College Festival Management Web Application

March 3, 2023.

Suchi Sharma
Vaishnovi Arun
Tapaswini Cherukuri
Tirzah Grace Yarranna

CS30202
DataBase Management system
IIT KHARAGPUR

Table of Contents

System Description

This system is accessible through a user-friendly website that serves as the application interface. The primary objective is to provide a comprehensive platform for efficiently organizing and participating in various festival activities.

Intended Users

External Participants: Participants external to the university can browse event schedules,and register for events either as participants. Additionally, logistics information such as accommodation and food arrangements can be accessed.

Internal Students: University students can browse event schedules, and register for events either as participants or volunteers.

Organizers: Responsible for the planning and management of specific festival events, organizers can log in to the system to oversee event details and manage various aspects of the events they are organizing.

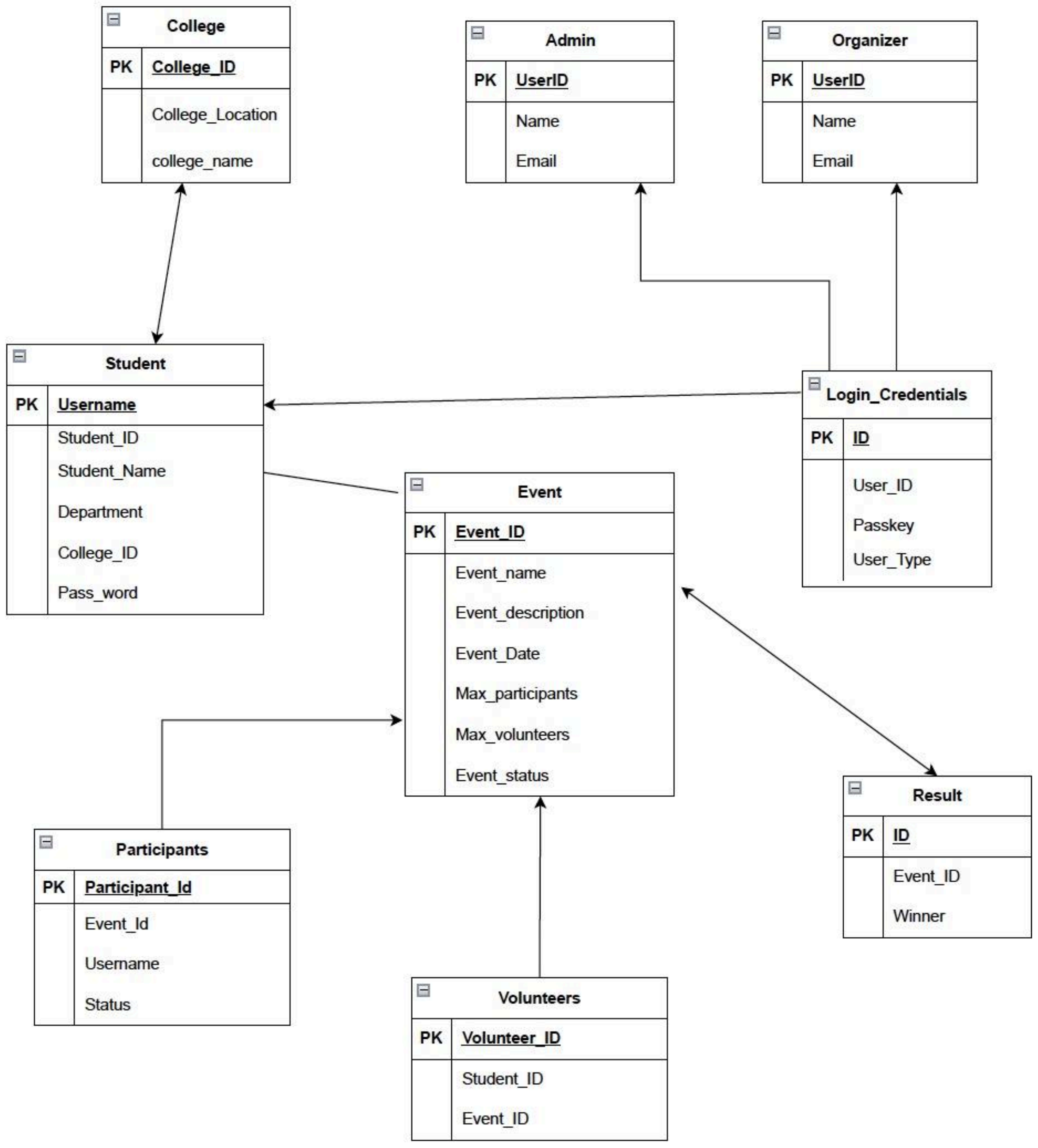
Administrators: Admin users have the authority to manage organizers, ensuring seamless coordination and execution of the festival.

