

College Event Website

COP 4710, 0001, Spring 2024

Rishi Gadhia, Jeyoos Jaison, Jeremy Rosales

Table of Contents

Project Description.....	
GUI.....	
ER-Model.....	
Relational Database Model.....	
Database.....	

Project Description

In response to the challenges faced by students in tracking and accessing university events, our team has developed a comprehensive web-based application – the College Event Website. This platform aims to streamline the process of discovering, managing, and participating in university events, catering to the diverse needs of students, administrators, and event organizers.

At the core of our application is a robust user authentication system, accommodating three distinct user levels: super admin, admin, and student. Super admins oversee the creation and management of university profiles, providing essential information such as location, description, and the number of students. Admins, affiliated with universities and registered student organizations (RSOs), possess the authority to host and manage events, each defined by unique attributes including name, category, date, time, and location. Meanwhile, students can explore a curated selection of public events, as well as those specific to their university and RSO memberships.

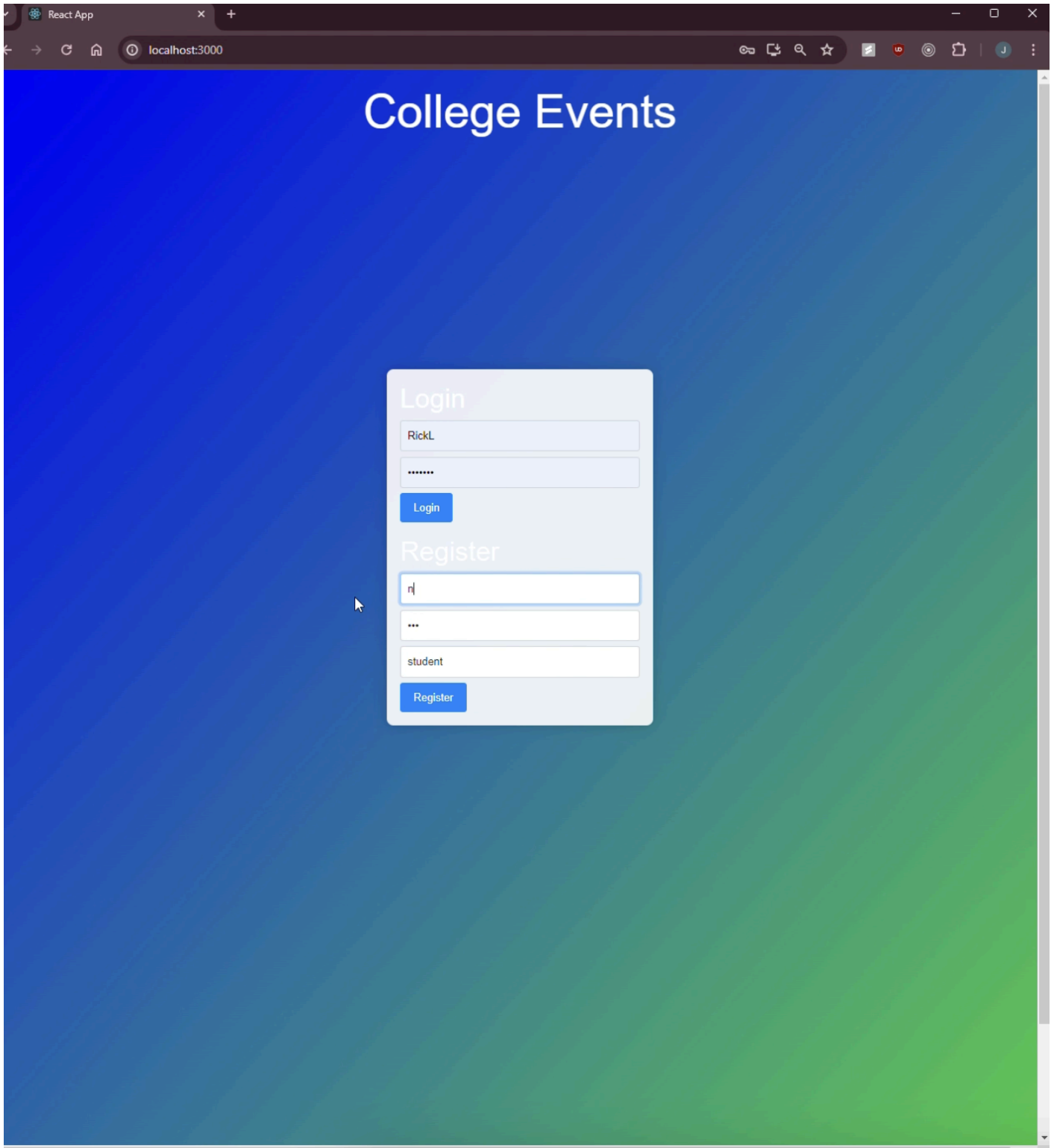
Our application leverages various technologies to enhance user experience and functionality. By integrating map services like Bing, Google, or OpenStreetMap, users can easily visualize event locations, facilitating navigation and attendance planning. Furthermore, social network integration enables seamless sharing and promotion of events across popular platforms like Facebook and Google.

