates
joins
sends
sends
makes
likes
Page
name: string
description: string
type: string
totalMembers: int
addRecommendation(): bool
Group
name: string
description: string
totalMembers: int
addMember(): bool
ConnectionInvitation
memberInvited: Member
status: ConnectionInvitationStatus
dateCreated: date
updateDate: date
acceptConnection(): bool
rejectConnection(): bool
Message
sendTo: Member[]
messageBody: string
media: blob[]
addMember(): bool
Post
text: string
totalLikes: int
totalShares: int
updateText(): bool
PrivacyList
name: string
creationDate: dateTime
addMember(): bool
visible to
creates
creates
creates
Recommendation
rating: int
description: string
createdAt: date
acceptRecommendation(): bool
creates
Notification
send(): bool
notificationId: int
createdOn: date
content: string
creates
Comment
text: string
totalLikes: int
updateText(): bool
Extends
Extends
EmailNotification
email: string
PushNotification
phoneNumber: string

Activity Diagrams:

- Add work experience to profile: Any Facebook member can perform this activity. Here are the steps to add work experience to a member's profile:



- Create a new post: Any Member can perform this activity. Here are the steps for creating a post:



Code:

AccountStatus.java

```java
package com.sai.facebook.enums;

public enum AccountStatus {
    ACTIVE,
    CLOSED,
    CANCELED,
    BLACKLISTED,
    DISABLED
}
```

ConnectionInvitationStatus.java

```java
package com.sai.facebook.enums;

public enum ConnectionInvitationStatus {
    PENDING,
    ACCEPTED,
    REJECTED,
    CANCELED
}
```

Search.java

```java
package com.sai.facebook.catalog;

import com.sai.facebook.feed.Group;
import com.sai.facebook.feed.Page;
import com.sai.facebook.feed.Post;
import com.sai.facebook.user.Member;

import java.util.List;

1 usage  1 implementation
public interface Search {
    1 implementation
    List<Member> searchMember(String name);
    1 implementation
    Group searchGroup(String name);
    1 implementation
    Page searchPage(String name);
    1 implementation
    Post searchPost(String word);
}
```

SearchIndex.java

```java
public class SearchIndex implements Search {

    4 usages
    private Map<String, List<Member>> member_names;
    1 usage
    private Map<String, Group> group_names;
    1 usage
    private Map<String, Page> page_titles;
    1 usage
    private Map<String, Post> posts;

    public void addMember(Member member) {
        if (member_names.containsKey(member.getName())) {
            member_names.get(member.getName()).add(member);
        } else {
            member_names.put(member.getName(), Arrays.asList(member));
        }
    }

    public void add_group(Group group) {

    }

    public void add_page(Page page) {

    }

    public void add_post(Post post) {

    }

    @Override
    public List<Member> searchMember(String name) { return member_names.get(name); }

    @Override
    public Group searchGroup(String name) { return group_names.get(name); }

    @Override
    public Page searchPage(String name) { return page_titles.get(name); }

    @Override
    public Post searchPost(String word) { return posts.get(word); }
}
```

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Address {
    private String streetAddress;
    private String city;
    private String state;
    private String zipCode;
    private String country;
}
```

```java
@Data
public class Group {
    1 usage
    private BigInteger groupId;
    1 usage
    private String name;
    1 usage
    private String description;
    1 usage
    private int totalMembers;
    1 usage
    private List<String> members;

    public Group(BigInteger id, String name, String description, int total_members) {
        this.groupId = id;
        this.name = name;
        this.description = description;
        this.totalMembers = total_members;
        this.members = new ArrayList<>();
    }

    public void addMember(String member) {
    }

    public void updateDescription(String description) {
    }
}
```

```java
public class Comment {
    1 usage
    private BigInteger commentId;
    1 usage
    private String text;
    1 usage
    private BigInteger totalLikes;
    1 usage
    private String owner;

    public Comment(BigInteger id, String text, BigInteger total_likes, String owner) {
        this.commentId = id;
        this.text = text;
        this.totalLikes = total_likes;
        this.owner = owner;
    }
}
public class Message {
    1 usage
    private BigInteger messageId;
    1 usage
    private String sentTo;
    1 usage
    private String messageBody;
    1 usage
    private String media;

    public Message(BigInteger id, String sent_to, String body, String media) {
        this.messageId = id;
        this.sentTo = sent_to;
        this.messageBody = body;
        this.media = media;
    }

    public void addMember(String member) {
    }
}
```

```java
@Data
@NoArgsConstructor
public class Page {
    1 usage
    private BigInteger pageId;
    1 usage
    private String name;
    1 usage
    private String description;
    1 usage
    private String type;
    1 usage
    private int totalMembers;
    1 usage
    private List<String> recommendation;

    public Page(BigInteger id, String name, String description, String type, int total_members) {
        this.pageId = id;
        this.name = name;
        this.description = description;
        this.type = type;
        this.totalMembers = total_members;
        this.recommendation = new ArrayList<>();
    }
}

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Post {
    private BigInteger id;
    private String text;
    private int total_likes;
    private int total_shares;
    private String owner;
}
```

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Recommendation {
    private int id;
    private String rating;
    private String description;
    private Date createdAt;
}
```

```java
3 usages
@NoArgsConstructor
public class Profile {
    1 usage
    private String profilePicture;
    1 usage
    private String coverPhoto;
    1 usage
    private String gender;
    1 usage
    private List<String> workExperiences;
    1 usage
    private List<String> educations;
    1 usage
    private List<String> places;
    1 usage
    private List<String> stats;

    public Profile(String profilePicture, String coverPhoto, String gender) {
        this.profilePicture = profilePicture;
        this.coverPhoto = coverPhoto;
        this.gender = gender;
        this.workExperiences = new ArrayList<>();
        this.educations = new ArrayList<>();
        this.places = new ArrayList<>();
        this.stats = new ArrayList<>();
    }

    public void addWorkExperience(String work) {
        // TODO
    }

    public void addEducation(String education) {
        // TODO
    }

    public void addPlace(String place) {
        // TODO
    }
}
```

```java

public class Work {
    1 usage
    private String title;
    1 usage
    private String company;
    1 usage
    private String location;
    1 usage
    private String dateFrom;
    1 usage
    private String dateTo;
    1 usage
    private String description;

    public Work(String title, String company, String location, String dateFrom, String dateTo, String description) {
        this.title = title;
        this.company = company;
        this.location = location;
        this.dateFrom = dateFrom;
        this.dateTo = dateTo;
        this.description = description;
    }
}

```

Account.java

```java
package com.sai.facebook.user;

import com.sai.facebook.enums.AccountStatus;

import java.math.BigInteger;

public class Account {
    private BigInteger id;
    private String password;
    private AccountStatus status;

    public Account(BigInteger id, String password, AccountStatus status) {
        this.id = id;
        this.password = password;
        this.status = status;
    }

    public void resetPassword() {
    }
}
```

Admin.java

```java
package com.sai.facebook.user;


public class Admin extends Person {
    public Admin(String name, String address, String email, String phone, String account) {
        super(name, address, email, phone, account);
    }

    public void blockUser(Person customer) {
    }

    public void unblockUser(Person customer) {
    }

    public void enablePage(String page) {
    }

    public void disablePage(String page) {
    }

    public void doSomething(String something) {
    }
}
```

ConnectionInvitation.java

```java
package com.sai.facebook.user;

import com.sai.facebook.enums.ConnectionInvitationStatus;
import lombok.Data;

import java.util.Date;

@Data
public class ConnectionInvitation {

    private Member memberInvited;
    private String name;
    private ConnectionInvitationStatus status;
    private Date dateCreated;
    private Date dateUpdated;


    public ConnectionInvitation(Member memberInvited, String name, ConnectionInvitationStatus status) {
        this.memberInvited = memberInvited;
        this.name = name;
        this.status = status;
        this.dateCreated = new Date();
        this.dateUpdated = new Date();
    }

    public void acceptConnection() {
    }
    public void rejectConnection() {
    }
}
```

Member.java

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Member extends Person {
    private BigInteger memberId;
    private Date dateOfMembership;
    private String name;
    private Profile profile;
    private List<Member> memberFollows;
    private List<Member> memberConnections;
    private List<Page> pageFollows;
    private List<Member> memberSuggestions;
    private List<ConnectionInvitation> connectionInvitations;
    private List<Group> groupFollows;

    public Member(BigInteger id, Date dateOfMembership, String name) {
        this.memberId = id;
        this.dateOfMembership = dateOfMembership;
        this.name = name;
        this.profile = new Profile();
        this.memberFollows = new ArrayList<>();
        this.memberConnections = new ArrayList<>();
        this.pageFollows = new ArrayList<>();
        this.memberSuggestions = new ArrayList<>();
        this.connectionInvitations = new ArrayList<>();
        this.groupFollows = new ArrayList<>();
    }

    public void sendMessage(Message message) {
        // implementation
    }

    public void createPost(Post post) {
        // implementation
    }

    public void sendConnectionInvitation(ConnectionInvitation invitation) {
        // implementation
    }

    public void searchMemberSuggestions() {
        // implementation
    }
}
```

Person.java

```java
package com.sai.facebook.user;

import lombok.NoArgsConstructor;

4 usages  2 inheritors
@NoArgsConstructor
public abstract class Person {
    1 usage
    private String name;
    1 usage
    private String address;
    1 usage
    private String email;
    1 usage
    private String phone;
    1 usage
    private String account;

    1 usage
    public Person(String name, String address, String email, String phone, String account) {
        this.name = name;
        this.address = address;
        this.email = email;
        this.phone = phone;
        this.account = account;
    }
}
```

Design Cricinfo:

Description:

Cricinfo is a sports news website exclusively for the game of cricket. The site features live coverage of cricket matches containing ball-by-ball commentary and a database for all the historic matches. The site also provides news and articles about cricket.

Requirements:

- The system should keep track of all cricket-playing teams and their matches.
- The system should show live ball-by-ball commentary of cricket matches.
- All international cricket rules should be followed.
- Any team playing a tournament will announce a squad (a set of players) for the tournament.
- For each match, both teams will announce their playing-eleven from the tournament squad.
- The system should be able to record stats about players, matches, and tournaments.
- The system should be able to answer global stats queries like, “Who is the highest wicket taker of all time?”, “Who has scored maximum numbers of 100s in test matches?”, etc.
- The system should keep track of all ODI, Test and T20 matches.

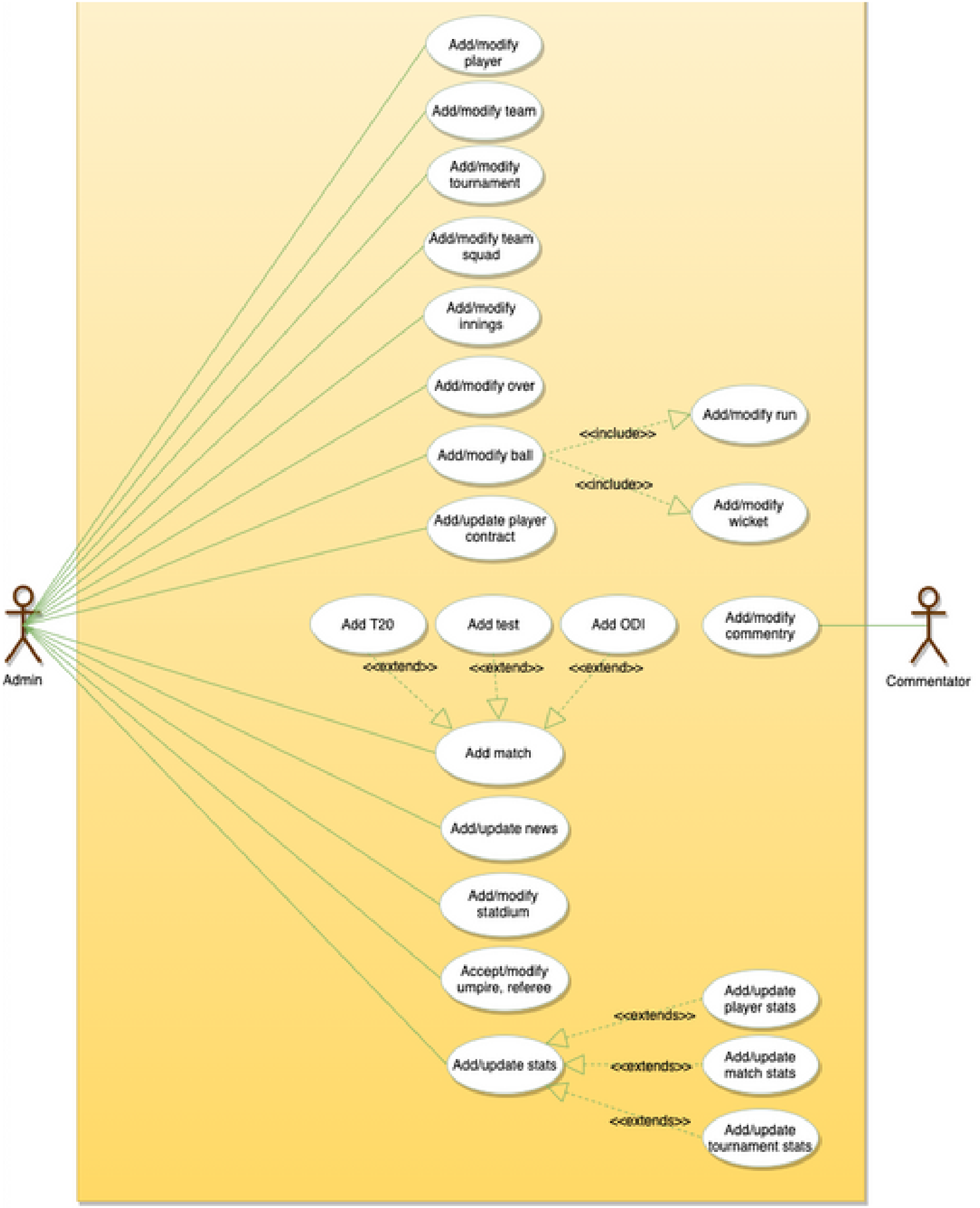
Use Case Diagram:

Main Actors:

- Admin: An Admin will be able to add/modify players, teams, tournaments, and matches, and will also record ball-by-ball details of each match.
- Commentator: Commentators will be responsible for adding ball-by-ball commentary for matches.

Top Use Cases:

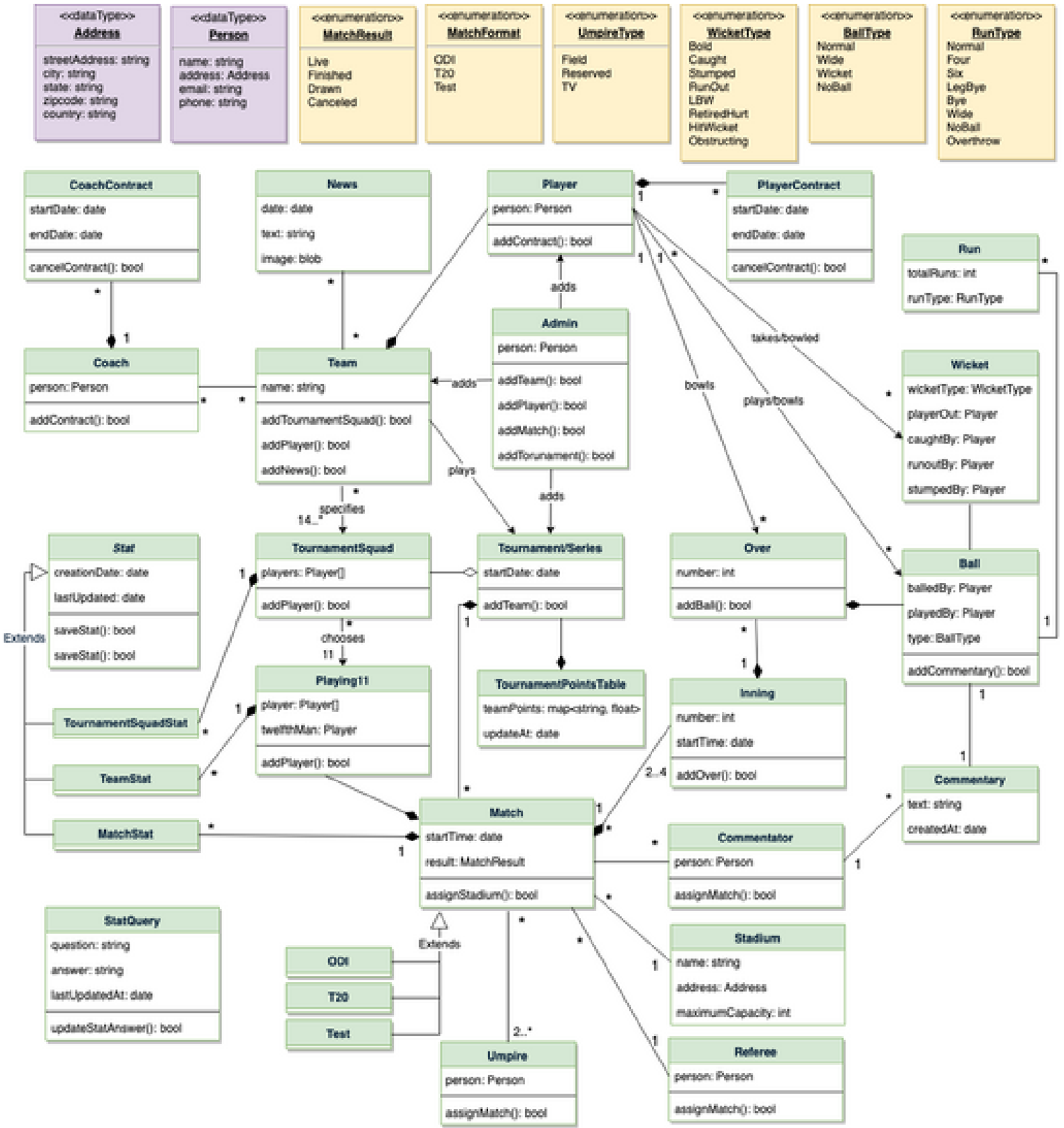
- Add/modify teams and players: An Admin will add players to teams and keeps up-to-date information about them in the system.
- Add tournaments and matches: Admins will add tournaments and matches in the system.
- Add ball: Admins will record ball-by-ball details of a match.
- Add stadium, umpire, and referee: The system will keep track of stadiums as well as of the umpires and referees managing the matches.
- Add/update stats: Admins will add stats about matches and tournaments. The system will generate certain stats.
- Add commentary: Add ball-by-ball commentary of matches.


Add/modify player
Add/modify team
Add/modify tournament
Add/modify team squad
Add/modify innings
Add/modify over
Add/modify ball
Add/update player contract
<<include>>
Add/modify run
<<include>>
Add/modify wicket
Admin
Add T20
Add test
Add ODI
Add/modify commentry
Commentator
<<extend>>
<<extend>>
<<extend>>
Add match
Add/update news
Add/modify statdium
Accept/modify umpire, referee
Add/update stats
<<extends>>
Add/update player stats
<<extends>>
Add/update match stats
<<extends>>
Add/update tournament stats

Class Diagram:

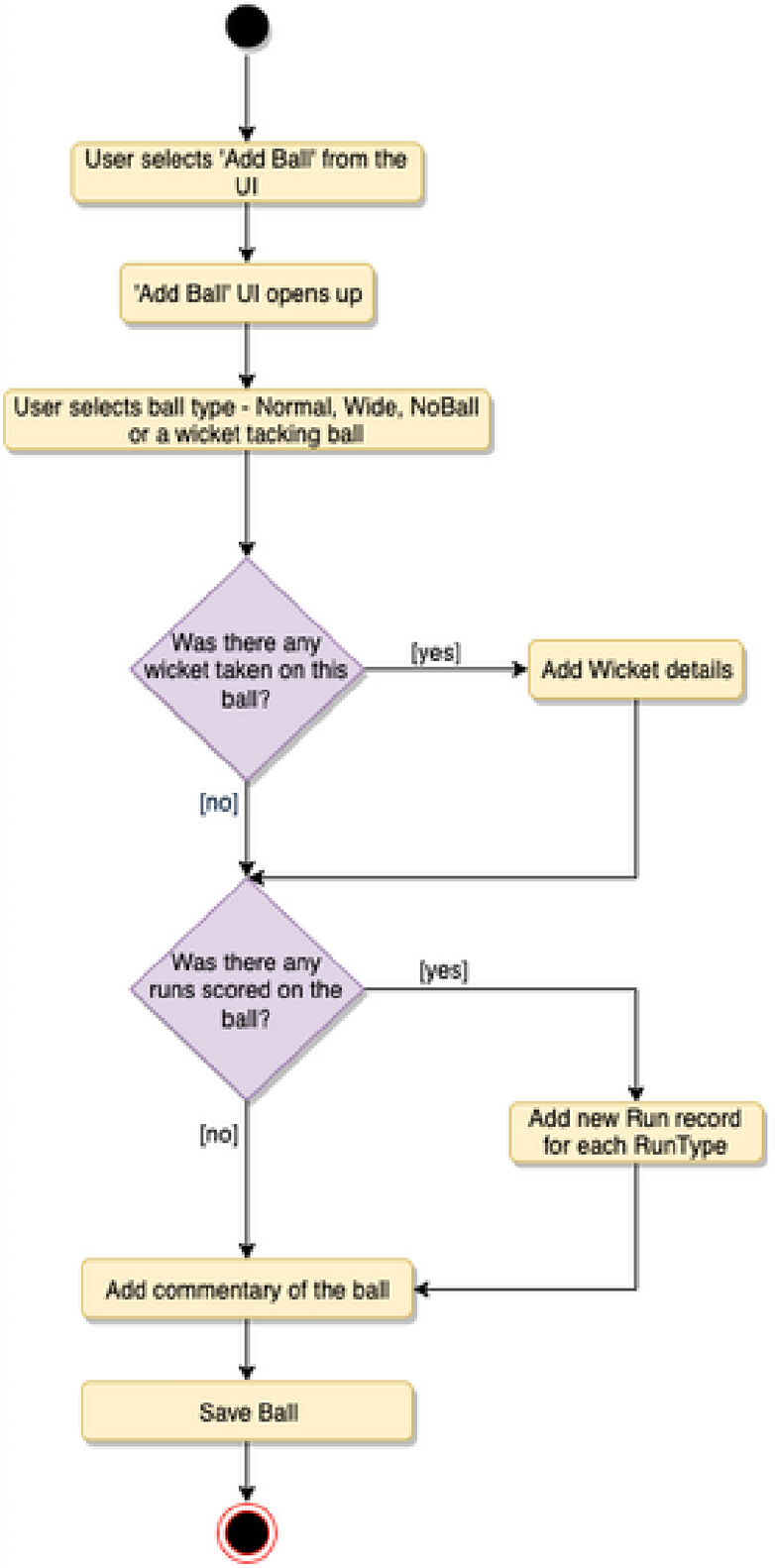
- Player: Keeps a record of a cricket player, their basic profile and contracts.
- Team: This class manages cricket teams.
- Tournament: Manages cricket tournaments and keeps track of the points table for all playing teams.
- TournamentSquad: Each team playing a tournament will announce a set of players who will be playing the tournament. TournamentSquad will encapsulate that.
- Playing11: Each team playing a match will select 11 players from their announced tournaments squad.
- Match: Encapsulates all information of a cricket match. Our system will support three match types: 1) ODI, 2) T20, and 3) Test
- Innings: Records all innings of a match.
- Over: Records details about an Over.
- Ball: Records every detail of a ball, such as the number of runs scored, if it was a wicket-taking ball, etc.
- Run: Records the number and type of runs scored on a ball. The different run types are: Wide, LegBy, Four, Six, etc.
- Commentator and Commentary: The commentator adds ball-by-ball commentary.
- Umpire and Referee: These classes will store details about umpires and referees, respectively.
- Stat: Our system will keep track of the stats for every player, match and tournament.

- StatQuery: This class will encapsulate general stat queries and their answers, like “Who has scored the maximum number of 100s in ODIs?” or, “Which bowler has taken the most wickets in test matches?”, etc.



Activity Diagram:

- Record a Ball of an Over: Here are the steps to record a ball of an over in the system:



Code:

```
from enum import Enum


class Address:
    def __init__(self, street, city, state, zip_code, country):
        self.__street_address = street
        self.__city = city
        self.__state = state
        self.__zip_code = zip_code
        self.__country = country


class Person():
    def __init__(self, name, address, email, phone):
        self.__name = name
        self.__address = address
        self.__email = email
        self.__phone = phone


class MatchFormat(Enum):
    ODI, T20, TEST = 1, 2, 3


class MatchResult(Enum):
    LIVE, FINISHED, DRAWN, CANCELLED = 1, 2, 3, 4


class UmpireType(Enum):
    FIELD, RESERVED, TV = 1, 2, 3


class WicketType(Enum):
    BOLD, CAUGHT, STUMPED, RUN_OUT, LBW, RETIRED_HURT, HIT_WICKET, OBSTRUCTING = 1, 2, 3, 4, 5, 6, 7, 8


class BallType(Enum):
    NORMAL, WIDE, WICKET, NO_BALL = 1, 2, 3, 4


class RunType(Enum):
    NORMAL, FOUR, SIX, LEG_BYE, BYE, NO_BALL, OVERTHROW = 1, 2, 3, 4, 5, 6, 7
```

```python
class Player:
    def __init__(self, person):
        self.__person = person
        self.__contracts = []

    def add_contract(self, contract):
        None


class Admin:
    def __init__(self, person):
        self.__person = person

    def add_match(self, match):
        None

    def add_team(self, team):
        None

    def add_tournament(self, tournament):
        None


class Umpire:
    def __init__(self, person):
        self.__person = person

    def assign_match(self, match):
        None


class Referee:
    def __init__(self, person):
        self.__person = person

    def assign_match(self, match):
        None


class Commentator:
    def __init__(self, person):
        self.__person = person

    def assign_match(self, match):
        None
```

```python
class Team:
    def __init__(self, name, coach):
        self.__name = name
        self.__players = []
        self.__news = []
        self.__coach = coach

    def add_tournament_squad(self, tournament_squad):
        None

    def add_player(self, player):
        None

    def add_news(self, news):
        None


class TournamentSquad:
    def __init__(self):
        self.__players = []
        self.__tournament_stats = []

    def add_player(self, player):
        None


class Playing11:
    def __init__(self):
        self.__players = []
        self.__twelfth_man = None

    def add_player(self, player):
        None
```

```python
from datetime import datetime
from abc import ABC
from .constants import MatchResult


class Over:
    def __init__(self, number):
        self.__number = number
        self.__balls = []

    def add_ball(self, ball):
        None


class Ball:
    def __init__(self, balled_by, played_by, ball_type, wicket, runs, commentary):
        self.__balled_by = balled_by
        self.__played_by = played_by
        self.__type = ball_type

        self.__wicket = wicket
        self.__runs = runs
        self.__commentary = commentary


class Wicket:
    def __init__(self, wicket_type, player_out, caught_by, runout_by, stumped_by):
        self.__wicket_type = wicket_type
        self.__player_out = player_out
        self.__caught_by = caught_by
        self.__runout_by = runout_by
        self.__stumped_by = stumped_by


class Commentary:
    def __init__(self, text, commentator):
        self.__text = text
        self.__created_at = datetime.date.today()
        self.__created_by = commentator


class Inning:
    def __init__(self, number, start_time):
        self.__number = number
        self.__start_time = start_time
        self.__overs = []

    def add_over(self, over):
        None
```

```
class Match(ABC):
    def __init__(self, number, start_time, referee):
        self.__number = number
        self.__start_time = start_time
        self.__result = MatchResult.LIVE

        self.__teams = []
        self.__innings = []
        self.__umpires = []
        self.__referee = referee
        self.__commentators = []
        self.__match_stats = []