Functional Workflow

1. External Participants:
    a. Create an account and/or Log in to the system
    b. Access detailed information about each event
    c. Register as participants for specific events
    d. Access logistics information such as accommodation and food
2. Students:
    a. Create an account and/or Log in to the system
    b. Access detailed information about each event
    c. Register for events as participants or volunteers

3. Organizers:
    a. Create an account and/or Log in to the system
    b. Manage and oversee the details of events they are responsible for.
4. Administrators:
    a. Manage Organizer accounts.

Schema



Functions Implemented

For each of the intended users, functions are implemented.

1. **Viewing Event Details:**

   Anyone visiting the website can browse and view details of all available events, their schedule, description.

2. **External Participants:**
   a. Viewing Event Details
   b. Account Creation and Login: External participants have the capability to create an account and securely log in.
   c. Event Registration: Once logged in, external participants can register for specific events of their choice.
   d. Event Information: Participants can access comprehensive information about each event, including event descriptions, winners, and other relevant details.
   e. Logistics Information: Participants can view logistics information related to accommodation and food.
   f. Profile Management: External participants have the flexibility to update their account details as needed.

3. **Students:**
   a. Viewing Event Details
   b. Login: Students can log in to their accounts using secure authentication.
   c. Register as Participant: Students have the option to register for events as a participant
   d. Register as Volunteer: Students have the option to register for events as a Volunteer
   e. Event Information: Students can access comprehensive information about each event, including event descriptions, winners, and other relevant details.
   f. Profile Management: Students have the flexibility to update their account details as needed.

4. **Organizers:**
    a. Account Creation and Login: Organizers can create an account and log in to the system.
    b. Event Management: Organizers can view details of events they are responsible for, including participants and volunteers.
    c. Event Updates: Organizers can update event status, schedule times, and manage event winners.
    d. Profile Management: Organizers have the ability to update their account details , password.

5. **Admin:**
    a. Login: Administrators can securely log in to the system.
    b. Organizer Management: Admins have the authority to add or delete organizers to ensure effective event coordination.

Technologies Used

Frontend

1. JavaScript
2. React
3. HTML
4. CSS

Backend

1. Node js
2. Express js
3. Database: MySQL

Triggers

Triggers are designed to automate specific actions in response to certain events or changes in the database.

1. Trigger on Remove organizer:
    a. Should be removed from both organizer and login_credentials tables

```
export async function removeOrganizer(userId) {
const sql1 = `DELETE FROM organizer WHERE userID = ?;`
const values = [userId];
await pool.query(sql1, values)
const sql2 = `DELETE FROM login_credentials WHERE userID = ?;`
await pool.query(sql2, values
}
```

2. Trigger on adding Winner:
    a. Need to update results, participant, their join table.

```
export async function addWinner(eventId, winner) {
 console.log(eventId,winner)
 const sql1 = "INSERT INTO results (event_id,winner) VALUES ( ? , ? );"
 const values1 = [eventId, winner];
 await pool.query(sql1, values1)
 const sql2 = "UPDATE participant SET status = 'won' WHERE participant_id
= ? ;"
 const values2 = [winner];
 await pool.query(sql2, values2)
 const sql3 = "SELECT r.winner, p.username, p.status FROM results r JOIN
participant p ON p.participant_id = r.winner AND p.event_id = r.event_id
WHERE p.event_id = ? ;"
 const values3 = [eventId];
 const [rows] = await pool.query(sql3, values3)
 return [rows]
}
```