To ensure data integrity and security, our database design adheres to best practices, including normalization, indexing, and constraint enforcement through triggers and assertions. Sample data, comprising users, RSOs, events, comments, and ratings, enriches the user experience and facilitates testing and evaluation.

In addition to meeting technical requirements, our project encompasses advanced features such as event feeds integration from university websites, crash recovery policies, and robust security measures. These enhancements elevate the platform's reliability, scalability, and resilience to meet the demands of multiple concurrent users.

In summary, the College Event Website offers a comprehensive solution to the challenges of event discovery and management within university communities. Through intuitive design, seamless integration, and robust functionality, our application aims to enhance the student experience and foster engagement within campus communities.

GUI



The screenshot shows a web browser window with the title 'React App' and the address bar displaying 'localhost:3000'. The main content area has a blue-to-green gradient background with the text 'College Events' at the top. In the center, there is a white modal form with two sections: 'Login' and 'Register'.

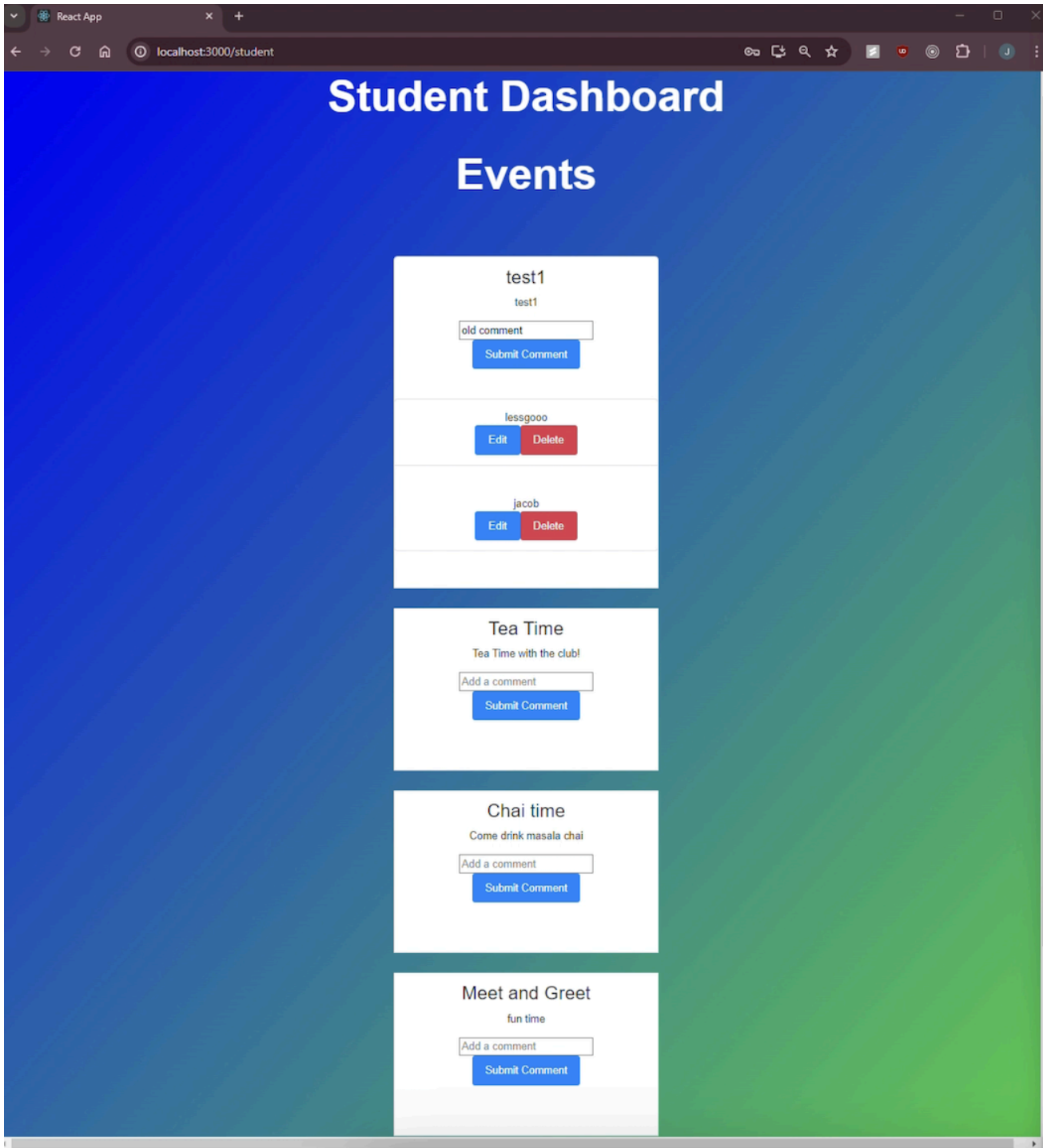
Login Section:

- Username input field: RickL
- Password input field: masked with asterisks
- Login button

Register Section:

- First name input field: n
- Last name input field: masked with asterisks
- Role input field: student
- Register button





Languages/Framework

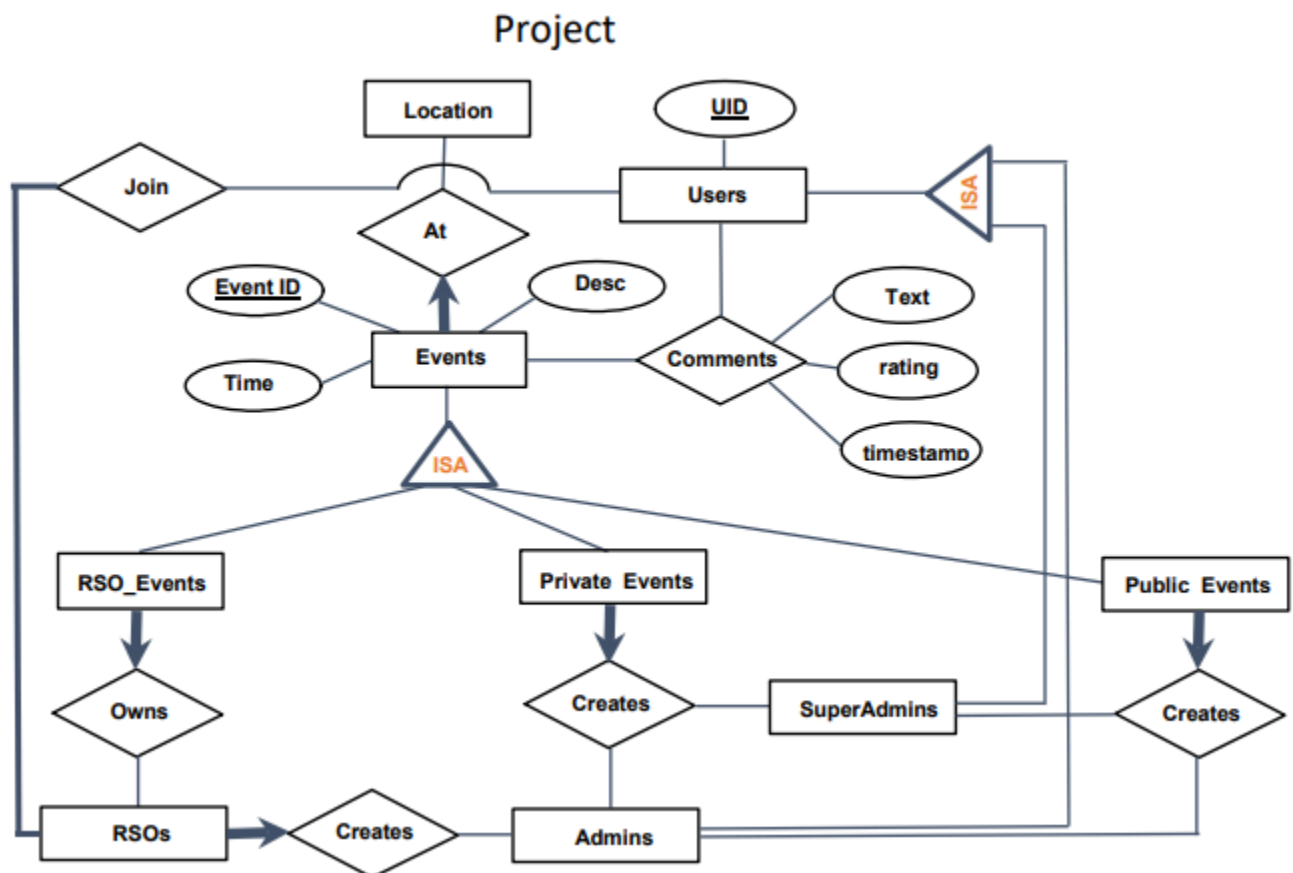
We used JavaScript and React to build the frontend of the application, as well as MySQL for setting up the database.