    def assign_stadium(self, stadium):
        None

    def assign_referee(self, referee):
        None


class ODI(Match):
    # ...
    pass


class Test(Match):
    # ...
    pass
```

Design Car Rental System:

Description:

A Car Rental System is a software built to handle the renting of automobiles for a short period of time, generally ranging from a few hours to a few weeks. A car rental system often has numerous local branches (to allow its user to return a vehicle to a different location), and primarily located near airports or busy city areas.

Requirements:

- The system will support the renting of different automobiles like cars, trucks, SUVs, vans, and motorcycles.
- Each vehicle should be added with a unique barcode and other details, including a parking stall number which helps to locate the vehicle.
- The system should be able to retrieve information like which member took a particular vehicle or what vehicles have been rented out by a specific member.
- The system should collect a late-fee for vehicles returned after the due date.
- Members should be able to search the vehicle inventory and reserve any available vehicle.
- The system should be able to send notifications whenever the reservation is approaching the pick-up date, as well as when the vehicle is nearing the due date or has not been returned within the due date.
- The system will be able to read barcodes from vehicles.

- Members should be able to cancel their reservations.
- The system should maintain a vehicle log to track all events related to the vehicles.
- Members can add rental insurance to their reservation.
- Members can rent additional equipment, like navigation, child seat, ski rack, etc.
- Members can add additional services to their reservation, such as roadside assistance, additional driver, wifi, etc.

Use Case Diagram:

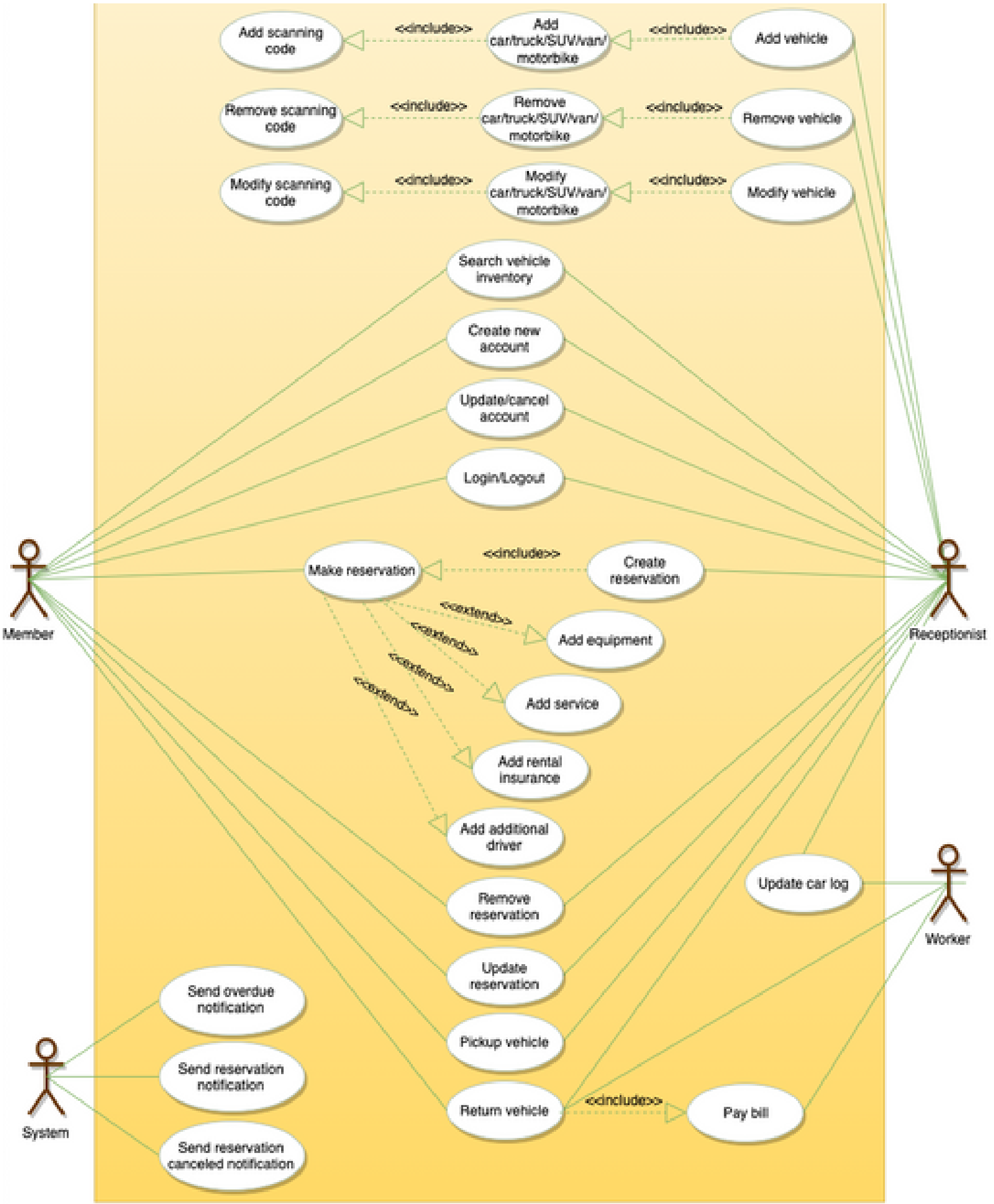
Main Actors:

- Receptionist: Mainly responsible for adding and modifying vehicles and workers. Receptionists can also reserve vehicles.
- Member: All members can search the catalog, as well as reserve, pick-up, and return a vehicle.
- System: Mainly responsible for sending notifications about overdue vehicles, canceled reservation, etc.
- Worker: Mainly responsible for taking care of a returned vehicle and updating the vehicle log.

Top Use Cases:

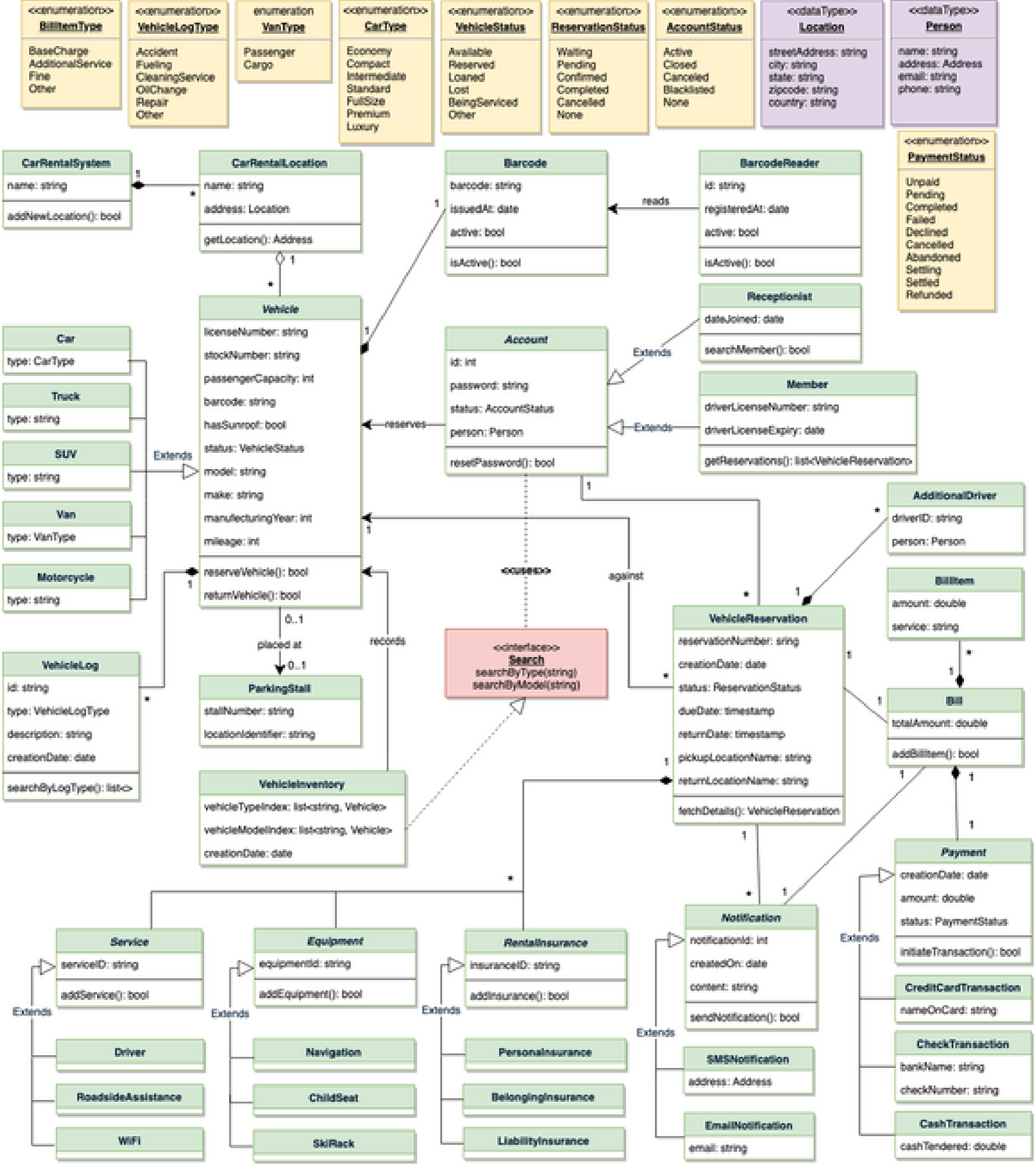
- Add/Remove/Edit vehicle: To add, remove or modify a vehicle.
- Search catalog: To search for vehicles by type and availability.
- Register new account/Cancel membership: To add a new member or cancel an existing membership.

- Reserve vehicle: To reserve a vehicle.
- Check-out vehicle: To rent a vehicle.
- Return a vehicle: To return a vehicle which was checked-out to a member.
- Add equipment: To add an equipment to a reservation like navigation, child seat, etc.
- Update car log: To add or update a car log entry, such as refueling, cleaning, damage, etc.


Add scanning code
<<include>>
Add car/truck/SUV/van/ motorbike
<<include>>
Add vehicle
Remove scanning code
<<include>>
Remove car/truck/SUV/van/ motorbike
<<include>>
Remove vehicle
Modify scanning code
<<include>>
Modify car/truck/SUV/van/ motorbike
<<include>>
Modify vehicle
Search vehicle inventory
Create new account
Update/cancel account
Login/Logout
Member
Make reservation
<<include>>
Create reservation
<<extend>>
<<extend>>
<<extend>>
<<extend>>
Add equipment
Add service
Add rental insurance
Add additional driver
Receptionist
Remove reservation
Update reservation
Pickup vehicle
Return vehicle
<<include>>
Pay bill
Update car log
Worker
System
Send overdue notification
Send reservation notification
Send reservation canceled notification

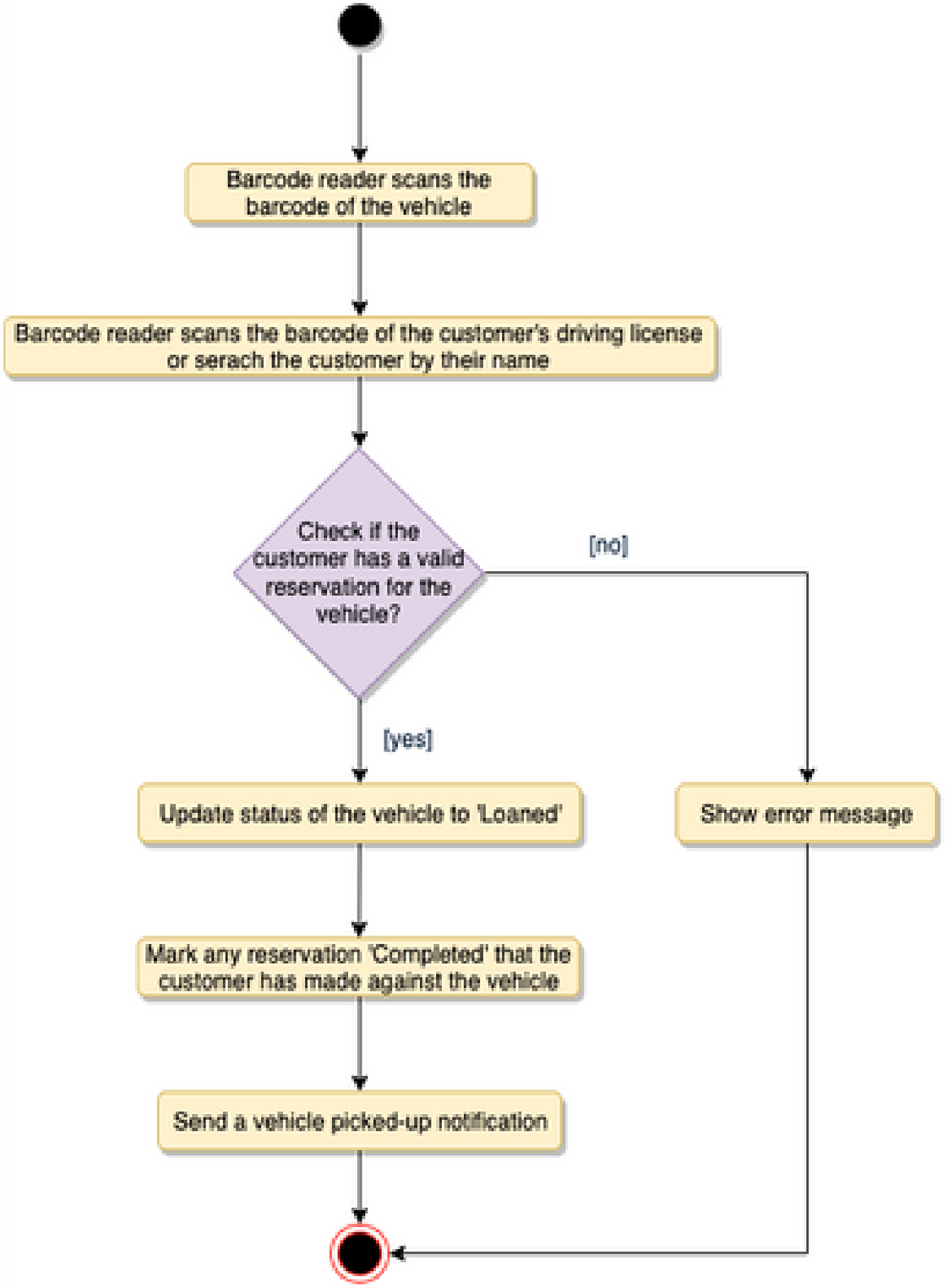
Class Diagram:

- CarRentalSystem: The main part of the organization for which this software has been designed.
- CarRentalLocation: The car rental system will have multiple locations, each location will have attributes like ‘Name’ to distinguish it from any other locations and ‘Address’ which defines the address of the rental location.
- Vehicle: The basic building block of the system. Every vehicle will have a barcode, license plate number, passenger capacity, model, make, mileage, etc. Vehicles can be of multiple types, like car, truck, SUV, etc.
- Account: Mainly, we will have two types of accounts in the system, one will be a general member and the other will be a receptionist. Another account can be of the worker taking care of the returned vehicle.
- VehicleReservation: This class will be responsible for managing reservations for a vehicle.
- Notification: Will take care of sending notifications to members.
- VehicleLog: To keep track of all the events related to a vehicle.
- RentalInsurance: Stores details about the various rental insurances that members can add to their reservation.
- Equipment: Stores details about the various types of equipment that members can add to their reservation.
- Service: Stores details about the various types of service that members can add to their reservation, such as additional drivers, roadside assistance, etc.
- Bill: Contains different bill-items for every charge for the reservation.

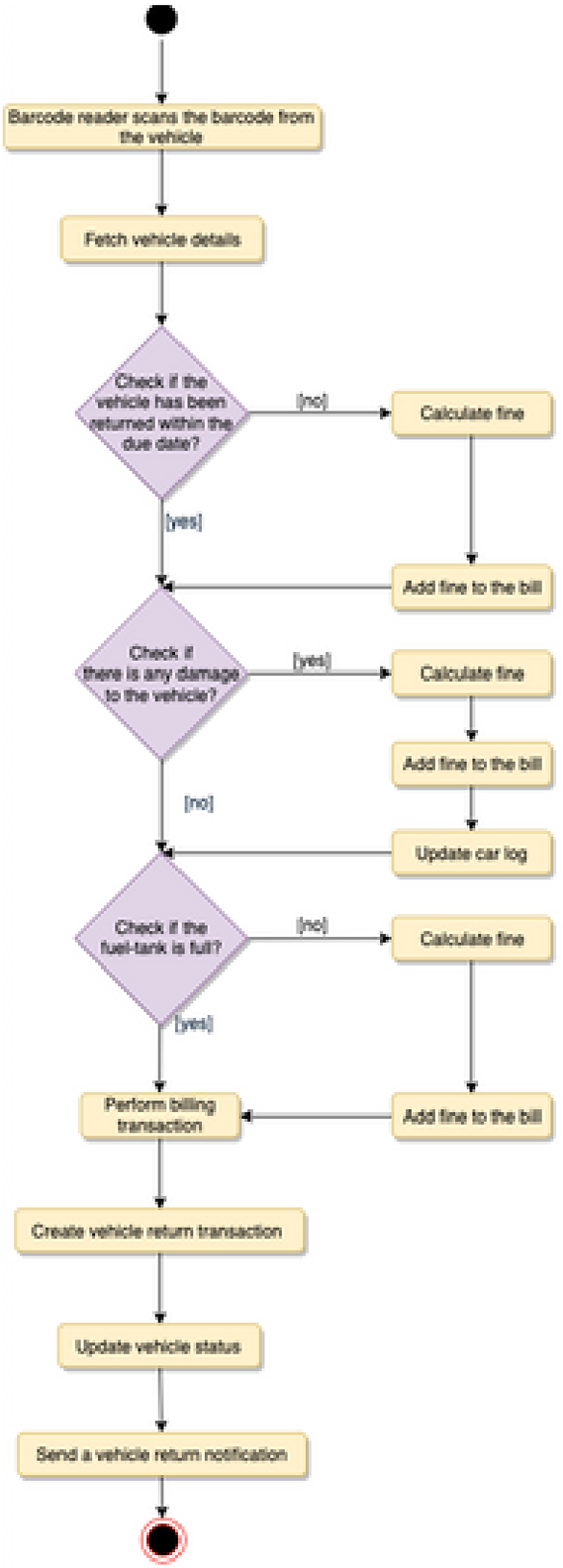

«enumeration»
BillItemType
BaseCharge
AdditionalService
Fine
Other
«enumeration»
VehicleLogType
Accident
Fueling
CleaningService
OilChange
Repair
Other
enumeration
VanType
Passenger
Cargo
«enumeration»
CarType
Economy
Compact
Intermediate
Standard
FullSize
Premium
Luxury
«enumeration»
VehicleStatus
Available
Reserved
Loaned
Lost
BeingServiced
Other
«enumeration»
ReservationStatus
Waiting
Pending
Confirmed
Completed
Cancelled
None
«enumeration»
AccountStatus
Active
Closed
Canceled
Blacklisted
None
«dataType»
Location
streetAddress: string
city: string
state: string
zipcode: string
country: string
«dataType»
Person
name: string
address: Address
email: string
phone: string
«enumeration»
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Cancelled
Abandoned
Settling
Settled
Refunded
CarRentalSystem
name: string
addNewLocation(): bool
CarRentalLocation
name: string
address: Location
getLocation(): Address
Barcode
barcode: string
issuedAt: date
active: bool
isActive(): bool
reads
BarcodeReader
id: string
registeredAt: date
active: bool
isActive(): bool
Vehicle
licenseNumber: string
stockNumber: string
passengerCapacity: int
barcode: string
hasSunroof: bool
status: VehicleStatus
model: string
make: string
manufacturingYear: int
mileage: int
reserveVehicle(): bool
returnVehicle(): bool
Account
id: int
password: string
status: AccountStatus
person: Person
resetPassword(): bool
reserves
Receptionist
dateJoined: date
searchMember(): bool
Extends
Member
driverLicenseNumber: string
driverLicenseExpiry: date
getReservations(): list<VehicleReservation>
Extends
Car
type: CarType
Truck
type: string
SUV
type: string
Van
type: VanType
Motorcycle
type: string
Extends
«uses»
«interface»
Search
searchByType(string)
searchByModel(string)
against
AdditionalDriver
driverID: string
person: Person
BillItem
amount: double
service: string
VehicleReservation
reservationNumber: sring
creationDate: date
status: ReservationStatus
dueDate: timestamp
returnDate: timestamp
pickupLocationName: string
returnLocationName: string
fetchDetails(): VehicleReservation
Bill
totalAmount: double
addBillItem(): bool
0..1
placed at
0..1
ParkingStall
stallNumber: string
locationIdentifier: string
records
VehicleLog
id: string
type: VehicleLogType
description: string
creationDate: date
searchByLogType(): list<>
VehicleInventory
vehicleTypeIndex: list<string, Vehicle>
vehicleModelIndex: list<string, Vehicle>
creationDate: date
Service
serviceID: string
addService(): bool
Equipment
equipmentId: string
addEquipment(): bool
RentalInsurance
insuranceID: string
addInsurance(): bool
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool
Payment
creationDate: date
amount: double
status: PaymentStatus
initiateTransaction(): bool
Extends
Driver
RoadsideAssistance
WiFi
Navigation
ChildSeat
SkiRack
PersonalInsurance
BelongingInsurance
LiabilityInsurance
SMSNotification
address: Address
EmailNotification
email: string
CreditCardTransaction
nameOnCard: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double

Activity Diagrams:

- Pick up a vehicle: Any member can perform this activity. Here are the steps to pick up a vehicle:



- Return a vehicle: Any worker can perform this activity. While returning a vehicle, the system must collect a late fee from the member if the return date is after the due date. Here are the steps for returning a vehicle:



```python
from enum import Enum


class BillItemType(Enum):
    BASE_CHARGE, ADDITIONAL_SERVICE, FINE, OTHER = 1, 2, 3, 4


class VehicleLogType(Enum):
    ACCIDENT, FUELING, CLEANING_SERVICE, OIL_CHANGE, REPAIR, OTHER = 1, 2, 3, 4, 5, 6


class VanType(Enum):
    PASSENGER, CARGO = 1, 2


class CarType(Enum):
    ECONOMY, COMPACT, INTERMEDIATE, STANDARD, FULL_SIZE, PREMIUM, LUXURY = 1, 2, 3, 4, 5, 6, 7


class VehicleStatus(Enum):
    AVAILABLE, RESERVED, LOANED, LOST, BEING_SERVICED, OTHER = 1, 2, 3, 4, 5, 6


class ReservationStatus(Enum):
    ACTIVE, PENDING, CONFIRMED, COMPLETED, CANCELLED, NONE = 1, 2, 3, 4, 5, 6


class AccountStatus(Enum):
    ACTIVE, CLOSED, CANCELED, BLACKLISTED, BLOCKED = 1, 2, 3, 4, 5


class PaymentStatus(Enum):
    UNPAID, PENDING, COMPLETED, FILLED, DECLINED, CANCELLED, ABANDONED, SETTLING, SETTLED,
    REFUNDED = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10


class Address:
    def __init__(self, street, city, state, zip_code, country):
        self.__street_address = street
        self.__city = city
        self.__state = state
        self.__zip_code = zip_code
        self.__country = country
```

```python
class Person():
    def __init__(self, name, address, email, phone):
        self.__name = name
        self.__address = address
        self.__email = email
        self.__phone = phone

from abc import ABC
from .constants import AccountStatus

class Account(ABC):
    def __init__(self, id, password, person, status=AccountStatus.NONE):
        self.__id = id
        self.__password = password
        self.__status = AccountStatus.NONE
        self.__person = person

    def reset_password(self):
        None


class Member(Account):
    def __init__(self):
        self.__total_vehicles_reserved = 0

    def get_reservations(self):
        None


class Receptionist(Account):
    def __init__(self, date_joined):
        self.__date_joined = date_joined

    def search_member(self, name):
        None


class AdditionalDriver:
    def __init__(self, id, person):
        self.__driver_id = id
        self.__person = person
```

```python
class CarRentalLocation:
    def __init__(self, name, address):
        self.__name = name
        self.__location = address

    def get_location(self):
        return self.__location


class CarRentalSystem:
    def __init__(self, name):
        self.__name = name
        self.__locations = []

    def add_new_location(self, location):
        None
```

```python
from abc import ABC
from datetime import datetime
from .constants import ReservationStatus


class Vehicle(ABC):
    def __init__(self, license_num, stock_num, capacity, barcode, has_sunroof, status, model, make, manufacturing_year,
                 mileage):
        self.__license_number = license_num
        self.__stock_number = stock_num
        self.__passenger_capacity = capacity
        self.__barcode = barcode
        self.__has_sunroof = has_sunroof
        self.__status = status
        self.__model = model
        self.__make = make
        self.__manufacturing_year = manufacturing_year
        self.__mileage = mileage
        self.__log = []

    def reserve_vehicle(self):
        None

    def return_vehicle(self):
        None


class Car(Vehicle):
    def __init__(self, license_num, stock_num, capacity, barcode, has_sunroof, status, model, make, manufacturing_year,
                 mileage, type):
        super().__init__(license_num, stock_num, capacity, barcode,
                         has_sunroof, status, model, make, manufacturing_year, mileage)
        self.__type = type
```

```python
class VehicleLog:
    def __init__(self, id, type, description, creation_date):
        self.__id = id
        self.__type = type
        self.__description = description
        self.__creation_date = creation_date

    def update(self):
        None

    def search_by_log_type(self, type):
        None


class VehicleReservation:
    def __init__(self, reservation_number):
        self.__reservation_number = reservation_number
        self.__creation_date = datetime.date.today()
        self.__status = ReservationStatus.ACTIVE
        self.__due_date = datetime.date.today()
        self.__return_date = datetime.date.today()
        self.__pickup_location_name = ""
        self.__return_location_name = ""
        self.__customer_id = 0
        self.__vehicle = None
        self.__bill = None
        self.__additional_drivers = []
        self.__notifications = []
        self.__insurances = []
        self.__equipments = []
        self.__services = []

    def fetch_reservation_details(self, reservation_number):
        None

    def get_additional_drivers(self):
        return self.__additional_drivers
```

```python
from abc import ABC


class Search(ABC):
    def search_by_type(self, type):
        None

    def search_by_model(self, model):
        None


class VehicleInventory(Search):
    def __init__(self):
        self.__vehicle_types = {}
        self.__vehicle_models = {}

    def search_by_type(self, query):
        # return all vehicles of the given type.
        return self.__vehicle_types.get(query)

