## Low Level Design Questions with Solutions

### 1. Library Management System

#### System Overview

Design a library management system that handles book inventory, member management, borrowing/returning books, and fine calculations.

**Core Entity Classes**

```
// Book Management
public class Book {
    private String ISBN;
    private String title;
    private String author;
    private String publisher;
    private BookCategory category;
    private int totalCopies;
    private int availableCopies;
    private String location;

    public boolean isAvailable() {
        return availableCopies > 0;
    }

    public void borrowBook() {
        if (isAvailable()) {
            availableCopies--;
        }
    }

    public void returnBook() {
        availableCopies++;
    }
}

// Member Management
public abstract class Member {
    protected String memberId;
    protected String name;
    protected String email;
```

```
    protected String phone;
    protected Address address;
    protected Date membershipDate;
    protected MemberStatus status;
    protected List<BookLending> borrowedBooks;

    public abstract int getMaxBooksAllowed();
    public abstract int getMaxLendingDays();

    public boolean canBorrowBook() {
        return borrowedBooks.size() < getMaxBooksAllowed();
    }
}

public class StudentMember extends Member {
    private String studentId;
    private String university;

    @Override
    public int getMaxBooksAllowed() { return 5; }

    @Override
    public int getMaxLendingDays() { return 14; }
}

public class FacultyMember extends Member {
    private String department;
    private String designation;

    @Override
    public int getMaxBooksAllowed() { return 10; }

    @Override
    public int getMaxLendingDays() { return 30; }
}

public class GeneralMember extends Member {
    @Override
    public int getMaxBooksAllowed() { return 3; }

    @Override
    public int getMaxLendingDays() { return 7; }
}
```

```
// Lending System
public class BookLending {
   private String lendingId;
   private Book book;
   private Member member;
   private Date issueDate;
   private Date dueDate;
   private Date returnDate;
   private LendingStatus status;
   private double fine;

   public boolean isOverdue() {
      return new Date().after(dueDate) && status == LendingStatus.ISSUED;
   }

   public double calculateFine() {
      if (isOverdue()) {
         long overdueDays = ChronoUnit.DAYS.between(
            dueDate.toInstant(),
            (returnDate != null ? returnDate : new Date()).toInstant()
         );
         return overdueDays * 1.0; // $1 per day
      }
      return 0.0;
   }

   public void returnBook() {
      this.returnDate = new Date();
      this.status = LendingStatus.RETURNED;
      this.fine = calculateFine();
      book.returnBook();
   }
}

// Library System
public class Library {
   private String libraryId;
   private String name;
   private Address address;
   private List<Book> books;
   private List<Member> members;
   private List<BookLending> lendings;
   private Map<String, Book> bookCatalog;
```

```
public List<Book> searchBooks(SearchCriteria criteria) {
    // Implementation for book search by title, author, ISBN, category
    return books.stream()
        .filter(book -> matchesCriteria(book, criteria))
        .collect(Collectors.toList());
}

public BookLending issueBook(String memberId, String ISBN) {
    Member member = findMember(memberId);
    Book book = findBook(ISBN);

    if (member == null || book == null) {
        throw new IllegalArgumentException("Invalid member or book");
    }

    if (!member.canBorrowBook()) {
        throw new RuntimeException("Member has reached borrowing limit");
    }

    if (!book.isAvailable()) {
        throw new RuntimeException("Book is not available");
    }

    BookLending lending = new BookLending();
    lending.setLendingId(generateLendingId());
    lending.setBook(book);
    lending.setMember(member);
    lending.setIssueDate(new Date());
    lending.setDueDate(calculateDueDate(member.getMaxLendingDays()));
    lending.setStatus(LendingStatus.ISSUED);

    book.borrowBook();
    member.getBorrowedBooks().add(lending);
    lendings.add(lending);

    return lending;
}

public double returnBook(String lendingId) {
    BookLending lending = findLending(lendingId);
    if (lending == null) {
        throw new IllegalArgumentException("Invalid lending ID");
    }
```

```
        lending.returnBook();
        lending.getMember().getBorrowedBooks().remove(lending);

        return lending.getFine();
    }

    public List<BookLending> getOverdueBooks() {
        return lendings.stream()
            .filter(BookLending::isOverdue)
            .collect(Collectors.toList());
    }
}

// Enums
public enum BookCategory {
    FICTION, NON_FICTION, SCIENCE, TECHNOLOGY, HISTORY, BIOGRAPHY
}

public enum MemberStatus {
    ACTIVE, SUSPENDED, EXPIRED, CANCELLED
}

public enum LendingStatus {
    ISSUED, RETURNED, LOST, RENEWED
}
```

### 2. Hotel Management System

#### System Overview

Design a hotel management system for room booking customer management, billing, and hotel services.

#### Core Entity Classes

```
// Room Management public abstract class Room { protected String roomNumber;
protected RoomType roomType; protected double basePrice; protected RoomStatus
status; protected List features; protected int floor;

public abstract double getPricePerNight();
public abstract int getMaxOccupancy();
```

```
public boolean isAvailable(Date checkIn, Date checkOut) {
    // Check if room is available for given dates
    return status == RoomStatus.AVAILABLE;
}


}
```

public class StandardRoom extends Room { @Override public double getPricePerNight() { return basePrice; }

```
@Override
public int getMaxOccupancy() { return 2; }


}
```

public class DeluxeRoom extends Room { private boolean hasBalcony;

```
@Override
public double getPricePerNight() { return basePrice * 1.5; }

@Override
public int getMaxOccupancy() { return 3; }


}
```

public class SuiteRoom extends Room { private int numberOfRooms; private boolean hasKitchen;

```
@Override
public double getPricePerNight() { return basePrice * 2.5; }

@Override
public int getMaxOccupancy() { return 6; }


}
```

// Guest Management public class Guest { private String guestId; private String firstName; private String lastName; private String email; private String phone; private Address address; private String idProof; private List bookings;

```
public String getFullName() {
    return firstName + " " + lastName;
}


}
```

// Booking System public class Booking { private String bookingId; private Guest primaryGuest; private List additionalGuests; private List rooms; private Date checkInDate; private Date checkOutDate; private int numberOfNights; private BookingStatus status; private double totalAmount; private List services; private Payment payment;

```
public double calculateTotalAmount() {
    double roomCharges = rooms.stream()
        .mapToDouble(room -> room.getPricePerNight() *
numberOfNights)
        .sum();

    double serviceCharges = services.stream()
        .mapToDouble(Service::getCost)
        .sum();

    double taxes = (roomCharges + serviceCharges) * 0.18; // 18% tax

    return roomCharges + serviceCharges + taxes;
}

public void checkIn() {
    if (status != BookingStatus.CONFIRMED) {
        throw new IllegalStateException("Booking must be confirmed
before check-in");
    }

    for (Room room : rooms) {
        room.setStatus(RoomStatus.OCCUPIED);
    }

    status = BookingStatus.CHECKED_IN;
}

public Invoice checkOut() {
    if (status != BookingStatus.CHECKED_IN) {
        throw new IllegalStateException("Guest must be checked in
before check-out");
```

```
    }

    for (Room room : rooms) {
        room.setStatus(RoomStatus.CLEANING);
    }

    status = BookingStatus.COMPLETED;
    totalAmount = calculateTotalAmount();

    return generateInvoice();
}


}
```

// Hotel Services public abstract class Service { protected String serviceId; protected String serviceName; protected double cost; protected ServiceStatus status;

```
public abstract void executeService();


}
```

public class RoomService extends Service { private String roomNumber; private List items; private Date orderTime;

```
@Override
public void executeService() {
    // Deliver food items to room
    status = ServiceStatus.COMPLETED;
}


}
```

public class LaundryService extends Service { private List items; private Date pickupTime; private Date deliveryTime;

```
@Override
public void executeService() {
    // Process laundry
    status = ServiceStatus.IN_PROGRESS;
}
```

```
}

// Hotel Management public class Hotel { private String hotelId; private String name;
private Address address; private int starRating; private List rooms; private List guests;
private List bookings; private List services;

public List<Room> searchAvailableRooms(Date checkIn, Date checkOut,
                                       RoomType roomType, int guests) {
    return rooms.stream()
        .filter(room -> room.getRoomType() == roomType)
        .filter(room -> room.getMaxOccupancy() >= guests)
        .filter(room -> room.isAvailable(checkIn, checkOut))
        .collect(Collectors.toList());
}

public Booking createBooking(BookingRequest request) {
    List<Room> availableRooms = searchAvailableRooms(
        request.getCheckInDate(),
        request.getCheckOutDate(),
        request.getRoomType(),
        request.getNumberOfGuests()
    );

    if (availableRooms.isEmpty()) {
        throw new RuntimeException("No rooms available for selected
dates");
    }

    Booking booking = new Booking();
    booking.setBookingId(generateBookingId());
    booking.setPrimaryGuest(request.getGuest());
    booking.setRooms(availableRooms.subList(0,
request.getNumberOfRooms()));
    booking.setCheckInDate(request.getCheckInDate());
    booking.setCheckOutDate(request.getCheckOutDate());
    booking.setStatus(BookingStatus.PENDING);

    bookings.add(booking);
    return booking;
}

public boolean cancelBooking(String bookingId) {
    Booking booking = findBooking(bookingId);
    if (booking != null && booking.getStatus() ==
BookingStatus.CONFIRMED) {
```

```
        booking.setStatus(BookingStatus.CANCELLED);
        // Release rooms
        for (Room room : booking.getRooms()) {
            room.setStatus(RoomStatus.AVAILABLE);
        }
        return true;
    }
    return false;
}


}

// Enums public enum RoomType { STANDARD, DELUXE, SUITE, PRESIDENTIAL }

public enum RoomStatus { AVAILABLE, OCCUPIED, RESERVED, OUT_OF_ORDER,
CLEANING }

public enum BookingStatus { PENDING, CONFIRMED, CHECKED_IN, COMPLETED,
CANCELLED, NO_SHOW }

public enum ServiceStatus { REQUESTED, IN_PROGRESS, COMPLETED, CANCELLED }
```

### 3. Online Food Delivery System

#### System Overview

Design a food delivery system like Uber Eats or DoorDash with restaurants, customers, delivery partners, and order management.