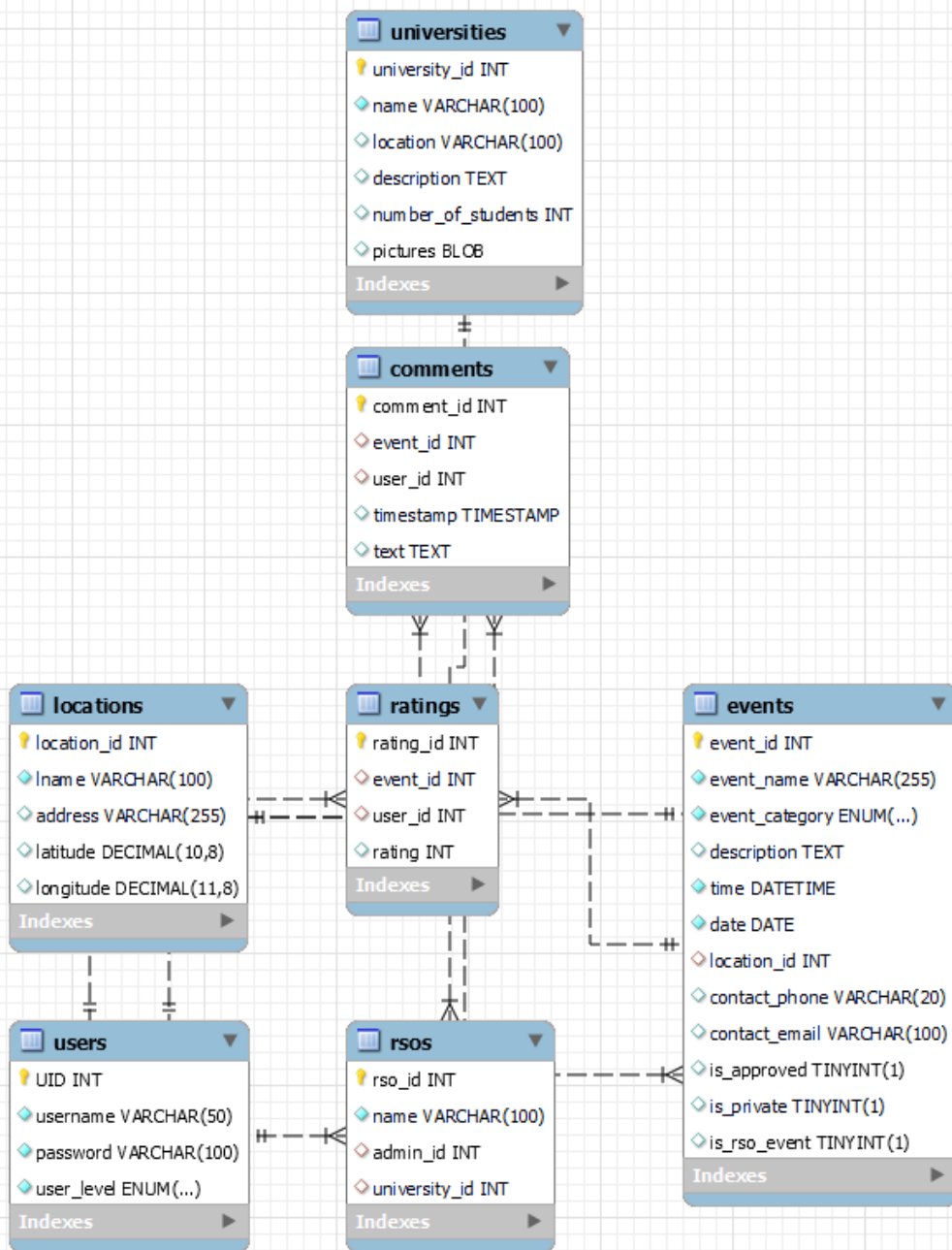
Platform/DBMS

We used a local database and hosting, and used MySQL as our DBMS. We used Workbench to view the database.

