21. Meeting Scheduler

Step 1: Outline Use Cases and Constraints

Use Cases

The system should support the following functionalities:

- Users should be able to schedule and cancel meetings.
- Users should be able to view scheduled meetings.
- Multiple users can be invited to meetings.
- Users should receive notifications when a meeting is scheduled or canceled.
- Each user should have a separate calendar.
- The system should check for conflicts before scheduling a meeting.

Out of Scope

- Integration with external calendar services (e.g., Google Calendar, Outlook).
- Recurring meetings.
- Advanced meeting analytics and reporting.

Constraints and Assumptions

- The system will support up to 10,000 concurrent users.
- The meeting room availability will be managed centrally.
- Users should be able to book a meeting room in advance.
- Notifications will be sent via email or push notifications.
- Meeting duration will be in predefined slots.

Step 2: Create a High-Level Design

Key Components:

1. User Service: Manages user-related operations.
2. Meeting Scheduler: Handles meeting creation, scheduling, and cancellations.
3. Room Management Service: Ensures room availability and handles room bookings.
4. Notification Service: Sends notifications to users about meetings.
5. Meeting DAO: Responsible for persisting meeting data.
6. Calendar Management: Tracks user schedules and prevents conflicts.

Data Flow:

1. User creates a meeting request.
2. System checks availability of the room and participants.
3. Meeting is scheduled and saved in the database.
4. Notifications are sent to all attendees.
5. Users can view, update, or cancel meetings.

Step 3: Design Core Components (According to the Use Cases)

Classes and Their Responsibilities:

1. **User**: Stores user details and meeting calendar.
2. **Meeting**: Stores meeting title, date, duration, attendees, and host.
3. **Meeting Room**: Stores room name, capacity, and scheduled meetings.
4. **Slot**: Represents a time slot for a meeting.
5. **MeetingDAO**: Handles data storage and retrieval for meetings.
6. **UserService**: Manages user calendars and prevents scheduling conflicts.
7. **RoomManagementService**: Handles room booking and availability.
8. **NotificationService**: Sends notifications to users.
9. **MeetingScheduler**: Orchestrates the scheduling process.

1. User Service - Manages user-related operations

```
public class UserService {
    private Map<String, User> users = new HashMap<>();

    public User createUser(String username, String email) {
        User user = new User(username, email);
        users.put(username, user);
        return user;
    }

    public User getUser(String username) {
        return users.get(username);
    }
}
```

2. User Class - Represents a user in the system

```
public class User {
    private String username;
    private String email;
    private Calendar calendar;

    public User(String username, String email) {
```

```
        this.username = username;
        this.email = email;
        this.calendar = new Calendar();
    }

    public String getUsername() {
        return username;
    }

    public String getEmail() {
        return email;
    }

    public Calendar getCalendar() {
        return calendar;
    }
}
```

3. Calendar Class - Manages user's schedule and prevents conflicts

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

public class Calendar {
    private List<Meeting> meetings;

    public Calendar() {
        this.meetings = new ArrayList<>();
    }

    public boolean hasConflict(Meeting newMeeting) {
        for (Meeting meeting : meetings) {
            if (meeting.getStartTime().equals(newMeeting.getStartTime()) ||
meeting.getEndTime().equals(newMeeting.getEndTime())) {
                return true;  // Conflict if the time overlaps
            }
        }
        return false;
    }

    public void addMeeting(Meeting meeting) {
        meetings.add(meeting);
    }

    public void removeMeeting(Meeting meeting) {
        meetings.remove(meeting);
    }
```

```
    public List<Meeting> getMeetings() {
        return meetings;
    }
}
```

4. Meeting Class - Represents a meeting

