College Festival Management Web Application Development

Team: Dextop

Dhruv Agja(21CS10022)
Galipelli Sai Mallikarjun(21CS30019)
Ishaan Sinha(21CS30064)
Vinayak Gupta(21CS10077)

1. Introduction

A **University Fest Management System (UFMS)** is a pivotal web application designed to streamline the intricate process of organizing and executing university cultural festivals. It serves as a **centralized platform for managing event schedules**, **participant registrations**, **volunteer coordination**, and **logistical arrangements**. The UFMS plays a critical role in enhancing the efficiency and effectiveness of festival planning, ensuring a memorable and seamless experience for participants, organizers, volunteers, and attendees.

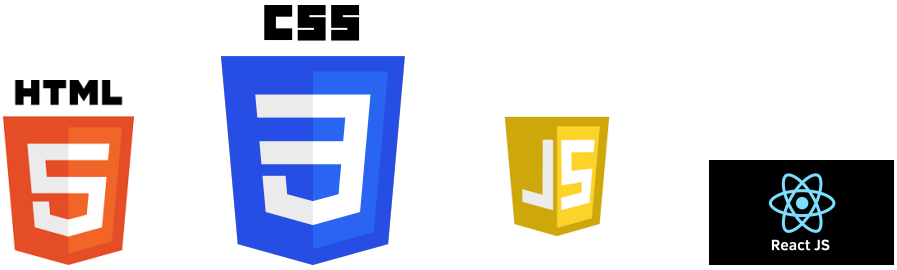
Database Management Systems (DBMS) constitute the foundational framework of the UFMS, **facilitating the storage**, **retrieval**, and **manipulation of vast quantities of festival-related data**. By leveraging robust DBMS functionalities, the UFMS empowers stakeholders to **access accurate and up-to-date information**, enabling informed decision-making and efficient resource allocation throughout the festival lifecycle.

In this project, our objective is to develop a comprehensive University Fest Management System that harnesses the power of modern technologies and database management principles. Our goal is to create an intuitive and user-friendly platform that caters to the diverse needs of **festival organizers**, **participants**, **volunteers**, and **administrators**. By prioritizing scalability, reliability, and usability, we aim to establish a UFMS that not only streamlines festival operations but also fosters community engagement and promotes the vibrant spirit of university cultural festivals. Through collaborative development and innovative design, we aspire to set new standards in festival management, ensuring the continued success and growth of university festivities for years to come.

2. Languages and tools used

2.1 Frontend

The front-end of a web application is developed using a combination of **HTML**, **CSS**, **JavaScript**, and **ReactJS**. These technologies work together to create a visually appealing and interactive user interface that allows users to interact with the application and access its various features and functions. HTML provides the **structure and content** of the web page, CSS is used for **styling and layout**, JavaScript enables **dynamic behavior and interactivity**, and ReactJS is a popular library for building **reusable UI components and managing application state**.



2.2 Backend

For the back-end, we used **Node.js** as a server-side JavaScript runtime. To simplify the development process, we also used the **Express.js** framework which provides tools for **handling HTTP requests and responses**, **routing**, and **middleware**. To interact with our PostgreSQL database, we used the **pg** library. Overall, this backend technology stack allowed us to build a fast and efficient web application with robust database connectivity and secure resource access.

2.3 Database

In our project, we utilized **PostgreSQL** as our preferred database management system for **storing and organizing schemas**. PostgreSQL is a widely used, open-source relational database management system that allows us to create and manage databases, tables, and other related objects. By using PostgreSQL, we were able to ensure that our **data is organized and easily accessible**, with the **ability to retrieve, modify, and analyze data efficiently**.

2.4 Operating System

Our application is designed to be compatible with multiple operating systems, including **Windows** and **Linux**. By ensuring cross-platform compatibility, we have made our application accessible to **a wider audience** and have made it **easier for users** to access its features and functions, regardless of the device or operating system they are using.

3. Database Schemas

3.1 Schema for Users

users (
id VARCHAR(255) NOT NULL,
password VARCHAR(255) NOT NULL,
role VARCHAR(255) NOT NULL,
PRIMARY KEY (id)
);
The above SQL schema defines a "users" table with three columns: id, passsword and role. id is the primary key. The table is used to store user information, including their unique id, role and encrypted password (using bcrypt).

