Project Documentation: Event Management System

Emre Demirci
150220317
Faculty of Computer and Informatics
Department of AI and Data Eng.

Abdülkadir Külçe
150210322
Faculty of Computer and Informatics
Department of AI and Data Eng.

Zeynep Serpil Vızvız
820210323
Faculty of Computer and Informatics
Department of Computer Eng.

Ahmet Enes Topçu
150210310
Faculty of Computer and Informatics
Department of AI and Data Eng.

Contents

1 Overview of the Project Idea

The Event Management System is a RESTful API-based platform designed to manage events, users, and groups efficiently. It allows users to register, create groups, manage events, post messages, and interact with the system through well-defined API endpoints. The system ensures secure and organized interaction with a MySQL database using Flask.

Key features include:

- User registration and authentication.
- Group creation and membership management.
- Event creation, attendance tracking, and feedback management.
- Message board for group communication.
- Tagging events and querying them by tags.

2 ER Diagram and Data Model Explanations

2.1 ER Diagram

- Check ER-Diagram.pdf file for details.

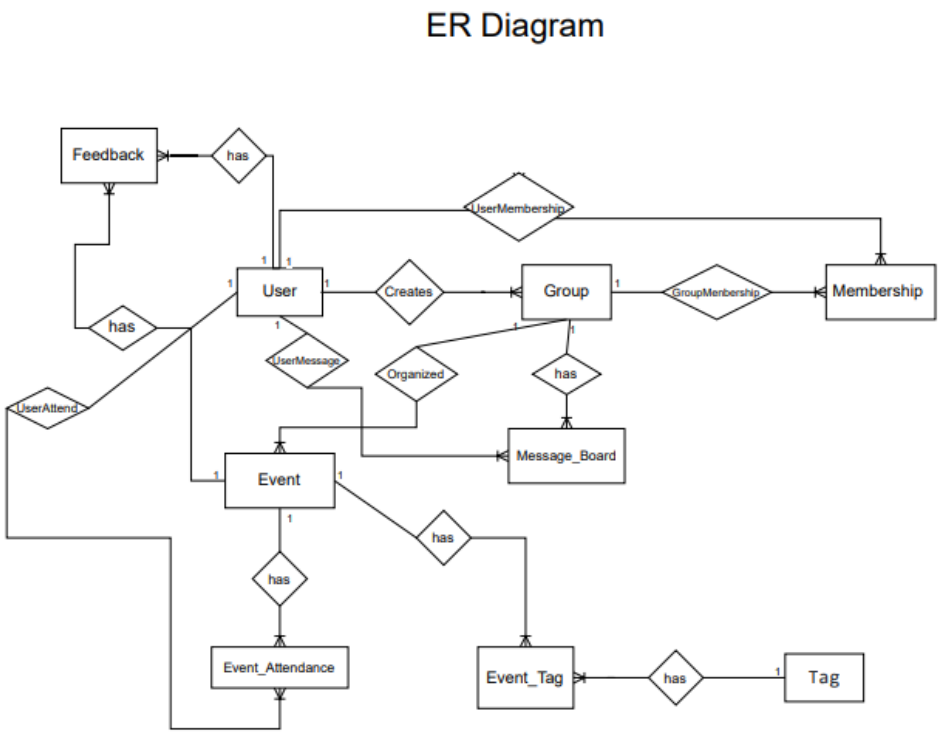

Figure 1: ER Diagram of the Event Management System

2.2 Data Model Explanations

- Check Data-Model.pdf for details.

- **User Table**: Contains user information including username, email, password (hashed), and bio.
- **Group Table**: Defines groups with a name, description, and creator.
- **Membership Table**: Manages user roles (Admin, Member, Guest) in groups.
- **Event Table**: Stores events associated with groups, including date, location, and description.
- **Event Attendance Table**: Tracks user attendance statuses for events.
- **Feedback Table**: Stores user feedback for events with ratings and comments.
- **Message Board Table**: Stores messages posted by users in groups.
- **Tag and Event Tag Tables**: Handles tagging of events for better searchability.