#### Core Entity Classes

```
// User Management public abstract class User { protected String userId; protected
String name; protected String email; protected String phone; protected Address address;
protected UserStatus status;

public abstract UserType getUserType();


}
```

public class Customer extends User { private List orderHistory; private PaymentMethod preferredPayment; private List

savedAddresses;

```
@Override
public UserType getUserType() {
    return UserType.CUSTOMER;
}

public Order placeOrder(Cart cart, Address deliveryAddress) {
    Order order = new Order();
    order.setOrderId(generateOrderId());
    order.setCustomer(this);
    order.setRestaurant(cart.getRestaurant());
    order.setItems(cart.getItems());
    order.setDeliveryAddress(deliveryAddress);
    order.setStatus(OrderStatus.PLACED);
    order.calculateTotalAmount();

    orderHistory.add(order);
    return order;
}


}
```

public class DeliveryPartner extends User { private String vehicleDetails; private String licenseNumber; private boolean isAvailable; private Location currentLocation; private List deliveryHistory; private double rating; private PartnerStatus partnerStatus;

```
@Override
public UserType getUserType() {
    return UserType.DELIVERY_PARTNER;
}

public void acceptOrder(Order order) {
    if (!isAvailable) {
        throw new IllegalStateException("Partner not available");
    }

    order.setDeliveryPartner(this);
    order.setStatus(OrderStatus.ACCEPTED);
    isAvailable = false;
}

public void markOrderPickedUp(Order order) {
```

```
    order.setStatus(OrderStatus.PICKED_UP);
    order.setPickupTime(new Date());
}

public void markOrderDelivered(Order order) {
    order.setStatus(OrderStatus.DELIVERED);
    order.setDeliveryTime(new Date());
    isAvailable = true;
}


}
```

// Restaurant Management public class Restaurant { private String restaurantId; private String name; private Address address; private String cuisine; private List menu; private boolean isOpen; private double rating; private int preparationTime; private double deliveryRadius;

```
public boolean canDeliverTo(Address deliveryAddress) {
    double distance = calculateDistance(address, deliveryAddress);
    return distance <= deliveryRadius;
}

public List<MenuItem> getAvailableItems() {
    return menu.stream()
        .filter(MenuItem::isAvailable)
        .collect(Collectors.toList());
}

public void acceptOrder(Order order) {
    order.setStatus(OrderStatus.CONFIRMED);
    order.setEstimatedPreparationTime(preparationTime);
}


}
```

// Menu Management public class MenuItem { private String itemId; private String name; private String description; private double price; private FoodCategory category; private List ingredients; private boolean isVegetarian; private boolean isAvailable; private String imageUrl; private List customizations;

```
public double calculatePrice(List<CustomizationOption>
selectedOptions) {
    double totalPrice = price;
    for (CustomizationOption option : selectedOptions) {
```

```
        totalPrice += option.getAdditionalPrice();
    }
    return totalPrice;
}


}
```

public class Customization { private String customizationId; private String name; private CustomizationType type; private List options; private boolean isRequired;

```
public boolean isValidSelection(List<CustomizationOption>
selectedOptions) {
    if (isRequired && selectedOptions.isEmpty()) {
        return false;
    }

    if (type == CustomizationType.SINGLE_SELECT &&
selectedOptions.size() > 1) {
        return false;
    }

    return true;
}


}
```

// Order Management public class Order { private String orderId; private Customer customer; private Restaurant restaurant; private List items; private Address deliveryAddress; private OrderStatus status; private DeliveryPartner deliveryPartner; private Date orderTime; private Date estimatedDeliveryTime; private Date deliveryTime; private Payment payment; private double itemTotal; private double deliveryFee; private double taxes; private double totalAmount; private String specialInstructions;

```
public void calculateTotalAmount() {
    itemTotal = items.stream()
        .mapToDouble(OrderItem::getTotalPrice)
        .sum();

    deliveryFee = calculateDeliveryFee();
    taxes = itemTotal * 0.08; // 8% tax
    totalAmount = itemTotal + deliveryFee + taxes;
}

private double calculateDeliveryFee() {
```

```
    double distance = calculateDistance(
        restaurant.getAddress(),
        deliveryAddress
    );
    return Math.max(2.0, distance * 0.5); // Minimum $2, $0.5 per km
}

public void updateStatus(OrderStatus newStatus) {
    this.status = newStatus;
    notifyCustomer();

    if (newStatus == OrderStatus.READY_FOR_PICKUP) {
        findAvailableDeliveryPartner();
    }
}


}
```

public class OrderItem { private MenuItem menuItem; private int quantity; private List selectedCustomizations; private String specialRequest; private double unitPrice; private double totalPrice;

```
public void calculateTotalPrice() {
    unitPrice = menuItem.calculatePrice(selectedCustomizations);
    totalPrice = unitPrice * quantity;
}


}
```

// Shopping Cart public class Cart { private String cartId; private Customer customer; private Restaurant restaurant; private List items; private double totalAmount;

```
public void addItem(MenuItem menuItem, int quantity,
                    List<CustomizationOption> customizations) {
    if (restaurant != null
&& !restaurant.equals(menuItem.getRestaurant())) {
        throw new IllegalArgumentException("Items from different
restaurants not allowed");
    }

    if (restaurant == null) {
        restaurant = menuItem.getRestaurant();
    }
```

```
    OrderItem orderItem = new OrderItem();
    orderItem.setMenuItem(menuItem);
    orderItem.setQuantity(quantity);
    orderItem.setSelectedCustomizations(customizations);
    orderItem.calculateTotalPrice();

    items.add(orderItem);
    calculateTotalAmount();
}

public void removeItem(String itemId) {
    items.removeIf(item ->
item.getMenuItem().getItemId().equals(itemId));
    calculateTotalAmount();
}

public void calculateTotalAmount() {
    totalAmount = items.stream()
        .mapToDouble(OrderItem::getTotalPrice)
        .sum();
}

public void clearCart() {
    items.clear();
    restaurant = null;
    totalAmount = 0.0;
}


}
```

// Enums public enum UserType { CUSTOMER, RESTAURANT_OWNER, DELIVERY_PARTNER, ADMIN }

public enum OrderStatus { PLACED, CONFIRMED, PREPARING, READY_FOR_PICKUP, ACCEPTED, PICKED_UP, OUT_FOR_DELIVERY, DELIVERED, CANCELLED }

public enum FoodCategory { APPETIZER, MAIN_COURSE, DESSERT, BEVERAGE, SNACK }

public enum CustomizationType { SINGLE_SELECT, MULTI_SELECT, TEXT_INPUT }

public enum PartnerStatus { ACTIVE, INACTIVE, SUSPENDED, ON_DELIVERY }

## 4. Elevator System

### System Overview

Design an elevator control system for a multi-floor building with multiple elevators, request handling, and optimization algorithms.