3.2 Schema for Student

Student (
Roll VARCHAR(255) NOT NULL,
Name VARCHAR(255) NOT NULL,
Dept VARCHAR(255) NOT NULL,
PRIMARY KEY (Roll)
);
The above SQL schema defines a "Student" table with three columns: "Roll", "Name", and "Dept". "Roll" is the unique identifier for each student and is marked as NOT NULL, ensuring that every student entry must have a roll number. "Name" represents the name of the student and is marked as NOT NULL, indicating that every student entry must have a name. "Dept" represents the department of the student and is marked as NOT NULL, ensuring that every student entry must have a department. The "Roll" column is the primary key of the table, ensuring that each student's roll number is unique within the table. The table is designed to store information about students, including their roll number, name, and department.

3.3 Schema for Event

Event (
EID INT NOT NULL,
EName VARCHAR(255) NOT NULL,
Date DATE,
Type VARCHAR(255) NOT NULL,
Description VARCHAR(255) NOT NULL,
PRIMARY KEY (EID)
);
The provided SQL schema defines an "Event" table with the following columns: EID: An integer column representing the Event ID. It is marked as NOT NULL, indicating that every event entry must have a unique ID.
EName: A variable-length string column representing the name of the event. It is marked as NOT NULL, ensuring that every event entry must have a name.
Date: A date column representing the date of the event. It allows NULL values, indicating that the date may not be specified for all events.
Type: A variable-length string column representing the type of event. It is marked as NOT NULL, ensuring that every event entry must have a type.
Description: A variable-length string column representing the description of the event. It is marked as NOT NULL, ensuring that every event entry must have a description.
The EID column serves as the primary key for the table, ensuring that each event has a unique identifier within the table. The table is designed to store information about events, including their ID, name, date, type, and description.

3.4 Schema for Role

Role (
RID INT NOT NULL,
RName VARCHAR(255) NOT NULL,
Description VARCHAR(255) NOT NULL,
PRIMARY KEY (RID)
);
The provided SQL schema defines a "Role" table with three columns: "RID","RName" and "Description". "RID" is an integer column representing the Role ID. It is marked as NOT NULL, indicating that every role entry must have a unique ID. "RName" is a variable-length string column representing the name of the role. It is marked as NOT NULL, ensuring that every role entry must have a name. "Description" is a variable-length string column representing the description of the role. It is marked as NOT NULL, ensuring that every role entry must have a description. The "RID" column serves as the primary key for the table, ensuring that each role has a unique identifier within the table. The table is designed to store information about roles, including their ID, name, and description.

3.5 Schema for Student Manage

Student_Manage (
Roll VARCHAR(255) NOT NULL,
EID INT NOT NULL,
RID INT NOT NULL,
PRIMARY KEY (Roll, EID, RID),
FOREIGN KEY (Roll) REFERENCES Student(Roll),
FOREIGN KEY (EID) REFERENCES Event(EID),
FOREIGN KEY (RID) REFERENCES Role(RID)
);
The provided SQL schema defines a "Student_Manage" table with the following columns:
Roll: A variable-length string column representing the roll number of the student. It is marked as NOT NULL, ensuring that every student entry must have a roll number.
EID: An integer column representing the event ID. It is marked as NOT NULL, ensuring that every entry must have an event ID.
RID: An integer column representing the role ID. It is marked as NOT NULL, ensuring that every entry must have a role ID.
The combination of (Roll, EID, RID) serves as the primary key for the table, ensuring that each entry is uniquely identified. The table includes foreign key constraints: The "Roll" column references the "Roll" column in the "Student" table. The "EID" column references the "EID" column in the "Event" table. The "RID" column references the "RID" column in the "Role" table. The table is designed to manage student participation in events with their respective roles.

3.6 Schema for College

College (
Name VARCHAR(255) NOT NULL,
Address VARCHAR(255) NOT NULL,
PRIMARY KEY (Name)
);
The provided SQL schema defines a "College" table with two columns: Name and Address. Name is the primary key. The table stores information about colleges, including their unique names and addresses.

3.7 Schema for Volunteer

Volunteer (
Roll VARCHAR(255) NOT NULL,
PRIMARY KEY (Roll),
FOREIGN KEY (Roll) REFERENCES Student(Roll)
);

The provided SQL schema defines a "Volunteer" table with one column: Roll. Roll is marked as NOT NULL which means every volunteer which is a student only must have a roll number. Roll serves as the primary key, establishing each volunteer's unique identity. Additionally, the table establishes a foreign key constraint referencing the 'Roll' column in the 'Student' table, connecting volunteers to student records in order to keep a record of which all students of the college are volunteers(in respective events).