    def search_by_model(self, query):
        # return all vehicles of the given model.
        return self.__vehicle_models.get(query)
```

Online Stock Brokerage System:

Description:

An Online Stock Brokerage System facilitates its users the trade (i.e. buying and selling) of stocks online. It allows clients to keep track of and execute their transactions, and shows performance charts of the different stocks in their portfolios. It also provides security for their transactions and alerts them to pre-defined levels of changes in stocks, without the use of any middlemen.

The online stock brokerage system automates traditional stock trading using computers and the internet, making the transaction faster and cheaper. This system also gives speedier access to stock reports, current market trends, and real-time stock prices.

Requirements:

- Any user of our system should be able to buy and sell stocks.
- Any user can have multiple watchlists containing multiple stock quotes.
- Users should be able to place stock trade orders of the following types: 1) market, 2) limit, 3) stop loss and, 4) stop limit.
- Users can have multiple 'lots' of a stock. This means that if a user has bought a stock multiple times, the system should be able to differentiate between different lots of the same stock.
- The system should be able to generate reports for quarterly updates and yearly tax statements.

- Users should be able to deposit and withdraw money either via check, wire, or electronic bank transfer.
- The system should be able to send notifications whenever trade orders are executed.

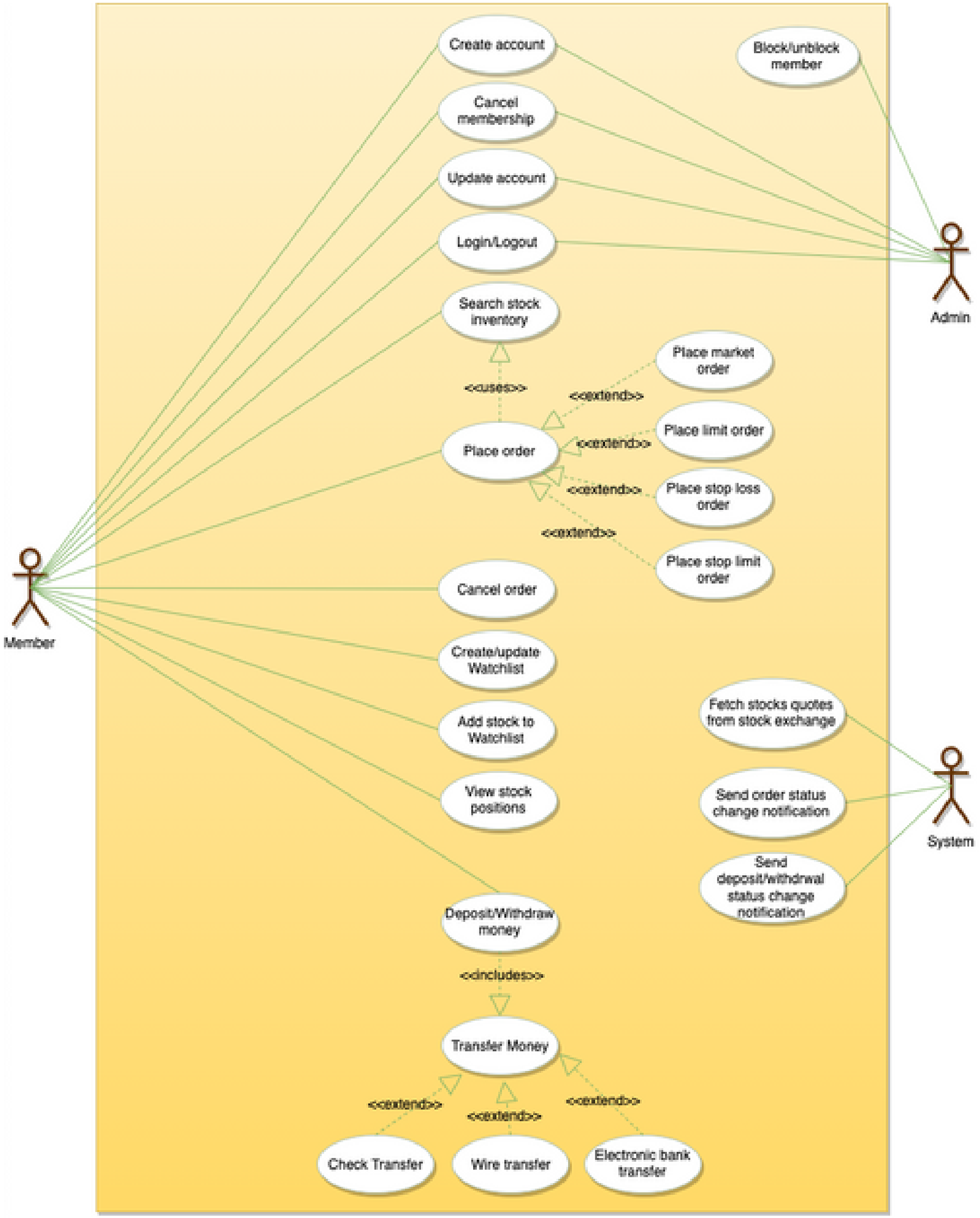
Use Case Diagram:

Main Actors:

- Admin: Mainly responsible for administrative functions like blocking or unblocking members.
- Member: All members can search the stock inventory, as well as buy and sell stocks. Members can have multiple watchlists containing multiple stock quotes.
- System: Mainly responsible for sending notifications for stock orders and periodically fetching stock quotes from the stock exchange.

Top Use Cases:

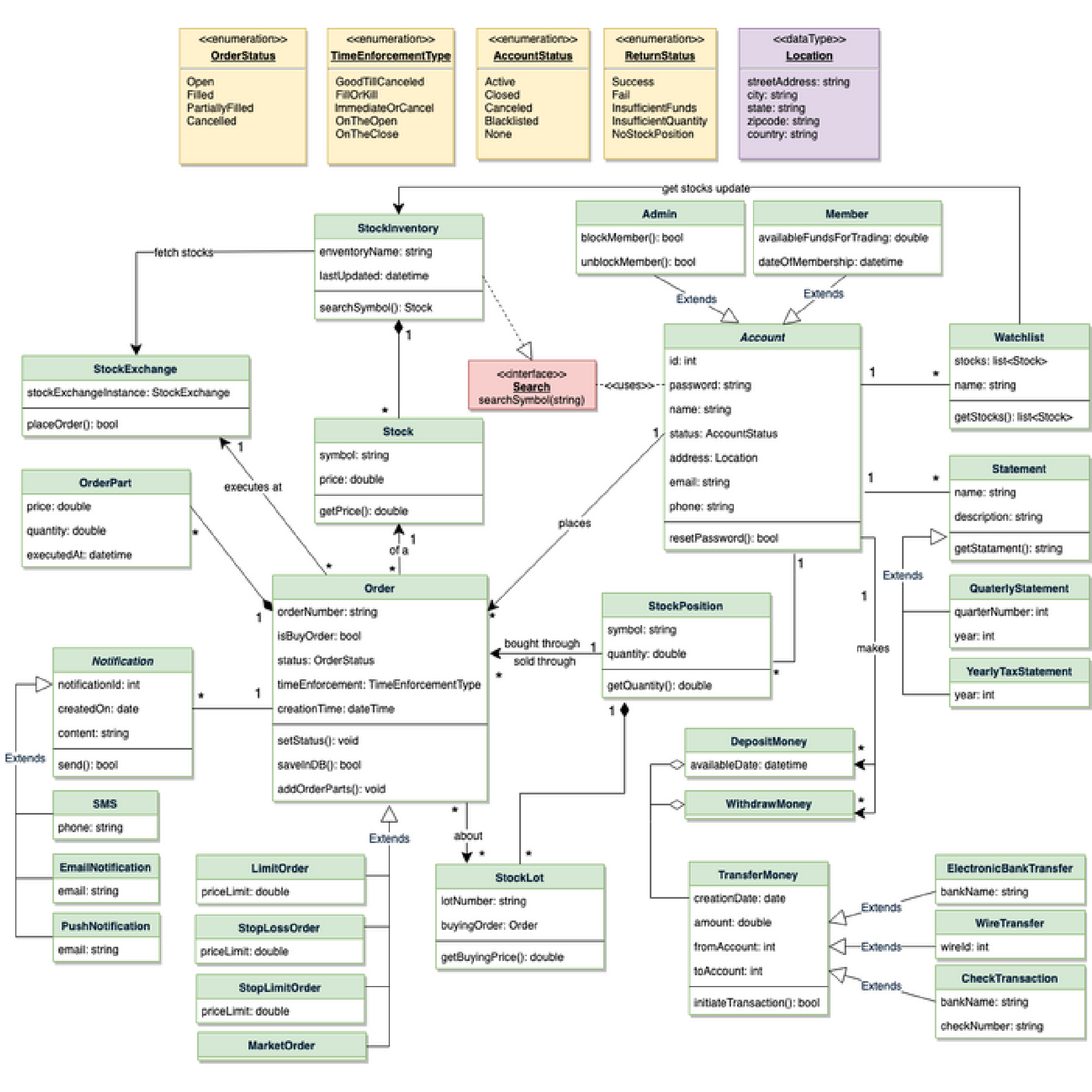
- Register new account/Cancel membership: To add a new member or cancel the membership of an existing member.
- Add/Remove/Edit watchlist: To add, remove or modify a watchlist.
- Search stock inventory: To search for stocks by their symbols.
- Place order: To place a buy or sell order on the stock exchange.
- Cancel order: Cancel an already placed order.
- Deposit/Withdraw money: Members can deposit or withdraw money via check, wire or electronic bank transfer.


Create account
Block/unblock member
Cancel membership
Update account
Login/Logout
Admin
Search stock inventory
Place market order
<<uses>>
<<extend>>
Place limit order
Place order
<<extend>>
<<extend>>
Place stop loss order
<<extend>>
Place stop limit order
Member
Cancel order
Create/update Watchlist
Add stock to Watchlist
Fetch stocks quotes from stock exchange
View stock positions
Send order status change notification
System
Send deposit/withdrwal status change notification
Deposit/Withdraw money
<<includes>>
Transfer Money
<<extend>>
<<extend>>
<<extend>>
Check Transfer
Wire transfer
Electronic bank transfer

Class Diagram:

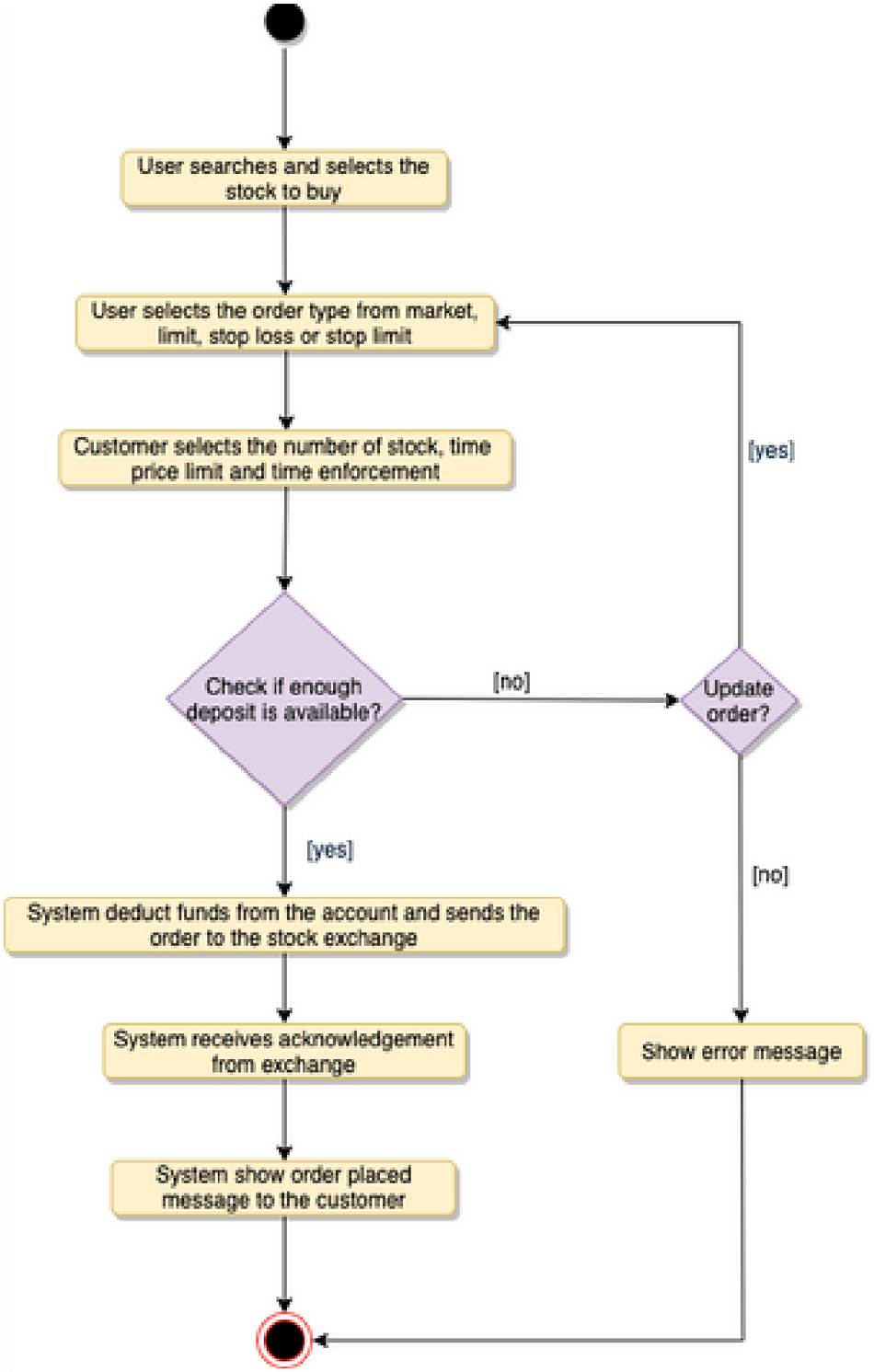
- Account: Consists of the member's name, address, e-mail, phone, total funds, funds that are available for trading, etc. We'll have two types of accounts in the system: one will be a general member, and the other will be an Admin. The Account class will also contain all the stocks the member is holding.
- StockExchange: The stockbroker system will fetch all stocks and their current prices from the stock exchange. StockExchange will be a singleton class encapsulating all interactions with the stock exchange. This class will also be used to place stock trading orders on the stock exchange.
- Stock: The basic building block of the system. Every stock will have a symbol, current trading price, etc.
- StockInventory: This class will fetch and maintain the latest stock prices from the StockExchange. All system components will read the most recent stock prices from this class.
- Watchlist: A watchlist will contain a list of stocks that the member wants to follow.
- Order: Members can place stock trading orders whenever they would like to sell or buy stock positions. The system would support multiple types of orders:
- Market Order: Market order will enable users to buy or sell stocks immediately at the current market price.
- Limit Order: Limit orders will allow a user to set a price at which they want to buy or sell a stock.

- Stop Loss Order: An order to buy or sell once the stock reaches a certain price.
- Stop Limit Order: The stop-limit order will be executed at a specified price or better after a given stop price has been reached. Once the stop price is reached, the stop-limit order becomes a limit order to buy or sell at the limit price or better.
- OrderPart: An order could be fulfilled in multiple parts. For example, a market order to buy 100 stocks could have one part containing 70 stocks at $10 and another part with 30 stocks at $10.05.
- StockLot: Any member can buy multiple lots of the same stock at different times. This class will represent these individual lots. For example, the user could have purchased 100 shares of AAPL yesterday and 50 more stocks of AAPL today. While selling, users will be able to select which lot they want to sell first.
- StockPosition: This class will contain all the stocks that the user holds.
- Statement: All members will have reports for quarterly updates and yearly tax statements.
- DepositMoney & WithdrawMoney: Members will be able to move money through check, wire or electronic bank transfers.
- Notification: Will take care of sending notifications to members.

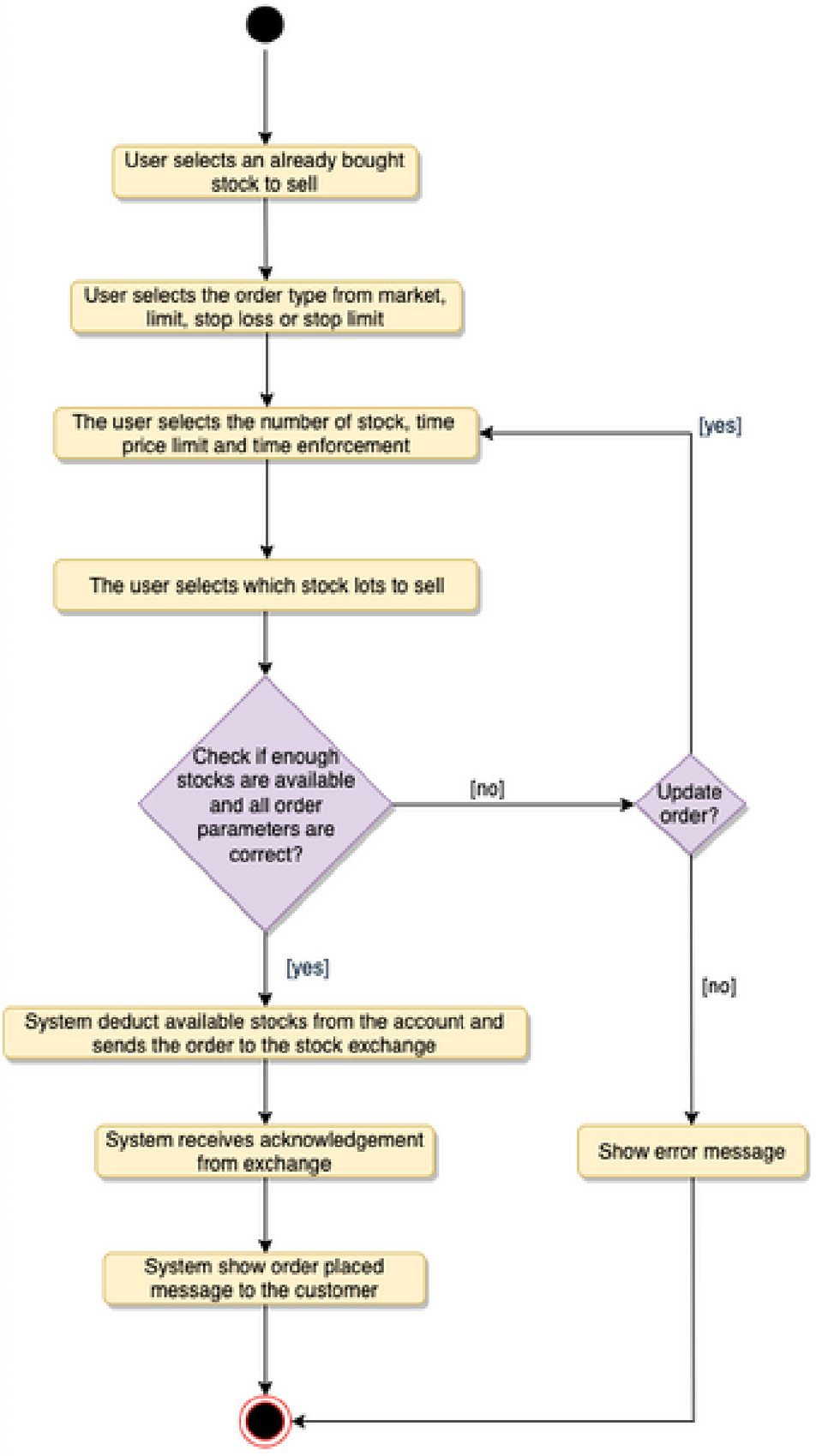

<<enumeration>>
OrderStatus
Open
Filled
PartiallyFilled
Cancelled
<<enumeration>>
TimeEnforcementType
GoodTillCanceled
FillOrKill
ImmediateOrCancel
OnTheOpen
OnTheClose
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
<<enumeration>>
ReturnStatus
Success
Fail
InsufficientFunds
InsufficientQuantity
NoStockPosition
<<dataType>>
Location
streetAddress: string
city: string
state: string
zipcode: string
country: string
get stocks update
StockInventory
enventoryName: string
lastUpdated: datetime
searchSymbol(): Stock
fetch stocks
Admin
blockMember(): bool
unblockMember(): bool
Member
availableFundsForTrading: double
dateOfMembership: datetime
Extends
Extends
StockExchange
stockExchangeInstance: StockExchange
placeOrder(): bool
<<interface>>
Search
searchSymbol(string)
<<uses>>
Account
id: int
password: string
name: string
status: AccountStatus
address: Location
email: string
phone: string
resetPassword(): bool
Watchlist
stocks: list<Stock>
name: string
getStocks(): list<Stock>
Stock
symbol: string
price: double
getPrice(): double
executes at
of a
places
OrderPart
price: double
quantity: double
executedAt: datetime
Statement
name: string
description: string
getStatament(): string
Extends
QuaterlyStatement
quarterNumber: int
year: int
YearlyTaxStatement
year: int
Order
orderNumber: string
isBuyOrder: bool
status: OrderStatus
timeEnforcement: TimeEnforcementType
creationTime: dateTime
setStatus(): void
saveInDB(): bool
addOrderParts(): void
bought through
sold through
StockPosition
symbol: string
quantity: double
getQuantity(): double
makes
Notification
notificationId: int
createdOn: date
content: string
send(): bool
Extends
SMS
phone: string
EmailNotification
email: string
PushNotification
email: string
Extends
about
StockLot
lotNumber: string
buyingOrder: Order
getBuyingPrice(): double
DepositMoney
availableDate: datetime
WithdrawMoney
TransferMoney
creationDate: date
amount: double
fromAccount: int
toAccount: int
initiateTransaction(): bool
Extends
Extends
Extends
ElectronicBankTransfer
bankName: string
WireTransfer
wireId: int
CheckTransaction
bankName: string
checkNumber: string
LimitOrder
priceLimit: double
StopLossOrder
priceLimit: double
StopLimitOrder
priceLimit: double
MarketOrder

Activity Diagrams:

- Place a buy order: Any system user can perform this activity. Here are the steps to place a buy order:



- Place a sell order: Any system user can perform this activity. Here are the steps to place a buy order:



Code:

```
from enum import Enum


class ReturnStatus(Enum):
    SUCCESS, FAIL, INSUFFICIENT_FUNDS, INSUFFICIENT_QUANTITY, NO_STOCK_POSITION = 1, 2, 3, 4, 5, 6


class OrderStatus(Enum):
    OPEN, FILLED, PARTIALLY_FILLED, CANCELLED = 1, 2, 3, 4


class TimeEnforcementType(Enum):
    GOOD_TILL_CANCELLED, FILL_OR_KILL, IMMEDIATE_OR_CANCEL, ON_THE_OPEN, ON_THE_CLOSE = 1, 2, 3, 4, 5


class AccountStatus(Enum):
    ACTIVE, CLOSED, CANCELED, BLACKLISTED, NONE = 1, 2, 3, 5


class Location:
    def __init__(self, street, city, state, zip_code, country):
        self.__street_address = street
        self.__city = city
        self.__state = state
        self.__zip_code = zip_code
        self.__country = country


class Constants:
    def __init__(self):
        self.__MONEY_TRANSFER_LIMIT = 100000
```

```
from .order import Order


class StockExchange:
    # singleton, used for restricting to create only one instance
    instance = None

    class __OnlyOne:
        def __init__(self):
            None

    def __init__(self):
        if not StockExchange.instance:
            StockExchange.instance = StockExchange.__OnlyOne()

    def place_order(self, order):
        return_status = self.get_instance().submit_order(Order)
        return return_status
```

```python
from abc import ABC
from datetime import datetime
from .constants import OrderStatus, TimeEnforcementType


class Order(ABC):
    def __init__(self, id):
        self.__order_id = id
        self.__is_buy_order = False
        self.__status = OrderStatus.OPEN
        self.__time_enforcement = TimeEnforcementType.ON_THE_OPEN
        self.__creation_time = datetime.now()

        self.__parts = {}

    def set_status(self, status):
        self.status = status

    def save_in_DB(self):
        None

    # save in the database

    def add_order_parts(self, parts):
        for part in parts:
            self.parts[part.get_id()] = part


class LimitOrder(Order):
    def __init__(self):
        self.__price_limit = 0.0
```

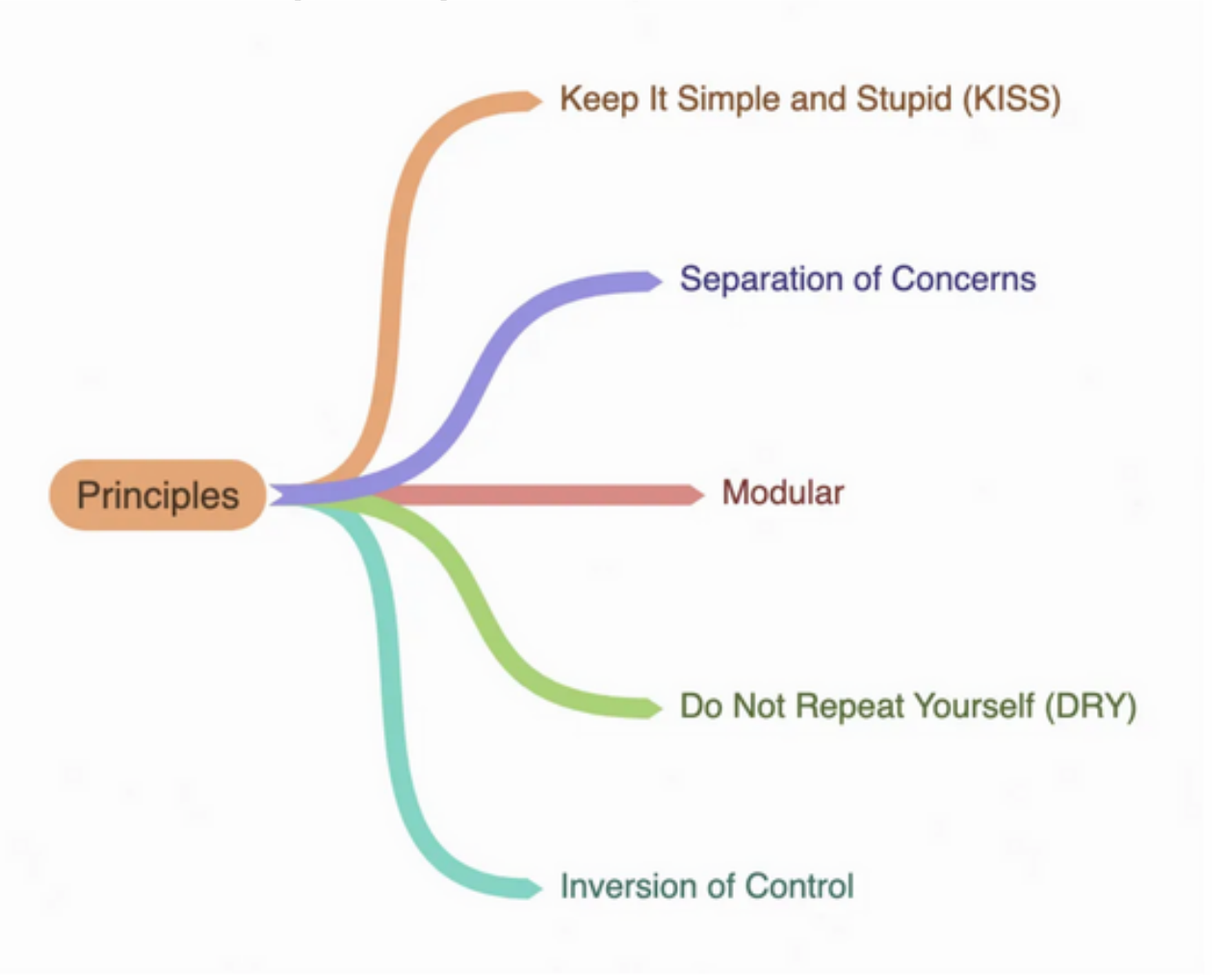
```python
from datetime import datetime
from abc import ABC
from .constants import OrderStatus, AccountStatus, ReturnStatus
from .order import LimitOrder
from .stock_exchange import StockExchange


class Account(ABC):
    def __init__(self, id, password, name, address, email, phone, status=AccountStatus.NONE):
        self.__id = id
        self.__password = password
        self.__name = name
        self.__address = address
        self.__email = email
        self.__phone = phone
        self.__status = AccountStatus.NONE

    def reset_password(self):
        None
```

```
class Member(Account):
    def __init__(self):
        self.__available_funds_for_trading = 0.0
        self.__date_of_membership = datetime.date.today()
        self.__stock_positions = {}
        self.__active_orders = {}

    def place_sell_limit_order(self, stock_id, quantity, limit_price, enforcement_type):
        # check if member has this stock position
        if stock_id not in self.__stock_positions:
            return ReturnStatus.NO_STOCK_POSITION

        stock_position = self.__stock_positions[stock_id]
        # check if the member has enough quantity available to sell
        if stock_position.get_quantity() < quantity:
            return ReturnStatus.INSUFFICIENT_QUANTITY

        order = LimitOrder(stock_id, quantity, limit_price, enforcement_type)
        order.is_buy_order = False
        order.save_in_DB()
        success = StockExchange.place_order(order)
        if success:
            order.set_status(OrderStatus.FAILED)
            order.save_in_DB()
        else:
            self.active_orders.add(order.get_order_id(), order)
        return success

    def place_buy_limit_order(self, stock_id, quantity, limit_price, enforcement_type):
        # check if the member has enough funds to buy this stock
        if self.__available_funds_for_trading < quantity * limit_price:
            return ReturnStatus.INSUFFICIENT_FUNDS

        order = LimitOrder(stock_id, quantity, limit_price, enforcement_type)
        order.is_buy_order = True
        order.save_in_DB()
        success = StockExchange.place_order(order)
        if not success:
            order.set_status(OrderStatus.FAILED)
            order.save_in_DB()
        else:
            self.active_orders.add(order.get_order_id(), order)
        return success

    # this function will be invoked whenever there is an update from
    # stock exchange against an order
    def callback_stock_exchange(self, order_id, order_parts, status):
        order = self.active_orders[order_id]
        order.add_order_parts(order_parts)
        order.set_status(status)
        order.update_in_DB()