### Core Entity Classes

```
// Elevator System
public class ElevatorController {
    private List<Elevator> elevators;
    private PriorityQueue<ElevatorRequest> upRequests;
    private PriorityQueue<ElevatorRequest> downRequests;
    private RequestProcessor requestProcessor;
    private ElevatorScheduler scheduler;

    public void requestElevator(int floor, Direction direction) {
        ElevatorRequest request = new ElevatorRequest(floor, direction);

        if (direction == Direction.UP) {
            upRequests.offer(request);
        } else {
            downRequests.offer(request);
        }

        scheduler.scheduleRequest(request);
    }

    public void processInternalRequest(int elevatorId, int targetFloor) {
        Elevator elevator = findElevator(elevatorId);
        elevator.addInternalRequest(targetFloor);
    }

    public Elevator findOptimalElevator(ElevatorRequest request) {
        return elevators.stream()
            .filter(elevator -> elevator.getStatus() != ElevatorStatus.MAINTENANCE)
            .min((e1, e2) -> compareElevatorEfficiency(e1, e2, request))
            .orElse(null);
    }

    private int compareElevatorEfficiency(Elevator e1, Elevator e2,
                                          ElevatorRequest request) {
```

```
        int distance1 = Math.abs(e1.getCurrentFloor() - request.getFloor());
        int distance2 = Math.abs(e2.getCurrentFloor() - request.getFloor());

        // Consider direction compatibility
        if (isDirectionCompatible(e1, request) &&
            !isDirectionCompatible(e2, request)) {
            return -1;
        }
        if (!isDirectionCompatible(e1, request) &&
            isDirectionCompatible(e2, request)) {
            return 1;
        }

        return Integer.compare(distance1, distance2);
    }
}

// Elevator Implementation
public class Elevator {
    private int elevatorId;
    private int currentFloor;
    private Direction currentDirection;
    private ElevatorStatus status;
    private int capacity;
    private int currentLoad;
    private PriorityQueue<Integer> upRequests;
    private PriorityQueue<Integer> downRequests;
    private List<ElevatorRequest> externalRequests;
    private Door door;
    private Display display;

    public void move() {
        if (hasRequests()) {
            int nextFloor = getNextFloor();

            if (nextFloor > currentFloor) {
                currentDirection = Direction.UP;
                currentFloor++;
            } else if (nextFloor < currentFloor) {
                currentDirection = Direction.DOWN;
                currentFloor--;
            }

            display.updateFloor(currentFloor);
```

```
      if (shouldStopAtFloor(currentFloor)) {
        stop();
      }
    } else {
      status = ElevatorStatus.IDLE;
      currentDirection = Direction.IDLE;
    }
  }

  private void stop() {
    status = ElevatorStatus.STOPPED;
    door.open();

    // Remove requests for current floor
    removeRequestsForFloor(currentFloor);

    // Simulate door close after delay
    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      @Override
      public void run() {
        door.close();
        status = ElevatorStatus.MOVING;
      }
    }, 3000); // 3 seconds
  }

  public void addInternalRequest(int floor) {
    if (floor < 1 || floor > Building.MAX_FLOORS) {
      throw new IllegalArgumentException("Invalid floor number");
    }

    if (currentDirection == Direction.UP ||
      (currentDirection == Direction.IDLE && floor > currentFloor)) {
      upRequests.offer(floor);
    } else {
      downRequests.offer(floor);
    }
  }

  public void addExternalRequest(ElevatorRequest request) {
    externalRequests.add(request);
```

```
    if (request.getDirection() == Direction.UP) {
      upRequests.offer(request.getFloor());
    } else {
      downRequests.offer(request.getFloor());
    }
  }

  private int getNextFloor() {
    if (currentDirection == Direction.UP && !upRequests.isEmpty()) {
      return upRequests.peek();
    } else if (currentDirection == Direction.DOWN && !downRequests.isEmpty()) {
      return Collections.max(downRequests);
    } else if (!upRequests.isEmpty()) {
      currentDirection = Direction.UP;
      return upRequests.peek();
    } else if (!downRequests.isEmpty()) {
      currentDirection = Direction.DOWN;
      return Collections.max(downRequests);
    }

    return currentFloor;
  }

  private boolean shouldStopAtFloor(int floor) {
    // Stop if there's an internal request for this floor
    if (upRequests.contains(floor) || downRequests.contains(floor)) {
      return true;
    }

    // Stop if there's a compatible external request
    return externalRequests.stream()
      .anyMatch(req -> req.getFloor() == floor &&
            (req.getDirection() == currentDirection ||
             currentDirection == Direction.IDLE));
  }

  public boolean isOverloaded() {
    return currentLoad > capacity;
  }

  public void emergencyStop() {
    status = ElevatorStatus.EMERGENCY;
    upRequests.clear();
    downRequests.clear();
```

```
        externalRequests.clear();
    }
}

// Supporting Classes
public class ElevatorRequest {
    private int floor;
    private Direction direction;
    private long timestamp;
    private RequestPriority priority;

    public ElevatorRequest(int floor, Direction direction) {
        this.floor = floor;
        this.direction = direction;
        this.timestamp = System.currentTimeMillis();
        this.priority = RequestPriority.NORMAL;
    }

    public int getWaitTime() {
        return (int) (System.currentTimeMillis() - timestamp) / 1000;
    }
}

public class Door {
    private DoorStatus status;
    private int elevatorId;

    public void open() {
        if (status == DoorStatus.CLOSED) {
            status = DoorStatus.OPENING;
            // Simulate opening time
            Timer timer = new Timer();
            timer.schedule(new TimerTask() {
                @Override
                public void run() {
                    status = DoorStatus.OPEN;
                }
            }, 2000);
        }
    }

    public void close() {
        if (status == DoorStatus.OPEN) {
            status = DoorStatus.CLOSING;
```

```
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
          @Override
          public void run() {
            status = DoorStatus.CLOSED;
          }
        }, 2000);
      }
    }

    public void forceClose() {
      status = DoorStatus.CLOSED;
    }
}

public class Display {
    private int currentFloor;
    private Direction direction;
    private String message;

    public void updateFloor(int floor) {
      this.currentFloor = floor;
      showFloorInfo();
    }

    public void updateDirection(Direction direction) {
      this.direction = direction;
      showDirectionInfo();
    }

    public void showMessage(String message) {
      this.message = message;
    }

    private void showFloorInfo() {
      System.out.println("Current Floor: " + currentFloor);
    }

    private void showDirectionInfo() {
      System.out.println("Direction: " + direction);
    }
}

public class ElevatorScheduler {
```

```
    private List<Elevator> elevators;
    private SchedulingAlgorithm algorithm;

    public void scheduleRequest(ElevatorRequest request) {
        Elevator optimalElevator = algorithm.findBestElevator(elevators, request);
        if (optimalElevator != null) {
            optimalElevator.addExternalRequest(request);
        }
    }

    public void setSchedulingAlgorithm(SchedulingAlgorithm algorithm) {
        this.algorithm = algorithm;
    }
}

// Scheduling Algorithms
public interface SchedulingAlgorithm {
    Elevator findBestElevator(List<Elevator> elevators, ElevatorRequest request);
}

public class SCANAlgorithm implements SchedulingAlgorithm {
    @Override
    public Elevator findBestElevator(List<Elevator> elevators,
                                     ElevatorRequest request) {
        return elevators.stream()
            .filter(e -> e.getStatus() == ElevatorStatus.IDLE ||
                    isDirectionCompatible(e, request))
            .min((e1, e2) -> Integer.compare(
                calculateCost(e1, request),
                calculateCost(e2, request)))
            .orElse(elevators.get(0));
    }

    private int calculateCost(Elevator elevator, ElevatorRequest request) {
        int distance = Math.abs(elevator.getCurrentFloor() - request.getFloor());
        int directionPenalty = isDirectionCompatible(elevator, request) ? 0 : 10;
        return distance + directionPenalty;
    }
}

public class FCFSAlgorithm implements SchedulingAlgorithm {
    @Override
    public Elevator findBestElevator(List<Elevator> elevators,
                                     ElevatorRequest request) {
```

```
        return elevators.stream()
            .filter(e -> e.getStatus() != ElevatorStatus.MAINTENANCE)
            .min((e1, e2) -> Integer.compare(
                Math.abs(e1.getCurrentFloor() - request.getFloor()),
                Math.abs(e2.getCurrentFloor() - request.getFloor())))
            .orElse(null);
    }
}

// Enums
public enum Direction {
    UP, DOWN, IDLE
}

public enum ElevatorStatus {
    IDLE, MOVING, STOPPED, MAINTENANCE, EMERGENCY
}

public enum DoorStatus {
    OPEN, CLOSED, OPENING, CLOSING
}

public enum RequestPriority {
    LOW, NORMAL, HIGH, EMERGENCY
}
```

### 5. Social Media Platform (Twitter/X Clone)

#### System Overview

Design a social media platform with user profiles, posts, following system, timeline generation, and notifications.