3.8 Schema for Volunteers of an Event

```
Event_Has_Volunteer (
EID INT NOT NULL,
Roll VARCHAR(255) NOT NULL,
PRIMARY KEY (EID, Roll),
FOREIGN KEY (EID) REFERENCES Event(EID),
FOREIGN KEY (Roll) REFERENCES Volunteer(Roll)
);
```

The "Event_Has_Volunteer" table schema consists of two columns: EID (Event ID) and Roll. Both columns are marked as NOT NULL to ensure data integrity. The combination of EID and Roll serves as the primary key, allowing each volunteer to be associated with a specific event uniquely. Additionally, the table establishes foreign key constraints referencing the EID column in the Event table and the Roll column in the Volunteer table, facilitating the relationship between events and volunteers.

3.9 Schema for Participant

```
Participant (
Name VARCHAR(255) NOT NULL,
PID VARCHAR(255) NOT NULL,
CollegeName VARCHAR(255) NOT NULL,
PRIMARY KEY (PID),
FOREIGN KEY (CollegeName) REFERENCES College(Name)
);
```

The "Participant" table schema comprises three columns: Name, PID (Participant ID), and CollegeName. Each column is marked as NOT NULL to ensure data integrity. PID serves as the primary key, allowing for the unique identification of participants. The CollegeName column establishes a foreign key constraint, referencing the Name column in the College table to associate participants with their respective colleges.

3.10 Schema for Student Participation

Student_Participates (
Roll VARCHAR(255) NOT NULL,
EID INT NOT NULL,
PRIMARY KEY (Roll, EID),
FOREIGN KEY (Roll) REFERENCES Student(Roll),
FOREIGN KEY (EID) REFERENCES Event(EID)
);
The "Student_Participates" table schema includes two columns: Roll and EID (Event ID). Both columns are marked as NOT NULL, ensuring data integrity. The combination of Roll and EID serves as the primary key, allowing each student to participate in multiple events uniquely. Additionally, the table establishes foreign key constraints referencing the Roll column in the Student table and the EID column in the Event table, facilitating the relationship between students and events they participate in.

3.11 Schema for External Participant's Participation in an Event

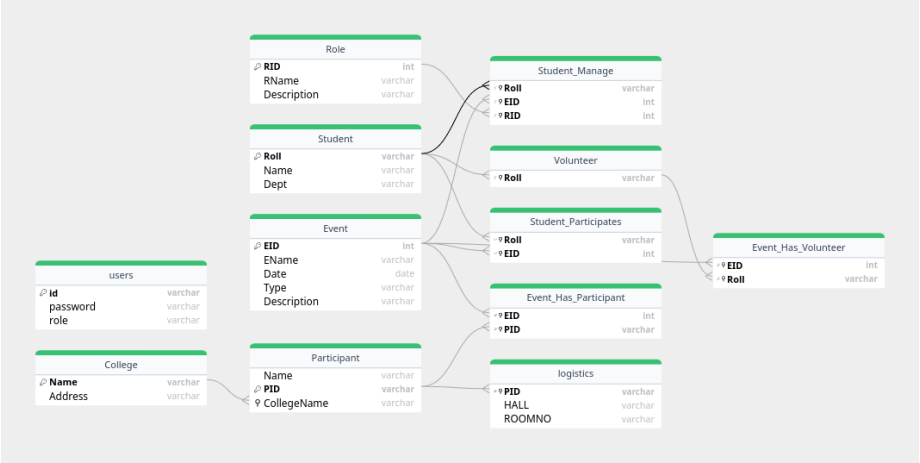
Event_Has_Participant (
EID INT NOT NULL,
PID VARCHAR(255) NOT NULL,
PRIMARY KEY (EID, PID),
FOREIGN KEY (EID) REFERENCES Event(EID),
FOREIGN KEY (PID) REFERENCES Participant(PID)
);
The "Event_Has_Participant" table schema consists of two columns: EID (Event ID) and PID (Participant ID). Both columns are marked as NOT NULL to enforce data integrity. The combination of EID and PID serves as the primary key, allowing each event-participant pair to be unique. Additionally, the table establishes foreign key constraints referencing the EID column in the Event table and the PID column in the Participant table, ensuring referential integrity and facilitating the association between events and participants.

3.12 Schema for Logistics

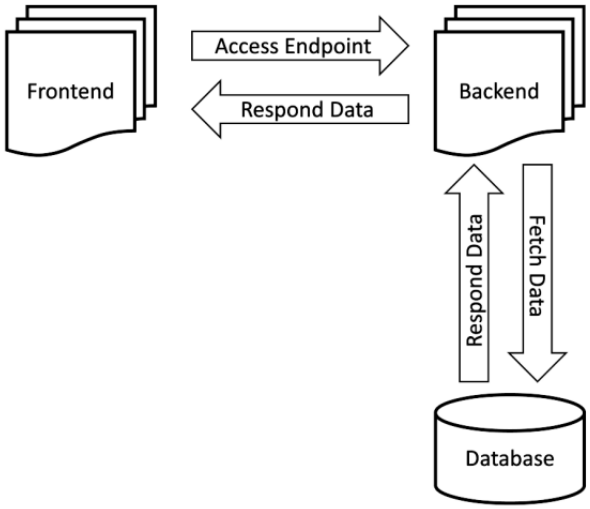
logistics (
PID VARCHAR(255) NOT NULL,
HALL VARCHAR(255) NOT NULL,
ROOMNO VARCHAR(255) NOT NULL,
PRIMARY KEY (PID),
FOREIGN KEY (PID) REFERENCES Participant(PID)
);