ER-Model

ER-Diagram





Create Table:

```
-- Users table to store student user information
CREATE TABLE Users (
    UID INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) NOT NULL,
    password VARCHAR(100) NOT NULL,
    user_level ENUM('student', 'admin', 'super_admin') NOT NULL
);

-- Universities table to store university profile information
CREATE TABLE Universities (
    university_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    location VARCHAR(100),
    description TEXT,
    number_of_students INT,
    pictures BLOB -- Assuming storing pictures as binary data
);

-- RSOs table to store Registered Student Organizations information
CREATE TABLE RSOs (
    rso_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    admin_id INT, -- Foreign key to Users table
    university_id INT, -- Foreign key to Universities table
    FOREIGN KEY (admin_id) REFERENCES Users(UID),
    FOREIGN KEY (university_id) REFERENCES Universities(university_id)
);

-- Locations table to store event locations
CREATE TABLE Locations (
    location_id INT PRIMARY KEY AUTO_INCREMENT,
    lname VARCHAR(100) NOT NULL,
    address VARCHAR(255),
    latitude DECIMAL(10, 8),
    longitude DECIMAL(11, 8)
);
```

```

-- Events table to store event information
CREATE TABLE Events (
    event_id INT PRIMARY KEY AUTO_INCREMENT,
    event_name VARCHAR(255) NOT NULL,
    event_category ENUM('social', 'fundraising', 'tech_talks', 'other') NOT NULL,
    description TEXT,
    time DATETIME NOT NULL,
    date DATE NOT NULL,
    location_id INT, -- Foreign key to Locations table
    contact_phone VARCHAR(20),
    contact_email VARCHAR(100),
    is_approved BOOLEAN DEFAULT TRUE, -- Whether the event is approved by super admin
    is_private BOOLEAN DEFAULT FALSE, -- Whether the event is private
    is_rso_event BOOLEAN DEFAULT FALSE, -- Whether the event is organized by an RSO
    FOREIGN KEY (location_id) REFERENCES Locations(location_id)
);

-- Comments table to store user comments on events
CREATE TABLE Comments (
    comment_id INT PRIMARY KEY AUTO_INCREMENT,
    event_id INT, -- Foreign key to Events table
    user_id INT, -- Foreign key to Users table
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    text TEXT,
    FOREIGN KEY (event_id) REFERENCES Events(event_id),
    FOREIGN KEY (user_id) REFERENCES Users(UID)
);

-- Ratings table to store event ratings by users
CREATE TABLE Ratings (
    rating_id INT PRIMARY KEY AUTO_INCREMENT,
    event_id INT, -- Foreign key to Events table
    user_id INT, -- Foreign key to Users table
    rating INT CHECK (rating >= 1 AND rating <= 5), -- Rating scale from 1 to 5 stars
    FOREIGN KEY (event_id) REFERENCES Events(event_id),
    FOREIGN KEY (user_id) REFERENCES Users(UID)
);

```

```

CREATE TABLE Events (
    event_id INT PRIMARY KEY,
    event_name VARCHAR(255),
    event_type VARCHAR(50) -- This column will store the type of event (RSO, Private,
    Public)
);

CREATE TABLE RSO_Events (
    event_id INT PRIMARY KEY,
    FOREIGN KEY (event_id) REFERENCES Events(event_id)
);

CREATE TABLE Private_Events (
    event_id INT PRIMARY KEY,
    FOREIGN KEY (event_id) REFERENCES Events(event_id)
);

CREATE TABLE Public_Events (
    event_id INT PRIMARY KEY,
    FOREIGN KEY (event_id) REFERENCES Events(event_id)
);

ALTER TABLE RSO_Events ADD CONSTRAINT fk_rso_events_events
    FOREIGN KEY (event_id) REFERENCES Events(event_id);

-- Check for intersection between RSO_Events and Private_Events
SELECT COUNT(*)
FROM RSO_Events r
JOIN Private_Events p ON r.event_id = p.event_id;
-- If the count is greater than 0, it means there are events that belong to both RSO
and Private, which violates the disjointness constraint.

-- Check for intersection between RSO_Events and Private_Events
SELECT COUNT(*)
FROM RSO_Events r
JOIN Private_Events p ON r.event_id = p.event_id;
-- If the count is greater than 0, it means there are events that belong to both RSO
and Private, which violates the disjointness constraint.