        if status == OrderStatus.FILLED or status == OrderStatus.CANCELLEd:
            self.active_orders.remove(order_id)
```

Architecture and Design Principles Behind the Swiggy's Delivery Partners app

First things first: The Design principles:

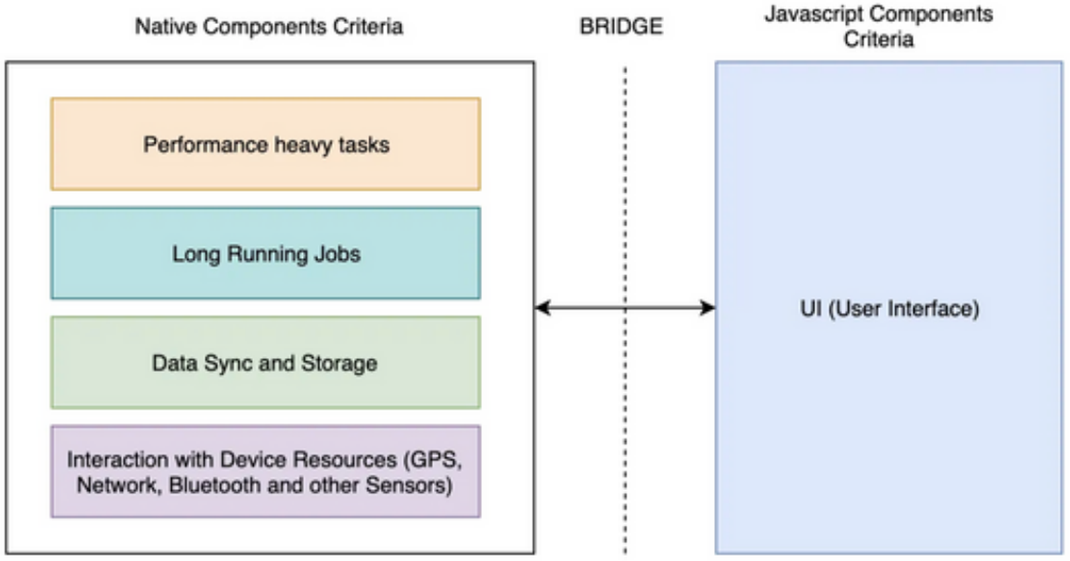
Keeping in mind to set good quality of the codebase standards, reusability of the code (we are lazy programmers!), futuristic approach (less rework), faster rollouts, A/B experimentation, plug-play-unplug use cases, unit testing, agility in integrating any 3rd party modules etc. we have laid out the below principles to stick to while building this app (Like how celestial bodies revolve around the Sun, our codebase revolves around these principles).



Core Design Principles (in-depth explanation below)

Keep it Simple and Stupid(KISS)

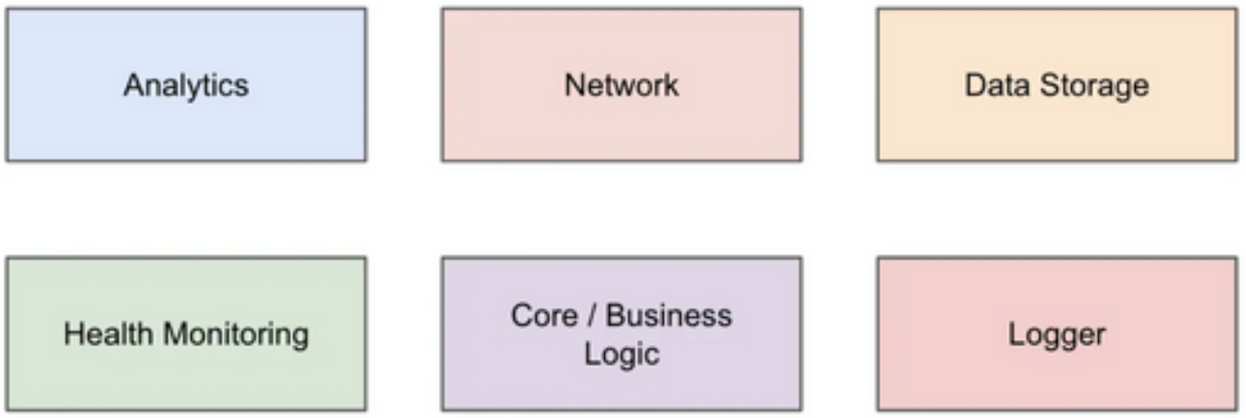
To maintain constant sync with our servers, our mobile application process needs to run in background for longer duration (on average 4–5 hours each session) and involves data such as tracking location info of our delivery partners (needed to assign trips, real-time tracking of an order, payment based on distance travelled etc.), order info (state of the order, item info, restaurant or store info, customer info etc.), earnings and incentives info etc. These components also interact with device resources such as GPS, network, bluetooth and other sensors for location tracking, proximity detection, distance travelled measurement, activity recognition etc. Performing these long-running/heavy operations on a single thread (UI / JS thread) isn't efficient and in the long run, will become a bottleneck for UI operations inducing jittery/laggy experience. So, to keep things simple we decided to build only UI components in React native and long running, background intensive, data storage components in Native (Android/iOS)



Division of Components based on key criteria

Separation of Concerns

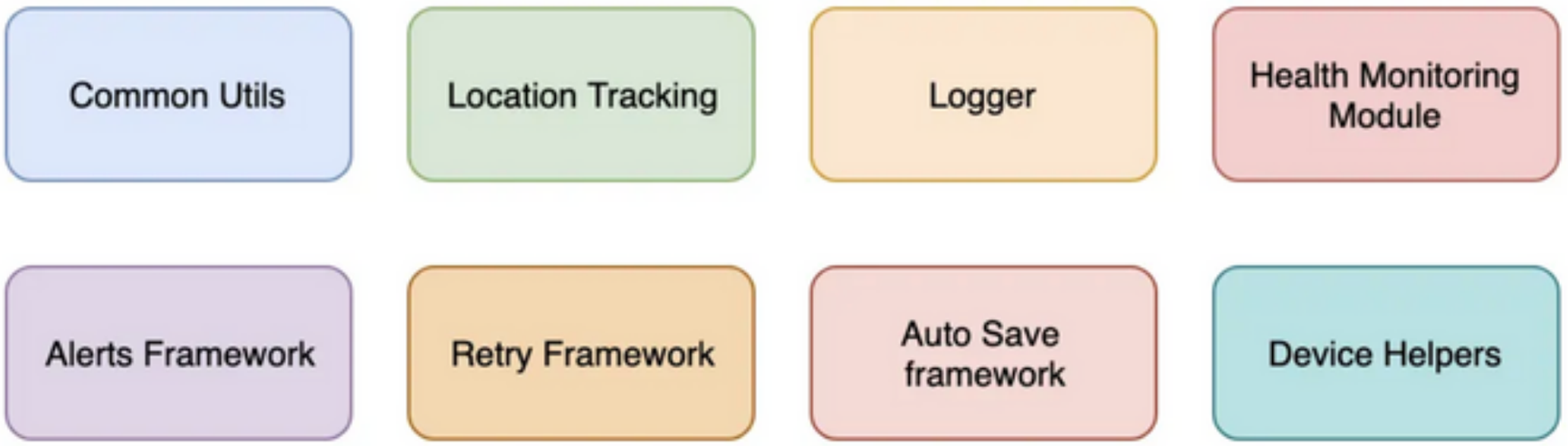
Our application is feature-rich and complex in serving the data needed to complete each delivery workflow. This involves handling user interactions, business logic, data sync and storage, network transactions, instrumentation of data such as user journey, touchpoints, real-time health metrics, etc. to name a few concerns. Instead of sprinkling these concerns all over the code base which becomes a developers nightmare to change/clean-up/revamp the code related to any concern, we separated these concerns either by creating a module or exposing a framework so that dependent modules can ignore the implementation details and be extensible to build upon.



Concerns involved in the App.

Modular and DRY (Do not repeat yourself)

At Swiggy, we are often working on multiple projects during which components that are built can be shared across multiple apps (Consumer, Delivery, Vendor, Daily etc.). Repeating the same code often involves development cycle, QA cycle, maintenance and stability monitoring, etc. To avoid these cycles and fast track production releases, we create components (however small it is) as agnostic to any specific application. So, based on this principle we decided to build below modules in which a couple of them are being shared across other apps.



Inversion of Control

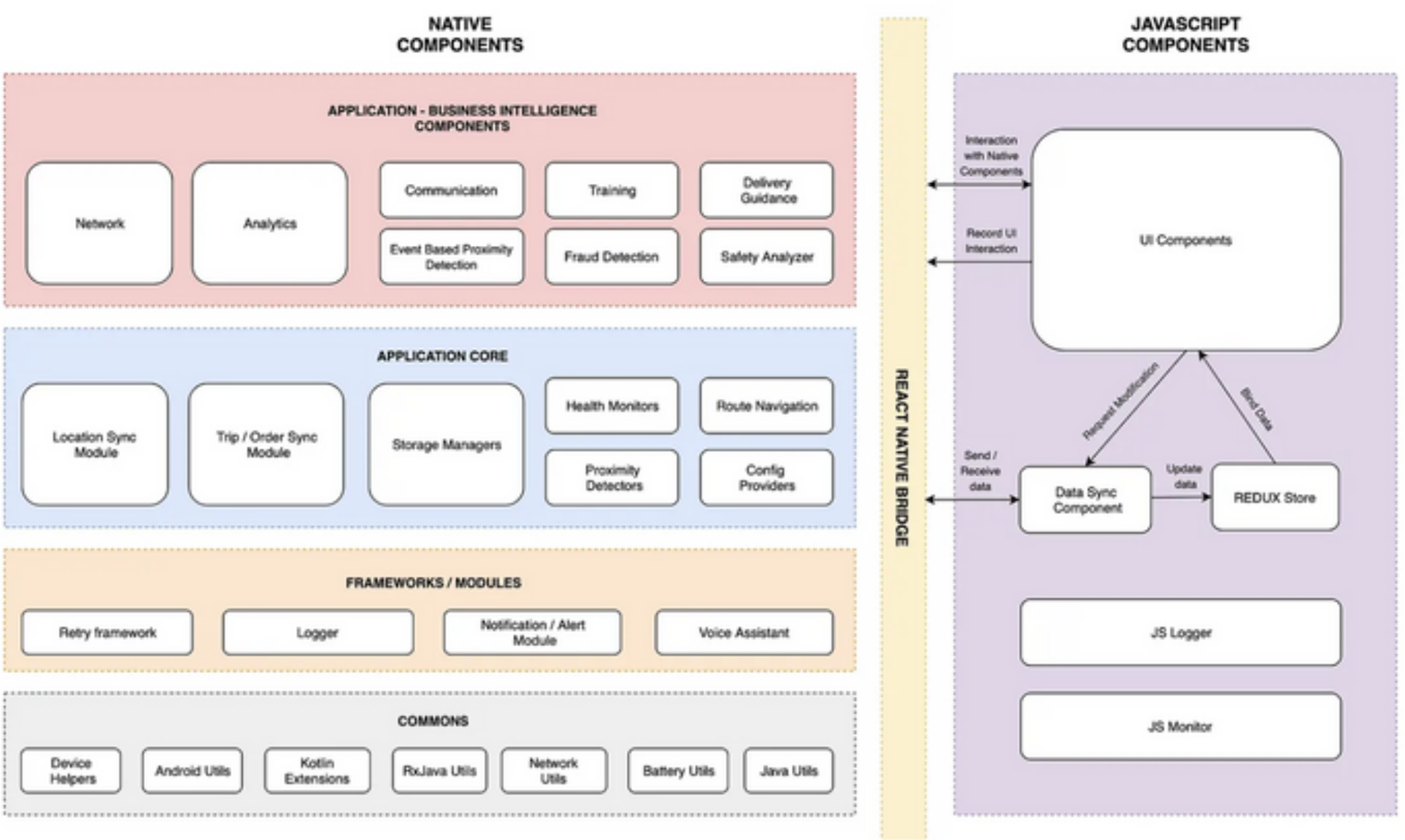
According to the above principles, we built smaller components with clear responsibilities and separation of concerns. But as we build more components in the future, the interaction between these components, dependency resolution and the flow of control will be cumbersome. So, we relied on this principle which is analogous to the Hollywood principle:

“Don’t call us, we will call you”.

Based on this principle, the flow of control is handled by the external components rather than the caller to reduce the complexity of handling the control. In order to decouple the dependency resolution from the callers and also to scope components better based on duty state, app state, etc; we integrated Dagger2 for dependency injection. As to delegate the flow to other components rather than creating a burden on the caller, we built few components as frameworks which invoke the caller functions appropriately or relied on events to trigger the flow. An example of this is the Retry framework which can be used by callers to retry a function unless successfully executed based on attributes like time, strategies, retry attempts etc. Overall, applying this principle helped us create smaller reusable components at the same time reduce the cumbersome problems involved while handling too many components.

High-Level Design

Based on our requirements and above principles we laid out the blueprint of the App before writing any piece of code. Below diagram shows the high-level design and major components involved.

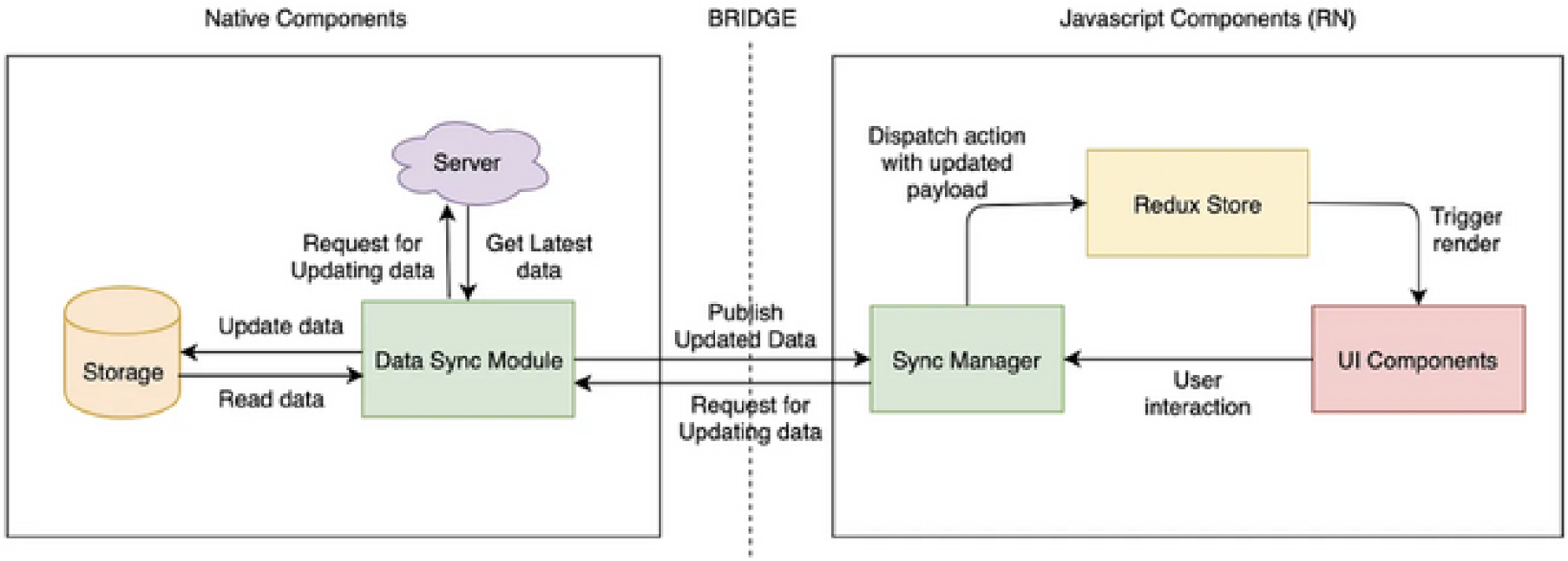


Blueprint of the App

The Design patterns

Unidirectional Data Flow:

As we have decided to build our sync modules, storage components in Native and UI in React native (based on above principles) we also need to stick to a pattern in the way data should be flowing across these components. Taking inspiration from the Flux pattern and especially the way REDUX works, we designed below blueprint of the data flow(server → storage → redux store → UI components).

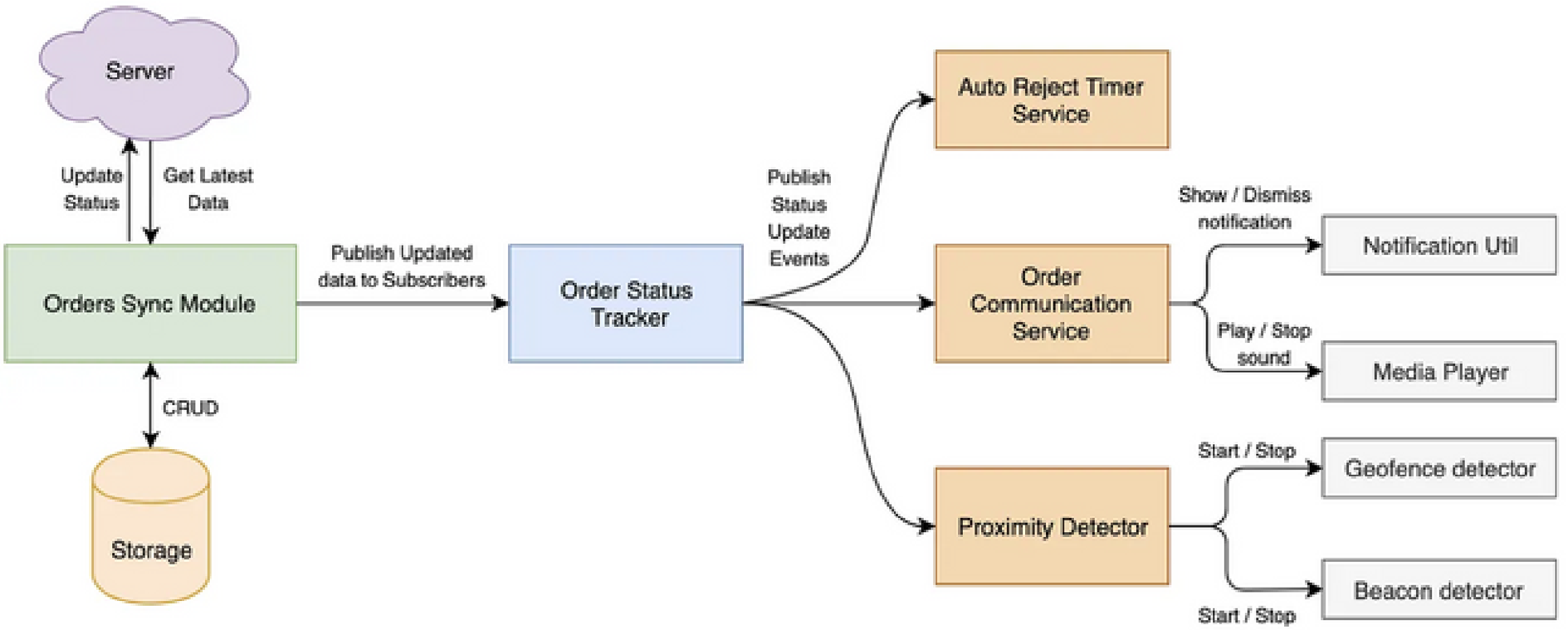


Blueprint of the data flow across the components

So, storage components in native acts as the single source of truth (w.r.t the client) and redux store in React native would be updated only after native storage components are updated from the server. So, any user interactions triggering data modification (ex: status update of an order) would not be dispatched directly to the redux store but rather be passed on to the sync module in Native which makes request to the server and accordingly update the storage components as-well-as publish the updates based on which redux store would be modified. This ensured us avoid any inconsistency in data across components and the reliability of the data is persisted.

Pub-Sub Pattern:

For the most part, our application is event driven and honours a finite state machine for each delivery flow. So, whenever any event or state transition occurs we might need to perform tasks like playing a sound and vibrating the phone as soon as an order is Assigned, showing a confirmation pop-up on the screen, uploading logs for real-time debugging, alerting partner when the battery level is critical or in poor network area, etc. Below is an example showcasing the usage of this pattern in one of our core components.



An example showcasing the usage of pub-sub pattern in one of our components

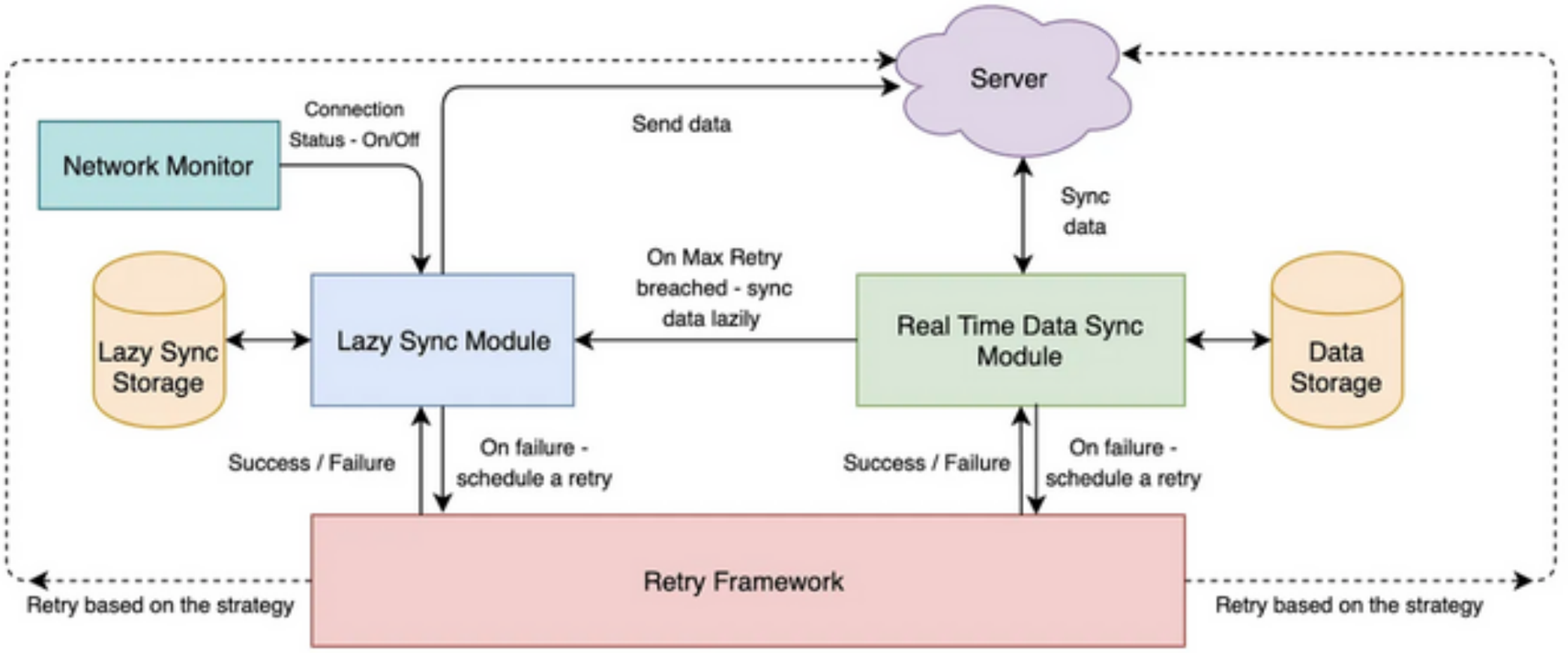
To simplify the reactive mechanisms mentioned above, we use RxJava extensively across all components for publishing events to subscribers, streaming data, asynchronous and background execution of a task, buffering or throttling of events, periodic scheduling of a job, etc.

Scalability and Real-time Aspects

As we have decided upon the design principles and patterns to be considered (explained above), we also had to take into consideration below explained aspects in detail before making the application live in the market.

Supporting offline/poor network conditions

Since our partners spend most of their day travelling on the roads delivering orders with a smile, through tech solutions we wanted to make their workflow on the app a buttery-smooth experience by providing support even in flaky/offline network conditions. In oder to achieve this, we made our critical flows lighter as well as removed complex flows from the critical path, built retry flows, lazy sync mechanisms and bcp modes(business continuity plan).

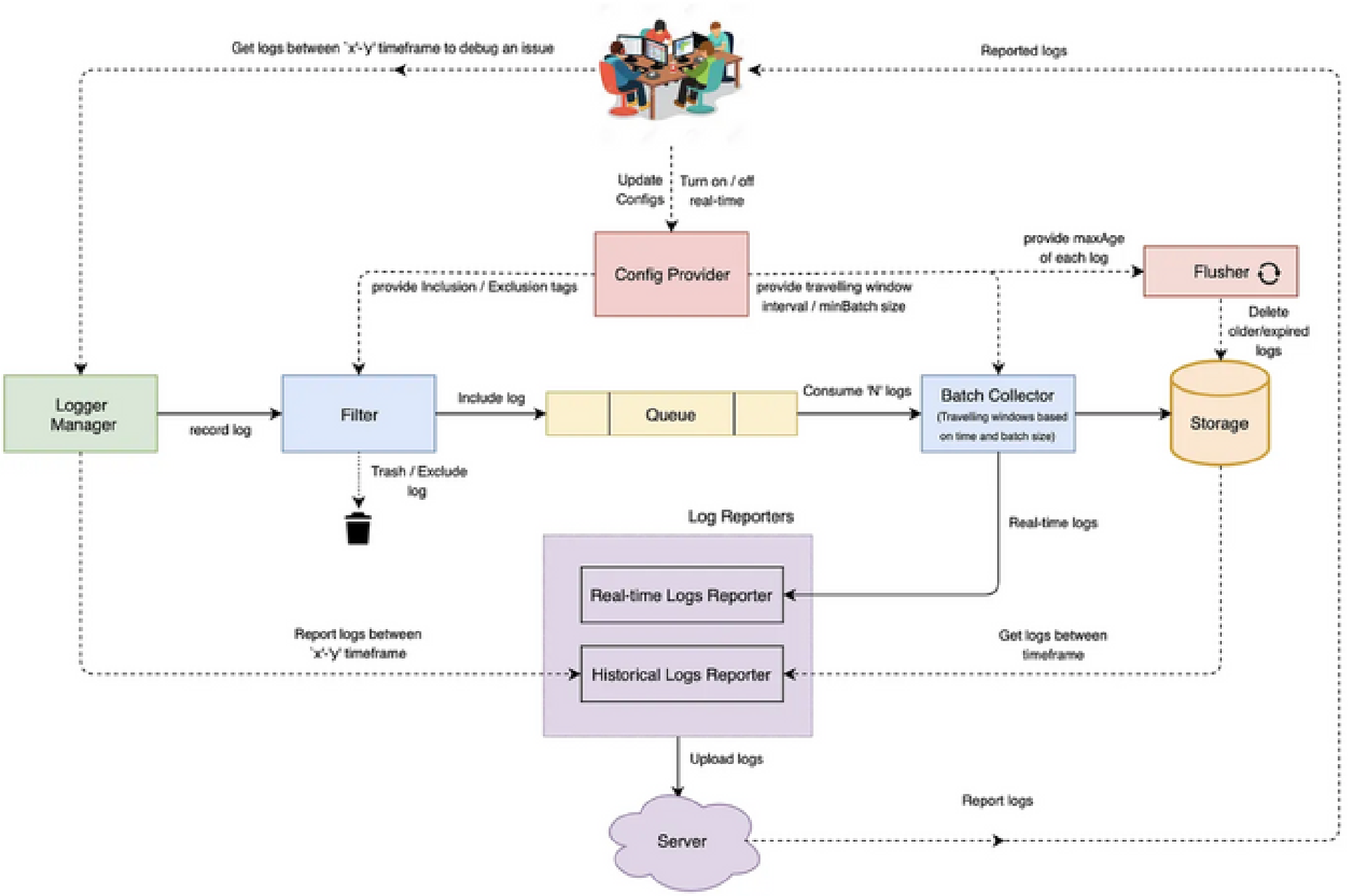


HLD for supporting offline/poor network scenarios

To make the above design more robust and use the same set up across flows/apps, we built our lazy sync module and retry framework as application agnostic which takes a configuration of a function that is to perform, metadata such as max number of attempts/retry limit, duration of the timeout, policies(exponential, linear and fibonacci...), conditions and criteria etc.

Real-time Debugging

At Swiggy, we operate at a tremendous scale during which an issue affecting one delivery partner will impact N orders/customers where N can be any number as we continue to grow at a rapid pace (in actual may be derived through parameters like demand-supply ratio, growth rate, time of the day, duration of the issue etc.). As our scale increases, our Turn around Time to identify the cause, length and breadth of an issue as well as finding the resolution should decrease to minimise the impact. For debugging issues on the mobile application, we need to analyse data such as user interactions, control/data flow, network interactions (API calls made), application state, device information, user and zone level information etc. Above data that is to be collected should also be reported in real time as time ticking is of utmost importance during an issue. To solve this requirement, we built a custom Logger solution which can be configured in real-time to report specific data of flow. Below is the high-level design of our solution.



HLD of Logger which records and reports real-time/historical logs collected

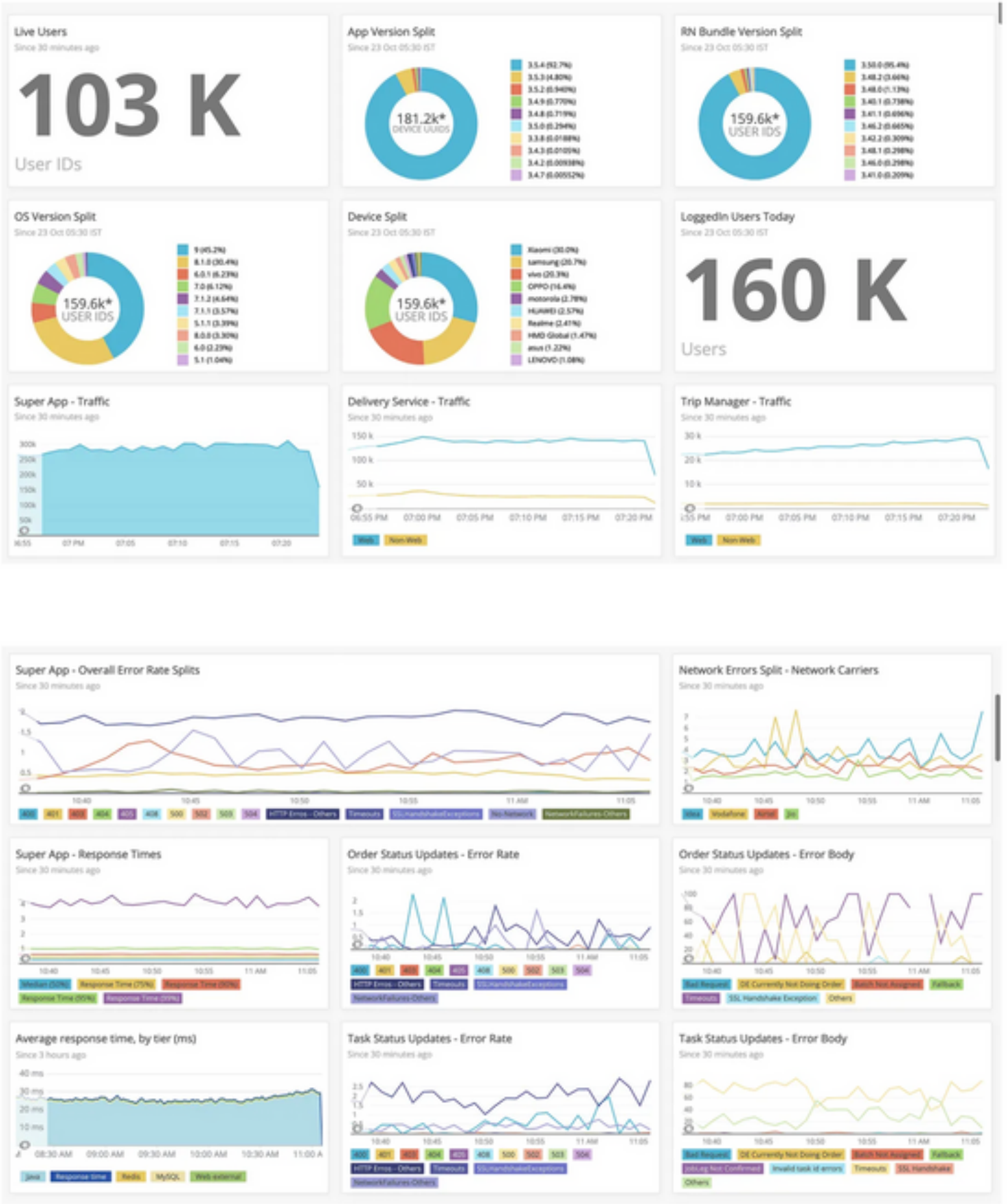
Above solution helps us collect data whenever we are notified with an alert (by PagerDuty/NewRelic/Firebase Crashlytics), reported by fault detection mechanisms baked in the app for critical flows or when an issue is reported by the Operations team from the ground. Based on the logs collected and real-time metrics (explained below) our team analyses the data quickly for faster resolution of the issue.

Real-time Health Monitoring

We operate 24x7 around the year, so it is important to keep a constant eye and stay alert on the application health all the time. So tracking metrics related to network, fatal/non-fatal errors, operational metrics each minute is crucial for our workflow. Below, we explained in-depth on how we constantly monitor each of these metrics.

Monitoring Network Metrics:

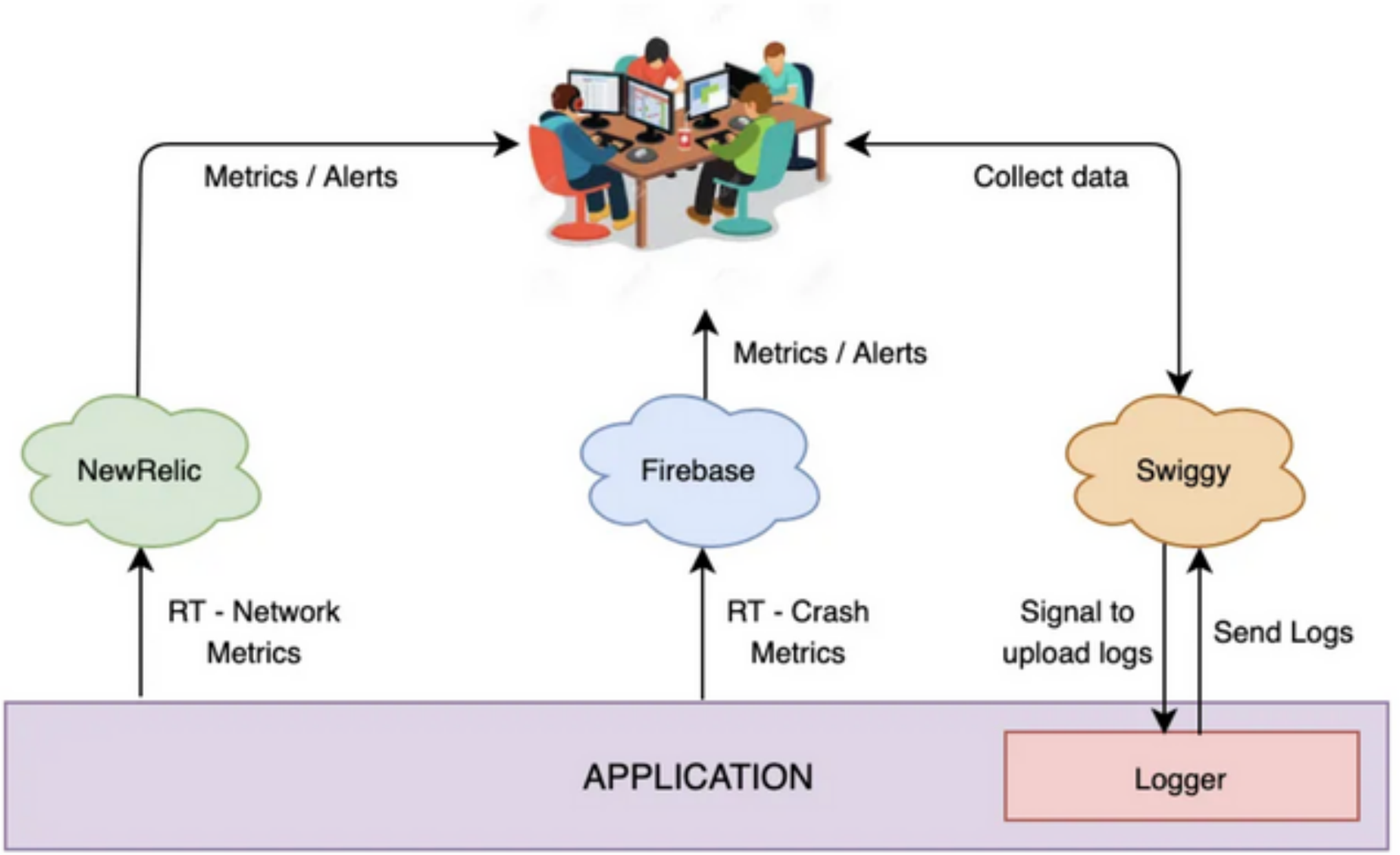
For tracking network metrics of all mobile applications across Swiggy, we've integrated NewRelic which helps in tracking info such as network error rates, traffic, response times for an API call and much more insights real-time as well as historic, provides us with alerting when any metric breaches the threshold, creating custom dashboards for a holistic picture of the application during any timeframe, etc. Below is one of the snapshots of our real-time dashboard which we created for our team to constantly keep an eye on / quickly refer during an ongoing issue.



Real-time dashboard (above are a couple of snapshots only)

Monitoring Fatal/Non-Fatal Errors:

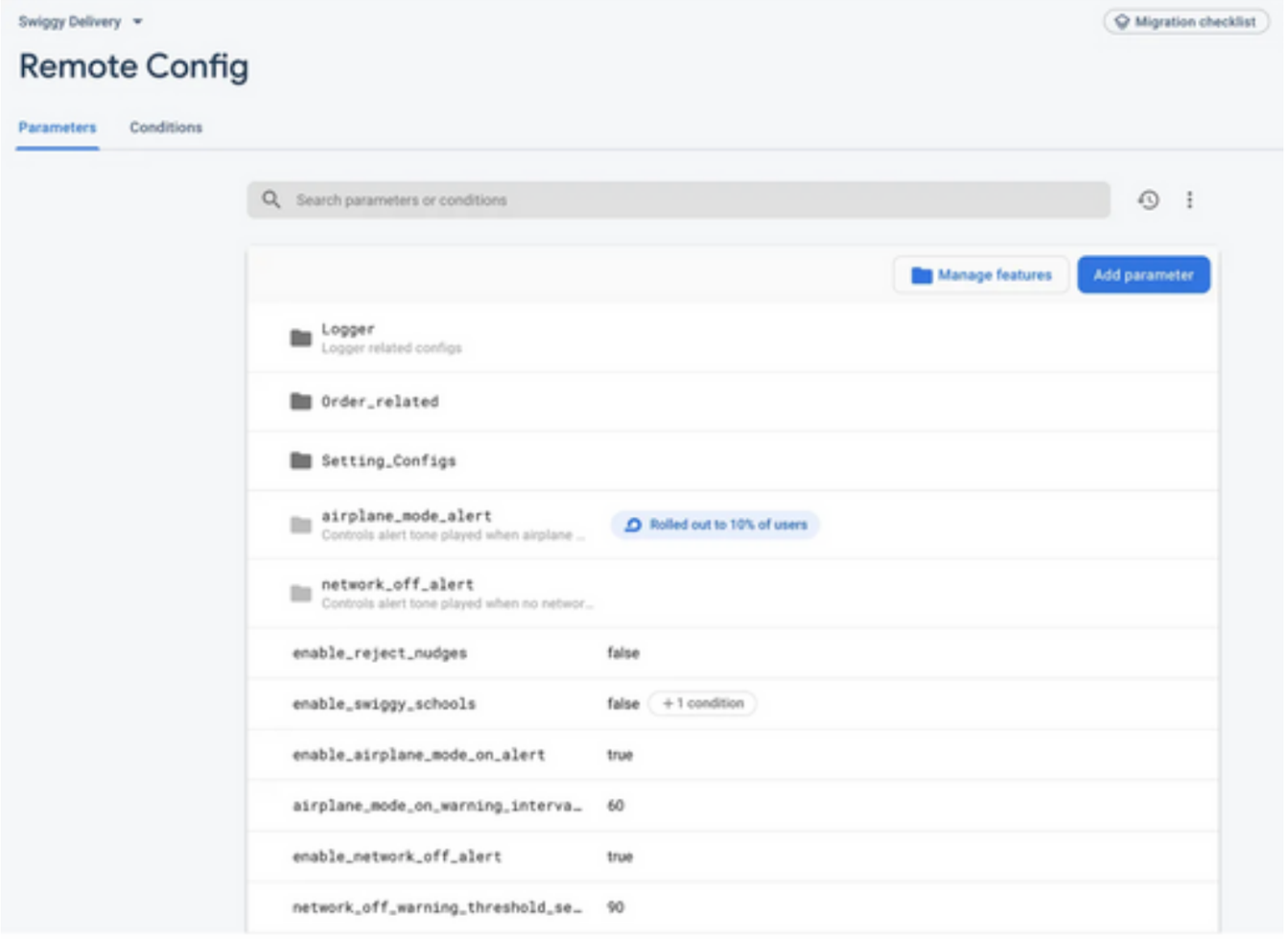
Firebase Crashlytics helps in tracking fatal and non-fatal errors along with recording vital information such as device details, user identity. It stitches firebase analytics data too on the same dashboard for each log making it easier to understand the user journey. It even offers real-time alerting based on velocity configuration through multiple channels (we configured Pager Duty and Slack for the same).



Realtime Monitoring and Debugging of the App

Configurations

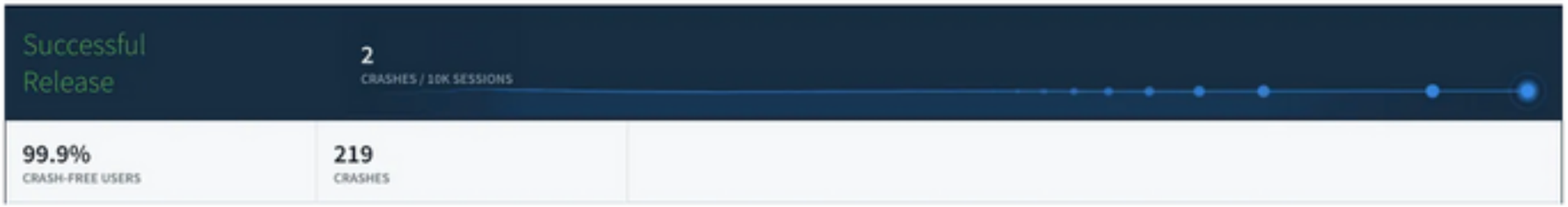
For any feature we rollout, we plan to keep a config for turning it on/off, controlling the feature to only a subset of users based on certain properties like the city, zone, os versions, devices etc. Nothing best fits our requirement than the features provided by Firebase Remote Config which gives us the flexibility to modify configs real time based on attributes such as user properties, device info etc.



Firebase remote config dashboard

Stability

As we handle one of the largest delivery fleets across India and process millions of orders daily, it is super important to be stable. So, to mitigate any risk while adopting from older version to this application, we slowly and steadily rolled out this new application to the entire fleet making changes iteratively based on the inputs collected and in the process improving stability at each update. As we are 100% live to the entire fleet, today our application is stable for 99.9% users. We were able to achieve this by sticking to our core principles, understanding the capabilities of React Native, real-time monitoring in place before GTM (go to market), setting up fallback mechanisms, BCP(business continuity plan) modes during an issue and above all a Super-Duper team handling our app.



Stability of our latest version of the App.

Design Twitter

Requirements:

- The user should be able to tweet in just a few seconds.
- The user should be able to see Tweet Timeline(s)
- Timeline: This can be divided into three parts...
- User timeline: User sees his/her own tweets and tweets user retweet. Tweets that users see when they visit their profile.
- Home Timeline: This will display the tweets from people users follow. (Tweets when you land on twitter.com)
- Search timeline: When users search some keywords or #tags and they see the tweets related to that particular keywords.
- The user should be able to follow another user.
- Users should be able to tweet millions of followers within a few seconds (5 seconds)

Naive Solution (Synchronous DB queries):

To design a big system like Twitter we will firstly talk about the Naive solution. That will help us in moving towards high-level architecture. You can design a solution for the two things:

- Data modeling: You can use a relational database like MySQL and you can consider two tables user table (id, username) and a tweet table[id, content, user(primary key of user table)]. User information will be stored in the user table and whenever a user will tweet a message it will be stored in the tweet table. Two relations are also necessary here. One is the user can follow each other, the other is each feed has a user owner. So there will be a one-to-many relationship between the user and the tweet table.
- Serve feeds: You need to fetch all the feeds from all the people a user follows and render them in chronological order.

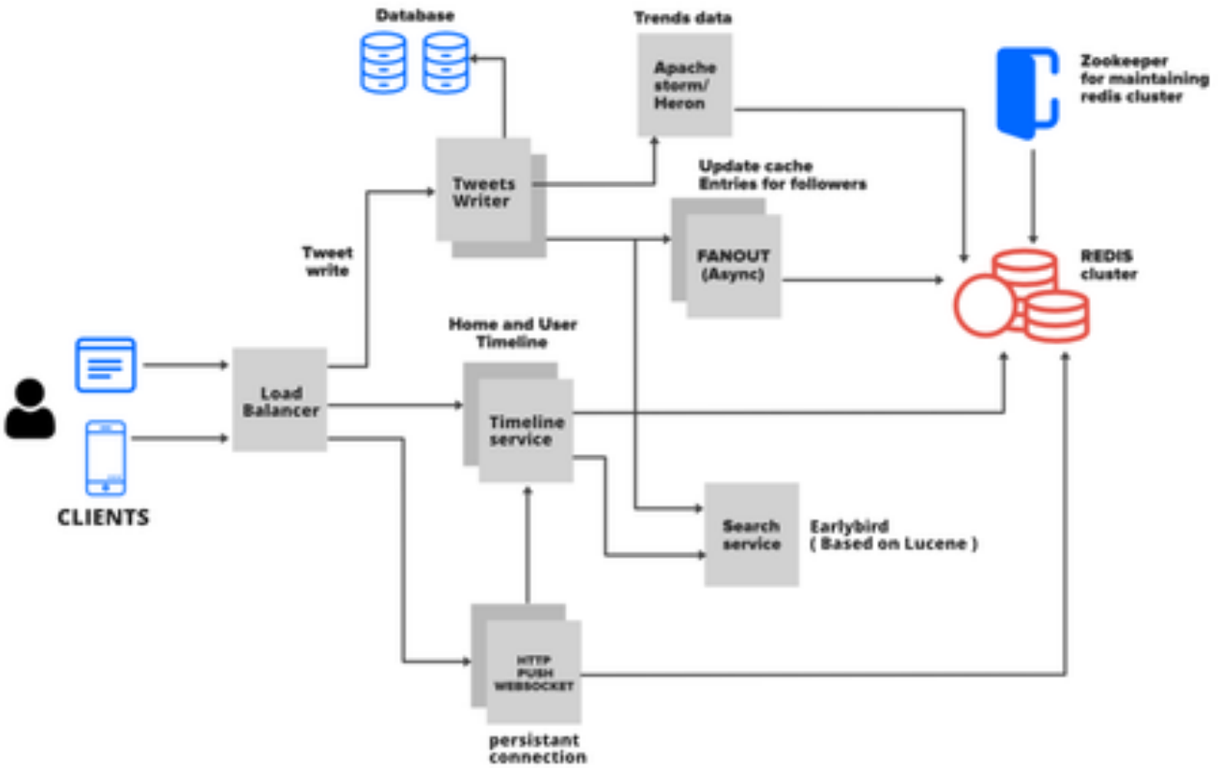
3. Limitation of Architecture (Point Out Bottleneck):

You will have to do a big select statement in the tweet table to get all the tweets for a specific user whomsoever he/she is following, and that's also in chronological order. Doing this every time will create a problem because the tweet table will have huge content with lots of tweets. We need to optimize this solution to solve this problem and for that, we will move to the high-level solution for this problem. Before that let's understand the characteristics of Twitter first.

Characteristics of Twitter (Traffic):

Twitter has 300M daily active users. On average, every second 6, 000 tweets are tweeted on Twitter. Every second 6, 00, 000 Queries are made to get the timelines. Each user has on average 200 followers and some users like some celebrities have millions of followers. This characteristic of Twitter clears the following points...

Twitter has a heavy read in comparison to writing so we should care much more about the availability and scale of the application for the heavy read on Twitter.
We can consider eventual consistency for this kind of system. It's completely ok if a user sees the tweet of his follower a bit delayed
Space is not a problem as tweets are limited to 140 characters.



High-Level Solution

As we have discussed that Twitter is read-heavy so we need a system that allows us to read the information faster and also it can scale horizontally. Redis is perfectly suitable for this requirement but we can not solely depend on Redis because we also need to store a copy of tweets and other users' related info in the Database. So here we will have the basic architecture of Twitter which consists of three tables… User Table, Tweet Table, and Followers Table.

- Whenever a user will create a profile on Twitter the entry will be stored in the User table.
- Tweets tweeted by a user will be stored in the Tweet table along with the User_id. Also, the User table will have 1 to many relationships with the Tweet table.
- When a user follows another user, it gets stored in Followers Table, and also caches it Redis. The User table will have 1 to many relationships with the Follower table.

1. User Timeline Architecture

To get the User Timeline simply go to the user table get the user_id, match this user_id in the tweet table and then get all the tweets. This will also include retweets, save retweets as tweets with original tweet references. Once this is done sort the tweet by date and time and then display the information on the user timeline.

As we have discussed that Twitter is read-heavy so the above approach won't work always. Here we need to use another layer i.e caching layer and we will save the data for user timeline queries in Redis. Also, keep saving the tweets in Redis, so when someone visits a user timeline he/she can get all the tweets made by that user. Getting the data from Redis is much faster so it's not much use to get it from DB always.

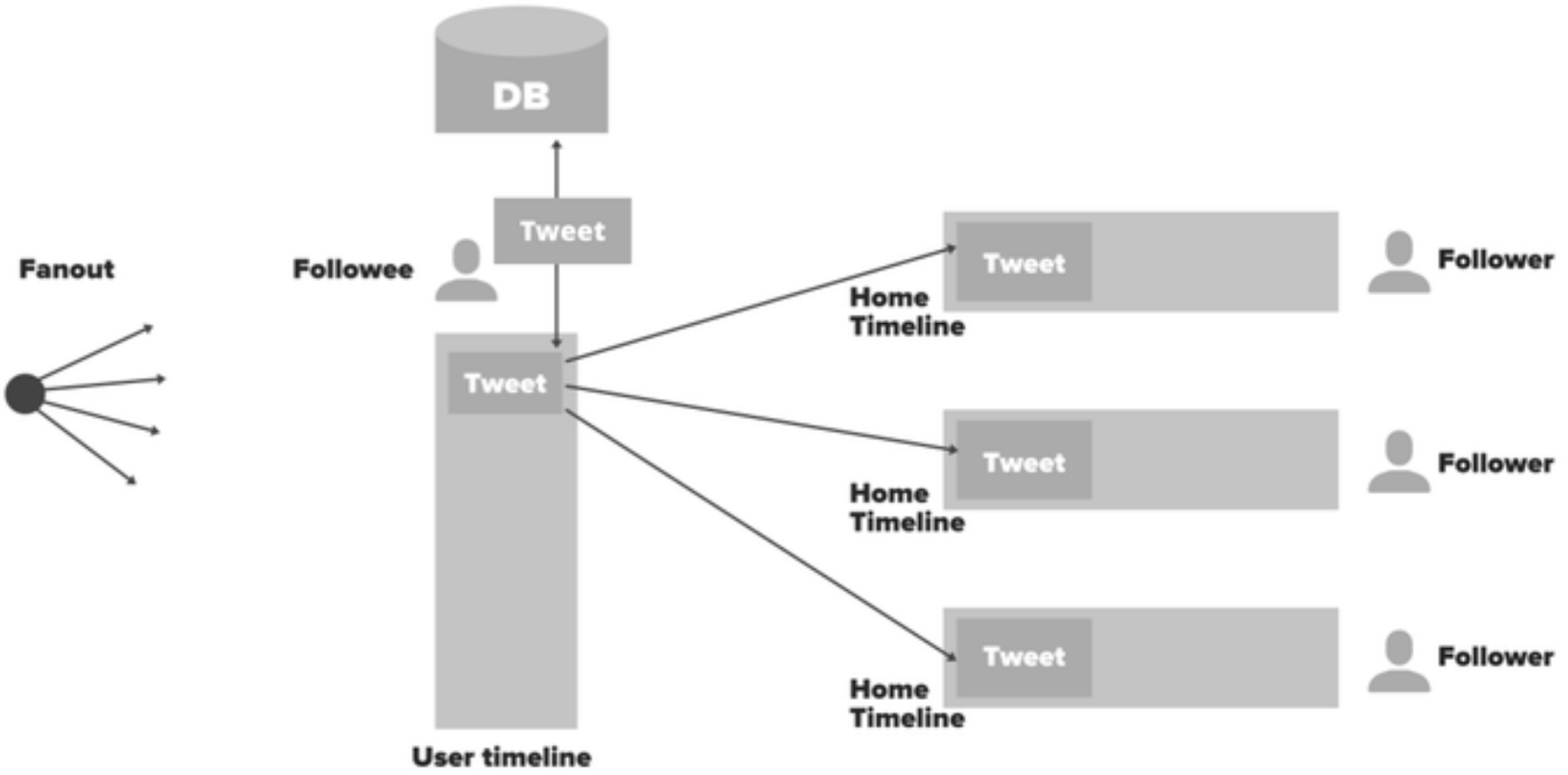
2. Home Timeline Architecture

A user's Home Timeline contains all the latest tweets of the person and the pages that the user follows. Well, here you can simply fetch the users whom a user is following, for each follower fetch all the latest tweets, then merge all the tweets, sort all these tweets by date and time and display them on the home timeline. This solution has some drawbacks. The Twitter home page loads much faster and these queries are heavier on the database so this huge search operation will take much more time once the tweet table grows to millions. Let's talk about the solution now for this drawback...

Fanout Approach: Fanout simply means spreading the data from a single point. Let's see how to use it. Whenever a tweet is made by a user (Followee) do a lot of preprocessing and distribute the data into different users (followers) home timelines. In this process, you won't have to make any database queries. You just need to go to the cache by user_id and access the home timeline data in Redis. So this process will be much faster and easier because it is an in-memory we get the list of tweets.

Here is the complete flow of this approach...

- User X is followed by three people and this user has a cache called user timeline. X Tweeted something.
- Through Load Balancer tweets will flow into back-end servers.
- Server node will save tweet in DB/cache
- The server node will fetch all the users that follow User X from the cache.
- The server node will inject this tweet into the in-memory timelines of his followers (fanout)
- All followers of User X will see the tweet of User X in their timeline. It will be refreshed and updated every time a user will visit his/her timeline.



What will happen if a celebrity will have millions of followers? Is the above method efficient in this scenario?
Weakness (Edge Case): The interviewer may ask the above question. If there is a celebrity who has millions of followers then Twitter can take up to 3-44 minutes for a tweet to flow from Eminem(a celebrity) to his million followers. You will have to update the millions of home timelines of followers which is not scalable. Here is the solution...
Solution [Mixed Approach (In-memory+Synchronous calls)]:

- Precompute the home timeline of User A (follower of Eminem) with everyone except Eminem's tweet(s)
- Every user maintains the list of celebrities in the cache as well as whom that user is following. When the request will arrive (tweet from the celebrity) you can get the celebrity from the list, fetch the tweet from the user timeline of the celebrity and then mix the celebrity's tweet at runtime with other tweets of User A.
- So when User A access his home timeline his tweet feed is merged with Eminem's tweet at load time. So the celebrity tweet will be inserted at runtime.

Other Optimization: For inactive users don't compute the timeline. People who don't log in to the system for a quite long time (say more than 20 days.).

3. Searching:

Twitter handles searching for its tweets and #tags using Earlybird which is a real-time, reverse index based on Lucene. Early Bird does an inverted full-text indexing operation. It means whenever a tweet is posted it is treated as a document. The tweet will be split into tags, words, and #tags, and then these words are indexed. This indexing is done at a big table or distributed table. In this table, each word has a reference to all the tweets which contain that particular word. Since the index is an exact string-match, unordered, it can be extremely fast. Suppose if a user searches for ‘election‘ then you will go through the table, you will find the word ‘election‘, then you’ll figure out all the references to all the tweets in the system, then it gives all the results that contain the word ‘election‘.

Twitter handles thousands of tweets per second so you can’t have just one big system or table to handle all the data so it should be handled through a distributed approach. Twitter uses the strategy scatter and gather where it set up the multiple servers or data center which allow indexing. When Twitter gets a query (let’s say #geeksforgeeks) it sends the query to all the servers or data centers and it queries every Early Bird shard. All the early bird that matches with the query return the result. The results are returned, sorted, merged, and reranked. The ranking is done based on the number of retweets, replies, and the popularity of the tweets.

Design Dropbox

1. Requirements:

- Users should be able to upload/download, update and delete the files
- File versioning (History of updates)
- File and folder sync

2. Traffic

- 12+ million unique users
- 100 million requests per day with lots of reads and write.

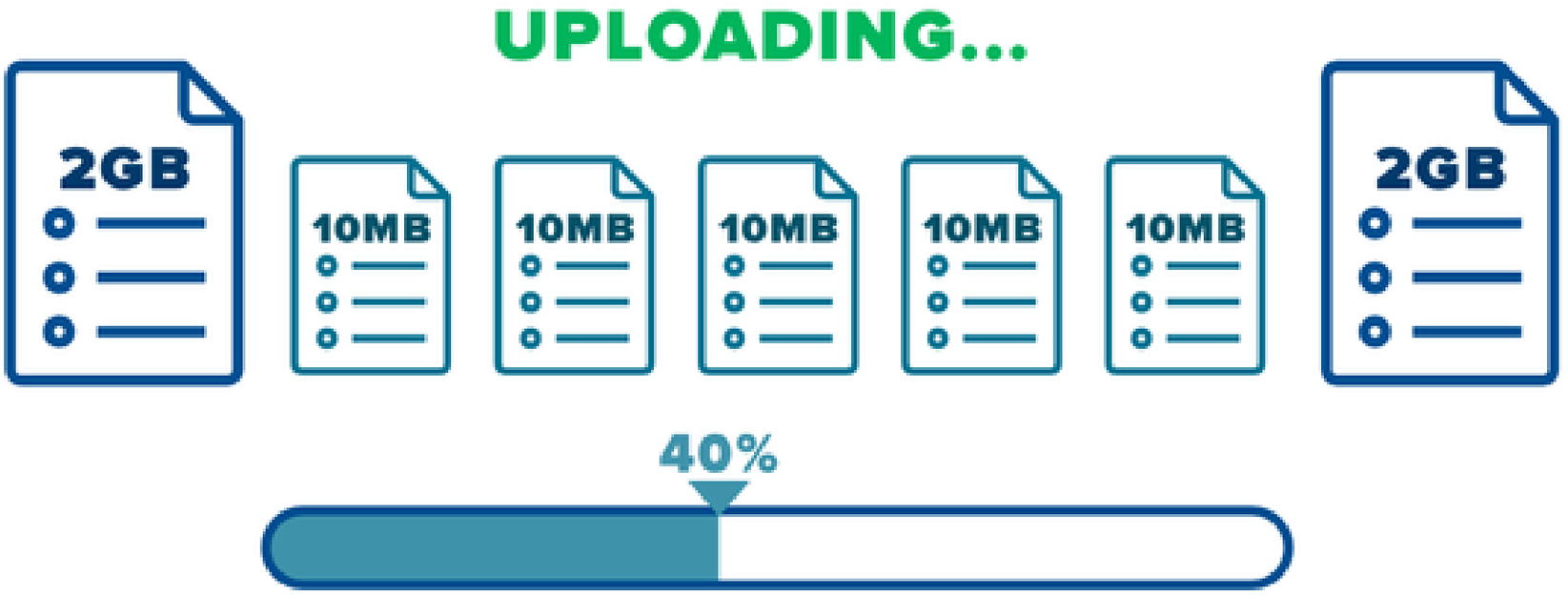
3. Problem Statement

A lot of people assume designing a dropbox is that all they just need to do is to use some cloud services, upload the file, and download the file whenever they want but that's not how it works. The core problem is "Where and how to save the files?". Suppose you want to share a file that can be of any size (small or big) and you upload it into the cloud. Everything is fine till here but later if you have to make an update in your file then it's not a good idea to edit the file and upload the whole file again and again into the cloud. The reason is...

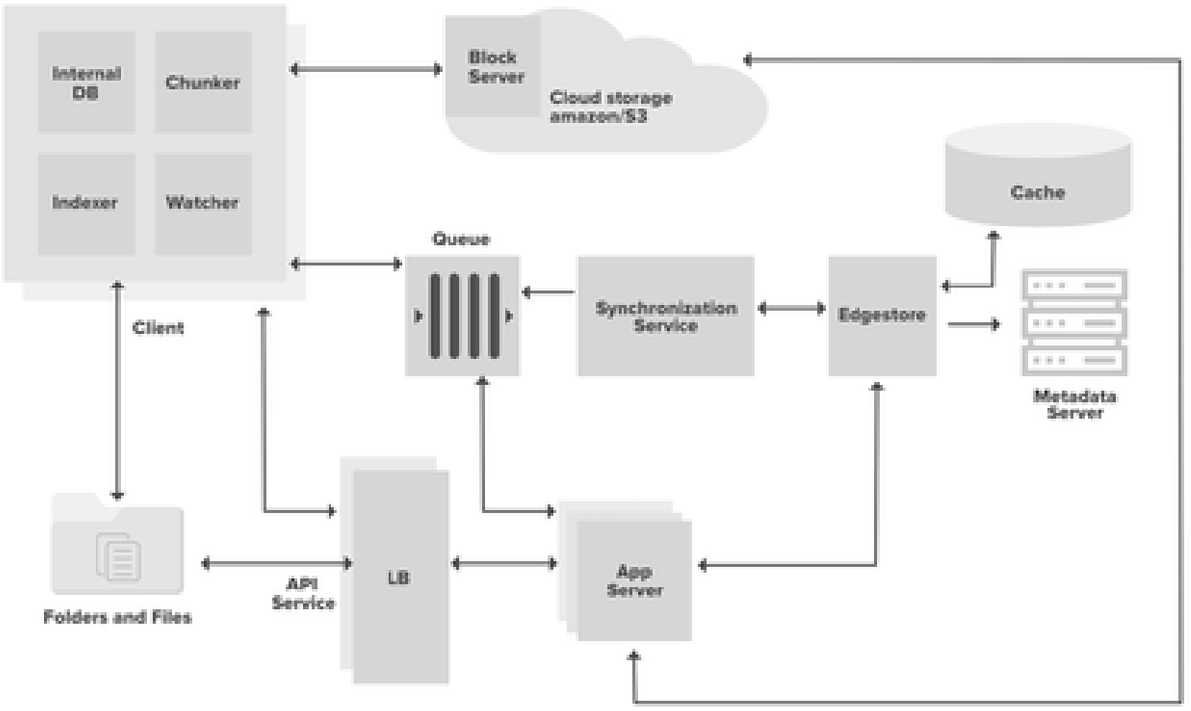
More bandwidth and cloud space utilization: To provide a history of the files you need to keep multiple versions of the files. This requires more bandwidth and more space in the cloud.

Even for the small changes in your file, you will have to back up and transfer the whole file into the cloud again and again which is not a good idea.

Latency or Concurrency Utilization: You can’t do time optimization as well. It will consume more time to upload a single file as a whole even if you make small changes in your file. It’s also not possible to make use of concurrency to upload/download the files using multi threads or multi processes.



We can break the files into multiple chunks to overcome the problem we discussed above. There is no need to upload/download the whole single file after making any changes in the file. You just need to save the chunk which is updated (this will take less memory and time). It will be easier to keep the different versions of the files in various chunks. We have considered one file which is divided into various chunks. If there are multiple files then we need to know which chunks belong to which file. To keep this information we will create one more file named a metadata file. This file contains the indexes of the chunks (chunk names and order information). You need to mention the hash of the chunks (or some reference) in this metadata file and you need to sync this file into the cloud. We can download the metadata file from the cloud whenever we want and we can recreate the file using various chunks. Now let's talk about the various components for the complete system design solution of the dropbox service.



Let’s assume we have a client installed on our computer (an app installed on your computer) and this client has 4 basic components. These basic components are Watcher, Chunker, Indexer, and Internal DB. We have considered only one client but there can be multiple clients belonging to the same user with the same basic components.

- The client is responsible for uploading/downloading the files, identifying the file changes in the sync folder, and handling conflicts due to offline or concurrent updates.
- The client is actively monitoring the folders for all the updates or changes happening in the files.
- To handle file metadata updates (e.g. file name, size, modification date, etc.) this client interacts with the Messaging services and Synchronization Service.
- It also interacts with remote cloud storage (Amazon S3 or any other cloud services) to store the actual files and to provide folder synchronization.

Discuss The Client Components

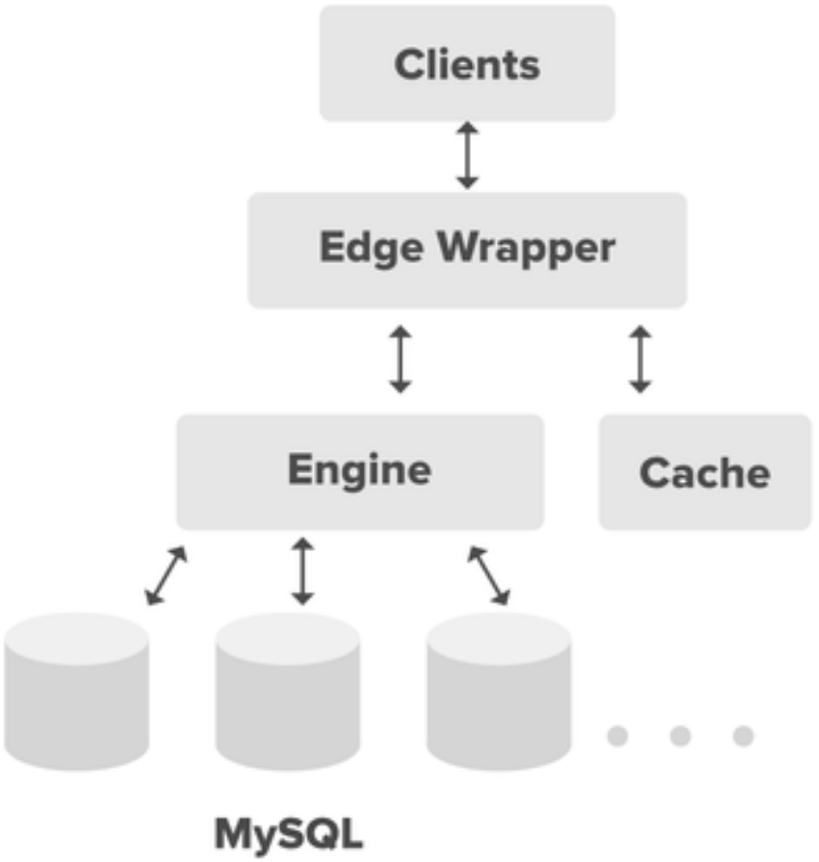
- Watcher is responsible for monitoring the sync folder for all the activities performed by the user such as creating, updating, or deleting files/folders. It gives notification to the indexer and chunker if any action is performed in the files or folders.
- Chunker breaks the files into multiple small pieces called chunks and uploads them to the cloud storage with a unique id or hash of these chunks. To recreate the files these chunks can be joined together. For any changes in the files, the chunking algorithm detects the specific chunk which is modified and only saves that specific part/chunk to the cloud storage. It reduces the bandwidth usage, synchronization time, and storage space in the cloud.
- Indexer is responsible for updating the internal database when it receives the notification from the watcher (for any action performed in folders/files). It receives the URL of the chunks from the chunker along with the hash and updates the file with modified chunks. Indexer communicates with the Synchronization Service using the Message Queuing Service, once the chunks are successfully submitted to the cloud Storage.
- Internal databases stores all the files and chunks of information, their versions, and their location in the file system.

Discuss The Other Components

1. Metadata Database

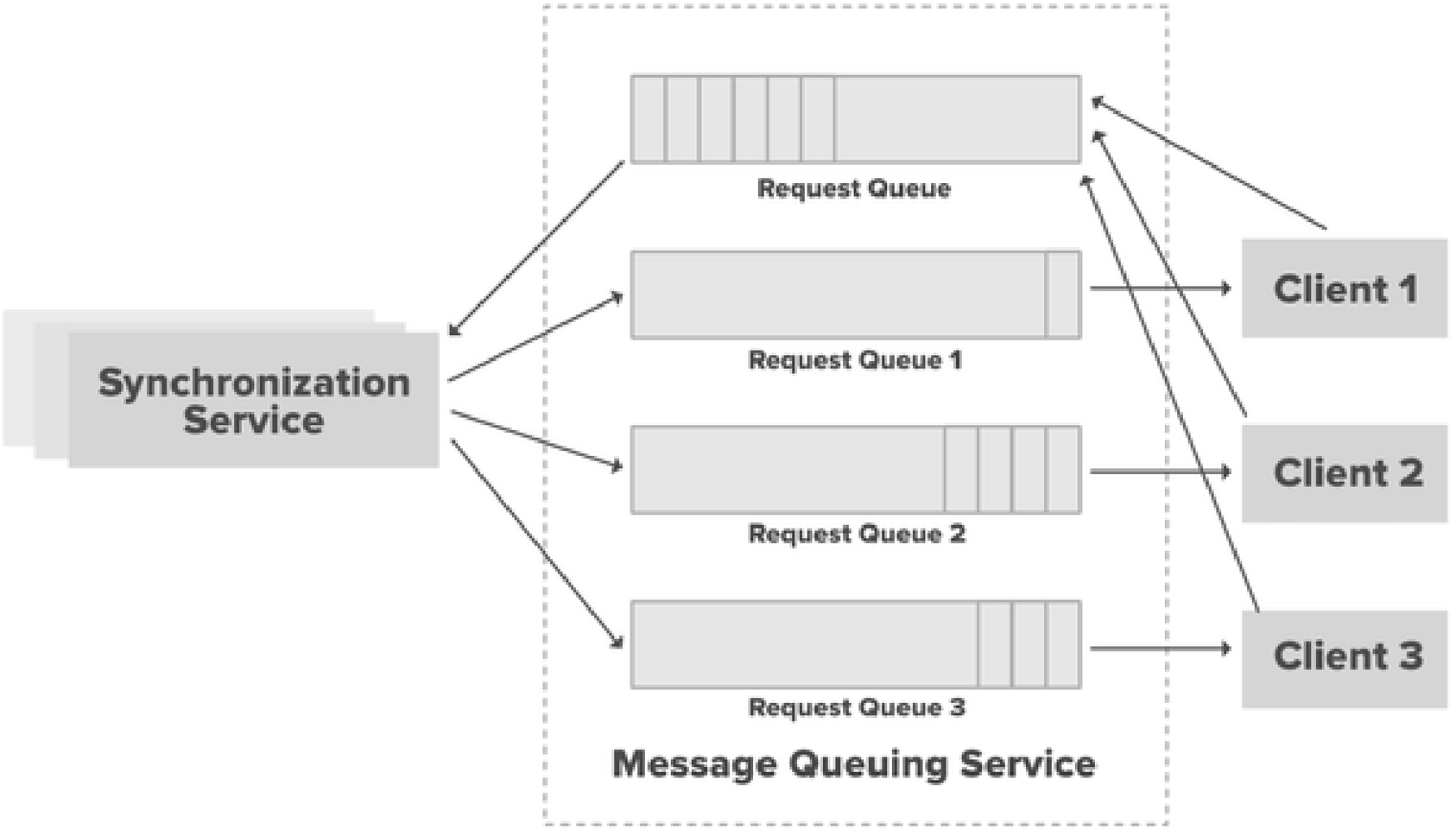
The metadata database maintains the indexes of the various chunks. The information contains files/chunks names, and their different versions along with the information of users and workspace. You can use RDBMS or NoSQL but make sure that you meet the data consistency property because multiple clients will be working on the same file. With RDBMS there is no problem with the consistency but with NoSQL, you will get eventual consistency. If you decide to use NoSQL then you need to do different configurations for different databases (For example, the Cassandra replication factor gives the consistency level). Relational databases are difficult to scale so if you're using the MySQL database then you need to use a database sharding technique (or master-slave technique) to scale the application. In database sharding, you need to add multiple MySQL databases but it will be difficult to manage these databases for any update or for any new information that will be added to the databases. To overcome this problem we need to build an edge wrapper around the sharded databases. This edge wrapper provides the ORM and the client can easily use this edge wrapper's ORM to interact with the database (instead of interacting with the databases directly).

Metadata



2. Message Queuing Service

The messaging service queue will be responsible for the asynchronous communication between the clients and the synchronization service.



Below are the main requirements of the Message Queuing Service.

- Ability to handle lots of reading and writing requests.
- Store lots of messages in a highly available and reliable queue.
- High performance and high scalability.
- Provides load balancing and elasticity for multiple instances of the Synchronization Service.

There will be two types of messaging queues in the service.

- Request Queue: This will be a global request queue shared among all the clients. Whenever a client receives any update or changes in the files/folder it sends the request through the request queue. This request is received by the synchronization service to update the metadata database.
- Response Queue: There will be an individual response queue corresponding to the individual clients. The synchronization service broadcast the update through this response queue and this response queue will deliver the updated messages to each client and then these clients will update their respective files accordingly. The message will never be lost even if the client will be disconnected from the internet (the benefit of using the messaging queue service). We are creating n number of response queues for n number of clients because the message will be deleted from the queue once it will be received by the client and we need to share the updated message with the various subscribed clients.

3. Synchronization Service

The client communicates with the synchronization services either to receive the latest update from the cloud storage or to send the latest request/updates to the Cloud Storage. The synchronization service receives the request from the request queue of the messaging services and updates the metadata database with the latest changes. Also, the synchronization service broadcast the latest update to the other clients (if there are multiple clients) through the response queue so that the other client's indexer can fetch back the chunks from the cloud storage and recreate the files with the latest update. It also updates the local database with the information stored in the Metadata Database. If a client is not connected to the internet or offline for some time, it polls the system for new updates as soon as it goes online.

4. Cloud Storage

You can use any cloud storage service like Amazon S3 to store the chunks of the files uploaded by the user. The client communicates with the cloud storage for any action performed in the files/folders using the API provided by the cloud provider. A lot of candidates get afraid of this round more than the coding round because they don't get the idea that what topics and tradeoffs they should cover within this limited timeframe. Firstly, remember that the system design round is extremely open-ended and there's no such thing as a standard answer. Even for the same question (System Design Dropbox), you'll have a different discussion with different interviewers.

Design a Rate Limiter API

Description:

A Rate Limiter API is a tool that developers can use to define rules that specify how many requests can be made in a given time period and what actions should be taken when these limits are exceeded.

Rate limiting is an essential technique used in software systems to control the rate of incoming requests. It helps to prevent the overloading of servers by limiting the number of requests that can be made in a given time frame.

It helps to prevent a high volume of requests from overwhelming a server or API. Here is a basic design for a rate limiter API In this article, we will discuss the design of a rate limiter API, including its requirements, high-level design, and algorithms used for rate limiting.

Why is rate limiting used?
Avoid resource starvation due to a Denial of Service (DoS) attack.
Ensure that servers are not overburdened. Using rate restriction per user
ensures fair and reasonable use without harming other users.
Control the flow of information, for example, prevent a single worker from
accumulating a backlog of unprocessed items while other workers are idle.

Functional requirements to Design a Rate Limiter API:

- The API should allow the definition of multiple rate-limiting rules.
- The API should provide the ability to customize the response to clients when rate limits are exceeded.
- The API should allow for the storage and retrieval of rate-limit data.
- The API should be implemented with proper error handling as in when the threshold limit of requests are crossed for a single server or across different combinations, the client should get a proper error message.

Non-functional requirements to Design a Rate Limiter API:

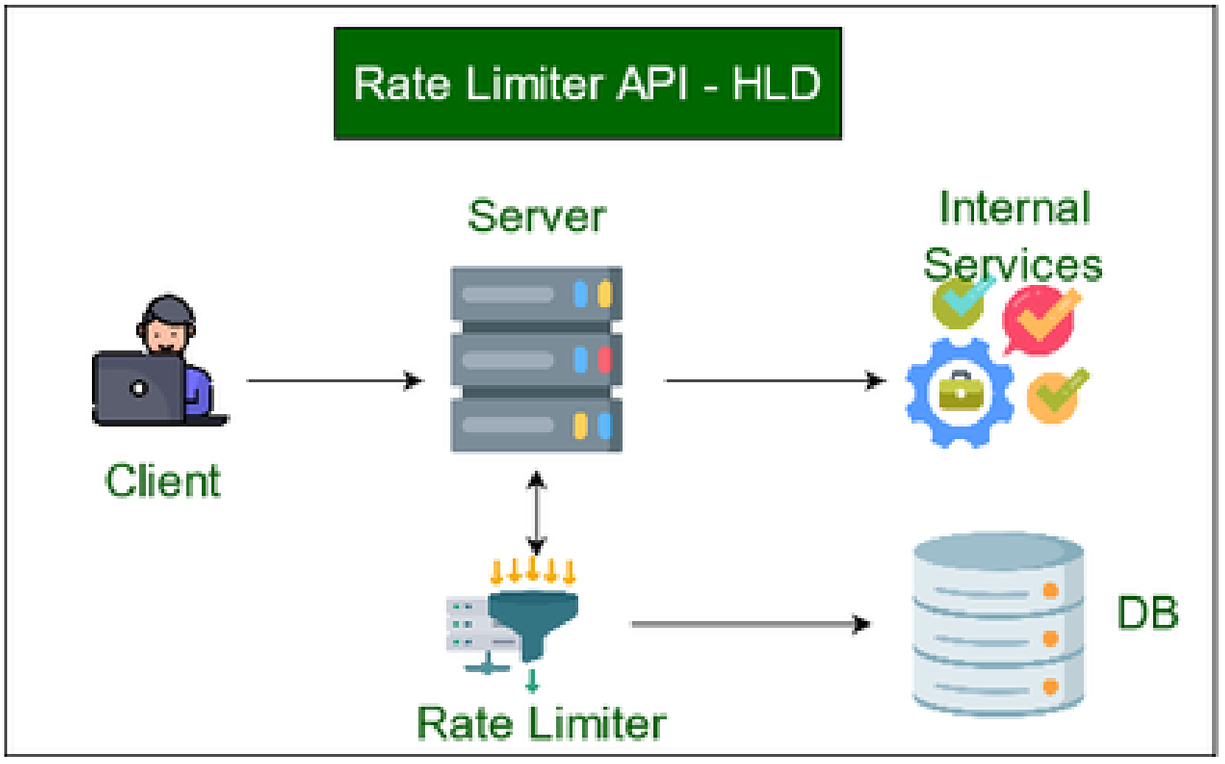
- The API should be highly available and scalable. Availability is the main pillar in the case of request fetching APIs.
- The API should be secure and protected against malicious attacks.
- The API should be easy to integrate with existing systems.
- There should be low latency provided by the rate limiter to the system, as performance is one of the key factors in the case of any system.

High Level Design (HLD) to Design a Rate Limiter API:

Where to place the Rate Limiter – Client Side or Server Side?

A rate limiter should generally be implemented on the server side rather than on the client side. This is because of the following points:

- Positional Advantage: The server is in a better position to enforce rate limits across all clients, whereas client-side rate limiting would require every client to implement their own rate limiter, which would be difficult to coordinate and enforce consistently.
- Security: Implementing rate limiting on the server side also provides better security, as it allows the server to prevent malicious clients from overwhelming the system with a large number of requests. If rate limiting were implemented on the client side, it would be easier for attackers to bypass the rate limit by just modifying or disabling the client-side code.
- Flexible: Server-side rate limiting allows more flexibility in adjusting the rate limits and managing resources. The server can dynamically adjust the rate limits based on traffic patterns and resource availability, and can also prioritize certain types of requests or clients over others. Thus, lends to better utilization of available resources, and also keeps performance good.



The overall basic structure of a rate limiter seems relatively simpler. We just need a counter associated with each user to track how many requests are being same submitted in a particular timeframe. The request is rejected if the counter value hits the limit.

Memory Structure/Approximation:

Thus, now let's think of the data structure which might help us. Since we need fast retrieval of the counter values associated with each user, we can use a hash-table. Considering we have a key-value pair. The key would contain hash value of each User Id, and the corresponding value would be the pair or structure of counter and the startTime, e.g., UserId -> {counter, startTime}

Now, each UserId let's say takes 8 bytes(long long) and the counter takes 2 bytes(int), which for now can count to 50k(limit). Now for the time if we store only the minute and seconds, it will also take 2 bytes. So in total, we would need 12 bytes to store each user's data.

Now considering the overhead of 10 bytes for each record in our hash-table, we would be needing to track at least 5 million users at any time(traffic), so the total memory in need would be:
(12+10)bytes*5 million = 110 MB

Key Components in the Rate Limiter:

- Define the rate limiting policy: The first step is to determine the policy for rate limiting. This policy should include the maximum number of requests allowed per unit of time, the time window for measuring requests, and the actions to be taken when a limit is exceeded (e.g., return an error code or delay the request).
- Store request counts: The rate limiter API should keep track of the number of requests made by each client. One way to do this is to use a database, such as Redis or Cassandra, to store the request counts.
- Identify the client: The API must identify each client that makes a request. This can be done using a unique identifier such as an IP address or an API key.
- Handle incoming requests: When a client makes a request, the API should first check if the client has exceeded their request limit within the specified time window.

If the limit has been reached, the API can take the action specified in the rate-limiting policy (e.g., return an error code). If the limit has not been reached, the API should update the request count for the client and allow the request to proceed.

- Set headers: When a request is allowed, the API should set appropriate headers in the response to indicate the remaining number of requests that the client can make within the time window, as well as the time at which the limit will be reset.
- Expose an endpoint: Finally, the rate limiter API should expose an endpoint for clients to check their current rate limit status. This endpoint can return the number of requests remaining within the time window, as well as the time at which the limit will be reset.

Where should we keep the counters?

Due to the slowness of Database operations, it is not a smart option for us. This problem can be handled by an in-memory cache such as Redis. It is quick and supports the already implemented time-based expiration technique.

We can rely on two commands being used with in-memory storage,

INCR: This is used for increasing the stored counter by 1.
EXPIRE: This is used for setting the timeout on the stored counter. This counter is automatically deleted from the storage when the timeout expires.

In this design, client requests pass through a rate limiter middleware, which checks against the configured rate limits. The rate limiter module stores and retrieves rate limit data from a backend storage system. If a client exceeds a rate limit, the rate limiter module returns an appropriate response to the client.

Algorithms to Design a Rate Limiter API:

Several algorithms are used for rate limiting, including

The Token bucket,
Leaky bucket,
Sliding window logs, and
Sliding window counters.

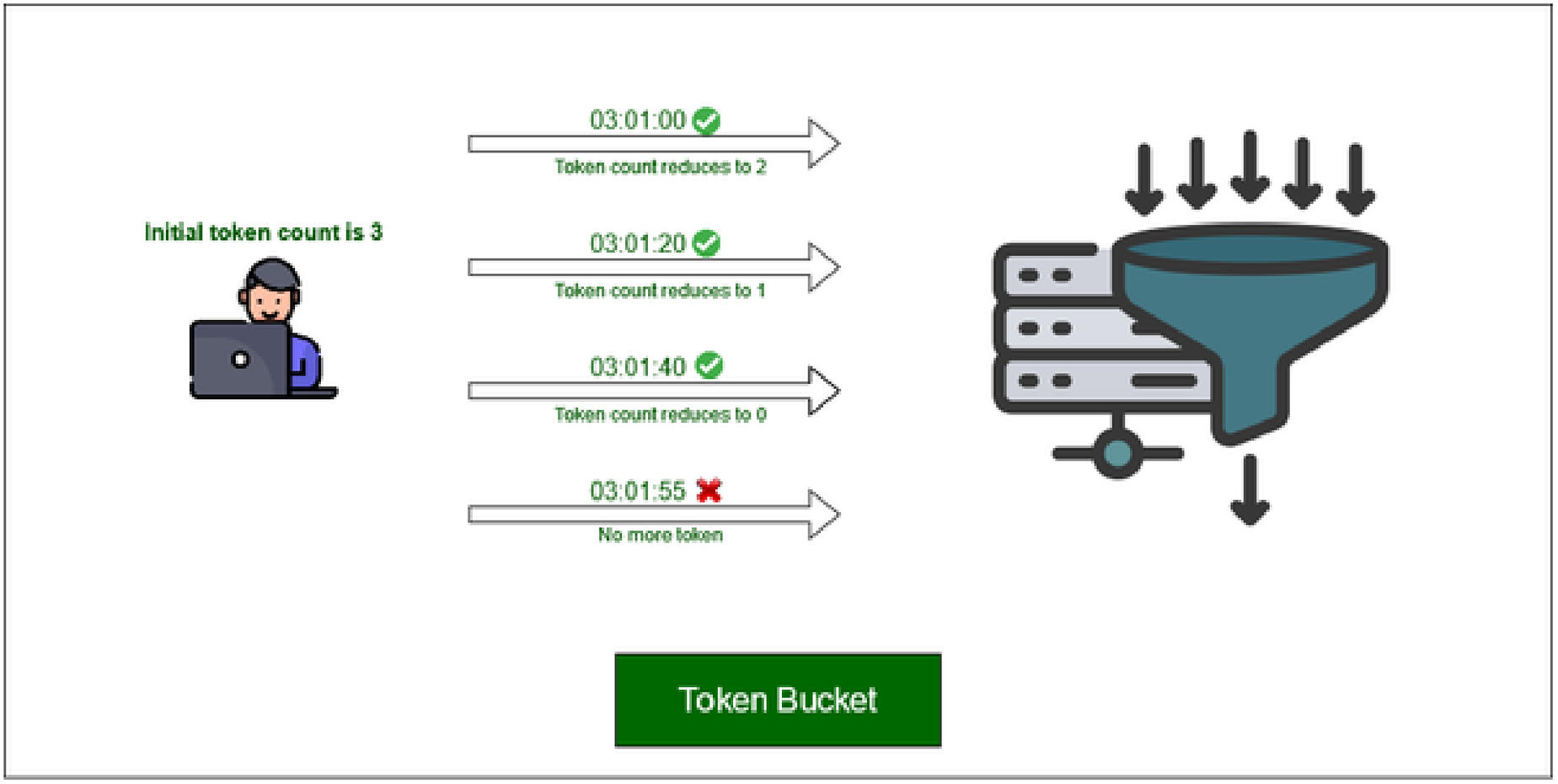
Token Bucket:

The token bucket algorithm is a simple algorithm that uses a fixed-size token bucket to limit the rate of incoming requests. The token bucket is filled with tokens at a fixed rate, and each request requires a token to be processed. If the bucket is empty, the request is rejected.

The token bucket algorithm can be implemented using the following steps:

- Initialize the token bucket with a fixed number of tokens.
- For each request, remove a token from the bucket.
- If there are no tokens left in the bucket, reject the request.
- Add tokens to the bucket at a fixed rate.

Thus, by allocating a bucket with a predetermined number of tokens for each user, we are successfully limiting the number of requests per user per time unit. When the counter of tokens comes down to 0 for a certain user, we know that he or she has reached the maximum amount of requests in a particular timeframe. The bucket will be auto-refilled whenever the new timeframe starts.



Token bucket example with initial bucket token count of 3 for each user in one minute

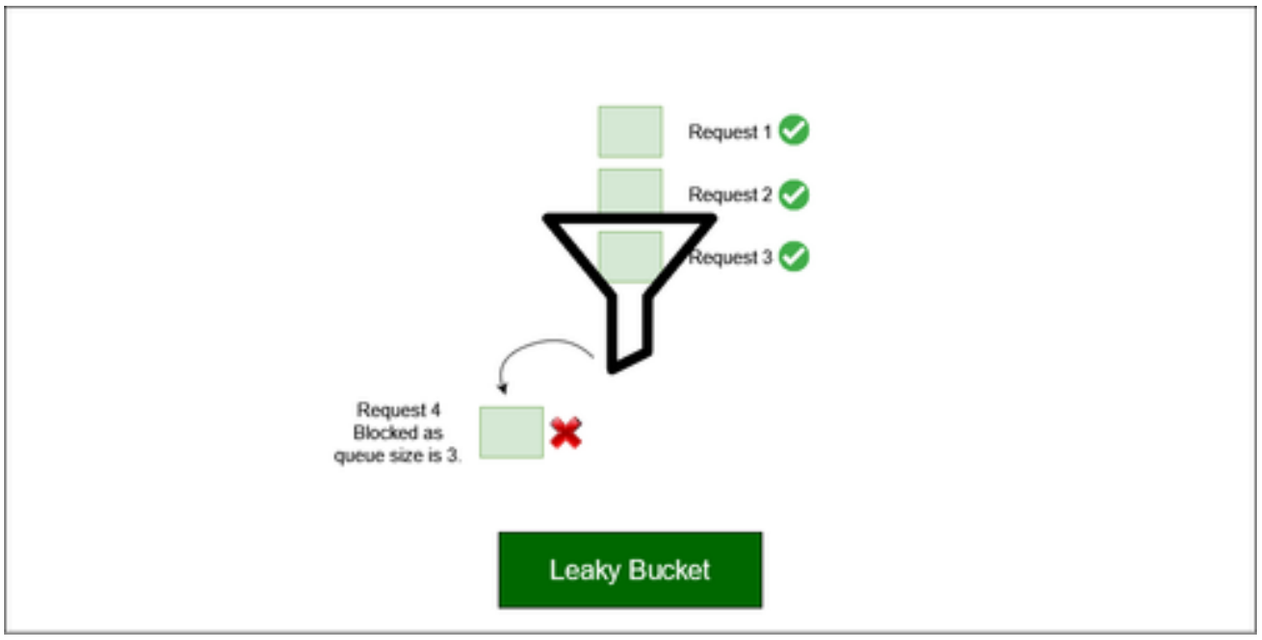
Leaky Bucket:
It is based on the idea that if the average rate at which water is poured exceeds the rate at which the bucket leaks, the bucket will overflow.

The leaky bucket algorithm is similar to the token bucket algorithm, but instead of using a fixed-size token bucket, it uses a leaky bucket that empties at a fixed rate. Each incoming request adds to the bucket's depth, and if the bucket overflows, the request is rejected.

One way to implement this is using a queue, which corresponds to the bucket that will contain the incoming requests. Whenever a new request is made, it is added to the queue's end. If the queue is full at any time, then the additional requests are discarded.

The leaky bucket algorithm can be separated into the following concepts:

- Initialize the leaky bucket with a fixed depth and a rate at which it leaks.
- For each request, add to the bucket's depth.
- If the bucket's depth exceeds its capacity, reject the request.
- Leak the bucket at a fixed rate.



Leaky bucket example with token count per user per minute is 3, which is the queue size.

Sliding Window Logs:

Another approach to rate limiting is to use sliding window logs. This data structure involves a “window” of fixed size that slides along a timeline of events, storing information about the events that fall within the window at any given time.

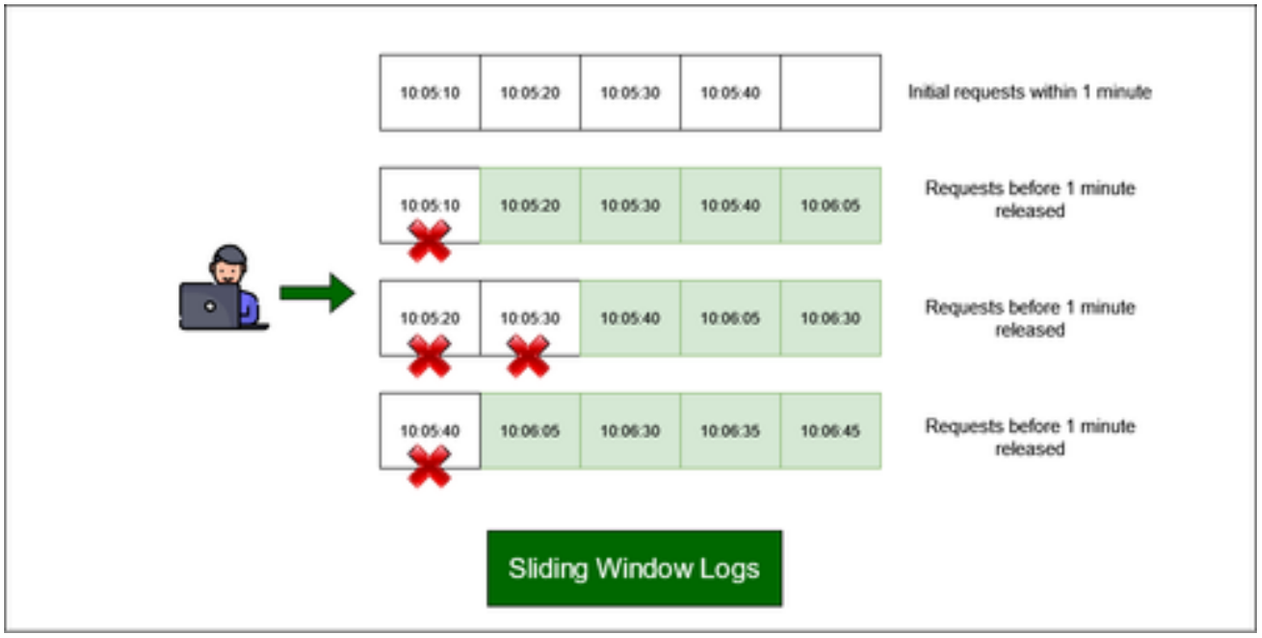