#### Core Entity Classes

```
// User Management public class User { private String userId; private String username; private String email; private String displayName; private String bio; private String profilePictureUrl; private Date joinDate; private boolean isVerified; private UserStatus status; private Privacy privacy; private List followers; private List following; private int followerCount; private int followingCount; private int postCount;
```

```
public void follow(User userToFollow) {
    if (userToFollow.getPrivacy() == Privacy.PRIVATE) {
        sendFollowRequest(userToFollow);
    } else {
        following.add(userToFollow.getUserId());
        userToFollow.getFollowers().add(this.userId);
        followingCount++;
        userToFollow.followerCount++;
    }
}

public void unfollow(User userToUnfollow) {
    following.remove(userToUnfollow.getUserId());
    userToUnfollow.getFollowers().remove(this.userId);
    followingCount--;
    userToUnfollow.followerCount--;
}

public boolean canViewProfile(User viewer) {
    if (privacy == Privacy.PUBLIC) return true;
    if (viewer.getUserId().equals(this.userId)) return true;
    return followers.contains(viewer.getUserId());
}

private void sendFollowRequest(User targetUser) {
    FollowRequest request = new FollowRequest();
    request.setRequestId(UUID.randomUUID().toString());
    request.setRequester(this);
    request.setTargetUser(targetUser);
    request.setStatus(RequestStatus.PENDING);
    request.setTimestamp(new Date());

    // Add to target user's pending requests
    NotificationService.sendNotification(targetUser,
        new FollowRequestNotification(request));
}


}
```

// Post Management public class Post { private String postId; private User author; private String content; private Date createdAt; private Date updatedAt; private PostType type; private List media; private PostVisibility visibility; private Location location; private int likeCount; private int retweetCount; private int replyCount; private List hashtags; private

List mentions; private Post originalPost; // For retweets private Post parentPost; // For replies

```
public boolean canBeViewedBy(User viewer) {
    if (visibility == PostVisibility.PUBLIC) return true;
    if (author.getUserId().equals(viewer.getUserId())) return true;

    if (visibility == PostVisibility.FOLLOWERS_ONLY) {
        return author.getFollowers().contains(viewer.getUserId());
    }

    return false;
}

public void like(User user) {
    Like like = new Like();
    like.setLikeId(UUID.randomUUID().toString());
    like.setUser(user);
    like.setPost(this);
    like.setTimestamp(new Date());

    likeCount++;

    // Notify post author
    if (!author.getUserId().equals(user.getUserId())) {
        NotificationService.sendNotification(author,
            new LikeNotification(like));
    }
}

public Post retweet(User user, String comment) {
    Post retweet = new Post();
    retweet.setPostId(UUID.randomUUID().toString());
    retweet.setAuthor(user);
    retweet.setContent(comment);
    retweet.setType(PostType.RETWEET);
    retweet.setOriginalPost(this);
    retweet.setCreatedAt(new Date());
    retweet.setVisibility(PostVisibility.PUBLIC);

    this.retweetCount++;

    // Notify original author
    NotificationService.sendNotification(author,
        new RetweetNotification(retweet));
```

```
    return retweet;
}

public Post reply(User user, String content) {
    Post reply = new Post();
    reply.setPostId(UUID.randomUUID().toString());
    reply.setAuthor(user);
    reply.setContent(content);
    reply.setType(PostType.REPLY);
    reply.setParentPost(this);
    reply.setCreatedAt(new Date());

    this.replyCount++;

    // Notify post author
    if (!author.getUserId().equals(user.getUserId())) {
        NotificationService.sendNotification(author,
            new ReplyNotification(reply));
    }

    return reply;
}

public void extractHashtagsAndMentions() {
    // Extract hashtags
    Pattern hashtagPattern = Pattern.compile("#\\w+");
    Matcher hashtagMatcher = hashtagPattern.matcher(content);
    while (hashtagMatcher.find()) {
        hashtags.add(hashtagMatcher.group().substring(1));
    }

    // Extract mentions
    Pattern mentionPattern = Pattern.compile("@\\w+");
    Matcher mentionMatcher = mentionPattern.matcher(content);
    while (mentionMatcher.find()) {
        mentions.add(mentionMatcher.group().substring(1));
    }
}


}
```

// Timeline Generation public class Timeline { private String timelineId; private User user; private List posts; private Date lastUpdated; private TimelineType type;

```
public void refreshTimeline() {
    switch (type) {
        case HOME:
            generateHomeTimeline();
            break;
        case USER:
            generateUserTimeline();
            break;
        case TRENDING:
            generateTrendingTimeline();
            break;
    }
    lastUpdated = new Date();
}

private void generateHomeTimeline() {
    List<Post> timelinePosts = new ArrayList<>();

    // Get posts from followed users
    for (String followedUserId : user.getFollowing()) {
        User followedUser = UserService.getUser(followedUserId);
        List<Post> userPosts = PostService.getUserPosts(followedUser,
50);
        timelinePosts.addAll(userPosts);
    }

    // Add user's own posts
    timelinePosts.addAll(PostService.getUserPosts(user, 20));

    // Sort by relevance and recency
    posts = timelinePosts.stream()
        .filter(post -> post.canBeViewedBy(user))
        .sorted((p1, p2) -> calculateRelevanceScore(p2) -
calculateRelevanceScore(p1))
        .limit(100)
        .collect(Collectors.toList());
}

private int calculateRelevanceScore(Post post) {
    int score = 0;

    // Recency factor
    long hoursAgo = ChronoUnit.HOURS.between(
        post.getCreatedAt().toInstant(),
        Instant.now()
```

```
    );
    score += Math.max(0, 100 - (int)hoursAgo);

    // Engagement factor
    score += post.getLikeCount() * 2;
    score += post.getRetweetCount() * 3;
    score += post.getReplyCount() * 1;

    // Author relationship factor
    if (user.getFollowing().contains(post.getAuthor().getUserId())) {
        score += 50;
    }

    return score;
}


}
```

// Notification System public abstract class Notification { protected String notificationId; protected User recipient; protected Date timestamp; protected boolean isRead; protected NotificationType type;

```
public abstract String getMessage();
public abstract void markAsRead();


}
```

public class LikeNotification extends Notification { private Like like;

```
public LikeNotification(Like like) {
    this.like = like;
    this.type = NotificationType.LIKE;
    this.timestamp = new Date();
    this.isRead = false;
}

@Override
public String getMessage() {
    return like.getUser().getDisplayName() + " liked your post";
}

@Override
public void markAsRead() {
    this.isRead = true;
```

```
}


}

public class FollowNotification extends Notification { private User follower;

@Override
public String getMessage() {
    return follower.getDisplayName() + " started following you";
}

@Override
public void markAsRead() {
    this.isRead = true;
}


}

public class RetweetNotification extends Notification { private Post retweet;

@Override
public String getMessage() {
    return retweet.getAuthor().getDisplayName() + " retweeted your
post";
}

@Override
public void markAsRead() {
    this.isRead = true;
}


}

// Trending and Search public class TrendingService { private Map<String, Integer>
hashtagCounts; private Map<String, Integer> mentionCounts; private List currentTrends;

public void updateTrends() {
    // Analyze recent posts for trending hashtags
    Date since = Date.from(Instant.now().minus(24,
ChronoUnit.HOURS));
    List<Post> recentPosts = PostService.getPostsSince(since);

    Map<String, Integer> hashtagFrequency = new HashMap<>();
```

```
    for (Post post : recentPosts) {
        for (String hashtag : post.getHashtags()) {
            hashtagFrequency.put(hashtag,
                hashtagFrequency.getOrDefault(hashtag, 0) + 1);
        }
    }

    // Create trending topics
    currentTrends = hashtagFrequency.entrySet().stream()
        .sorted((e1, e2) -> e2.getValue().compareTo(e1.getValue()))
        .limit(10)
        .map(entry -> new TrendingTopic(entry.getKey(),
entry.getValue()))
        .collect(Collectors.toList());
}

public List<TrendingTopic> getTrendingTopics() {
    return currentTrends;
}


}
```

public class SearchService { public List searchPosts(String query, User searcher, SearchFilter filter) { List results = new ArrayList<>();

```
   // Search in post content
    List<Post> contentMatches = PostService.searchByContent(query);

    // Search in hashtags
    if (query.startsWith("#")) {
        String hashtag = query.substring(1);
        List<Post> hashtagMatches =
PostService.getPostsByHashtag(hashtag);
        results.addAll(hashtagMatches);
    }

    // Search in mentions
    if (query.startsWith("@")) {
        String username = query.substring(1);
        User mentionedUser = UserService.getUserByUsername(username);
        if (mentionedUser != null) {
            List<Post> mentionMatches =
PostService.getPostsWithMention(mentionedUser);
            results.addAll(mentionMatches);
```

```
        }
    }

    results.addAll(contentMatches);

    // Filter results based on visibility
    return results.stream()
        .filter(post -> post.canBeViewedBy(searcher))
        .filter(post -> applySearchFilter(post, filter))
        .distinct()
        .sorted((p1, p2) ->
p2.getCreatedAt().compareTo(p1.getCreatedAt()))
        .collect(Collectors.toList());
}

public List<User> searchUsers(String query, User searcher) {
    return UserService.searchUsers(query).stream()
        .filter(user -> user.canViewProfile(searcher))
        .collect(Collectors.toList());
}


}
```