```

Assertions:

Disjointness Constraint between RSO Events and Private Events:

This assertion ensures that an event cannot belong to both RSO events and private events simultaneously.

```
CREATE ASSERTION Disjointness_RSOPrivate
CHECK (
    NOT EXISTS (
        SELECT 1
        FROM RSO_Events r
        JOIN Private_Events p ON r.event_id = p.event_id
    )
);
```

Triggers:

Trigger to Ensure RSO Events are Approved by Super Admin:

This trigger ensures that whenever an event is inserted into RSO_Events table, it is marked as approved by the super admin.

```
CREATE TRIGGER Approve_RSOPrivate
AFTER INSERT ON RSO_Events
FOR EACH ROW
BEGIN
    UPDATE Events
    SET is_approved = TRUE
    WHERE event_id = NEW.event_id;
END;
```

Trigger to Prevent Deletion of Approved Events:

This trigger prevents the deletion of events that have been approved by the super admin.

```
CREATE TRIGGER Prevent_Delete_ApprovedEvents
BEFORE DELETE ON Events
FOR EACH ROW
BEGIN
    IF OLD.is_approved = TRUE THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete an approved event.';
    END IF;
END;
```

Database Demo

The database demo including demonstration of all the different requirements is included in the project demo.

SQL

```
const express = require("express");
const cors = require("cors");
const mysql = require("mysql2");
const app = express();

app.use(express.json());
app.use(cors());

const db = mysql.createConnection(
  {
    host: "localhost",
    user: "root",
    password: "password",
    database: "test"
  })

// Route for registering a new user
app.post("/register", (req, res) => {
  const { username, password, user_level } = req.body;
  const sql = "INSERT INTO Users (username, password, user_level) VALUES (?, ?, ?)";
  db.query(sql, [username, password, user_level], (err, data) => {
    if (err) {
      console.log(err);
      return res.status(500).json({ error: "Error registering user" });
    }
    return res.status(200).json({ message: "User registered successfully" });
  });
});

// Route for logging in
app.post("/login", (req, res) => {
  const { username, password } = req.body;
```

```

const sql = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.query(sql, [username, password], (err, data) => {
  if (err) {
    console.log(err);
    return res.status(500).json({ error: "Error logging in" });
  }
  if (data.length === 0) {
    return res.status(401).json({ error: "Invalid credentials" });
  }
  return res.status(200).json({ message: "Login successful", user: data[0] });
});
});

// Route for fetching list of RSOS
app.get("/rso", (req, res) => {
  const sql = "SELECT * FROM RSOs";
  db.query(sql, (err, data) => {
    if (err) {
      console.error('Error fetching RSOs:', err);
      return res.status(500).json({ error: "Error fetching RSOs" });
    }
    return res.status(200).json(data);
  });
});

// Route for fetching events for a selected RSO
app.get("/rso/:rsoId/events", (req, res) => {
  const rsoId = req.params.rsoId;
  const sql = "SELECT * FROM Events WHERE rso_id = ?";
  db.query(sql, [rsoId], (err, data) => {
    if (err) {
      console.error('Error fetching events:', err);
      return res.status(500).json({ error: "Error fetching events" });
    }
    return res.status(200).json(data);
  });
});
});