The window can be thought of as a buffer of limited size that holds the most recent events or changes that have occurred. As new events or changes occur, they are added to the buffer, and old events that fall outside of the window are removed. This ensures that the buffer stays within its fixed size, and only contains the most recent events.

This rate limitation keeps track of each client’s request in a time-stamped log.

These logs are normally stored in a time-sorted hash set or table.

The sliding window logs algorithm can be implemented using the following steps:

A time-sorted queue or hash table of timestamps within the time range of the most recent window is maintained for each client making the requests.
When a certain length of the queue is reached or after a certain number of minutes, whenever a new request comes, a check is done for any timestamps older than the current window time.
The queue is updated with new timestamp of incoming request and if number of elements in queue does not exceed the authorised count, it is proceeded otherwise an exception is triggered.



Sliding window logs in a timeframe of 1 minute

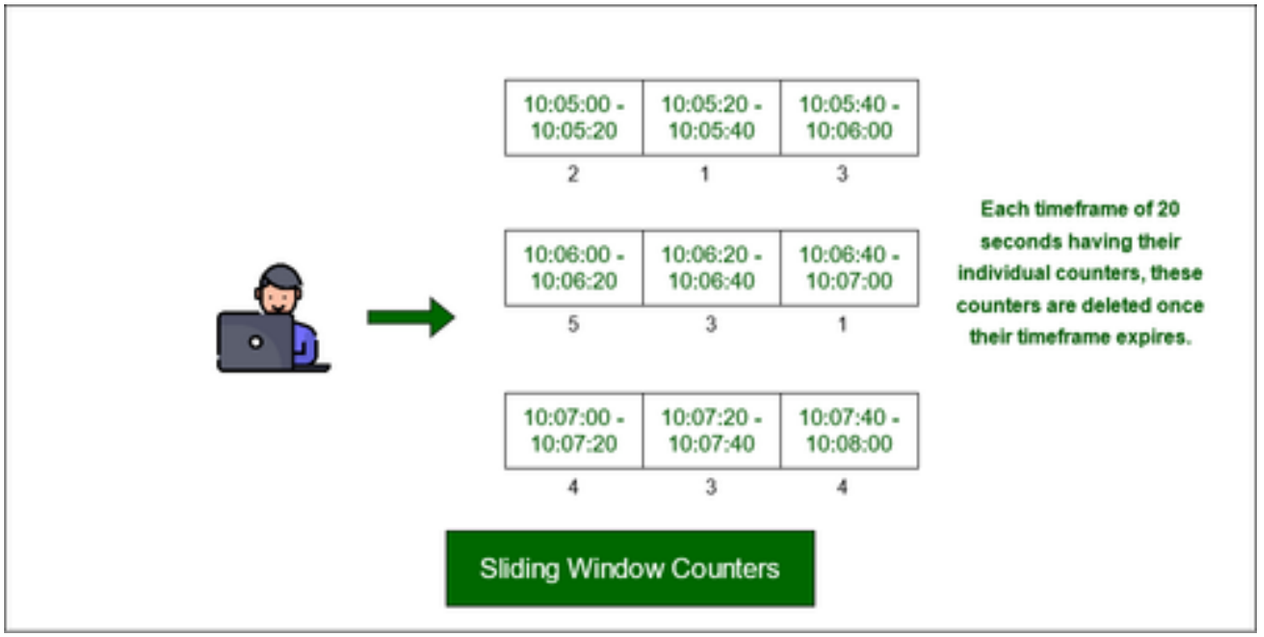
Sliding Window Counters:

The sliding window counter algorithm is an optimization over sliding window logs. As we can see in the previous approach, memory usage is high. For example, to manage numerous users or huge window timeframes, all the request timestamps must be kept for a window time, which eventually uses a huge amount of memory. Also, removing numerous timestamps older than a particular timeframe means high complexity of time as well.

To reduce surges of traffic, this algorithm accounts for a weighted value of the previous window's request based on timeframe. If we have a one-minute rate limit, we can record the counter for each second and calculate the sum of all counters in the previous minute whenever we get a new request to determine the throttling limit.

The sliding window counters can be separated into the following concepts:

Remove all counters which are more than 1 minute old.
If a request comes which falls in the current bucket, the counter is increased.
If a request comes when the current bucket has reached it's throat limit, the request is blocked.



sliding window counters with a timeframe of 20 seconds

Examples of Rate Limiting APIs used worldwide:

- Google Cloud Endpoints: It is a platform for building APIs that includes a built-in rate limiter to help prevent excessive API usage.
- AWS API Gateway: Amazon Web Services (AWS) API Gateway includes a feature called Usage Plans that allows for rate limiting and throttling of API requests.
- Akamai API Gateway: Akamai API Gateway is a cloud-based platform that includes a rate limiter feature for controlling API requests.
- Cloudflare Rate Limiting: Cloudflare's Rate Limiting feature helps prevent DDoS attacks and other types of abusive traffic by limiting the number of requests that can be made to an API.
- Redis: It is an in-memory data structure store that can be used as a database, cache, and message broker. It includes several features that make it useful for implementing a rate limiter, such as its ability to store data in memory for fast access and its support for atomic operations.

Load Balancing Algorithms

A load balancer is a critical component in system design, as it is responsible for distributing incoming requests to multiple servers to ensure that no single server becomes overwhelmed and that the overall system performance remains optimal.

In a system design interview, a candidate may be asked to design a load balancing system, or explain how they would integrate a load balancer into an existing system. When designing a load balancing system, the following key considerations should be taken into account:

- Load balancing algorithm: The algorithm used to distribute incoming requests to the servers, such as round-robin, least connections, or IP hash.
- Health checks: How the load balancer will determine the health of each server and ensure that only healthy servers receive incoming requests.
- Session persistence: How the load balancer will ensure that subsequent requests from the same user are directed to the same server, to maintain session state.
- Scalability: How the load balancing system will scale as the number of incoming requests increases, for example by adding more load balancers or using a more scalable load balancing algorithm.
- High availability: How the load balancing system will ensure that it remains available even if one or more components fail, for example by using a redundant load balancing setup or by employing a load balancing service provided by a cloud provider.

- Performance: How the load balancing system will impact the overall performance of the system, including the latency and throughput of incoming requests.

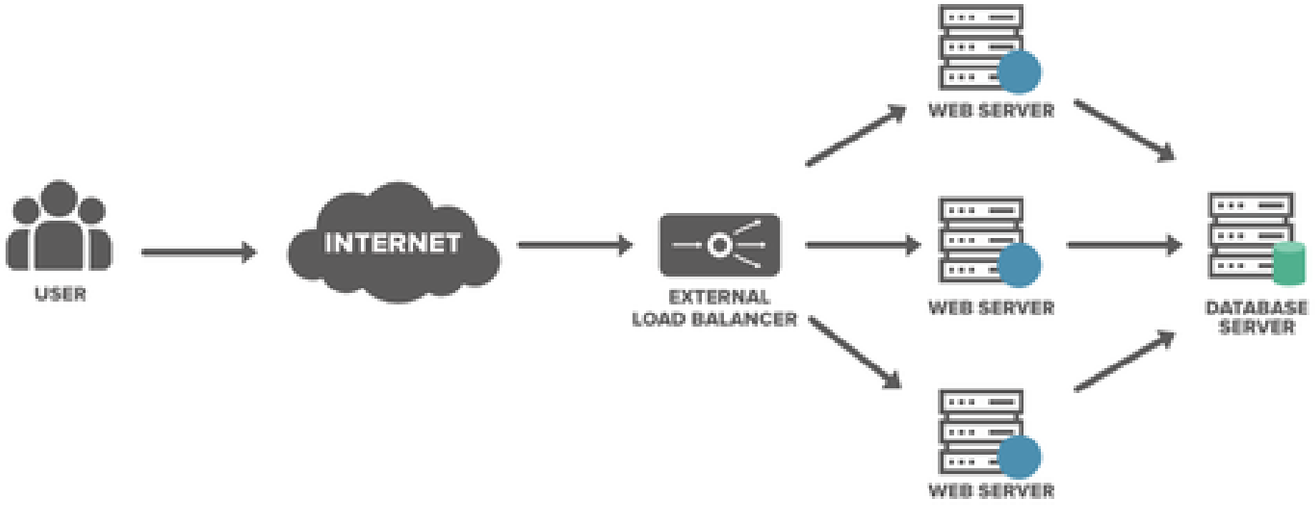
In summary, a load balancer is a key component in system design, as it helps to distribute incoming requests and ensure that the system remains highly available and performant. A strong understanding of load balancing is an important part of any system design interview, and is viewed as a core requirement for successful system design.

What is a Load Balancer?

A load balancer works as a “traffic cop” sitting in front of your server and routing client requests across all servers. It simply distributes the set of requested operations (database write requests, cache queries) effectively across multiple servers and ensures that no single server bears too many requests that lead to degrading the overall performance of the application. A load balancer can be a physical device or a virtualized instance running on specialized hardware or a software process.
Consider a scenario where an application is running on a single server and the client connects to that server directly without load balancing. It will look something like the one below...

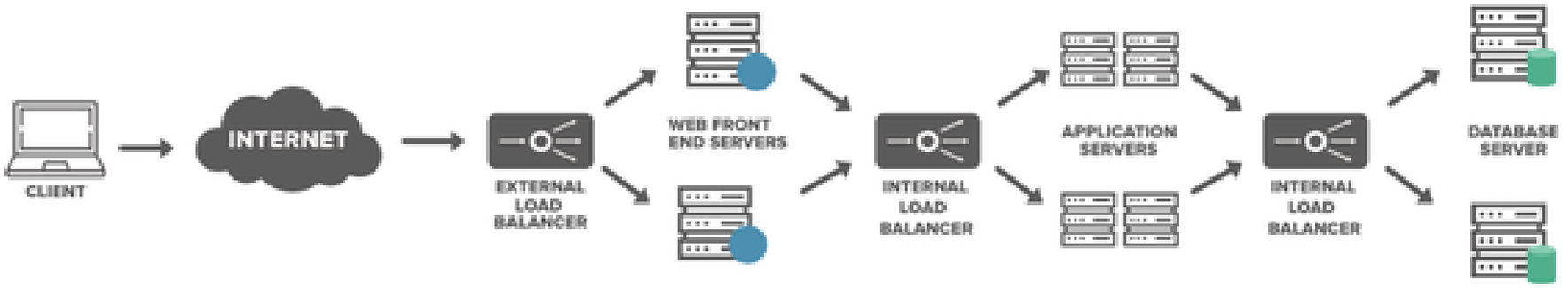
What is a Load Balancer?

A load balancer works as a “traffic cop” sitting in front of your server and routing client requests across all servers. It simply distributes the set of requested operations (database write requests, cache queries) effectively across multiple servers and ensures that no single server bears too many requests that lead to degrading the overall performance of the application. A load balancer can be a physical device or a virtualized instance running on specialized hardware or a software process.
Consider a scenario where an application is running on a single server and the client connects to that server directly without load balancing. It will look something like the one below...



- Load balancers minimize server response time and maximize throughput.
- Load balancer ensures high availability and reliability by sending requests only to online servers
- Load balancers do continuous health checks to monitor the server's capability of handling the request.
- Depending on the number of requests or demand load balancers add or remove the number of servers.

Where Are Load Balancers Typically Placed?



- In between the client application/user and the server
- In between the server and the application/job servers
- In between the application servers and the cache servers
- In between the cache servers the database servers

Types of Load Balancers

We can achieve load balancing in three ways. These are...

1. Software Load Balancers in Clients

As the name suggests all the logic of load balancing resides on the client application (Eg. A mobile phone app). The client application will be provided with a list of web servers/application servers to interact with. The application chooses the first one in the list and requests data from the server. If any failure occurs persistently (after a configurable number of retries) and the server becomes unavailable, it discards that server and chooses the other one from the list to continue the process. This is one of the cheapest ways to implement load balancing.

2. Software Load Balancers in Services

These load balancers are the pieces of software that receive a set of requests and redirect these requests according to a set of rules. This load balancer provides much more flexibility because it can be installed on any standard device (Ex: Windows or Linux machine). It is also less expensive because there is no need to purchase or maintain the physical device, unlike hardware load balancers. You can have the option to use the off-the-shelf software load balancer or you can write your custom software (Ex: load balance Active Directory Queries of Microsoft Office365) for load balancing.

3. Hardware Load Balancers
As the name suggests we use a physical appliance to distribute the traffic across the cluster of network servers. These load balancers are also known as Layer 4-7 Routers and these are capable of handling all kinds of HTTP, HTTPS, TCP, and UDP traffic. HLDs provide a virtual server address to the outside world. When a request comes from a client application, it forwards the connection to the most appropriate real server doing bi-directional network address translation (NAT). HLDs can handle a large volume of traffic but it comes with a hefty price tag and it also has limited flexibility.

HLDs keep doing the health checks on each server and ensure that each server is responding properly. If any of the servers don't produce the desired response, it immediately stops sending the traffic to the servers. These load balancers are expensive to acquire and configure, that is the reason a lot of service providers use them only as the first entry point of user requests. Later the internal software load balancers are used to redirect the data behind the infrastructure wall.

Different Categories of Load Balancing
Generally, load balancers are grouped into three categories...

1. Layer 4 (L4) Load Balancer
In the OSI model layer 4 is the transport layer(TCP/SSL) where the routing decisions are made. Layer 4 load balancer is also referred to as Network Load Balancing and as the name suggests it leverages network layer information to make the routing decision for the traffic. It can control millions of requests per second and it handles all forms of TCP/UDP traffic. The decision will be based on the TCP or UDP ports that packets use along with their source and destination IP addresses. The L4 load balancer also performs Network Address Translation (NAT) on the request packet but it doesn't inspect the actual contents of each packet. This category of load balancer maximizes the utilization and availability by distributing the traffic across IP addresses, switches, and routers.

2. Layer 7 (L7) Load Balancer
Layer 7 load balancer is also referred to as Application Load Balancer or HTTP(S) Load Balancer. It is one of the oldest forms of load balancing. In the OSI model, Layer 7 is the application layer (HTTP/HTTPS) where the routing decisions execute. Layer 7 adds content switching to load balancing and it uses information such as HTTP header, cookies, uniform resource identifier, SSL session ID, and HTML form data to decide the routing request across the servers.

3. Global Server Load Balancing (GSLB)

Today a lot of applications are hosted in cloud data centers in multiple geographic locations. This is the reason a lot of organizations are moving to a different load balancer that can deliver applications with greater reliability and lower latency to any device or location. With the significant change in the capability of the load balancers, GSLB fulfills these expectations of IT organizations. GSLB extends the capability of L4 and L7 servers in different geographic locations and distributes a large amount of traffic across multiple data centers efficiently. It also ensures a consistent experience for end-users when they are navigating multiple applications and services in a digital workspace.

Load Balancing Algorithms:

We need a load balancing algorithm to decide which request should be redirected to which backend server. The different system uses different ways to select the servers from the load balancer. Companies use varieties of load balancing algorithm techniques depending on the configuration. Some of the common load balancing algorithms are given below:

1. Round Robin
Requests are distributed across the servers in a sequential or rotational manner. For example, the first request goes to the first server, the second one goes to the second server, the third request goes to the third server and it continues further for all the requests. It is easy to implement but it doesn't consider the load already on a server so there is a risk that one of the servers receives a lot of requests and becomes overloaded.

2. Weighted Round Robin
It is much similar to the round-robin technique. The only difference is, that each of the resources in a list is provided a weighted score. Depending on the weighted score the request is distributed to these servers. So in this method, some of the servers get a bigger share of the overall request.

3. Least Connection Method
In this method, the request will be directed to the server with the fewest number of requests or active connections. To do this load balancer needs to do some additional computing to identify the server with the least number of connections. This may be a little bit costlier compared to the round-robin method but the evaluation is based on the current load on the server. This algorithm is most useful when there is a huge number of persistent connections in the traffic unevenly distributed between the servers.

4. Least Response Time Method

This technique is more sophisticated than the Least connection method. In this method, the request is forwarded to the server with the fewest active connections and the least average response time. The response time taken by the server represents the load on the server and the overall expected user experience.

5. Source IP Hash

In this method, the request is sent to the server based on the client's IP address. The IP address of the client and the receiving compute instance are computed with a cryptographic algorithm.

How to Use Load Balancing During System Design Interviews?

In your system design interview, you'll be asked some sort of scalability question where you'll have to explain how load balancers help distribute the traffic and how it ensures scalability and availability of services in your application. The overall concept that you need to keep in mind from this article is...

A load balancer enables elastic scalability which improves the performance and throughput of data. It allows you to keep many copies of data (redundancy) to ensure the availability of the system. In case a server goes down or fails you'll have the backup to restore the services. Load balancers can be placed at any software layer. Many companies use both hardware and software to implement the load balancers, depending on the different scale points in their system.

Eventual Consistency in Distributive Systems

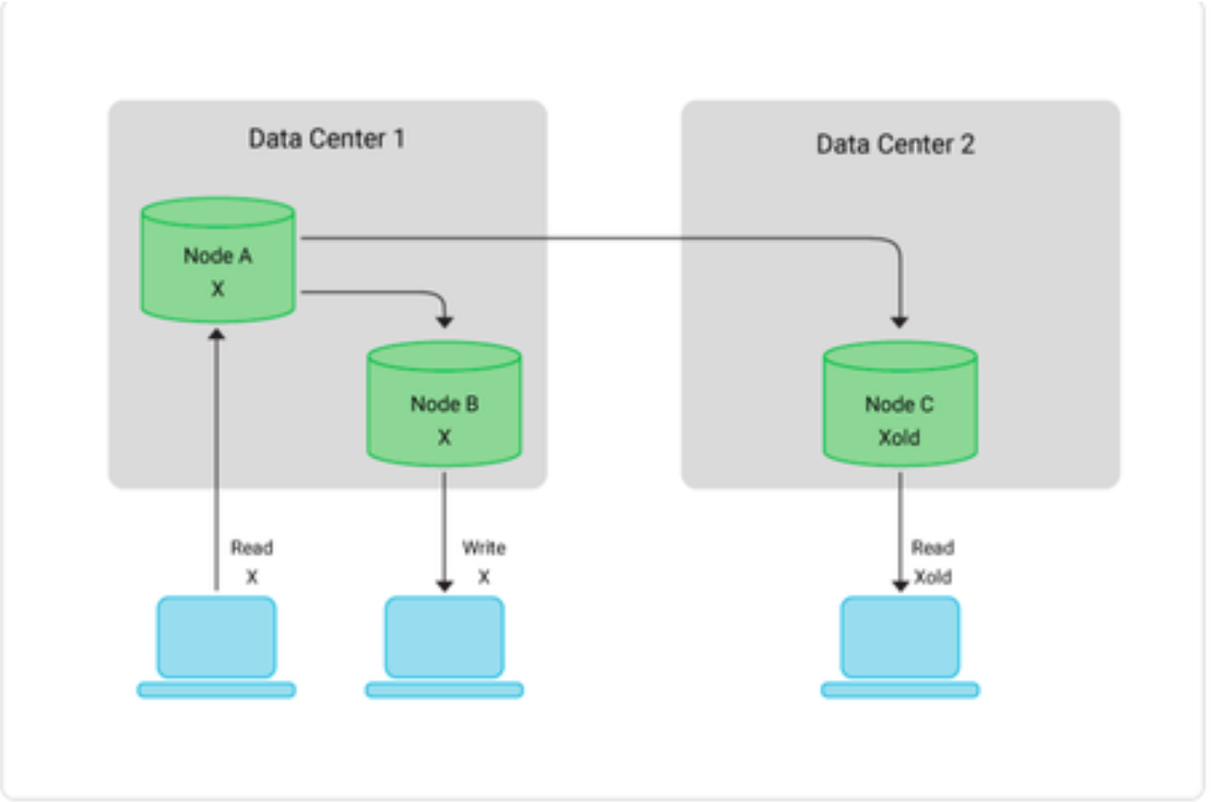
Distributed systems are composed of multiple nodes that cooperate to achieve a common goal. One of the primary challenges in distributed systems is to ensure that all nodes have a consistent view of shared data. In this context, consistency refers to ensuring that all nodes in the system see the same values for shared data at the same time.

Consistency in a distributed system refers to the property that the data stored on different nodes of the system is always in agreement with each other. In other words, all nodes in the system have the same view of the data at all times.

Consistency is a crucial aspect of distributed systems, as it ensures that all nodes have access to the same, up-to-date information. Without consistency, nodes may have different views of the data, which can lead to conflicts, errors, and even data loss.

Eventual consistency: Eventual consistency is a consistency model that allows for temporary inconsistencies between nodes in a distributed system, but guarantees that these inconsistencies will eventually be resolved and all nodes will eventually converge to a consistent state. Under eventual consistency, updates made to a node's local copy of the data are propagated asynchronously to other nodes in the system, and it is possible for different nodes to see different versions of the data at any given time. However, as long as updates continue to be made and propagated, eventually, all nodes will see the same version of the data.

Eventual consistency is often used in systems that prioritize availability and partition tolerance over strict consistency.



Consistency Models:
There are different models of consistency that can be used to achieve this property. Some of the most common consistency models include:

Strong consistency: In this model, all nodes in the system have the same view of the data at all times. Any update to the data is immediately propagated to all nodes, and all nodes agree on the ordering of updates. This model provides the strongest form of consistency but can be expensive to achieve in a distributed system.
Weak consistency: In this model, nodes may have different views of the data at different times, but eventually, all nodes will converge to the same view.

This model allows for more scalability and availability in a distributed system but can result in temporary inconsistencies.
Eventual consistency: This model is a form of weak consistency that guarantees that all nodes will eventually converge to the same view of the data, as long as there are no new updates. This model allows for high scalability and availability in a distributed system while still providing some level of consistency.

There are various consistency models that can be used to define the level of consistency required by an application in a distributed system. The most strict consistency model is linearizability, which ensures that all operations appear to execute atomically in a globally agreed-upon order. However, achieving linearizability can be challenging in a distributed system.

There are weaker consistency models that relax the strict ordering requirements but still provide some guarantees on consistency. One such model is eventual consistency, which allows for temporary inconsistencies between nodes in a distributed system, but guarantees that these inconsistencies will eventually be resolved and all nodes will eventually converge to a consistent state.

Overall, the choice of consistency model depends on the requirements of the system, the trade-off between consistency and availability, and the cost of implementing each model.

Implementation: Eventual consistency in a distributed system

One common approach to implementing eventual consistency in a distributed system is through the use of conflict-free replicated data types (CRDTs). A CRDT is a data structure that can be replicated across multiple nodes in a distributed system and allows for concurrent updates to the data without requiring explicit coordination between nodes.

Let us consider an example of a CRDT-based system that maintains a set of counters, each of which can be incremented or decremented by any node in the system. Each node in the system maintains a local copy of the set of counters, and updates are propagated asynchronously to other nodes in the system.

When a node increments or decrements a counter, it generates an update message that includes the counter identifier, the amount by which to increment or decrement the counter, and a unique identifier for the update. This unique identifier is used to resolve conflicts that may arise when different nodes make conflicting updates to the same counter concurrently.

When a node receives an update message, it applies the update to its local copy of the data using a predefined merge function. The merge function combines updates from different nodes, resolving conflicts based on the unique identifiers assigned to each update.

Over time, as updates are propagated through the system, all nodes will eventually converge to the same set of counters with the same values. This provides eventual consistency, as different nodes may temporarily have different versions of the counter data, but all nodes will eventually see the same version of the data as updates are propagated throughout the system.

In this example, CRDTs and the use of unique identifiers provide a mechanism for achieving eventual consistency in a distributed system, without requiring explicit coordination between nodes.

Example: CRDT-based counter system in Python