// Supporting Classes public class Media { private String mediaId; private MediaType type; private String url; private String thumbnail; private int width; private int height; private long fileSize; private String altText;

```
public boolean isValid() {
    return url != null && !url.isEmpty() &&
           type != null && fileSize > 0;
}


}
```

public class Like { private String likeId; private User user; private Post post; private Date timestamp; }

public class FollowRequest { private String requestId; private User requester; private User targetUser; private RequestStatus status; private Date timestamp;

```
public void approve() {
    requester.getFollowing().add(targetUser.getUserId());
    targetUser.getFollowers().add(requester.getUserId());
    requester.followingCount++;
```

```
    targetUser.followerCount++;
    status = RequestStatus.APPROVED;

    // Send notification
    NotificationService.sendNotification(requester,
        new FollowApprovedNotification(this));
}

public void reject() {
    status = RequestStatus.REJECTED;
}


}
```

public class TrendingTopic { private String topic; private int postCount; private Date lastUpdated; private TrendCategory category;

```
public TrendingTopic(String topic, int postCount) {
    this.topic = topic;
    this.postCount = postCount;
    this.lastUpdated = new Date();
}


}
```

// Enums public enum PostType { ORIGINAL, RETWEET, REPLY, QUOTE_TWEET }

public enum PostVisibility { PUBLIC, FOLLOWERS_ONLY, PRIVATE, MENTIONED_ONLY }

public enum MediaType { IMAGE, VIDEO, GIF, AUDIO }

public enum UserStatus { ACTIVE, SUSPENDED, DEACTIVATED, BANNED }

public enum Privacy { PUBLIC, PRIVATE, PROTECTED }

public enum NotificationType { LIKE, RETWEET, REPLY, FOLLOW, MENTION, DIRECT_MESSAGE }

public enum TimelineType { HOME, USER, TRENDING, SEARCH_RESULTS }

public enum RequestStatus { PENDING, APPROVED, REJECTED, CANCELLED }

public enum TrendCategory { GENERAL, SPORTS, ENTERTAINMENT, TECHNOLOGY, NEWS, POLITICS }

## 6. Chat/Messaging System

### System Overview

Design a real-time messaging system supporting one-on-one chats, group chats, message delivery, and online presence.

### Core Entity Classes

// User and Presence Management public class User { private String userId; private String username; private String displayName; private String email; private String profilePictureUrl; private UserStatus status; private Date lastSeen; private boolean isOnline; private List blockedUsers; private UserSettings settings;

```
public void updateOnlineStatus(boolean online) {
    this.isOnline = online;
    if (!online) {
        this.lastSeen = new Date();
    }

    // Notify contacts about status change
    PresenceService.updateUserPresence(this);
}

public boolean canReceiveMessageFrom(User sender) {
    return !blockedUsers.contains(sender.getUserId()) &&
           !sender.getBlockedUsers().contains(this.userId);
}

public void blockUser(User userToBlock) {
    blockedUsers.add(userToBlock.getUserId());
}

public void unblockUser(User userToUnblock) {
    blockedUsers.remove(userToUnblock.getUserId());
}


}
```

// Chat Management public abstract class Chat { protected String chatId; protected List participants; protected List messages; protected Date createdAt; protected Date lastActivity; protected ChatType type; protected ChatSettings settings;

```
public abstract boolean canUserJoin(User user);
public abstract boolean canUserSendMessage(User user);

public void addMessage(Message message) {
    messages.add(message);
    lastActivity = new Date();

    // Update message delivery status
    updateMessageDeliveryStatus(message);

    // Send push notifications to participants
    notifyParticipants(message);
}

protected void updateMessageDeliveryStatus(Message message) {
    for (String participantId : participants) {
        if (!participantId.equals(message.getSender().getUserId())) {
            User participant = UserService.getUser(participantId);
            if (participant.isOnline()) {
                message.markAsDelivered(participant);
            } else {
                // Queue for delivery when user comes online
                MessageDeliveryService.queueForDelivery(message,
participant);
            }
        }
    }
}

protected void notifyParticipants(Message message) {
    for (String participantId : participants) {
        if (!participantId.equals(message.getSender().getUserId())) {
            User participant = UserService.getUser(participantId);
            NotificationService.sendPushNotification(participant,
message);
        }
    }
}


}
```

public class DirectMessage extends Chat { private String user1Id; private String user2Id;

```
public DirectMessage(User user1, User user2) {
    this.chatId = generateChatId(user1.getUserId(),
user2.getUserId());
    this.user1Id = user1.getUserId();
    this.user2Id = user2.getUserId();
    this.participants = Arrays.asList(user1Id, user2Id);
    this.type = ChatType.DIRECT_MESSAGE;
    this.createdAt = new Date();
    this.messages = new ArrayList<>();
}

@Override
public boolean canUserJoin(User user) {
    return user.getUserId().equals(user1Id) ||
           user.getUserId().equals(user2Id);
}

@Override
public boolean canUserSendMessage(User user) {
    if (!canUserJoin(user)) return false;

    User otherUser = getUserByIdBut(user.getUserId());
    return user.canReceiveMessageFrom(otherUser);
}

public User getOtherUser(String currentUserId) {
    String otherUserId = currentUserId.equals(user1Id) ? user2Id :
user1Id;
    return UserService.getUser(otherUserId);
}


}
```

public class GroupChat extends Chat { private String groupName; private String groupDescription; private String groupImageUrl; private String creatorId; private List admins; private GroupSettings groupSettings; private int maxParticipants;

```
public GroupChat(String name, User creator) {
    this.chatId = UUID.randomUUID().toString();
    this.groupName = name;
    this.creatorId = creator.getUserId();
    this.participants = new ArrayList<>();
    this.admins = new ArrayList<>();
```

```
    this.type = ChatType.GROUP_CHAT;
    this.createdAt = new Date();
    this.messages = new ArrayList<>();
    this.maxParticipants = 256;

    // Add creator as first participant and admin
    participants.add(creator.getUserId());
    admins.add(creator.getUserId());
}

@Override
public boolean canUserJoin(User user) {
    if (participants.contains(user.getUserId())) return true;
    if (participants.size() >= maxParticipants) return false;

    return groupSettings.isPublic() ||
           groupSettings.getAllowInvites();
}

@Override
public boolean canUserSendMessage(User user) {
    return participants.contains(user.getUserId()) &&
           !groupSettings.isReadOnly() ||
           admins.contains(user.getUserId());
}

public void addParticipant(User user, User addedBy) {
    if (!canUserJoin(user)) {
        throw new IllegalStateException("User cannot join this
group");
    }

    if (!admins.contains(addedBy.getUserId()) &&
        !groupSettings.getAllowInvites()) {
        throw new SecurityException("Only admins can add
participants");
    }

    participants.add(user.getUserId());

    // Send system message
    Message joinMessage = createSystemMessage(
        user.getDisplayName() + " joined the group"
    );
    addMessage(joinMessage);
```

```
}

public void removeParticipant(User user, User removedBy) {
    if (!participants.contains(user.getUserId())) return;

    boolean canRemove =
removedBy.getUserId().equals(user.getUserId()) || // Self leave
                        admins.contains(removedBy.getUserId()) || //
Admin action
                        removedBy.getUserId().equals(creatorId); //
Creator action

    if (!canRemove) {
        throw new SecurityException("Not authorized to remove
participant");
    }

    participants.remove(user.getUserId());
    admins.remove(user.getUserId()); // Remove admin rights if any

    // Send system message
    Message leaveMessage = createSystemMessage(
        user.getDisplayName() + " left the group"
    );
    addMessage(leaveMessage);
}

public void promoteToAdmin(User user, User promotedBy) {
    if (!admins.contains(promotedBy.getUserId())) {
        throw new SecurityException("Only admins can promote
members");
    }

    if (participants.contains(user.getUserId()) &&
        !admins.contains(user.getUserId())) {
        admins.add(user.getUserId());
    }
}

private Message createSystemMessage(String content) {
    Message message = new Message();
    message.setMessageId(UUID.randomUUID().toString());
    message.setContent(content);
    message.setType(MessageType.SYSTEM);
    message.setTimestamp(new Date());
```

```
    return message;
}


}
```