The "logistics" table schema consists of three columns: PID (Participant ID), HALL, and ROOMNO. Each column is marked as NOT NULL to ensure data integrity. PID serves as the primary key, allowing for the unique identification of logistics entries. Additionally, the table establishes a foreign key constraint referencing the PID column in the Participant table, ensuring referential integrity between logistics and participant records.

4. ER Diagram


Role
RID int
RName varchar
Description varchar
Student
Roll varchar
Name varchar
Dept varchar
Event
EID int
EName varchar
Date date
Type varchar
Description varchar
users
Id varchar
password varchar
role varchar
College
Name varchar
Address varchar
Participant
Name varchar
PID varchar
CollegeName varchar
Student_Manage
Roll varchar
EID int
RID int
Volunteer
Roll varchar
Student_Participates
Roll varchar
EID int
Event_Has_Participant
EID int
PID varchar
logistics
PID varchar
HALL varchar
ROOMNO varchar
Event_Has_Volunteer
EID int
Roll varchar

5. Triggers and Workflows



5.1 Creating Users, Students and Participants

User:
"INSERT INTO users (id,password,role)VALUES($1, $2, $3) RETURNING *", [id, hashedPassword, req.body.role]
This SQL query inserts a new record into the "users" table with the provided values for the "id", "password", and "role" columns. The values for these columns are provided as parameters: $1 corresponds to the id, $2 corresponds to the hashedPassword, and $3 corresponds to the role, as specified by the placeholders in the query. The RETURNING clause is used to return the inserted row back to the client-side application after the insertion operation is completed. The array [id, hashedPassword, req.body.role] contains the actual values that

will be inserted into the respective columns: the "id" value is taken from the variable "id", the "password" value is taken from the variable "hashedPassword", and the "role" value is taken from the request body's "role" parameter. Additionally, it's worth noting that the hashed password is generated using bcrypt, a cryptographic hash function designed for secure password hashing. Bcrypt is commonly used to protect passwords by producing a hash that is computationally expensive to reverse, thereby enhancing security. This trigger is typically used in the context of user registration or user creation operations, where new user information is provided by the client-side application and then securely inserted into the database for storage.

Students:
"INSERT INTO student (roll, name, dept) VALUES ($1, $2, $3) RETURNING *, [id, name, dept]
The trigger represents an SQL query designed to insert data into the "student" table within the database. It utilizes the "INSERT INTO" clause to indicate the intention of adding new records to the table. The specific columns targeted for insertion are denoted within parentheses following the table name: "roll", "name", and "dept". These columns will receive the corresponding values specified in the subsequent "VALUES" clause. The "VALUES" clause is structured to accept placeholders, indicated by "$1", "$2", and "$3". These placeholders serve as variables that will be replaced with actual values during the execution of the query. The values to be inserted are provided within an array, where "id", "name", and "dept" represent the values corresponding to the respective columns. Upon successful insertion, the "RETURNING " clause retrieves the newly inserted row(s) from the "student" table. This clause allows for the retrieval of all columns () associated with the inserted row(s), providing an opportunity for further processing or validation of the newly added data. Overall, this trigger is commonly utilized to add new records to the "student" table, facilitating the storage of student-related information within the database. It enables efficient data management and retrieval processes within the database system.

Participants:
"INSERT INTO participant (name, pid, collegename) VALUES ($1, $2, $3) RETURNING *", [name, id, collegename]
The trigger comprises an SQL query intended to insert data into the "participant" table within the database. It begins with the "INSERT INTO" clause, indicating the insertion of new records into the specified table. The target columns for insertion, namely "name", "pid", and "collegename", are enclosed within parentheses following the table name declaration. The "VALUES" clause accompanies the trigger and serves to receive the values intended for insertion into the respective columns of the "participant" table. Within the clause, placeholders denoted by "$1", "$2", and "$3" are utilized to represent variables awaiting actual values during query execution. These values, referenced within an array, include "name", "id", and "collegename", which correspond to the data

to be inserted into the respective columns. Following the insertion operation, the "RETURNING *" clause is invoked to retrieve the newly inserted row(s) from the "participant" table. This clause facilitates the retrieval of all columns associated with the inserted row(s), allowing for subsequent processing or validation of the newly added data. Overall, this trigger is commonly employed to facilitate the addition of new records to the "participant" table, thereby enabling efficient management and retrieval of participant-related information within the database.