```

```

// Route for creating a new RSO
app.post("/rso", (req, res) => {
  const { name, admin_id, university_id } = req.body;
  const sql = "INSERT INTO RSOs (name, admin_id, university_id) VALUES (?, ?, ?)";
  db.query(sql, [name, admin_id, university_id], (err, data) => {
    if (err) {
      console.error('Error creating RSO:', err);
      return res.status(500).json({ error: "Error creating RSO" });
    }
    return res.status(200).json({ message: "RSO created successfully" });
  });
});

// Route for updating an existing RSO
app.put("/rso/:rsoId", (req, res) => {
  const rsoId = req.params.rsoId;
  const { name, admin_id, university_id } = req.body;
  const sql = "UPDATE RSOs SET name = ?, admin_id = ?, university_id = ? WHERE rso_id = ?";
  db.query(sql, [name, admin_id, university_id, rsoId], (err, data) => {
    if (err) {
      console.error('Error updating RSO:', err);
      return res.status(500).json({ error: "Error updating RSO" });
    }
    return res.status(200).json({ message: "RSO updated successfully" });
  });
});

// Route for deleting an existing RSO
app.delete("/rso/:rsoId", (req, res) => {
  const rsoId = req.params.rsoId;
  const sql = "DELETE FROM RSOs WHERE rso_id = ?";
  db.query(sql, [rsoId], (err, data) => {
    if (err) {
      console.error('Error deleting RSO:', err);
      return res.status(500).json({ error: "Error deleting RSO" });
    }
    return res.status(200).json({ message: "RSO deleted successfully" });
  });
});

```

```
// Route for creating a new event
app.post("/events", (req, res) => {
  const { event_name, event_category, description, time, date, location_id, contact_phone,
contact_email, is_approved, is_private, is_rso_event } = req.body;
  const sql = "INSERT INTO Events (event_name, event_category, description, time, date,
location_id, contact_phone, contact_email, is_approved, is_private, is_rso_event) VALUES (?,
?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";
  db.query(sql, [event_name, event_category, description, time, date, location_id,
contact_phone, contact_email, is_approved, is_private, is_rso_event], (err, data) => {
    if (err) {
      console.error('Error creating event:', err);
      return res.status(500).json({ error: "Error creating event" });
    }
    return res.status(200).json({ message: "Event created successfully" });
  });
});

// API to get all events
app.get("/events", (req, res) => {
  const sql = "SELECT * FROM Events";
  db.query(sql, (err, result) => {
    if (err) {
      console.error("Error fetching events:", err);
      res.status(500).json({ error: "Error fetching events" });
    } else {
      res.status(200).json(result);
    }
  });
});

// Route for updating an existing event
app.put("/events/:eventId", (req, res) => {
  const eventId = req.params.eventId;
  const { event_name, event_category, description, time, date, location_id, contact_phone,
contact_email, is_approved, is_private, is_rso_event } = req.body;
  const sql = "UPDATE Events SET event_name = ?, event_category = ?, description = ?, time
= ?, date = ?, location_id = ?, contact_phone = ?, contact_email = ?, is_approved = ?,
is_private = ?, is_rso_event = ? WHERE event_id = ?";
```



```

    db.query(sql, [event_name, event_category, description, time, date, location_id,
contact_phone, contact_email, is_approved, is_private, is_rso_event, eventId], (err, data) =>
{
    if (err) {
        console.error('Error updating event:', err);
        return res.status(500).json({ error: "Error updating event" });
    }
    return res.status(200).json({ message: "Event updated successfully" });
});
});

// Route for deleting an existing event
app.delete("/events/:eventId", (req, res) => {
    const eventId = req.params.eventId;
    const sql = "DELETE FROM Events WHERE event_id = ?";
    db.query(sql, [eventId], (err, data) => {
        if (err) {
            console.error('Error deleting event:', err);
            return res.status(500).json({ error: "Error deleting event" });
        }
        return res.status(200).json({ message: "Event deleted successfully" });
    });
});

// Route for adding comments to an event
app.post("/events/:eventId/comments", (req, res) => {
    const event_id = req.params.eventId;
    const { user_id, text } = req.body;
    const sql = "INSERT INTO Comments (event_id, user_id, text) VALUES (?, ?, ?)";
    db.query(sql, [event_id, user_id, text], (err, data) => {
        if (err) {
            console.error('Error adding comment:', err);
            return res.status(500).json({ error: "Error adding comment" });
        }
        return res.status(200).json({ message: "Comment added successfully" });
    });
});
});