3 CRUD Operations and Implementations

3.1 Create Operations

- **Create User:** `POST /users` - Registers a new user. Ensures that the username and email are unique. Passwords are hashed before storage.
- **Create Group:** `POST /groups` - Creates a new group and assigns the 'Admin' role to the authenticated user. Validates that the group name is unique within the system.
- **Create Event:** `POST /events` - Creates a new event. Checks that the authenticated user is a member of the group. Validates the event date and ensures the group exists.
- **Add Attendance:** `POST /events/{event_id}/attendance` - Adds attendance to an event. Verifies that the user is a member of the group associated with the event. Ensures duplicate attendance entries are not created.
- **Create Message:** `POST /messages` - Posts a new message to the group. Checks that the user is a member of the group before posting.
- **Create Tag:** `POST /tags` - Creates a new tag. Ensures that the tag name is unique and avoids duplication.
- **Add Tags to Event:** `POST /events/tags` - Adds multiple tags to an event. Checks the validity of the event and tag IDs. Ensures the user is a member of the group associated with the event.
- **Add User to Group:** `POST /membership/` - Adds the authenticated user to a group as a member. Verifies that the group exists and ensures the user is not already a member.

- **Add Feedback:** `POST /feedback` - Adds new feedback for an event. Checks that the user attended the event and validates the rating range (1-5).

3.2 Read Operations

- **Get Users:** `GET /users` - Retrieves users with advanced search options. Allows filtering by username oremail and supports pagination.
- **Get Single User:** `GET /users/{username}` - Retrieves a single user by username.
- **Get Groups:** `GET /groups` - Retrieves all groups. Supports optional search and pagination to limit the results.
- **Get Single Group:** `GET /groups/{group_id}` - Retrieves a group by its ID.
- **Retrieve Events for User Groups:** `GET /events/user/groups` - Retrieves events for groups the authenticated user belongs to. Filters past and upcoming events.
- **Retrieve Events by Tag:** `GET /events/tags/events/{tag_name}` - Retrieves events associated with a specific tag.
- **Retrieve Popular Tags:** `GET /events/tags/popular` - Retrieves the most used tags in the system. Tags are ranked by usage frequency.
- **List Messages:** `GET /messages` - Lists all messages ordered from newest to oldest. Verifies that the user is a member of the group before returning messages.
- **Filter Messages:** `GET /messages/filter` - Filters messages by date range with pagination. Checks group membership before returning results.
- **List Attendance for Event:** `GET /events/{event_id}/attendance` - Lists users attending or interested in an event. Ensures the event exists and validates group membership.
- **List User Attendance:** `GET /users/{user_id}/attendance` - Lists events a user has attended or shown interest in. Filters results based on attendance status.
- **Get Feedback for Event:** `GET /feedback/events/{event_id}` - Retrieves all feedback for a specific event.
- **Get Feedback by User:** `GET /feedback/users/{user_id}` - Retrieves feedback given by a specific user.

3.3 Update Operations

- **Update User:** `PUT /users` - Updates the authenticated user's account. Ensures that the new email is unique if provided. Updating username is not allowed.
- **Update Group:** `PUT /groups/{group_id}` - Updates an existing group by ID. Checks that the authenticated user is an admin of the group.

- **Update Event:** `PUT /events/{event_id}` - Updates an existing event. Ensures the user is a group admin and validates the new event details.
- **Update Attendance:** `PUT /events/{event_id}/attendance` - Updates the attendance status for an event. Validates the new status and ensures group membership.
- **Update Message:** `PUT /messages/{message_id}` - Updates a message by its ID. Checks that the user is the message creator.
- **Update Tag:** `PUT /tags/{tag_id}` - Updates a tag by its ID. Ensures that the new tag name is unique.
- **Update Feedback:** `PUT /feedback/{feedback_id}` - Updates feedback for an event. Ensures the user is the feedback creator and validates the new rating.