// Message Management public class Message { private String messageId; private User sender; private String content; private MessageType type; private Date timestamp; private Date editedAt; private boolean isEdited; private boolean isDeleted; private List attachments; private Message replyToMessage; private Map<String, MessageStatus> deliveryStatus; private Map<String, Date> readReceipts; private List reactions; private MessagePriority priority;

```
public void markAsDelivered(User user) {
    deliveryStatus.put(user.getUserId(), MessageStatus.DELIVERED);
}

public void markAsRead(User user) {
    deliveryStatus.put(user.getUserId(), MessageStatus.READ);
    readReceipts.put(user.getUserId(), new Date());
}

public void editMessage(String newContent, User editor) {
    if (!sender.getUserId().equals(editor.getUserId())) {
        throw new SecurityException("Only sender can edit message");
    }

    if (ChronoUnit.MINUTES.between(
        timestamp.toInstant(),
        Instant.now()
    ) > 15) {
        throw new IllegalStateException("Cannot edit message after 15
minutes");
    }

    this.content = newContent;
    this.isEdited = true;
    this.editedAt = new Date();
}

public void deleteMessage(User deleter) {
    if (!sender.getUserId().equals(deleter.getUserId())) {
        throw new SecurityException("Only sender can delete
message");
    }
```

```
    this.isDeleted = true;
    this.content = "This message was deleted";
    this.attachments.clear();
}

public void addReaction(User user, String emoji) {
    MessageReaction reaction = reactions.stream()
        .filter(r -> r.getEmoji().equals(emoji))
        .findFirst()
        .orElse(null);

    if (reaction == null) {
        reaction = new MessageReaction(emoji);
        reactions.add(reaction);
    }

    reaction.addUser(user);
}

public void removeReaction(User user, String emoji) {
    reactions.stream()
        .filter(r -> r.getEmoji().equals(emoji))
        .findFirst()
        .ifPresent(reaction -> {
            reaction.removeUser(user);
            if (reaction.getUsers().isEmpty()) {
                reactions.remove(reaction);
            }
        });
}

public boolean isDeliveredToAll(List<String> participants) {
    return participants.stream()
        .filter(p -> !p.equals(sender.getUserId()))
        .allMatch(p -> deliveryStatus.getOrDefault(p,
MessageStatus.SENT) ==
                      MessageStatus.DELIVERED ||
                      deliveryStatus.getOrDefault(p,
MessageStatus.SENT) ==
                      MessageStatus.READ);
}

public boolean isReadByAll(List<String> participants) {
    return participants.stream()
```

```
        .filter(p -> !p.equals(sender.getUserId()))
        .allMatch(p -> deliveryStatus.getOrDefault(p,
MessageStatus.SENT) ==
                      MessageStatus.READ);
}


}
```

// Attachment Handling public class Attachment { private String attachmentId; private String fileName; private String fileUrl; private AttachmentType type; private long fileSize; private String mimeType; private String thumbnail; private Map<String, String> metadata;

```
public boolean isValid() {
    return fileUrl != null && !fileUrl.isEmpty() &&
           type != null && fileSize > 0 && fileSize <=
getMaxFileSize();
}

private long getMaxFileSize() {
    switch (type) {
        case IMAGE: return 10 * 1024 * 1024; // 10MB
        case VIDEO: return 100 * 1024 * 1024; // 100MB
        case AUDIO: return 50 * 1024 * 1024; // 50MB
        case DOCUMENT: return 20 * 1024 * 1024; // 20MB
        default: return 5 * 1024 * 1024; // 5MB
    }
}


}
```

// Message Reactions public class MessageReaction { private String emoji; private List users; private Date firstReactionTime;

```
public MessageReaction(String emoji) {
    this.emoji = emoji;
    this.users = new ArrayList<>();
    this.firstReactionTime = new Date();
}

public void addUser(User user) {
    if (!users.contains(user)) {
        users.add(user);
    }
}
```

```
public void removeUser(User user) {
    users.remove(user);
}

public int getCount() {
    return users.size();
}


}
```

// Real-time Message Delivery public class MessageDeliveryService { private static Map<String, List> pendingMessages = new HashMap<>(); private static WebSocketConnectionManager connectionManager;

```
public static void sendMessage(Message message, Chat chat) {
    for (String participantId : chat.getParticipants()) {
        if (!participantId.equals(message.getSender().getUserId())) {
            User participant = UserService.getUser(participantId);

            if (participant.isOnline() &&
connectionManager.isConnected(participantId)) {
                // Send via WebSocket
                connectionManager.sendMessage(participantId,
message);
                message.markAsDelivered(participant);
            } else {
                // Queue for later delivery
                queueForDelivery(message, participant);
            }
        }
    }
}

public static void queueForDelivery(Message message, User user) {
    pendingMessages.computeIfAbsent(user.getUserId(), k -> new
ArrayList<>())
                   .add(message);
}

public static void deliverPendingMessages(User user) {
    List<Message> pending = pendingMessages.get(user.getUserId());
    if (pending != null && !pending.isEmpty()) {
        for (Message message : pending) {
            connectionManager.sendMessage(user.getUserId(), message);
```

```
            message.markAsDelivered(user);
        }
        pending.clear();
    }
}

public static void handleMessageRead(String messageId, User reader) {
    Message message = MessageService.getMessage(messageId);
    if (message != null) {
        message.markAsRead(reader);

        // Notify sender about read receipt
        if (reader.getSettings().isSendReadReceipts()) {
            ReadReceipt receipt = new ReadReceipt(message, reader,
new Date());

connectionManager.sendReadReceipt(message.getSender().getUserId(),
receipt);
        }
    }
}


}
```

// Supporting Classes public class UserSettings { private boolean sendReadReceipts; private boolean allowGroupInvites; private boolean showLastSeen; private boolean allowMessagePreview; private NotificationSettings notificationSettings;

```
// Getters and setters


}
```

public class GroupSettings { private boolean isPublic; private boolean allowInvites; private boolean isReadOnly; private boolean allowMembersToAddOthers; private boolean showMemberList;

```
// Getters and setters


}
```

public class ChatSettings { private boolean isMuted; private boolean isArchived; private boolean isPinned; private String customName; private Date muteUntil;

```
// Getters and setters


}

// Enums public enum ChatType { DIRECT_MESSAGE, GROUP_CHAT, BROADCAST }

public enum MessageType { TEXT, IMAGE, VIDEO, AUDIO, DOCUMENT, LOCATION,
CONTACT, SYSTEM, DELETED }

public enum MessageStatus { SENT, DELIVERED, READ, FAILED }

public enum AttachmentType { IMAGE, VIDEO, AUDIO, DOCUMENT, LOCATION,
CONTACT }

public enum MessagePriority { LOW, NORMAL, HIGH, URGENT }

public enum UserStatus { ONLINE, AWAY, BUSY, OFFLINE, INVISIBLE }
```

```
*/

//
==========================================================================
===== // SOCIAL MEDIA PLATFORM - SPRING BOOT IMPLEMENTATION

//
==========================================================================
=====

// Domain Entities @Entity @Table(name = "users") @Data @NoArgsConstructor
@AllArgsConstructor @EqualsAndHashCode(callSuper = false) public class User
extends BaseEntity { @Id private String userId;

@Column(unique = true, nullable = false)
private String username;

@Column(unique = true, nullable = false)
private String email;

@Column(nullable = false)
private String displayName;

@Column(length = 500)
private String bio;
```

```
private String profilePictureUrl;

@CreationTimestamp
private LocalDateTime joinDate;

private boolean isVerified = false;

@Enumerated(EnumType.STRING)
private UserStatus status = UserStatus.ACTIVE;

@Enumerated(EnumType.STRING)
private Privacy privacy = Privacy.PUBLIC;

@ElementCollection
@CollectionTable(name = "user_followers", joinColumns =
@JoinColumn(name = "user_id"))
@Column(name = "follower_id")
private Set<String> followers = new HashSet<>();

@ElementCollection
@CollectionTable(name = "user_following", joinColumns =
@JoinColumn(name = "user_id"))
@Column(name = "following_id")
private Set<String> following = new HashSet<>();

private int followerCount = 0;
private int followingCount = 0;
private int postCount = 0;

@OneToMany(mappedBy = "author", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
private List<Post> posts = new ArrayList<>();

public void follow(User userToFollow) {
    if (userToFollow.getPrivacy() == Privacy.PRIVATE) {
        // Send follow request logic here
        return;
    }

    this.following.add(userToFollow.getUserId());
    userToFollow.getFollowers().add(this.userId);
    this.followingCount++;
    userToFollow.followerCount++;
}
```

```
public void unfollow(User userToUnfollow) {
    this.following.remove(userToUnfollow.getUserId());
    userToUnfollow.getFollowers().remove(this.userId);
    this.followingCount--;
    userToUnfollow.followerCount--;
}

public boolean canViewProfile(User viewer) {
    if (privacy == Privacy.PUBLIC) return true;
    if (viewer.getUserId().equals(this.userId)) return true;
    return followers.contains(viewer.getUserId());
}


}
```

@Entity @Table(name = "posts") @Data @NoArgsConstructor @AllArgsConstructor @EqualsAndHashCode(callSuper = false) public class Post extends BaseEntity { @Id private String postId;

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "author_id")
private User author;

@Column(length = 2000)
private String content;

@CreationTimestamp
private LocalDateTime createdAt;

@UpdateTimestamp
private LocalDateTime updatedAt;

@Enumerated(EnumType.STRING)
private PostType type = PostType.ORIGINAL;

@OneToMany(mappedBy = "post", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
private List<Media> media = new ArrayList<>();

@Enumerated(EnumType.STRING)
private PostVisibility visibility = PostVisibility.PUBLIC;

@Embedded
```

```
private Location location;

private int likeCount = 0;
private int retweetCount = 0;
private int replyCount = 0;

@ElementCollection
@CollectionTable(name = "post_hashtags", joinColumns =
@JoinColumn(name = "post_id"))
@Column(name = "hashtag")
private Set<String> hashtags = new HashSet<>();