```

```

// Route for fetching comments for an event
app.get("/events/:eventId/comments", (req, res) => {
  const event_id = req.params.eventId;
  const sql = "SELECT * FROM Comments WHERE event_id = ?";
  db.query(sql, [event_id], (err, data) => {
    if (err) {
      console.error('Error fetching comments:', err);
      return res.status(500).json({ error: "Error fetching comments" });
    }
    return res.status(200).json(data);
  });
});

// Route for updating a comment
app.put("/events/:eventId/comments/:commentId", (req, res) => {
  const eventId = req.params.eventId;
  const { comment_id, text } = req.body;
  const sql = "UPDATE Comments SET text = ? WHERE event_id = ? AND comment_id = ?";
  db.query(sql, [text, eventId, comment_id], (err, data) => {
    if (err) {
      console.error('Error updating comment:', err);
      return res.status(500).json({ error: "Error updating comment" });
    }
    return res.status(200).json({ message: "Comment updated successfully" });
  });
});

// Route for deleting a comment
app.delete("/events/:eventId/comments/:commentId", (req, res) => {
  const eventId = req.params.eventId;
  const commentId = req.params.commentId;
  const sql = "DELETE FROM Comments WHERE event_id = ? AND comment_id = ?";
  db.query(sql, [eventId, commentId], (err, result) => {
    if (err) {
      console.error('Error deleting comment:', err);
      return res.status(500).json({ error: "Error deleting comment" });
    }
    if (result.affectedRows === 0) {
      return res.status(404).json({ error: "Comment not found" });
    }
  });
});

```

```
    }  
    return res.status(200).json({ message: "Comment deleted successfully" });  
  });  
});  
  
app.listen(8081, () => { console.log("Server is running on port 8081"); });
```

SQL Message Screenshots

More examples included in the project demo.

The screenshot displays a web application interface with a blue gradient background. At the top, a dark notification box shows the message "localhost:3000 says Login failed" with an "OK" button. Below this, a light blue form contains two sections: "Login" and "Register". The "Login" section has input fields for "Username" (containing "RickL") and "Password" (containing "*****"), followed by a blue "Login" button. The "Register" section has input fields for "Username", "Password", and "Role" (containing "student, admin or super_admin"), followed by a blue "Register" button.

localhost:3000 says

Login successful

OK

Login

Login

Register

student, admin or super_admin

Register

localhost:3000 says
Event created successfully

OK

Create RSO

RSO Name

Create RSO

Create Event

test

social

test

30-Apr-2024 01:17 PM

12-Apr-2024

1

232234234

test@gmail.com

Create Event

Events

test1 - social

Delete

Tea Time - social

Delete

Chai time - social

Delete

Meet and Greet - social

Delete

Help the poor - fundraising

Delete

RSOs

American Social Club

Delete

Blue Socials

Delete

Hispanic Social Club

Delete

Indian Social Club

Delete

Soccer Club

Delete

New RSO

Delete

localhost:3000 says
RSO created successfully

OK

Create RSO

New RSO

Create RSO

Create Event

Event Name

Event Category: social/fundraising

Description

dd----yyyy --:-- --

dd----yyyy

Location ID

Contact Phone

Contact Email

Create Event

Events

test1 - social

Delete

Tea Time - social

Delete

Chai time - social

Delete

Meet and Greet - social

Delete

Help the poor - fundraising

Delete

RSOs

American Social Club

Delete

Blue Socials

Delete

Hispanic Social Club

Delete

Indian Social Club

Delete

Soccer Club

Delete

Conclusion

Database Performance:

Query response time was satisfactory overall around 5ms average, but certain queries may benefit from indexing to improve performance. Suggested indexes include those on frequently queried columns such as event times, event locations, and RSO names.

Desired Features/Functionalities:

- Security (Login): Implementing a secure login system would enhance user authentication and access control.
- Event Feed from University Websites: Integrating event feeds from university websites, such as XML feeds from <http://events.ucf.edu/>, would provide users with a comprehensive view of all campus events.
- Social Network Integration: Integrating with social networks like Facebook and Twitter would allow users to easily share events and engage with their social circles.

Problems Encountered and Lessons Learned:

- Database Schema Design: Striking a balance between normalization and performance optimization proved challenging. We learned the importance of carefully designing the database schema to ensure both data integrity and efficient query execution.
- Error Handling: Managing error messages and user feedback effectively requires attention to detail and thorough testing. We gained insights into the importance of robust error handling mechanisms to enhance user experience.
- Continuous Learning: The project highlighted the need for ongoing learning and skill development in database management and web development technologies. We recognized the importance of staying updated with industry best practices and emerging trends.

Retrospective Improvements:

- Refined Data Model: In retrospect, we would have conducted a more comprehensive analysis of the data model to identify potential areas for improvement in terms of normalization and performance.
- Enhanced User Experience: We would prioritize implementing additional features to enhance the user experience, such as advanced search functionality, cleaner UI, additional features such as posting to social media, etc.