MILESTONE II

1. User Table

Purpose: Stores information about the users in the system, including credentials and optional bio.

- **Columns**:
  - `user_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `username`: VARCHAR(30), NOT NULL, UNIQUE
  - `email`: VARCHAR(30), NOT NULL, UNIQUE
  - `password`: VARCHAR(20), NOT NULL
  - `bio`: VARCHAR(255)

---

2. Group Table

Purpose: Manages groups created by users and holds group-related metadata.

- **Columns**:
  - `group_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `group_name`: VARCHAR(30), NOT NULL
  - `group_description`: TEXT, NULLABLE
  - `created_by`: INT, FOREIGN KEY REFERENCES `User(user_id)` (ON DELETE SET NULL)

---

3. Membership Table

Purpose: Tracks which users belong to which groups and their roles within the groups.

- **Columns**:
  - `membership_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `user_id`: INT, FOREIGN KEY REFERENCES `User(user_id)` (ON DELETE CASCADE)
  - `group_id`: INT, FOREIGN KEY REFERENCES `Group(group_id)` (ON DELETE CASCADE)
  - `user_role`: ENUM('Member', 'Admin', 'Guest'), NOT NULL

    Constraints:

  - UNIQUE(user_id, group_id) – A user can only join a group once.

---

4. Event Table

Purpose: Stores information about events organized within groups.

- **Columns**:
  - `event_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `group_id`: INT, FOREIGN KEY REFERENCES `Group(group_id)` (ON DELETE CASCADE)
  - `event_name`: VARCHAR(255), NOT NULL
  - `event_description`: TEXT, NULLABLE
  - `event_date`: DATE, NOT NULL
  - `event_location`: VARCHAR(255)

---

5. Event_Attendance Table

Purpose: Tracks user attendance or interest in events.

- **Columns**:
  - `attendance_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `user_id`: INT, FOREIGN KEY REFERENCES `User(user_id)` (ON DELETE CASCADE)
  - `event_id`: INT, FOREIGN KEY REFERENCES `Event(event_id)` (ON DELETE CASCADE)
  - `event_status`: ENUM('Attended', 'Interested', 'Not Attended'), NOT NULL – The user's attendance status.
- **Constraints**:
  - UNIQUE(user_id, event_id) – A user can respond to an event only once.

---

6. Feedback Table

Purpose: Allows users to provide feedback for events they attended.

- **Columns**:
  - `feedback_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `user_id`: INT, FOREIGN KEY REFERENCES `User(user_id)` (ON DELETE CASCADE)
  - `event_id`: INT, FOREIGN KEY REFERENCES `Event(event_id)` (ON DELETE CASCADE)
  - `rating`: INT, CHECK(rating BETWEEN 1 AND 5)
  - `feedback`: TEXT, NULLABLE – Detailed feedback from the user.

---

7. Message_Board Table

Purpose: Stores messages shared by users within group message boards.

- **Columns**:
  - `message_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `group_id`: INT, FOREIGN KEY REFERENCES `Group(group_id)` (ON DELETE CASCADE)
  - `user_id`: INT, FOREIGN KEY REFERENCES `User(user_id)` (ON DELETE SET NULL)
  - `user_message`: TEXT, NOT NULL
  - `message_time`: DATETIME, DEFAULT CURRENT_TIMESTAMP

---

8. Tag Table

Purpose: Stores tags used for categorizing events.

- **Columns**:
  - `tag_id`: INT, PRIMARY KEY, AUTO_INCREMENT
  - `tag_name`: VARCHAR(32), NOT NULL, UNIQUE

---

9. Event_Tag Relationship Table

Purpose: Establishes a many-to-many relationship between events and tags.

- **Columns**:
  - `event_id`: INT, FOREIGN KEY REFERENCES `Event(event_id)` (ON DELETE CASCADE)
  - `tag_id`: INT, FOREIGN KEY REFERENCES `Tag(tag_id)` (ON DELETE CASCADE)
- **Constraints**:
  - PRIMARY KEY(event_id, tag_id) – Ensures each event-tag pair is unique.

Complex Query Examples

1. Find All Events for Groups Where User 5 is Admin

```
SELECT e.event_name, e.event_date, e.event_location, g.group_name
FROM Event e
JOIN `Group` g ON e.group_id = g.group_id
JOIN Membership m ON g.group_id = m.group_id
WHERE m.user_id = 5 AND m.user_role = 'Admin';
```

2.Find the Most 5 Active Users in Terms of Group Memberships

```
SELECT u.username, COUNT(m.membership_id) AS group_count
FROM `User` u
JOIN Membership m ON u.user_id = m.user_id
GROUP BY u.user_id, u.username
ORDER BY group_count DESC
LIMIT 5;
```

3.Find the Average Feedback Rating for Each Group in Descending Order

```
SELECT g.group_name, AVG(f.rating) AS avg_rating
FROM `Group` g
JOIN Event e ON g.group_id = e.group_id
JOIN Feedback f ON e.event_id = f.event_id
GROUP BY g.group_id, g.group_name
ORDER BY avg_rating DESC;
```

4.Identify Users Who Are Not Members of Any Group

```
SELECT u.username
FROM `User` u
LEFT JOIN Membership m ON u.user_id = m.user_id
WHERE m.membership_id IS NULL;
```