5.2 During Login

"SELECT id, password FROM users WHERE users.id = $1 AND users.role = $2", [id, req.body.role]
In essence, this trigger retrieves the "id" and "password" of a user from the "users" table based on specific conditions. The conditions are that the "id" of the user must match the value provided by the variable $1, and the "role" of the user must match the value provided by the variable $2, which is taken from the request body (req.body.role). This is basically for providing authentication during login process

5.3 Displaying Events on Home Page

"SELECT * FROM event"
This SQL query selects all columns (*) from the "event" table. It retrieves all rows and columns from the "event" table without any filtering conditions. We are not applying any filtering condition as we want each and every attributes of the event to be displayed on home page. So as and when a student or a participant or an organizer logs in, this trigger is responsible for fetching all the required necessities from database using the above SQL query.

5.4 Registering for an Event

For Students:
"INSERT INTO student_participates (roll, eid) VALUES ($1, $2) RETURNING *", [req.params.id, eid]
The given SQL trigger is an INSERT INTO statement intended to insert data into the "student_participates" table. It attempts to insert values into the "roll" and "eid" columns of the table, with placeholders $1 and $2 representing the values to be inserted. The actual values for insertion are taken from the request parameters, where req.params.id corresponds to the value to be inserted into the "roll" column, and "eid" corresponds to the value to be inserted into the "eid" column. The RETURNING * clause retrieves the newly inserted row(s) from the "student_participates" table. This means that after the insertion operation is completed, the trigger will return the newly inserted row(s), including

all columns, to the caller. In summary, the trigger attempts to add a new entry into the "student_participates" table(specifying the "roll" and "eid" values) whenever a student registers for a particular event by clicking on the register button(given in the home page).

For External Participants:
"INSERT INTO event_has_participant (eid, pid) VALUES ($1, $2) RETURNING *", [eid, req.params.id]
The provided SQL statement is an INSERT INTO query aimed at inserting data into the "event has participant" table. This operation endeavors to input values into the "eid" and "pid" columns of the table, with placeholders $1 and $2 designating the values to be inserted. The actual values for insertion are derived from the request parameters, where req.params.id denotes the value intended for the "pid" column, and "eid" signifies the value intended for the "eid" column. The RETURNING * clause is included to retrieve the newly inserted row(s) from the "event has participant" table. Following the completion of the insertion operation, the trigger is set to return the newly inserted row(s), encompassing all columns, back to the caller. In essence, the trigger's purpose is to append a new entry into the "event has participant" table, specifying the values for "eid" and "pid". This operation occurs when an external participant registers for a specific event, typically through an action such as clicking the register button provided on the homepage.

5.5 Displaying Registered Events

For Students:
"SELECT * FROM event, student_participates WHERE event.eid = student_participates.eid AND student_participates.roll = $1", [id]
Whenever a student clicks on the registered events in the navigation bar provided on the homepage it will be redirected to a page where only those events with necessary details will be displayed for which the student has registered, this functionality has been implemented using the above SQL trigger which will fetch the list of registered events(along with necessary attributes) by that particular student, whenever the student clicks registered events. The provided SQL trigger is a SELECT query intended to retrieve data from two tables, "event" and "student_participates", based on specific conditions. The SELECT statement retrieves all columns (*) from the "event" and "student_participates" tables. The FROM clause specifies the tables from which the data is retrieved: "event" and "student_participates".

The WHERE clause filters the rows based on two conditions:
1. event.eid = student_participates.eid: This condition ensures that only rows with matching event IDs in both tables are selected.
2. student_participates.roll = $1: This condition matches the "roll" column from the "student_participates" table with the value provided in the array [id].

For External Participants:
SELECT * FROM event, event_has_participant WHERE event.eid = event_has_-participant.eid AND event_has_participant.pid = $1", [id]
Whenever an external participant clicks on the registered events link in the navigation bar provided on the homepage, they are directed to a dedicated page displaying events for which they have registered. This functionality is implemented using the following SQL trigger, which retrieves the list of registered events, along with their necessary attributes, for that specific participant whenever they access the registered events section. The provided SQL trigger is a SELECT query designed to fetch data from two tables, "event" and "event_has_participant", under specific conditions. The SELECT statement retrieves all columns (*) from the "event" and "event_has_participant" tables. The FROM clause specifies the tables involved in the query: "event" and "event_has_participant".