```
import java.time.LocalDateTime;

public class Meeting {
    private String title;
    private LocalDateTime startTime;
    private LocalDateTime endTime;
    private List<User> participants;

    public Meeting(String title, LocalDateTime startTime, LocalDateTime endTime) {
        this.title = title;
        this.startTime = startTime;
        this.endTime = endTime;
        this.participants = new ArrayList<>();
    }

    public String getTitle() {
        return title;
    }

    public LocalDateTime getStartTime() {
        return startTime;
    }

    public LocalDateTime getEndTime() {
        return endTime;
    }

    public List<User> getParticipants() {
        return participants;
    }

    public void addParticipant(User user) {
        participants.add(user);
    }
}
```

5. Meeting Scheduler Service - Handles meeting creation, scheduling, and cancellation

```
public class MeetingSchedulerService {
    private UserService userService;

    public MeetingSchedulerService(UserService userService) {
        this.userService = userService;
    }

    public boolean scheduleMeeting(String title, LocalDateTime startTime, LocalDateTime endTime,
List<String> usernames) {
        List<User> participants = new ArrayList<>();
        for (String username : usernames) {
            User user = userService.getUser(username);
            if (user != null && !user.getCalendar().hasConflict(new Meeting(title, startTime, endTime))) {
                participants.add(user);
            } else {
                System.out.println("Conflict for user: " + username);
                return false;
            }
        }

        // Schedule meeting for each participant
        Meeting meeting = new Meeting(title, startTime, endTime);
        for (User user : participants) {
            user.getCalendar().addMeeting(meeting);
            meeting.addParticipant(user);
        }

        sendNotification(meeting);
        return true;
    }

    public boolean cancelMeeting(Meeting meeting) {
        for (User user : meeting.getParticipants()) {
            user.getCalendar().removeMeeting(meeting);
        }
        sendCancellationNotification(meeting);
        return true;
    }

    private void sendNotification(Meeting meeting) {
        // Placeholder method to send notifications
        System.out.println("Notification: Meeting '" + meeting.getTitle() + "' has been scheduled.");
    }
```

```
    private void sendCancellationNotification(Meeting meeting) {
        // Placeholder method to send cancellation notifications
        System.out.println("Notification: Meeting '" + meeting.getTitle() + "' has been canceled.");
    }
}
```

6. Main Class - To test the system

```
import java.time.LocalDateTime;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        UserService userService = new UserService();
        User user1 = userService.createUser("Alice", "alice@example.com");
        User user2 = userService.createUser("Bob", "bob@example.com");

        MeetingSchedulerService scheduler = new MeetingSchedulerService(userService);

        LocalDateTime startTime = LocalDateTime.of(2025, 2, 5, 10, 0);
        LocalDateTime endTime = LocalDateTime.of(2025, 2, 5, 11, 0);

        boolean meetingScheduled = scheduler.scheduleMeeting("Project Sync", startTime, endTime,
Arrays.asList("Alice", "Bob"));
        if (meetingScheduled) {
            System.out.println("Meeting scheduled successfully.");
        } else {
            System.out.println("Failed to schedule meeting due to conflicts.");
        }

        // Attempt to cancel the meeting
        scheduler.cancelMeeting(new Meeting("Project Sync", startTime, endTime));
    }
}
```

Explanation:

- **UserService**: Manages user creation and retrieval.
- **User**: Stores information about a user and their calendar.
- **Calendar**: Manages meetings and prevents conflicts.
- **Meeting**: Represents a meeting with a title, time, and participants.
- **MeetingSchedulerService**: Handles scheduling and canceling meetings, checking for conflicts, and sending notifications.
- **Main**: A simple test to create users, schedule, and cancel meetings.

Step 4: Scale the Design

Scalability Considerations:

- **Database Sharding**: Store user calendars and meeting data in a distributed manner.
- **Caching**: Use Redis to store frequently accessed meeting schedules.
- **Load Balancing**: Distribute traffic across multiple servers.
- **Event-Driven Notifications**: Use a message queue like Kafka for sending notifications asynchronously.
- **Microservices Architecture**: Split services into independent units (User Service, Meeting Service, Notification Service, etc.).

By following this structured approach, the meeting scheduler ensures smooth scheduling, prevents conflicts, and provides a scalable solution for managing meetings effectively.