3. Trigger on Inserting into Logistics and Students :
    a. The procedure insert_into_logistics_and_students_proc() on calling finds out one accommodation venue and a food venue which are not full(i.e., less than maximum capacity) and allocates it

```
export async function insertIntoLogisticsAndStudents(userId){
 await pool.query('DROP TRIGGER IF EXISTS logistics_trigger');
 await pool.query('DROP PROCEDURE IF EXISTS
insert_into_logistics_and_students_proc');
 const sqlProcedure = `
 CREATE PROCEDURE insert_into_logistics_and_students_proc()
 BEGIN

     DECLARE v_accomodation_id INT;
     DECLARE v_food_id INT;


     SELECT a.accomodation_id INTO v_accomodation_id
     FROM accomodation a
     LEFT JOIN (
         SELECT accomodation_id, COUNT(*) AS logistics_count
         FROM logistics
         GROUP BY accomodation_id
     ) l ON a.accomodation_id = l.accomodation_id
     WHERE l.accomodation_id IS NULL OR l.logistics_count < a.max_capacity
     LIMIT 1;

     SELECT f.food_id INTO v_food_id
     FROM food f
     LEFT JOIN (
         SELECT food_id, COUNT(*) AS logistics_count
         FROM logistics
         GROUP BY food_id
     ) l ON f.food_id = l.food_id
     WHERE l.food_id IS NULL OR l.logistics_count < f.max_capacity
     LIMIT 1;


     SELECT v_accomodation_id AS accomodation_id_value, v_food_id AS
food_id_value;
```

```
     INSERT INTO logistics (accomodation_id, food_id) VALUES
(v_accomodation_id, v_food_id);


 END`;

 await pool.query(sqlProcedure);
 const sqlTrigger = `
 CREATE TRIGGER logistics_trigger
 AFTER INSERT ON logistics
 FOR EACH ROW
 BEGIN
 INSERT INTO logistics_students (logistics_id, student_id)
 VALUES (NEW.logistics_id, ?);
 END;
`;

await pool.query(sqlTrigger,[userId]);
const[rows]=await pool.query('CALL
insert_into_logistics_and_students_proc();');



   /*const proc_call='CALL insert_into_logistics_and_students_proc();'
   await pool.query(proc_call);*/

   //const[rows]=await pool.query(sqlProcedure);
   //const[rows]=await pool.query(sqlTrigger,[userId]);
   return [rows];
}
```

Queries

Queries play a crucial role in connecting frontend requests to the backend database. The flow typically involves the following steps:

1. User Interaction in Frontend
2. Handling Frontend Request in Backend
3. Execution of Queries
4. Database Interaction
5. Retrieval of Results
6. Data Transformation and Response

Example of Queries used in the code:

```
export async function getEvents() {
   const [rows] = await pool.query("SELECT * FROM events")
   return [rows]
}

export async function getLoginData(userId,password) {
 const sql = "SELECT * FROM login_credentials WHERE userId = ? AND passkey = ?";
 const [rows] = await pool.query(sql, [userId, password]);
 return [rows]
}
export async function getOrganizers() {
 const [rows] = await pool.query("SELECT * FROM organizer")
 return [rows]
}

export async function addOrganizer(userId) {
 console.log(typeof(userId))
 const sql = `INSERT INTO login_credentials (userId, passkey, user_type) VALUES
('${userId}','abc','organizer');`
```

```
 // const values = [userId,'abc','organizer'];
 await pool.query(sql)
 return [{'default_password':'abc'}]
}

export async function removeOrganizer(userId) {
 // console.log(userId)
 const sql1 = `DELETE FROM organizer WHERE userID = ?;`
 const values = [userId];
 await pool.query(sql1, values)
 const sql2 = `DELETE FROM login_credentials WHERE userID = ?;`
 await pool.query(sql2, values)
}

export async function getParticipants(eventId) {
 const sql = `SELECT p.participant_id, s.username, s.student_name, c.college_name
 FROM participant p
 JOIN student s ON p.username = s.username
 JOIN college c ON s.college_id = c.college_id
 WHERE p.event_id = ?;`
 const values = [eventId];
 const [rows] = await pool.query(sql, values)
 return [rows]
}

export async function getVolunteers(eventId) {
 const sql = `SELECT v.volunteer_id, s.student_id, s.student_name, s.department FROM
volunteer v JOIN student s ON v.student_id = s.username WHERE v.event_id = ? ;`
 const values = [eventId];
 const [rows] = await pool.query(sql, values)
 return [rows]
}

export async function getEventStatus(eventId) {
 const sql = `SELECT e.event_status FROM events e WHERE e.event_id = ?;`
 const values = [eventId];
 const [rows] = await pool.query(sql, values)
 return [rows]
}

export async function getStudentLogin(userId, password) {
 const sql = "SELECT * FROM student WHERE username = ? AND pass_word = ?";
 const [rows] = await pool.query(sql, [userId, password]);
 return [rows]
}

export async function getProfileDetails(userId, collegeId) {
 const sql = "SELECT * FROM student WHERE username = ? AND college_id = ?";
 const [rows] = await pool.query(sql, [userId, collegeId]);
 return [rows]
}

export async function RegisterEvent(userId, collegeId, eventId) {
 const sql = "INSERT INTO participant (participant_id, username, event_id, college_id,
status) VALUES (CONCAT(?,?,?),?,?,?,?);";
 const [rows] = await pool.query(sql, [userId,
collegeId,eventId,userId,eventId,collegeId,'registered']);
 return [rows]
}

export async function VolunteerEvent(userId,eventId) {
 const sql = "INSERT INTO volunteer (volunteer_id, student_id, event_id) VALUES
(CONCAT(?,?),?,?);";
 const [rows] = await pool.query(sql, [userId,eventId,userId,eventId]);
 return [rows]
}
```