The WHERE clause filters the rows based on two conditions:
1. event.eid = event_has_participant.eid: This condition ensures that only rows with matching event IDs in both tables are selected.
2. event_has_participant.pid = $1: This condition matches the "pid" column from the "event_has_participant" table with the value provided in the array [id].

5.6 Volunteering for an Event

Note: Only a student can volunteer for any event

"INSERT INTO event_has_volunteer (eid,,roll) VALUES($1, $2) RETURNING *", [eid, roll]
The given SQL trigger is an INSERT INTO statement intended to insert data into the "event_has_volunteer" table. It attempts to insert values into the "eid" and "roll" columns of the table, with placeholders $1 and $2 representing the values to be inserted. The actual values for insertion are taken from the parameters provided. In detail, the "eid" corresponds to the event ID to which the volunteer is associated, while the "roll" represents the identifier of the volunteer. These values are derived from the respective variables, "eid" and "roll", passed in the array during the trigger execution. Similar to the trigger for registering events, the RETURNING * clause retrieves the newly inserted row(s) from the "event_has_volunteer" table. This implies that upon the completion of the insertion operation, the trigger will return the newly inserted row(s), including all columns, to the caller. In summary, the trigger endeavors to add a new entry into the "event_has_volunteer" table, specifying the "eid" and "roll" values, whenever a volunteer offers to participate in a particular event by engaging with the volunteering feature provided on the platform.

5.7 Displaying Volunteered Events

"SELECT * FROM event, event_has_volunteer WHERE event.eid = event_has_-volunteer.eid AND event_has_volunteer.roll = $1", [id]
Whenever a student clicks on the volunteered events in the navigation bar provided on the homepage, they are directed to a page where only those events with necessary details will be displayed for which the student has volunteered. This functionality has been implemented using the above SQL trigger, which will fetch the list of volunteered events (along with necessary attributes) by that particular student, whenever the student clicks on volunteered events. The provided SQL trigger is a SELECT query intended to retrieve data from two tables, "event" and "event_has_volunteer", based on specific conditions. The SELECT statement retrieves all columns (*) from the "event" and "event_has_volunteer" tables. The FROM clause specifies the tables from which the data is retrieved: "event" and "event_has_volunteer".

The WHERE clause filters the rows based on two conditions:
1. event.eid = event_has_volunteer.eid: This condition ensures that only rows with matching event IDs in both tables are selected.
2. event_has_volunteer.roll = $1: This condition matches the "roll" column from the "event_has_volunteer" table with the value provided in the array [id].

5.8 Displaying the Logistics of an External Participant to him/her

"SELECT * FROM logistics WHERE pid = $1", [id]
This trigger basically provide the functionality of displaying the logistics like hall allotted, room number and participant id to the participant whenever he click on logistics in navigation bar provided on the homepage of an external participant. The SQL query basically retrieves all columns from the "logistics" table where the participant ID matches the value provided in the array [id]. The SELECT statement fetches all columns (*) from the "logistics" table. The WHERE clause filters the rows based on the condition "pid = $1".This condition ensures that only rows with matching participant IDs in the "logistics" table are selected. The value of $1 is derived from the parameter [id]. In summary, the trigger aims to provide the participant with detailed logistics information based on their participant ID, allowing them to access relevant details regarding hall allocation and room numbers during the event.

5.9 Displaying the List of Volunteers to an Organizer

Note: An Organizer can only see the volunteers of those events which he/she is organizing. No such volunteer of any other event which is

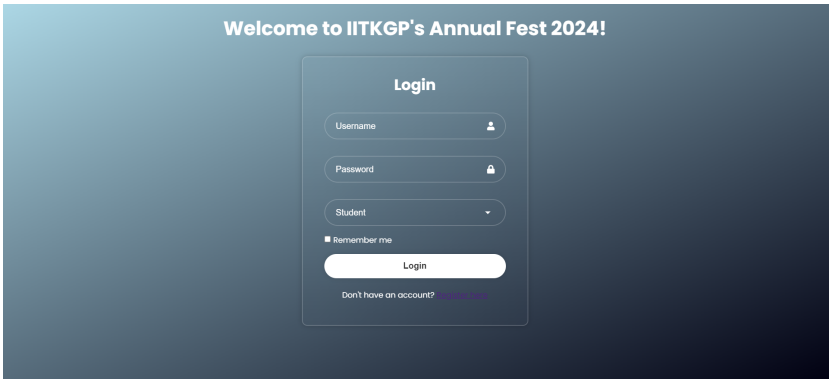
not organized by that organizer should be visible to the organizer.
"SELECT event.eid, event_has_volunteer.roll, event.ename, event.type FROM event_has_volunteer, event, student_manage WHERE event_has_volunteer.eid = event.eid AND event.eid = student_manage.eid AND student_manage.roll = $1", [id]
This trigger is designed to retrieve the list of volunteers along with their specific information, whenever an organizer from its homepage clicks on volunteers provided in the navigation bar. The purpose of this trigger is to display relevant details about events for which volunteers have been assigned and only those volunteers who are volunteering in any of the events organized by that organizer. The SQL query retrieves data from three tables: "event_has_volunteer", "event", and "student_manage". The SELECT statement specifies the columns to be retrieved: event.eid, event_has_volunteer.roll, event.ename, and event.type. The FROM clause indicates the tables involved in the query: "event_has_volunteer", "event", and "student_manage".