3.4 Delete Operations

- **Delete User:** `DELETE /users` - Deletes the authenticated user's account.
- **Delete Group:** `DELETE /groups/{group_id}` - Deletes a group by ID. Verifies that the authenticated user is the group admin.
- **Delete Event:** `DELETE /events/{event_id}` - Deletes an event by ID. Ensures the user is an admin of the group associated with the event.
- **Delete Attendance:** `DELETE /events/{event_id}/attendance` - Deletes attendance for an event. Ensures the user has an existing attendance record.
- **Delete Message:** `DELETE /messages/{message_id}` - Deletes a message by its ID. Validates that the user is the creator or a group admin.
- **Delete Tag:** `DELETE /tags/{tag_id}` - Deletes a tag by its ID.
- **Delete Feedback:** `DELETE /feedback/{feedback_id}` - Deletes feedback for an event. Ensures the user is the feedback creator or the event creator.

4 API Design and Endpoint Explanations

The API is architected to prioritize scalability, clarity, and RESTful principles, ensuring reliable performance as the system grows. Key features include:

Authentication and Security

JWT-based authentication ensures that all sensitive operations are securely accessed. Users must include the issued token in the `Authorization` header to access protected data.

Error Handling and Feedback

The API is designed to provide clear and actionable error messages. For instance:

- `400 Bad Request`: Indicates invalid inputs or missing required fields.
- `403 Forbidden`: Indicates insufficient permissions for the operation.
- `500 Internal Server Error`: Highlights unexpected server issues and includes the explanation if the error is expected by the server (e.g duplicate usernames are restricted).

Detailed Documentation

For a full list of endpoints, request parameters, and response formats, refer to:

- The `Swagger-Documentation.pdf` and `Swagger-with-Parameters.pdf` files included with this documentation.
- The Swagger UI available within the application for interactive exploration and testing.

5 Examples of Complex Queries with Explanations

5.1 Top Users by Event Attendance

SQL Query:

```
SELECT ea.user_id, u.username, COUNT(*) AS total_attendance
FROM Event_Attendance ea
INNER JOIN User u ON ea.user_id = u.user_id
GROUP BY ea.user_id
ORDER BY total_attendance DESC
LIMIT 10
```

Explanation: This query identifies the top 10 users who have attended the most events. It achieves this by joining the `User` and `Event_Attendance` tables, grouping results by user, and counting attendance records. The query then sorts users in descending order of attendance count, highlighting the most active participants.

5.2 List All Admins

SQL Query:

```
SELECT g.group_name, u.username, m.user_role
FROM Membership m
JOIN ‘User‘ u ON m.user_id = u.user_id
JOIN ‘Group‘ g ON m.group_id = g.group_id
WHERE m.user_role = ’Admin’;
```

Explanation: This query retrieves all users with the role of "Admin" across different groups. It joins the `Membership`, `User`, and `Group` tables to gather the group name, username, and user role.

5.3 Get Popular Tags

SQL Query:

```
SELECT t.tag_name, COUNT(et.tag_id) AS usage_count
FROM Tag t
INNER JOIN Event_Tag et ON t.tag_id = et.tag_id
GROUP BY t.tag_id
ORDER BY usage_count DESC
LIMIT 10;
```

Explanation: This query identifies the 10 most popular tags based on their usage in events. It joins the `Tag` and `Event_Tag` tables to count how many times each tag is associated with an event. The results are grouped by tag and sorted in descending order of usage count, highlighting the most frequently used tags in the system.

6 Challenges and Solutions Encountered

6.1 Deployment Issues

To streamline the deployment process, we developed automated scripts to handle database creation, including tables and triggers. Additionally, we implemented scripts to populate the database with dummy data using the API endpoints, ensuring that the system's functionality could be tested during deployment.

6.2 Maintenance and Scalability

To ensure smooth maintenance and scalability, we structured the project to support collaborative development. The codebase was modularized, allowing multiple team members to work on different components independently without conflicts. Clear boundaries between modules, such as routes, database interactions, and utility scripts, ensured that changes in one part of the system did not affect others. This approach improved productivity and streamlined the integration of new features.

7 Summary

This project implements a comprehensive event management system with secure authentication, modular design, and efficient database interactions. The development process involved thoughtful API design, robust CRUD operations, and meaningful error handling.