```python
import random
from collections import defaultdict

class Counter:
    def __init__(self):
        self.counters = defaultdict(int)

    def increment(self, counter_id, amount=1):
        update = (counter_id, amount, random.randint(1, 1000000))
        self.apply_update(update)

    def decrement(self, counter_id, amount=1):
        update = (counter_id, -amount, random.randint(1, 1000000))
        self.apply_update(update)

    def apply_update(self, update):
        counter_id, amount, update_id = update
        self.counters[counter_id] += amount

    def merge(self, other_counters):
        for counter_id, value in other_counters.items():
            self.counters[counter_id] += value

    def get_counters(self):
        return self.counters
```

Code Explanation:

In this implementation, the Counter class maintains a set of counters as a defaultdict(int). The increment and decrement methods generate update messages that include the counter identifier, the amount by which to increment or decrement the counter, and a unique identifier for the update. These updates are then applied to the local copy of the data using the apply_update method.

The merge method is used to combine updates from different nodes. It takes a dictionary of counters from another node and merges them into the local copy of the data. Conflicts are resolved by simply adding the values of each counter together.

The get_counters method simply returns the current set of counters.

This implementation is a simplified example of how CRDTs can be used to implement eventual consistency in a distributed system. In a real-world system, there would be more complexity and error handling to handle network partitions, node failures, and other issues that can arise in a distributed environment.

Sequential Consistency

What is Consistency?

Consistency is a fundamental property of distributed systems that ensures that all replicas of a shared data store have the same value. In a distributed system, data is typically replicated across multiple nodes to improve availability and fault tolerance. However, maintaining consistency across all replicas can be challenging, especially in the presence of concurrent updates and network delays.

Inconsistency problem in distributed systems:

When multiple processes access shared resources concurrently, where the order of execution is not deterministic, the problem arises due to the need for maintaining consistency across these processes.

This challenge is further complicated by the fact that distributed systems are prone to various types of failures, such as message losses, network delays, and crashes, which can lead to inconsistencies in the execution order.

So, how to solve this inconsistency problem in distributed systems?
To ensure that shared resources are accessed in a consistent order by consistency, there are several consistency models that can be used in distributed systems. These models define the level of consistency that is guaranteed for shared resources in the presence of concurrent access by multiple processes.

What is the Consistency Model?

A consistency model is a set of rules or guidelines that determine how a distributed system should behave in terms of the ordering and visibility of updates made to shared data.

In a distributed system, where multiple nodes can access and modify the same data, it's important to ensure that all nodes have a consistent view of the data to avoid conflicts and inconsistencies. A consistency model defines the level of consistency required for the system and specifies the mechanisms used to achieve it.

Different consistency models offer different trade-offs between consistency and performance, and they may prioritize factors such as availability, partition tolerance, or the level of coordination required between nodes.

Following are the different kinds of consistency models:

Eventual Consistency

- Weakest form of consistency.
- This ensures high availability and returns the same value for READ requests.
- One popular example is Cassandra – (NoSQL, highly available).

Causal Consistency

- Has consistency stronger than eventual consistency.
- Preserves the order of casually-related(dependent) operations.
- So, that's why it does not ensure the ordering of operations that are non-causal. One example is Comment replies.

Sequential Consistency

- Stronger than causal consistency.
- Doesn't ensure writes are visible instantaneously or in the same order as according to some global block.
- One example is that of facebook posts of friends (sequential for a particular friend, but not for overall)

Strict Consistency/ Linearizability

- This is the strongest consistency model.
- All read requests are consistent as they get the latest write value
- One popular example is password update of someone's bank account.

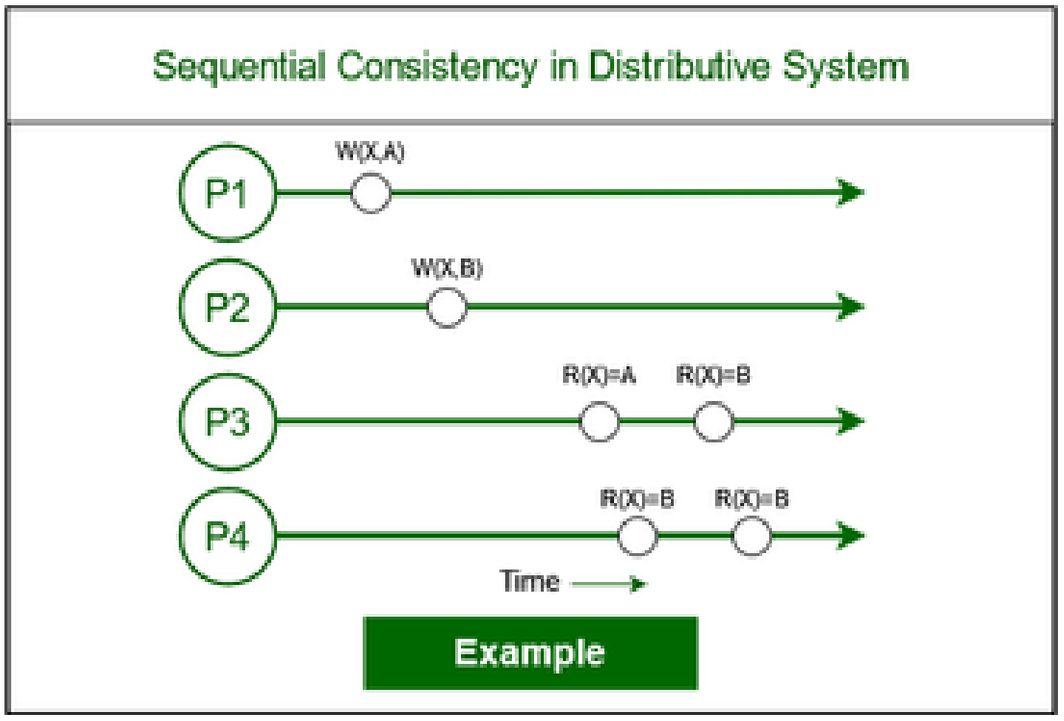
Why to use sequential consistency as a consistency model in distributive systems?

Every concurrent execution of a program should ensure that the results of the execution are to be in some sequential order. It provides an intuitive model of how the distributed systems should behave, which makes it easier for users to understand and reason about the behavior of the system.

Sequential Consistency:

Sequential consistency is a consistency model that ensures that the order in which operations are executed by different processes in the system appears to be consistent with a global order of execution. It ensures that the result of any execution of a distributive system is the same as if the operations of all the processes in the system were executed in some sequential order, one after the other.

Example of Sequential Consistency



Sequential Consistency Example

Let us consider a distributive system here for four different processes, P1, P2, P3, and P4. Here, several different operations are occurring simultaneously, such as W(X, A) updating the value of X to be A, and R(X) reading the value of X.

Now, if we follow the given global order of operations, we get a sequentially consistent system.

W(X,A), R(X) = A, W(Y,B), R(Y) = B

With this sequential order, it makes sense that P3 reads A first and then B in subsequent read operations.

Techniques to implement Sequential Consistency
The following techniques can be used to implement sequential consistency in distributed systems:

- Two-Phase Locking: Two-phase locking is a technique used to ensure that concurrent access to shared data is prevented. In this technique, locks are used to control access to shared data. Each process acquires a lock before accessing the shared data. Once a process acquires a lock, it holds the lock until it has completed its operation. Two-phase locking ensures that no two processes can access the same data at the same time, which can prevent inconsistencies in the data.
- Timestamp Ordering: Timestamp ordering is a technique used to ensure that operations are performed in the same order across all nodes. In this technique, each operation is assigned a unique timestamp. The system ensures that all operations are performed in the order of their timestamps. Timestamp ordering ensures that the order of operations is preserved across all nodes.
- Quorum-based Replication: Quorum-based replication is a technique used to ensure that data is replicated across different nodes. In this technique, the system ensures that each write operation is performed on a subset of nodes. To ensure consistency, the system requires that a majority of nodes agree on the value of the data. Quorum-based replication ensures that the data is replicated across different nodes, and inconsistencies in the data are prevented.

- Vector Clocks: Vector clocks are a technique used to ensure that operations are performed in the same order across all nodes. In this technique, each node maintains a vector clock that contains the timestamp of each operation performed by the node. The system ensures that all operations are performed in the order of their vector clocks. Vector clocks ensure that the order of operations is preserved across all nodes.

Some common challenges in achieving Sequential Consistency:

- Network Latency: Communication between different nodes in a distributed system can take time, and this latency can result in inconsistencies in the data.
- Node Failure: In a distributed system, nodes can fail, which can result in data inconsistencies.
- Concurrent Access: Concurrent access to shared data can lead to inconsistencies in the data.
- Replication: Replication of data across different nodes can lead to inconsistencies in the data.

Achievement after implementing Sequential Consistency:

After implementing sequential consistency in a distributed system, the system then provides a high level of consistency guarantee for shared resources. This means that the system ensures that the order of operations on a shared resource is consistent with the order in which they were issued by processes. As a result, the final state of the resource is the same as if the operations had been executed serially, even though they may have been executed concurrently by multiple processes.

Thus, this ensures that the system behaves correctly and consistently which is really essential and critical for several applications in fields of financial systems and mission-critical systems, also in databases of big applications. Thus, it ensures high reliability and consistency of the system.

Durability

Distributed System term defines the concept that multiple independent Computer Systems are been distributed to various locations and all these nodes can be operated from a single system. Due to their capacity to offer high scalability, fault tolerance, and load balancing, these systems are growing in popularity. They do, however, also have their own set of difficulties, such as assuring data durability.

Data Durability is an important factor in Distributed Systems. This factor assures that the data is even safe after the system failure or crashed state. 3 different techniques can be applied to recover the data in the situation of system failure.

- Replication,
- Backup, and
- Write-Ahead Logging(WAL)

provide a reliable approach for ensuring data durability in distributed systems. By applying this technique, organizations can safeguard their data and recover the data in the situation of system failure.

Replication in Systems:

The practice of making and keeping numerous copies of data on various nodes in a distributed system is known as replication.

Data is often dispersed over several nodes in distributed systems, and replication makes sure that each node has a copy of the data. The replication technique can increase the Scalability of System Performance and Availability which results in restoring the data in case of failure. Replication also provides the feature of Fault Tolerance which is discussed below in the article.

Key Features of Replication:

- Fault tolerance: Replication can help ensure that data is long-lasting and resistant to failure. With multiple copies of the data stored on different nodes, if one copy becomes corrupted or lost, the data can be recovered using another copy.
- Improved performance: The replication technique can be used to increase the performance of the system by decreasing the amount of data that is transferred across the network channel. By storing copies of the data on multiple nodes, data can be read from the closest node, reducing network latency and improving response times.
- High availability: Replication can help to ensure that data remains accessible even if a node fails. Because multiple copies of the data are stored on different nodes, if one node fails, another node can take over and provide access to the requested data avoiding the situation of system failure.

Applications of Replication:

- Data replication: In a distributed system, replication is commonly used to replicate information across different computer systems. This helps to ensure that data is available even if a node fails.
- Load balancing: Replication can be used to distribute workload across multiple nodes, assisting in load balancing and system performance.
- Disaster recovery: By creating copies of data in different geographical locations, replication can be used for disaster recovery. Even after the occurrence of the disaster or cyber-attack the data can be recovered from various replicated copies. These copies hold the same information that is being lost.

Replication is an important technique for ensuring the durability of distributed systems. Replication ensures that data remains available and durable even in the event of failures or other disruptions by maintaining multiple copies of the data.

Fault tolerance is one of the most important advantages of replication in terms of durability. If one node fails, another node can take over and provide access to the data because data is replicated across multiple nodes. This helps to ensure that data remains available and durable even if a node fails.

Data consistency is another advantage of replication in terms of durability.

Maintaining data consistency in a distributed system can be difficult. By ensuring that all nodes have the same copy of the data, replication can help ensure consistency. This helps to prevent data inconsistencies and ensures that the data is long-lasting and dependable.

Backup in Systems::

In the concept of Distributed System, backups are one of the most significant aspects of safeguarding data durability. The ability of information to persist and be accessible in situations of failure or any other interruptions is known as Data Durability. Backups are nothing but copies of information that are been preserved in various geographical locations or data centers, which aims to recover the data in the event of data corruption or loss.

Key Features of Backups:

- Redundancy: Redundancy is the process of creating multiple copies or replicas of data. Redundancy assures that if any one copy of the data is been lost or additional due to various factors like cyber threats, corruption, disaster, etc. then still the additional copies of data can be restored and reused to access the sensitive information without any problem.
- Automatic Backups: In the current scenario, many distributed systems come with the mechanism of Automated backups which results in allowing the information to be backed up at regular periods. This guarantees that the backups of data are always updated and can be restored at any point of failure without disruption.

- Incremental Backups: Some backup systems only backup changes to data that have occurred since the last backup. The incremental Backup approach decreases the amount of information that must be backed up, leading to a faster and more efficient backup operation. This results in the faster retrieval of data and the performance of the distributed systems is enhanced.

Applications of Backups:

- Compliance: Many industries and government regulations necessitate the creation and storage of backups for a set period. Backups ensure that data adheres to these regulations. For Example, to comply with HIPAA regulations, the healthcare industry requires backups to be made and stored for seven years.
- Archiving: Backups provide the facility to keep the data in an Archive state. As the data which is Archived is no longer necessary for use in day-to-day functioning. But this can also be used for various historical or legal purposes. Achieving backups can be beneficial for information that is needed to be restored for a longer period, such as Economical Records, Research and development data.
- Testing and Development: For testing and development, backups are helpful. Through the use of backups, developers can test new software or updates without interfering with the live environment. In order to give developers precise data and configurations to work with, backups can also be utilized to construct development environments that are exact replicas of the production environment.

Backups are used in a variety of applications and are a crucial component of ensuring data durability. Backups are essential for data protection and sustaining business operations, from disaster recovery and compliance to data replication and archiving.

WAL or Write-Ahead Logging in Systems:
WAL in Distributive Systems stands for Write-Ahead Logging, which is a method for ensuring Data Durability in Distributed Systems. In this process, the data which is being changed is first written into the log file, before saving the data to data centers or stores. The goal of this is to ensure that, if the data is being lost while committing to a data store, still it can be retrieved from the log file. This achieves Data Durability in Distributed Systems. In the sectors of Finance and healthcare, the most important factor is data consistency, so to maintain this consistency WAL can be used.

Below we have mentioned some key features of WAL:

Key Features of WAL:

- Atomicity: In the process of WAL, Atomicity specifies that the means of change in the data store are either complete or are fully rolled back and remain consistent and reliable.
- High Write Performance: WAL needs writing to a log file instead of immediately updating the data storage, this can provide enhanced writing performance. The reason for better performance is that the process of writing into a log file is much faster than updating the data store. Additional processing and I/O operations are needed while updating the Data Store.
- Scalability: WAL has the ability to scale up or scale down a large amount of data volumes and has high write throughput by distributing the log across different multiple nodes. This assures that logs can be written and accessed quickly from larger distributed systems without any disruption.

Applications of WAL:

- Databases: Write-Ahead Logging can be used in databases to assure data durability and consistency. Many databases like PostgreSQL, SQLite, and Oracle databases use the WAL approach for ensuring data durability. WAF guarantees that the change in the data store is properly written in the log file before saving it to the database.
- Messaging Systems: WAL in messaging systems assures that the message is been processed and delivered to the authorized person properly. Changes in the message queue are first written into a log file before saving the message queue. Due to this process, the risk of data loss and system failure has been minimized.

- Financial Applications: WAL in the Financial sector is considered one of the important approaches as the transactions in banks are processed and recorded rapidly. Changes to the bank transaction are initially written into the log file and then committed to the transaction log. This maintains the durability and also integrity of the data.

Conclusion

Distributed Systems are responsible for fault-tolerant, which means that if any failure occurs then the complete system should not be crashed. The operation is transferred to another node rather than stopping the operation. Because distributed systems are intended to be fault-tolerant, or to continue operating even if one or more of their components fail, durability is a crucial criterion. To preserve the system's general availability and dependability, the data stored in it must be robust and resilient to errors.

Because data is typically kept across several nodes, which may be spread out over various physical locations and connected by unreliable networks, ensuring durability in distributed systems is difficult. To ensure that data is persistent even in the face of such errors, distributed systems must use sophisticated approaches to conserve the durability of the system.

LLD of the Snake and Ladder game:

Problem Statement :

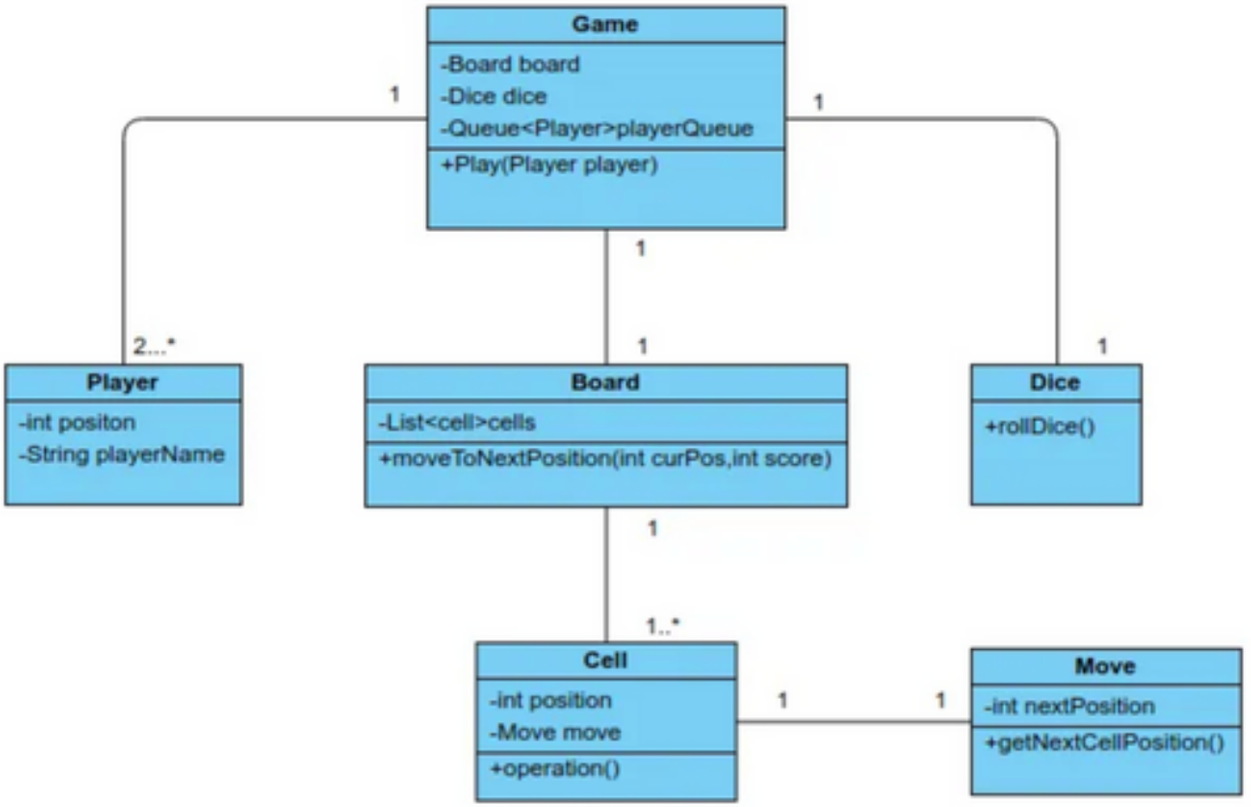
Basic Board: On a board (Of size 100), for a dice throw a player should move from the initial position by the number on dice throw.

Add a snake on the board: A snake moves a player from its start position to end position. where start position > end position
Test data: Add a snake at position 14 moving the player
to position 7.

Make A Crooked Dice: A dice that only throws Even numbers.
The can game can be started with normal dice or crooked dice.

Implementation:

Before jump into the design, we will figure out entities. The first entity that came into my mind is Game. The game will consist of Players, Board, and Dice. Let's break each entity to find new entities. If we take the player we didn't find any special entity. Then take the board, we see one entity a list of cells. In dice, we see just one method that will return 1–6 random numbers for now. One more class is for moving the object from higher cell to lower (call snake) or vice versa (ladder). You can see these all entities in the diagram below.



1.Game

class diagram snake and ladder game

```
import exception.GameOverException;
import exception.InvalidPlayerException;
import model.Board;
import model.Player;

import strategy.Dice;

import java.util.*;

public class Game {
    private final Board board;
    private Dice dice;
    private final Map<Integer, Player> res;
    private int winnerCount;
    private final Queue<Player> playerQueue;

    public Queue<Player> getPlayerQueue() {
        return playerQueue;
    }

    public Map<Integer, Player> getRes() {
        return res;
    }


    public void setDice(Dice dice) {
        this.dice = dice;
    }
```

```
    public Game(List<Player> playerList, Board board, Dice dice) {
        this.board = board;
        this.dice = dice;
        this.res = new HashMap<>();
        this.playerQueue = new ArrayDeque<>(playerList);
    }

    public Player getNextPlayerToPlay() throws GameOverException {
        if (playerQueue.size() <= 1)
            throw new GameOverException("Game is already over");
        return playerQueue.poll();
    }

    public void play(Player player) throws InvalidPlayerException {
        if (!checkReachToEnd(player)) {
            int nextPosition =
board.moveToNextPosition(player.getPosition(), dice.rollDice());
            player.setPosition(nextPosition);
            updateGameStatus(player);
        } else throw new InvalidPlayerException("player already reach
end of game");
    }

    private void updateGameStatus(Player player) {
        if (checkReachToEnd(player)) {
            res.put(++winnerCount, player);
            if (playerQueue.size() == 1) {
                res.put(++winnerCount, playerQueue.poll());
            }
        } else playerQueue.add(player);
    }

    private void declareResult(Map<Integer, Player> res) {
        res.forEach((k, v) -> System.out.println(v + " ranks " + k));
    }

    private boolean checkReachToEnd(Player player) {
        return player.getPosition() == 100;
    }
}
```

2.Board

```
import java.util.List;
import java.util.Optional;

public class Board {
    List<Cell> cells;

    public List<Cell> getCells() {
        return cells;
    }

    public Board(List<Cell> cells) {
        this.cells = cells;
    }

    public int moveToNextPosition(int currentPosition, int score) {
        if (currentPosition < 0)
            throw new IllegalArgumentException("Position should be
greater than zero");
        Optional<Cell> nextCellByPosition =
getNextCellByPosition(currentPosition + score);
        return
nextCellByPosition.map(Cell::nextPosition).orElse(currentPosition);
    }

    private Optional<Cell> getNextCellByPosition(int position) {
        return cells.stream().filter(cell -> cell.getPosition() ==
position)
                .findFirst();
    }
}
```

3.Player

```
import java.util.Objects;

public class Player {
    private int position;
    private final String playerName;

    public String getPlayerName() {
        return playerName;
    }

    public int getPosition() {
        return position;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Player)) return false;
        Player player = (Player) o;
        return position == player.position &&
                Objects.equals(playerName, player.playerName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(position, playerName);
    }

    public Player(int position, String playerName) {
        this.position = position;
        this.playerName = playerName;
    }

    @Override
    public String toString() {
        return "Player :" + playerName;
    }

    public void setPosition(int position) {
        this.position = position;
    }
}
```

4. Dice

Dice will be an interface with different implementations. Here is normal dice implementation.