The WHERE clause filters the rows based on the following conditions:
1. event_has_volunteer.eid = event.eid: This condition ensures that the event ID matches between the "event_has_volunteer" and "event" tables.
2. event.eid = student_manage.eid: This condition ensures that the event ID matches between the "event" and "student_manage" tables.
3. student_manage.roll = $1: This condition matches the roll of the student in the "student_manage" table with the value provided in the array [id].

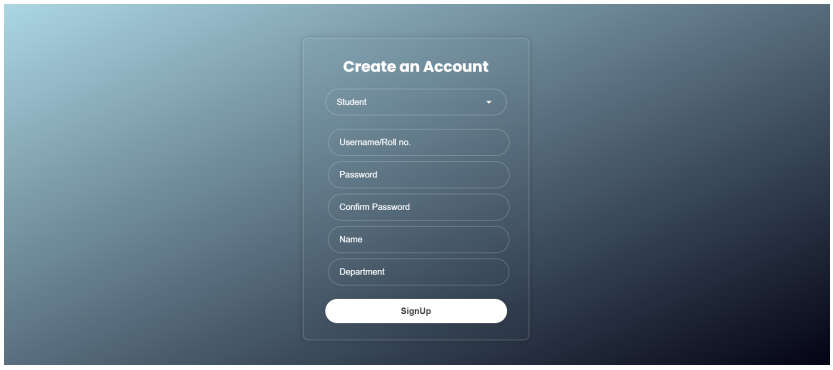
In summary, the trigger aims to retrieve event-related information where a specific student, identified by their roll, is involved. This information can be useful for managing volunteers and organizing events efficiently.