@ElementCollection
@CollectionTable(name = "post_mentions", joinColumns =
@JoinColumn(name = "post_id"))
@Column(name = "username")
private Set<String> mentions = new HashSet<>();

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "original_post_id")
private Post originalPost; // For retweets

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "parent_post_id")
private Post parentPost; // For replies

@OneToMany(mappedBy = "post", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
private List<Like> likes = new ArrayList<>();

public boolean canBeViewedBy(User viewer) {
    if (visibility == PostVisibility.PUBLIC) return true;
    if (author.getUserId().equals(viewer.getUserId())) return true;

    if (visibility == PostVisibility.FOLLOWERS_ONLY) {
        return author.getFollowers().contains(viewer.getUserId());
    }

    return false;
}

@PrePersist
@PreUpdate
public void extractHashtagsAndMentions() {
    if (content != null) {
```

```
        // Extract hashtags
        Pattern hashtagPattern = Pattern.compile("#\\w+");
        Matcher hashtagMatcher = hashtagPattern.matcher(content);
        while (hashtagMatcher.find()) {

hashtags.add(hashtagMatcher.group().substring(1).toLowerCase());
        }

        // Extract mentions
        Pattern mentionPattern = Pattern.compile("@\\w+");
        Matcher mentionMatcher = mentionPattern.matcher(content);
        while (mentionMatcher.find()) {

mentions.add(mentionMatcher.group().substring(1).toLowerCase());
        }
    }
}


}
```

@Entity @Table(name = "likes") @Data @NoArgsConstructor @AllArgsConstructor public class Like { @Id private String likeId;

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "user_id")
private User user;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "post_id")
private Post post;

@CreationTimestamp
private LocalDateTime timestamp;

@PrePersist
public void generateId() {
    if (likeId == null) {
        likeId = UUID.randomUUID().toString();
    }
}


}
```

// Repository Layer @Repository public interface UserRepository extends JpaRepository<User, String> { Optional findByUsername(String username); Optional findByEmail(String email); List findByUsernameContainingIgnoreCaseOrDisplayNameContainingIgnoreCase( String username, String displayName);

```
@Query("SELECT u FROM User u WHERE u.userId IN :userIds")
List<User> findByUserIds(@Param("userIds") Set<String> userIds);

@Query("SELECT COUNT(u) FROM User u WHERE u.status = 'ACTIVE'")
long countActiveUsers();


}
```

@Repository public interface PostRepository extends JpaRepository<Post, String> { List findByAuthor_UserIdOrderByCreatedAtDesc(String authorId);

```
@Query("SELECT p FROM Post p WHERE p.author.userId IN :followedUsers
" +
       "AND p.visibility = 'PUBLIC' ORDER BY p.createdAt DESC")
Page<Post> findTimelinePosts(@Param("followedUsers") Set<String>
followedUsers, Pageable pageable);

@Query("SELECT p FROM Post p JOIN p.hashtags h WHERE h = :hashtag
ORDER BY p.createdAt DESC")
List<Post> findByHashtag(@Param("hashtag") String hashtag);

@Query("SELECT p FROM Post p WHERE p.content LIKE %:keyword% AND
p.visibility = 'PUBLIC'")
List<Post> searchByContent(@Param("keyword") String keyword);

@Query("SELECT COUNT(p) FROM Post p WHERE p.createdAt >= :since")
long countPostsSince(@Param("since") LocalDateTime since);


}
```

@Repository public interface LikeRepository extends JpaRepository<Like, String> { Optional findByUser_UserIdAndPost_PostId(String userId, String postId); List findByPost_PostIdOrderByTimestampDesc(String postId); boolean existsByUser_UserIdAndPost_PostId(String userId, String postId);

```
@Modifying
@Query("DELETE FROM Like l WHERE l.user.userId = :userId AND
l.post.postId = :postId")
```

```
void deleteByUserIdAndPostId(@Param("userId") String userId,
@Param("postId") String postId);


}

// Service Layer @Service @Transactional @Slf4j public class SocialMediaService {

@Autowired
private UserRepository userRepository;

@Autowired
private PostRepository postRepository;

@Autowired
private LikeRepository likeRepository;

@Autowired
private NotificationService notificationService;

@Autowired
private TimelineService timelineService;

@EventListener
@Async
public void handlePostCreated(PostCreatedEvent event) {
    timelineService.updateTimelinesForPost(event.getPost());
    log.info("Post created: {}", event.getPost().getPostId());
}

public User createUser(CreateUserRequest request) {
    // Check if username or email already exists
    if
(userRepository.findByUsername(request.getUsername()).isPresent()) {
        throw new UsernameAlreadyExistsException("Username already
exists");
    }

    if (userRepository.findByEmail(request.getEmail()).isPresent()) {
        throw new EmailAlreadyExistsException("Email already
exists");
    }

    User user = User.builder()
        .userId(UUID.randomUUID().toString())
        .username(request.getUsername())
```

```
        .email(request.getEmail())
        .displayName(request.getDisplayName())
        .bio(request.getBio())
        .build();

    return userRepository.save(user);
}

public Post createPost(String userId, CreatePostRequest request) {
    User author = userRepository.findById(userId)
        .orElseThrow(() -> new UserNotFoundException("User not
found"));

    Post post = new Post();
    post.setPostId(UUID.randomUUID().toString());
    post.setAuthor(author);
    post.setContent(request.getContent());
    post.setType(request.getType());
    post.setVisibility(request.getVisibility());

    // Handle reply
    if (request.getParentPostId() != null) {
        Post parentPost =
postRepository.findById(request.getParentPostId())
            .orElseThrow(() -> new PostNotFoundException("Parent post
not found"));
        post.setParentPost(parentPost);
        parentPost.setReplyCount(parentPost.getReplyCount() + 1);
        postRepository.save(parentPost);
    }

    // Handle retweet
    if (request.getOriginalPostId() != null) {
        Post originalPost =
postRepository.findById(request.getOriginalPostId())
            .orElseThrow(() -> new PostNotFoundException("Original
post not found"));
        post.setOriginalPost(originalPost);
        originalPost.setRetweetCount(originalPost.getRetweetCount() +
1);
        postRepository.save(originalPost);
    }

    Post savedPost = postRepository.save(post);
```

```
    // Update author's post count
    author.setPostCount(author.getPostCount() + 1);
    userRepository.save(author);

    // Publish event for timeline updates
    applicationEventPublisher.publishEvent(new
PostCreatedEvent(savedPost));

    return savedPost;
}

public void likePost(String userId, String postId) {
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new UserNotFoundException("User not
found"));

    Post post = postRepository.findById(postId)
        .orElseThrow(() -> new PostNotFoundException("Post not
found"));

    if (!post.canBeViewedBy(user)) {
        throw new UnauthorizedAccessException("Cannot view this
post");
    }

    // Check if already liked
    if (likeRepository.existsByUser_UserIdAndPost_PostId(userId,
postId)) {
        throw new PostAlreadyLikedException("Post already liked");
    }

    Like like = new Like();
    like.setUser(user);
    like.setPost(post);
    likeRepository.save(like);

    // Update like count
    post.setLikeCount(post.getLikeCount() + 1);
    postRepository.save(post);

    // Send notification to post author
    if (!post.getAuthor().getUserId().equals(userId)) {
        notificationService.sendLikeNotification(post.getAuthor(),
user, post);
    }
```

```
    log.info("User {} liked post {}", userId, postId);
}

public void unlikePost(String userId, String postId) {
    likeRepository.deleteByUserIdAndPostId(userId, postId);

    Post post = postRepository.findById(postId)
        .orElseThrow(() -> new PostNotFoundException("Post not
found"));

    post.setLikeCount(Math.max(0, post.getLikeCount() - 1));
    postRepository.save(post);

    log.info("User {} unliked post {}", userId, postId);
}

public void followUser(String followerId, String followeeId) {
    if (followerId.equals(followeeId)) {
        throw new InvalidFollowException("Cannot follow yourself");
    }

    User follower = userRepository.findById(followerId)
        .orElseThrow(() -> new UserNotFoundException("Follower not
found"));

    User followee = userRepository.findById(followeeId)
        .orElseThrow(() -> new UserNotFoundException("User to follow
not found"));

    if (follower.getFollowing().contains(followeeId)) {
        throw new AlreadyFollowingException("Already following this
user");
    }

    follower.follow(followee);
    userRepository.save(follower);
    userRepository.save(followee);

    // Send notification
    notificationService.sendFollowNotification(followee, follower);

    log.info("User {} started following {}", followerId, followeeId);
}
```

```
public void unfollowUser(String followerId, String followeeId) {
    User follower = userRepository.findById(followerId)
        .orElseThrow(() -> new UserNotFoundException("Follower not
found"));

    User followee = userRepository.findById(followeeId)
        .orElseThrow(() -> new UserNotFoundException("User to
unfollow not found"));

    follower.unfollow(followee);
    userRepository.save(follower);
    userRepository.save(followee);

    log.info("User {} unfollowed {}", followerId, followeeId);
}

public List<Post> getUserTimeline(String userId, Pageable pageable) {
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new UserNotFoundException("User not
found"));

    return timelineService.generateHomeTimeline(user, pageable);
}

public List<User> searchUsers(String query) {
    return
userRepository.findByUsernameContainingIgnoreCaseOrDisplayNameContain
ingIgnoreCase(
        query, query);
}

public List<Post> searchPosts(String query) {
    if (query.startsWith("#")) {
        String hashtag = query.substring(1).toLowerCase();
        return postRepository.findByHashtag(hashtag);
    }

    return postRepository.searchByContent(query);
}


}
```