Forms

Forms play a crucial role in facilitating user interaction within the web application. They are employed to collect user input and initiate various actions.

Following are the forms implemented:

1. **Student and External Participant Forms**
   a. **Login Form**
   b. **Signup Form**

   # login_signup form combined.

```
<form onSubmit={handleSubmit}>
  <div className="inputs">
    {action === 'Sign Up' && ( // Conditional rendering for signup inputs
      <>
        <div className="input">
          <img src={name_icon} alt="" />
          <input type="text" placeholder="Name" />
        </div>
        <div className="input">
          <img src={id_icon} alt="" />
          <input type="text" placeholder="ID Number" />
        </div>
        <div className="input">
          <img src={school_icon} alt="" />
          <input type="text" placeholder="College Name" />
        </div>
      </> )}
    <div className="input"> <img src={user_icon} alt="" />
        <input type="text" placeholder="username" onChange={e =>
        setUserId(e.target.value)} />
    </div> <div className="input"> <img src={lock_icon} alt="" />
        <input type="password" placeholder='password' onChange={e =>
        setPassword(e.target.value)} />
    </div>
  </div>
  {action === 'Login' &&
        <div className="forgot-password">Lost Password? <span>Click
        Here!</span></div>}
  <button type="submit" className="btn" >Submit</button>
  <div className="submit-container">
        <div className={action == "Login" ? "submit gray" : "submit"} onClick={()
        => { setAction("Sign Up") }}>Sign Up</div>
         <div className={action == "Sign Up" ? "submit gray" : "submit"}
        onClick={() => { setAction("Login") }}>Login</div>
  </div>
</form>
```

c. Profile View Form

```
<form>
       <div className="form-group">
             <label for="exampleInputEmail1">Name</label>
             <input type="text" className="form-control"
             id="exampleInputEmail1" aria-describedby="emailHelp"
             value={Name} />
       </div>
       <div className="form-group">
         <label for="exampleInputPassword1">Your Student ID</label>
             <input type="text" className="form-control"
             id="exampleInputPassword1" value={studentId} />
       </div>
       <div className="form-group">
         <label for="exampleInputPassword2">Your Department</label>
         <input type="text" className="form-control"  value={deptName} />
       </div>
       <div className="form-group">
       <button type="submit" className="btn btn-primary">Submit</button>
       </div>
 </form>
```

2. Organizer and Admin Forms

a. Login Form

```
<form onSubmit={handleSubmit}>
  <div class="mb-3">
      <label for="exampleInputEmail1" class="form-label">UserID</label>
```

```
        <input type="text" class="form-control" id="exampleInputEmail1"
        onChange={e=> data.setUserId(e.target.value)} />
    </div>
    <div class="mb-3">
        <label for="exampleInputPassword1" class="form-label">Password</label>
        <input type="password" class="form-control" id="exampleInputPassword1"
        onChange={e=> setPassword(e.target.value)} />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

b. Add Organizer Form

```
<form class="container" onSubmit={addOrganizer}>
    <div class="mb-3">
      <label for="exampleInputEmail1" class="form-label">Enter the
      userId of the new member</label>
       <input type="text" class="form-control" id="exampleInputEmail1"
       onChange={e=> setUserId(e.target.value)} />
    </div>
    <button type="submit" class="btn btn-primary">Add Organizer</button>
</form>
```

3. **User Interaction and Submission**

   Users interact with these forms by entering their credentials, and upon submission, the data is sent to the respective backend routes for processing.

4. **Backend Processing**

   The backend routes associated with these forms execute functions to validate credentials, manage user sessions, and perform other actions based on the form's purpose.

By utilizing these forms, the web application ensures seamless communication between users and the backend, enhancing the overall user experience.