```
import java.util.Random;

public class NormalDice implements Dice {
    @Override
    public int rollDice() {
        Random random = new Random();
        return 1+random.nextInt(6);
    }
}
```

5. Move

The move will abstract class and their implementations will be different types of snake and ladders.

```
import exception.InvalidPositionException;

public class DefaultMove extends Move {

    public DefaultMove() {
        super(0);
    }
    @Override
    public boolean isValidPosition(int cellPosition) throws
InvalidPositionException {
        return false;
    }
}
```

6. GreenSnake

As already mention Move can extended to different types of snakes, we have one implementation for a green snake, a snake that bites only once, if once it bites once, it will be ineffective in the rest of the game.

```
import exception.InvalidPositionException;

public class GreenSnake extends Move {
    private boolean canMove = true;

    public GreenSnake(int nextPosition) {
        super(nextPosition);
    }

    @Override
    public boolean isValidPosition(int cellPosition) throws
InvalidPositionException {
        return false;
    }

    @Override
    public int getNextPosition() {
        int next = nextPosition;
        if (canMove) {
            nextPosition = 0;
            canMove = false;
        }
        return next;
    }
}
```

Test: We can test this game using test cases.

```
import exception.GameOverException;
import exception.InvalidPlayerException;
import model.Board;
import model.Cell;
import model.DefaultMove;
import model.Player;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import strategy.Dice;
import strategy.MockDice;

import java.util.ArrayList;
import java.util.List;

public class GameTest {
    private Game game;
    List<Player> playerList;
    Board board;
    Dice dice;

    @Before
    public void before() {
        playerList = new ArrayList<>();
        playerList.add(new Player(0, "A"));
        playerList.add(new Player(0, "B"));
        List<Cell> cells = getCells(100);
        board = new Board(cells);
        dice = new MockDice();
        game = new Game(playerList, board, dice);
    }

    @Test
    public void playTest() throws InvalidPlayerException {
        Player p = new Player(1, "Test");
        game.setDice(new MockDice());
        game.play(p);
        Assert.assertEquals(p.getPosition(), 26);
    }
```

```
    @Test
    public void endToEndTest() throws InvalidPlayerException,
GameOverException {
        Player p1 = playerList.get(0);
        Player p2 = playerList.get(1);
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(25, p1.getPosition());
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(25, p2.getPosition());
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(50, p1.getPosition());
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(50, p2.getPosition());
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(75, p1.getPosition());
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(75, p2.getPosition());
        game.play(game.getNextPlayerToPlay());
        Assert.assertEquals(100, p1.getPosition());
        Assert.assertEquals(p1, game.getRes().get(1));
        p1.setPosition(0);
        p2.setPosition(0);
        game.getPlayerQueue().add(p1);
    }

    @Test(expected = GameOverException.class)
    public void testGameOver() throws InvalidPlayerException,
GameOverException {
        Player p1 = game.getNextPlayerToPlay();
        p1.setPosition(75);
        game.play(p1);
        game.getNextPlayerToPlay();
    }

    @Test(expected = InvalidPlayerException.class)
    public void testInvalidPlayerPlay() throws InvalidPlayerException
{
        Player player = new Player(1, "Test");
        player.setPosition(100);
        game.play(player);
    }

    public static List<Cell> getCells(int numberOfCells) {
        List<Cell> cells = new ArrayList<>();
        for (int i = 1; i <= numberOfCells; i++) {
            cells.add(new Cell(i, new DefaultMove()));
        }
        return cells;
    }
}
```

Low-Level Design for Payment tracking app like Splitwise

Prioritized Requirements

- Users can add expenses.
- Users can edit expenses.
- Also, users can settle expenses.
- Allow users to make groups and add, edit and settle expenses in the group.

Requirements not a part of our design

- Comments for records.
- Activity log for every event.
- Authentication service.

Objects definition

- User object

```
struct User{
userID uid;
string ImageURI;
string bio;
}
```

Balance object

There are 2 types of people:

- People who will receive money from others. They have a positive balance.
- People who will pay money to others. They have a negative balance.

So our balance object should be able to handle both positive and negative values.

```
struct Balance{
string currency;
int amount; //For simplicity we are using integers here.
// We can also use float or double but in that case, we also ne
}
```

Expense object

Expense objects must map each user to their balance.

```
  struct Expense{
  ExpenseID eid;
  bool isSettled;
  map<User,Balance>;
  GroupID gid; //This expense belongs to which group

  //Metadata
  string title;
  int timestamp;
  string imageURI;
}
```

Group object

```
struct Group{
    GroupID gid;
    List<User> users;

    //metadata

    string ImageURI;
    string title;
    string description;
}
```

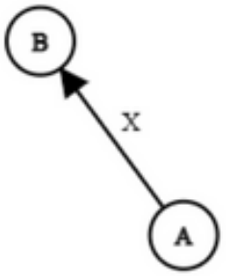
Behavior definition

- **Add Expense** We have user and balance objects, and then we can persist them in the database.
- **Edit Expense** Each expense object has a unique ID. We can use the ID to change the mapping or other metadata.
- **Settle Expense** We make the `` `isSettled` `` flag true. We will use a balancing algorithm to settle expenses.
- **Add, Settle and Edit expenses in group** Each expense object has a groupID. So we can add, settle and edit expenses in a group.

Balancing algorithm

- **Problem** Let us denote each user as a node and each payment as an edge in a graph.

  So we need to **minimize the number of edges in the graph**



It means A paid x amount to B.

- **Solution**

  Note: When we talk about balance we are talking about the sum of all transactions for a user.

  First, let's divide the user into two categories

  - People who have a positive balance.
  - People who have a negative balance.

  If at the point any user has 0 balance that means his/her expenses are settled, and we can remove that node from the graph.

At each step we will pick the largest absolute value from each category and add an edge between them. Let's say we pick `A` from the first category and `B` from the second category. So will add an edge from B to A (because B will pay and A will receive) and then update the balances. If the new balance of any node becomes 0 then we will not consider the node again.

To implement this we can use **heap data structure.**

```
List<PaymentNode> makePaymentGraph(){

  Max_Heap<Node> firstCategory;
  Max_Heap<Node> secondCategory; //We are storing the absoulte

  List<PaymentNode> graph;

  //The sum of balance of all users always results in 0 so if
  loop(firstCategory.isNotEmpty()){
      receiver = firstCategory.top();
      sender = secondCategory.top();

      //Removing the element
      firstCategory.pop();
      secondCategory.pop();
```

amountTransferred = min(sender.finalBalance , receiver.finalBalance);

graph.insert(PaymentNode(sender.uid, receiver.uid, amountTransferred));

```
        sender.finalBalance -= amountTransfered;
        receiver.finalBalance -= amountTransfered;

        if(sender.finalBalance != 0)
            secondCategory.push(sender);

        if(receiver.finalBalance != 0)
            firstCategory.push(receiver);

    }

    return PaymentGraph;

}
```

- **Edge case**

Let us consider a case

A	B	C	D	E	F	G
80	25	-25	-20	-20	-20	-20

If we use our approach then the payments will be

- C will pay $25 to A
- D will pay $20 to A
- E will pay $20 to A
- F will pay $20 to B
- G will pay $15 to A
- G will pay $5 to B

So there are **6 transactions.**

But it can be done in only **5** transactions

- C will pay $25 to B
- D will pay $20 to A
- E will pay $20 to A
- F will pay $20 to A
- G will pay $20 to A

API Design

- **Expenses[] getGroupExpenses(GroupID gid)**
- **PaymentGraph getGroupPaymentGraph(GroupID gid)**
- User getUser(UserID uid)
- User[] getUsersInGroup(GroupID gid)
- ExpenseID addExpense(GroupID gid, UserID uid, int amount, string currency)
- void editExpense(ExpenseID eid, UserID uid, int amount, string currency)
- void settleExpense(ExpenseID eid)
- GroupID makeGroup(User[] users, string name, string imageURI, string description)
- Group getGroup(GroupID gid)
- Expense getExpense(ExpenseID eid)

Caching

We want to cache responses that are **required by many users** or are **expensive to calculate**. So will cache the response of the following APIs.

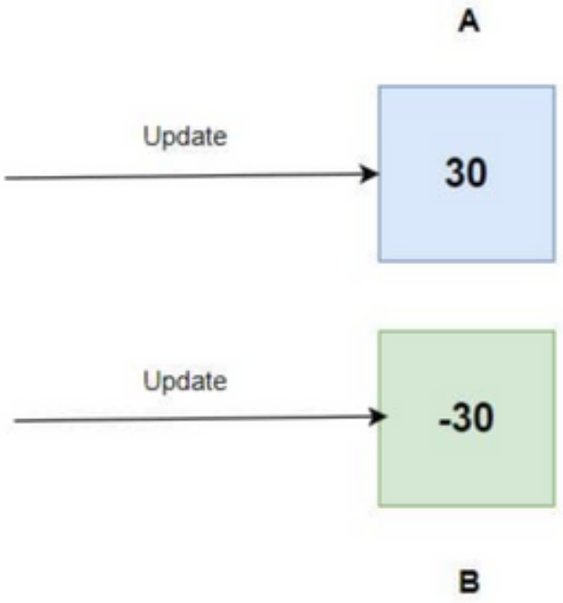
1. Expenses[] getGroupExpenses(GroupID gid)
2. PaymentGraph getGroupPaymentGraph(GroupID gid)

Data consistency (Add example)

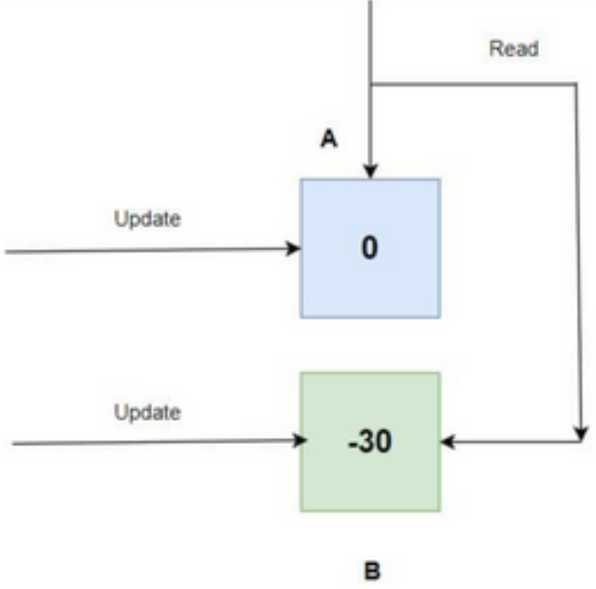
When performing simultaneous read and update operations the data users might get is incorrect. This is called **data inconsistency.**

Let's understand this using an example:

We have two users A and B. A received $30 from B. So we perform an update expression



Now the update operation on A was completed but not for B. And at the exact moment, there was a read request from a user.

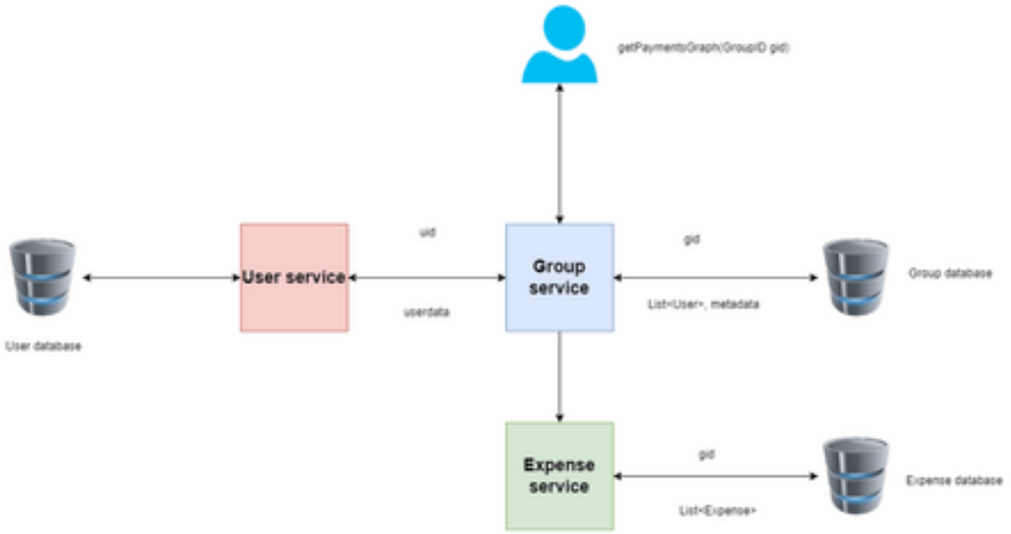


The sum of all balances should be 0 but in this case, it is -30. So the data is inconsistent.

There are **2 ways we can solve this problem**

- **Put read lock when updating expenses** While updating the expense objects we will not allow users to read data. It will make sure that the data users get will be consistent.
- **Make objects immutable** So whenever there is an edit request we will create a new object and update data in a new object. Once the update is completed we will point the reference to the new object. Users might get old data however we can be sure that data will be consistent. This will make our system **eventually consistent**.

Diagram



Low Level Design of Food delivery app - Zomato, Swiggy, UberEats

Description:

Food delivery applications allow customers to order food online. People can enjoy different varieties of food at home. They can order from the restaurants they wish for. All the available food delivery apps in the market are providing different types of functionalities for the customers. Some are providing discounts, some are providing free delivery, some are providing scheduled delivery.

Some of the available products for food delivery are DoorDash, Zomato, UberEats

Requirements:

- A food delivery system requires the interaction of the restaurants, customers and delivery boys with the admin.
- Restaurants can register themselves.
- Users can create, update, delete, get their profiles.
- User can search for the restaurant using a restaurant name, city name.
- Restaurants can add, update the food menu.
- User can see the food menu. User can get the food items based on Meal type or Cuisine type.
- User can add/remove items to/from the cart. User can get all the items of the cart.
- User can place or cancel the order. User can get all the orders ordered by him/her.

- User can apply the coupons. User can get the detailed bill containing tax details.
- User can make a payment using different modes of payment — credit card, wallet, etc.
- Delivery boy can get all the deliveries made by him using his Id.
- User can get the order status anytime. Success, Out for Delivery, Delivered, etc.

Services:

- Restaurant Service

```java
@Service
public class RestaurantService {
    private RestaurantData restaurantData;

    @Autowired
    public RestaurantService(RestaurantData restaurantData) { this.restaurantData = restaurantData; }

    public void addRestaurant(@NonNull final Restaurant restaurant) {...}

    public List<Restaurant> getAllRestaurants() { return new ArrayList<>(restaurantData.getRestaurantById().values()); }

    public Restaurant getRestaurantById(@NonNull final String restaurantId) {...}

    public List<Restaurant> getRestaurantsByName(@NonNull final String restaurantName) {...}

    public List<Restaurant> getRestaurantsByCity(@NonNull final String city) {...}
}
```

Restaurant Service

All the functionalities related to the restaurants will be handled by the **Restaurant Service**. It will interact with **Restaurant Data** only. It will render the first page of the application, i.e., the list of all restaurants or the searched restaurants. It will also allow restaurants to register and Admin to manage.

> *Restaurants can register themselves — **addRestaurant***
>
> *Customers can see the list of all the restaurants — **getAllRestaurants***
>
> *Admin can search for the restaurant using Id — **getRestaurantById***
>
> *Customers can search for restaurants using name — **getRestaurantsByName***
>
> *Customers can search for restaurants using city — **getRestaurantsByCity***

User Service

```java
@Service
public class UserService {
    private UserData userData;

    @Autowired
    public UserService(UserData userData) { this.userData = userData; }

    public void addUser(@NonNull final User user) {...}

    public void deleteUser(@NonNull final String userId) {...}

    public void updateUser(@NonNull final User user) {...}

    public User getUser(@NonNull final String userId) {...}
}
```

User Service

User profile related features will be provided by **User Service**. It will interact with **User Data** only. It will allow Customers and Delivery boys to register and update their profiles.

> *Users can register themselves — **addUser***
>
> *Users can delete their profiles — **deleteUser***
>
> *Users can update their profiles — **updateUser***
>
> *Users can get info of their profiles — **getUser***

Food Menu Service

```java
@Service
public class FoodMenuService {
    private FoodMenuData foodMenuData;

    @Autowired
    public FoodMenuService(FoodMenuData foodMenuData) { this.foodMenuData = foodMenuData; }

    public void addMenuByRestaurantId(@NonNull final String menuId, @NonNull final String restaurantId,
                                      @NonNull final List<MenuItem> menuItemList) {...}

    public FoodMenu getMenuById(@NonNull final String menuId) {...}

    public FoodMenu getMenuByRestaurantId(@NonNull final String restaurantId) {...}

    public void addMenuItemsByMenuId(@NonNull final String menuId, @NonNull final List<MenuItem> menuItemList) {...}

    public void addMenuItemsByRestaurantId(@NonNull final String restaurantId,
                                           @NonNull final List<MenuItem> menuItemList) {...}

    public List<MenuItem> getMenuItemsByRestaurantIdAndCuisine(@NonNull final String restaurantId,
                                                               @NonNull final CuisineType cuisineType) {...}

    public List<MenuItem> getMenuItemsByRestaurantIdAndMealType(@NonNull final String restaurantId,
                                                                @NonNull final MealType mealType) {...}

    public MenuItem getMenuItemById(@NonNull final String itemId) {...}
}
```

Food Menu Service

Once the customer selected the restaurant, the second page of the application, i.e., the food menu will be rendered by the **Food Menu Service**. It will also allow customers to search for the items on the menu basis on some category. It will interact with **Food Menu Data** only.

> *Restaurants can add the food menu — addMenuByRestaurantId*
>
> *Admin can search for the food menu using Id — getMenuById*
>
> *Customers can see the menu — getMenuByRestaurantId*
>
> *Admin can add more items in the menu — addMenuItemsByMenuId*
>
> *Restaurants can add more items — addMenuItemsByRestaurantId*

> *Customers can search for the items using cuisine type — **getMenuItemsByRestaurantIdAndCuisine***
>
> *Customers can search for the items using meal type — **getMenuItemsByRestaurantIdAndMealType***
>
> *Admin can search for the item using Id — **getMenuItemById***

. . .

Cart Service

```
@Service
public class CartService {
    private CartData cartData;
    private List<CartCommandExecutor> cartCommandExecutorList;
    private FoodMenuService foodMenuService;

    @Autowired
    public CartService(CartData cartData, List<CartCommandExecutor> cartCommandExecutorList,
                       FoodMenuService foodMenuService) {...}

    public void clearCart(@NonNull final String userId, @NonNull final String restaurantId) {...}

    public void updateCart(@NonNull final String userId, @NonNull final String restaurantId,
                           @NonNull final String itemId, @NonNull final CartCommandType cartCommandType) {...}

    public List<MenuItem> getAllItemsOfCart(@NonNull final String userId, @NonNull final String restaurantId) {...}
}
```

Cart Service will allow Customers to add or remove the items in or from the cart. It will render the third page of the application, i.e., items in the cart. It will interact with the **Cart Data** and will also call the **Food Menu Service** to get the items using Id. We can use the **Command Pattern** to handle the add or remove commands.

> *Customers can add/remove items — **updateCart***
>
> *Customers can empty the cart — **clearCart***
>
> *Customers can see the cart items — **getAllItemsOfCart***

Pricing Service

```java
@Service
public class PricingService {
    private List<PricingStrategy> pricingStrategyList;
    private CartService cartService;

    @Autowired
    public PricingService(List<PricingStrategy> pricingStrategyList, CartService cartService) {...}

    public Bill getBill(@NonNull final String userId, @NonNull final String restaurantId,
                        @NonNull final CouponCode couponCode) {...}
}
```

Pricing Service

On the cart page, **Pricing Service** will allow Customers to see the bill details. It will call the **Cart Service** to get all the items of the cart to render the bill. We can use a **Strategy Pattern** and have different types of strategies — TwentyPercentOff, FiveHundredOff, etc.

> *Customers can see the bill details — **getBill***

Order Service

```java
@Service
public class OrderService {
    private OrderData orderData;
    private List<OrderCommandExecutor> orderCommandExecutorList;

    @Autowired
    public OrderService(OrderData orderData, List<OrderCommandExecutor> orderCommandExecutorList) {...}

    public void updateOrder(@NonNull final Order order, @NonNull final OrderCommandType orderCommandType) {...}

    public Order getOrderById(@NonNull final String orderId) {...}

    public List<Order> getAllOrdersByRestaurantId(@NonNull final String userId, @NonNull final String restaurantId) {...}

    public List<Order> getAllOrders(@NonNull final String userId) {...}

}
```

Order Service

Customers can place or cancel the order, once the cart is finalized, using **Order Service**. It will interact with **Order Data** only. It will allow customers to see the **order history** also. Customers can select the history based on the restaurant also. We can use the **Command Pattern** to handle place or cancel commands.

> *Customers can place or cancel orders — **updateOrder***
>
> *Admin can search for the order — **getOrderById***
>
> *Customers can see their order history for a restaurant — **getAllOrdersByRestautantId***
>
> *Customers can see their all order history — **getAllOrders***

Payment Service

```java
@Service
public class PaymentService {
    private PaymentData paymentData;
    private PricingService pricingService;
    private OrderService orderService;

    @Autowired
    public PaymentService(PaymentData paymentData, PricingService pricingService, OrderService orderService) {...}

    public void addPayment(@NonNull final Payment payment) {...}

    public Payment getPaymentById(@NonNull final String paymentId) {...}

    public Payment getPaymentByOrderId(@NonNull final String orderId) {...}

    private boolean validatePayment(Payment payment, Order order) {...}
}
```

Payment Service

Payments can be made to the restaurants using **Payment Service**. It will interact with the **Payment Data** and also call the **Pricing Service** to validate the payment made & the **Order Service** to update the order status. It will allow Customers to add the Payment against their order.

> *Customers can pay for their orders — **addPayment***
>
> *Admin can search for the payment using Id — **getPaymentById***
>
> *Customers can see the payment made — **getPaymentByOrderId***
>
> *Payment must match the bill — **validatePayment***

. . .

Delivery Service

```
@Service
public class DeliveryService {
    private DeliveryData deliveryData;
    private OrderService orderService;

    @Autowired
    public DeliveryService(DeliveryData deliveryData, OrderService orderService) {...}

    public void addDelivery(@NonNull final Delivery delivery) {...}

    public Delivery getDeliveryById(@NonNull final String deliveryId) {...}

    public List<Delivery> getDeliveriesByDeliveryBoyId(@NonNull final String deliveryBoyId) {...}

    public OrderStatus getOrderStatus(@NonNull final String deliveryId) {...}

    private void updateOrderStatus(@NonNull final Delivery delivery, @NonNull final Order order) {...}
}
```

Delivery Service will deal with all the functionalities related to the order delivery. It will interact with the **Delivery Data** and also call the **Order Service** to get the order to be delivered.

> *Admin/Restaurants can add delivery — **addDelivery***
>
> *Admin can get the delivery by Id — **getDeliveryById***
>
> *Delivery boy can see all the deliveries made — **getDeliveriesByDeliveryBoyId***
>
> *Customers can track the order status — **getOrderStatus***

Model Classes

Restaurant

```java
@AllArgsConstructor
@Getter
@Builder
@ToString
public class Restaurant {
    private String id;
    private String name;
    private Address address;
}
```

Restaurant Model

To identify the restaurant, we require the following attributes-

- Restaurant id — *unique for each* restaurant
- Restaurant name
- Address

> *We can also add Review, Rating, Open-close timings.*

Address

```java
@AllArgsConstructor
@Builder
@Getter
@ToString
public class Address {
    private String id;
    private String streetAddress;
    private String city;
    private String zipCode;
    private Location location;
}
```

Address Model

- Address id — *unique for each*
- Address
- City
- Zipcode
- Location — can contain google maps link or longitude and latitude values.

> *There is* ***One-to-Many*** *mapping from* ***Address to Restaurant.*** *For eg- restaurants in a mall.*

User

```java
@AllArgsConstructor
@Getter
@Builder
@ToString
public class User {
    private String id;
    private String name;
    private long phoneNo;
    private Address address;
}
```

User Model

- User id — *unique for each*
- User name
- Phone no
- Address

> *We can also add First name, Last name, Gender, Bio and some more details related to the user.*

Food Menu

```java
@AllArgsConstructor
@Builder
@Getter
@ToString
public class FoodMenu {
    private String id;
    private List<String> restaurantIds;
    private List<MenuItem> menuItemList;
}
```

Food Menu Model

- Food menu id — *unique for each*
- Restaurants ids — *multiple*
- Menu Items — *multiple*

> *One-to-Many mapping from* ***FoodMenu to Restaurant****. For eg- Restaurant having franchises in multiple cities.*

Menu Item

```java
@AllArgsConstructor
@Builder
@Getter
@ToString
public class MenuItem {
    private String id;
    private String itemName;
    private CuisineType cuisineType;
    private MealType mealType;
    private double price;
}
```

- Menu Id — *unique for each*
- Name of Item
- Cuisine Type
- Meal Type
- Price

> *We can also add Rating, Review of items, Number of times ordered a particular item.*

```
public enum CuisineType {
    INDIAN_CUISINE,
    INDIAN_CHINESE,
    ITALIAN_CUISINE,
    ITALIAN_AMERICAN,
    CHINESE_CUISINE,
    GUJARATI,
    HYDERABAD,
    JAPANESE,
    SOUTH_INDIAN,
    KOREAN
}
```

```
public enum MealType {
    BREAKFAST,
    BRUNCH,
    ELEVENSES,
    LUNCH,
    DINNER,
    SUPPER,
    AFTERNOON_TEA,
    HIGH_TEA
}
```

Cuisines and Meals

Order

```
@Getter
@ToString
public class Order {
    private String id;
    private String userId;
    private String restaurantId;
    private List<MenuItem> menuItemList;
    private OrderStatus orderStatus;
```

Order Model

- Order id — *unique for each order*
- User id — *who orders*
- Restaurant id — *orders from*
- Menu Items to be ordered
- Status of Order

. . .

Bill

```
@AllArgsConstructor
@Builder
@Getter
@ToString
public class Bill {
    private String id;
    private double totalCost;
    private double discount;
    private double tax;
    private double amountToBePaid;
}
```

- Bill Id — *unique for each bill*
- Total Cost of the Bill
- Discount applied
- Total tax
- Amount to be paid

> *We can also add tax bifurcation — CGST, SGST for India. Discount bifurcation — card discount, coupon discount.*

Payment

```
@Getter
@ToString
public class Payment {
    private String id;
    private String orderId;
    private Map<PaymentType, Double> amountPaid;
    private CouponCode couponCode;
    private PaymentStatus paymentStatus;
```

Payment Model

- Payment Id — *unique for each*
- Order Id — *for order details*
- Amount Paid by user
- Coupon code applied
- Status of Payment

Delivery

```
@AllArgsConstructor
@Builder
@Getter
@ToString
public class Delivery {
    private String id;
    private String deliveryBoyId;
    private String userId;
    private String orderId;
    private Date deliveryTime;
}
```

- Delivery Id — *unique for each*
- Delivery boy Id — *User model*
- User Id — *for address details*
- Order Id — *for order details*
- Delivery Time — *for a status update*

> *We can also add feedback, ratings for the delivery.*

```
public enum OrderStatus {
    PENDING,
    WAITING_FOR_PAYMENT,
    PLACED,
    CANCELLED,
    OUT_FOR_DELIVERY,
    DELIVERED
}
```

```
public enum PaymentStatus {
    PENDING,
    APPROVED,
    DECLINED,
    TIMEOUT
}
```

```
public enum PaymentType {
    CREDIT_CARD,
    DEBIT_CARD,
    CASH_ON_DELIVERY,
    AMAZON_PAY,
    PAYTM,
    WALLET,
    PAYPAL
}
```

```
public enum CouponCode {
    TWENTY_PERCENT_OFF,
    FIVE_HUNDRED_OFF
}
```

Conclusion

Always think of the requirements as

> *Customer Point of View*

Almost every one of us has used some food delivery app at least once. So it is easy for us to relate things in that perspective only. Another point that needs to be considered is

Always follow the SOLID Design Principle