@Service @Slf4j public class TimelineService {

```
@Autowired
private PostRepository postRepository;

@Autowired
private RedisTemplate<String, Object> redisTemplate;

private static final String TIMELINE_KEY = "timeline:user:";
private static final int TIMELINE_SIZE = 100;

public List<Post> generateHomeTimeline(User user, Pageable pageable)
{
    String cacheKey = TIMELINE_KEY + user.getUserId();

    // Try to get from cache first
    List<Post> cachedTimeline = (List<Post>)
redisTemplate.opsForValue().get(cacheKey);
    if (cachedTimeline != null && !cachedTimeline.isEmpty()) {
        return cachedTimeline.stream()
            .skip(pageable.getOffset())
            .limit(pageable.getPageSize())
            .collect(Collectors.toList());
    }

    // Generate timeline from database
    Set<String> followedUsers = user.getFollowing();
    followedUsers.add(user.getUserId()); // Include own posts

    Page<Post> timelinePosts = postRepository.findTimelinePosts(
        followedUsers,
        PageRequest.of(0, TIMELINE_SIZE, Sort.by(Sort.Direction.DESC,
"createdAt"))
    );

    List<Post> posts = timelinePosts.getContent();

    // Cache the timeline
    redisTemplate.opsForValue().set(cacheKey, posts,
Duration.ofMinutes(30));

    return posts.stream()
        .skip(pageable.getOffset())
        .limit(pageable.getPageSize())
        .collect(Collectors.toList());
}
```

```
@Async
public void updateTimelinesForPost(Post post) {
    // Update timelines for all followers
    for (String followerId : post.getAuthor().getFollowers()) {
        String timelineKey = TIMELINE_KEY + followerId;
        redisTemplate.delete(timelineKey); // Invalidate cache
    }

    // Also invalidate author's timeline
    String authorTimelineKey = TIMELINE_KEY +
post.getAuthor().getUserId();
    redisTemplate.delete(authorTimelineKey);

    log.info("Invalidated timelines for post: {}", post.getPostId());
}


}
```

// Controller Layer @RestController @RequestMapping("/api/v1/social") @Validated @Slf4j public class SocialMediaController {

```
@Autowired
private SocialMediaService socialMediaService;

@PostMapping("/users")
public ResponseEntity<ApiResponse<User>> createUser(
        @Valid @RequestBody CreateUserRequest request) {

    try {
        User user = socialMediaService.createUser(request);
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(ApiResponse.success(user, "User created
successfully"));
    } catch (UsernameAlreadyExistsException |
EmailAlreadyExistsException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT)
            .body(ApiResponse.error("USER_EXISTS", e.getMessage()));
    }
}

@PostMapping("/users/{userId}/posts")
public ResponseEntity<ApiResponse<Post>> createPost(
        @PathVariable String userId,
        @Valid @RequestBody CreatePostRequest request) {
```

```
    Post post = socialMediaService.createPost(userId, request);
    return ResponseEntity.status(HttpStatus.CREATED)
        .body(ApiResponse.success(post, "Post created
successfully"));
}

@PostMapping("/users/{userId}/posts/{postId}/like")
public ResponseEntity<ApiResponse<Void>> likePost(
        @PathVariable String userId,
        @PathVariable String postId) {

    try {
        socialMediaService.likePost(userId, postId);
        return ResponseEntity.ok(ApiResponse.success(null, "Post
liked successfully"));
    } catch (PostAlreadyLikedException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT)
            .body(ApiResponse.error("ALREADY_LIKED",
e.getMessage()));
    }
}

@DeleteMapping("/users/{userId}/posts/{postId}/like")
public ResponseEntity<ApiResponse<Void>> unlikePost(
        @PathVariable String userId,
        @PathVariable String postId) {

    socialMediaService.unlikePost(userId, postId);
    return ResponseEntity.ok(ApiResponse.success(null, "Post unliked
successfully"));
}

@PostMapping("/users/{followerId}/follow/{followeeId}")
public ResponseEntity<ApiResponse<Void>> followUser(
        @PathVariable String followerId,
        @PathVariable String followeeId) {

    try {
        socialMediaService.followUser(followerId, followeeId);
        return ResponseEntity.ok(ApiResponse.success(null, "User
followed successfully"));
    } catch (AlreadyFollowingException e) {
        return ResponseEntity.status(HttpStatus.CONFLICT)
            .body(ApiResponse.error("ALREADY_FOLLOWING",
e.getMessage()));
```

```
    }
}

@DeleteMapping("/users/{followerId}/follow/{followeeId}")
public ResponseEntity<ApiResponse<Void>> unfollowUser(
        @PathVariable String followerId,
        @PathVariable String followeeId) {

    socialMediaService.unfollowUser(followerId, followeeId);
    return ResponseEntity.ok(ApiResponse.success(null, "User
unfollowed successfully"));
}

@GetMapping("/users/{userId}/timeline")
public ResponseEntity<ApiResponse<List<Post>>> getUserTimeline(
        @PathVariable String userId,
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "20") int size) {

    Pageable pageable = PageRequest.of(page, size);
    List<Post> timeline = socialMediaService.getUserTimeline(userId,
pageable);

    return ResponseEntity.ok(ApiResponse.success(timeline, "Timeline
retrieved successfully"));
}

@GetMapping("/search/users")
public ResponseEntity<ApiResponse<List<User>>> searchUsers(
        @RequestParam String query) {

    List<User> users = socialMediaService.searchUsers(query);
    return ResponseEntity.ok(ApiResponse.success(users, "Users
found"));
}

@GetMapping("/search/posts")
public ResponseEntity<ApiResponse<List<Post>>> searchPosts(
        @RequestParam String query) {

    List<Post> posts = socialMediaService.searchPosts(query);
    return ResponseEntity.ok(ApiResponse.success(posts, "Posts
found"));
}
```

```
}

// Request/Response DTOs @Data @Builder @NoArgsConstructor @AllArgsConstructor
public class CreateUserRequest { @NotBlank(message = "Username is required")
@Size(min = 3, max = 20, message = "Username must be between 3 and 20 characters")
@Pattern(regexp = "^[a-zA-Z0-9_]+$", message = "Username can only contain letters,
numbers, and underscores") private String username;

@NotBlank(message = "Email is required")
@Email(message = "Invalid email format")
private String email;

@NotBlank(message = "Display name is required")
@Size(min = 1, max = 50, message = "Display name must be between 1
and 50 characters")
private String displayName;

@Size(max = 500, message = "Bio must be less than 500 characters")
private String bio;


}

@Data @Builder @NoArgsConstructor @AllArgsConstructor public class
CreatePostRequest { @NotBlank(message = "Content is required") @Size(max = 2000,
message = "Post content must be less than 2000 characters") private String content;

@NotNull(message = "Post type is required")
private PostType type = PostType.ORIGINAL;

private PostVisibility visibility = PostVisibility.PUBLIC;

private String parentPostId; // For replies
private String originalPostId; // For retweets

private List<String> mediaUrls;
private Location location;


}

// Events @Data @AllArgsConstructor public class PostCreatedEvent { private Post
post; }
```

// Base Entity @MappedSuperclass @Data public abstract class BaseEntity { @CreationTimestamp @Column(updatable = false) private LocalDateTime createdAt;

```
@UpdateTimestamp
private LocalDateTime updatedAt;

@Version
private Long version;


}
```

// Utility Classes @Data @Builder @NoArgsConstructor @AllArgsConstructor @JsonInclude(JsonInclude.Include.NON_NULL) public class ApiResponse { private boolean success; private String message; private T data; private String errorCode; private Map<String, Object> metadata;

```
public static <T> ApiResponse<T> success(T data, String message) {
    return ApiResponse.<T>builder()
        .success(true)
        .message(message)
        .data(data)
        .build();
}

public static <T> ApiResponse<T> error(String errorCode, String
message) {
    return ApiResponse.<T>builder()
        .success(false)
        .errorCode(errorCode)
        .message(message)
        .build();
}

public static <T> ApiResponse<T> error(String errorCode, String
message, Map<String, Object> metadata) {
    return ApiResponse.<T>builder()
        .success(false)
        .errorCode(errorCode)
        .message(message)
        .metadata(metadata)
        .build();
}
```

```
}
```