6. Web Interface

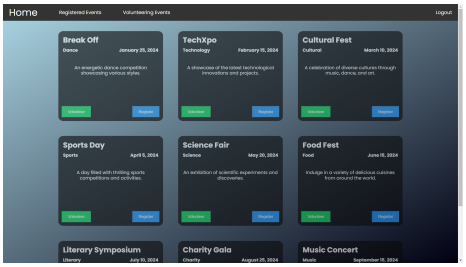
6.1 Login Page



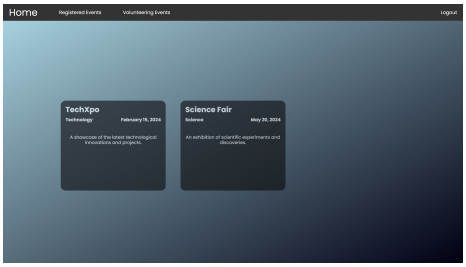
6.2 Signin Page



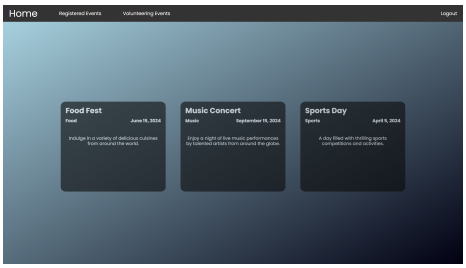
6.3 Interface for Student



Home Page

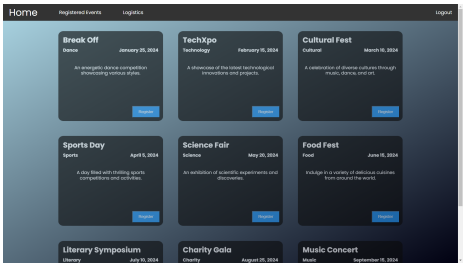


List of Registered Events

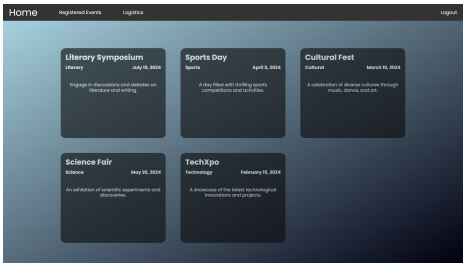


List of Volunteered Events

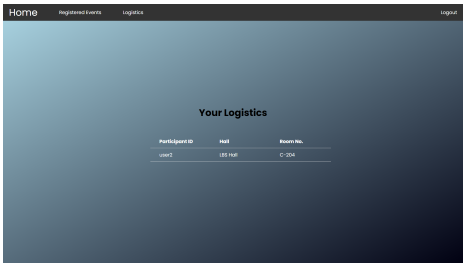
6.4 Interface for External Participant



Home Page

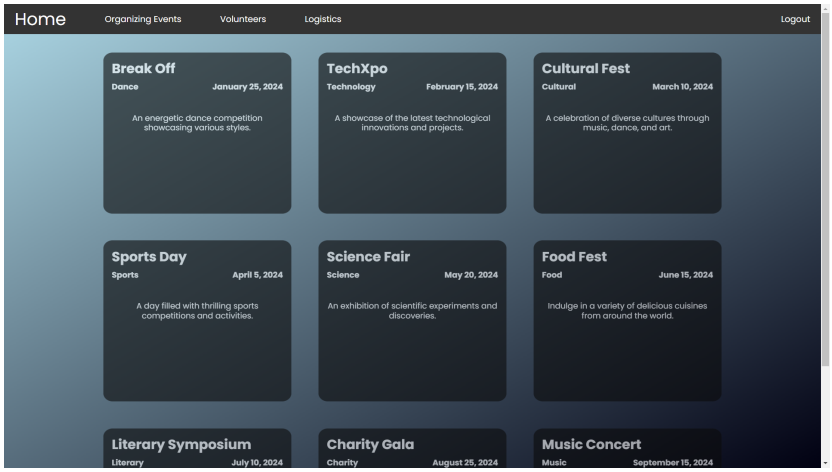


List of Registered Events

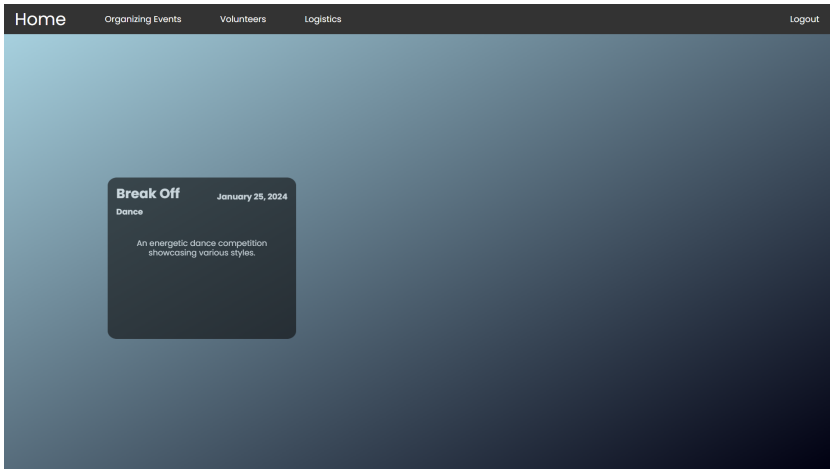


Logistics of the participant

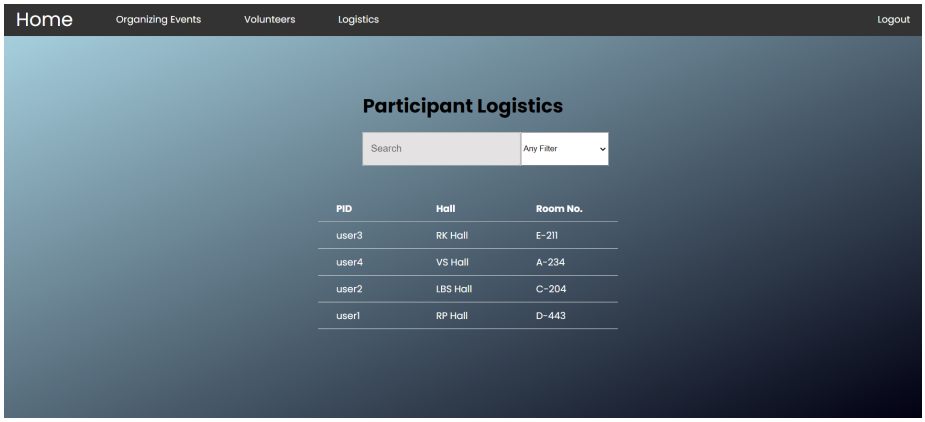
6.5 Interface for Organizer



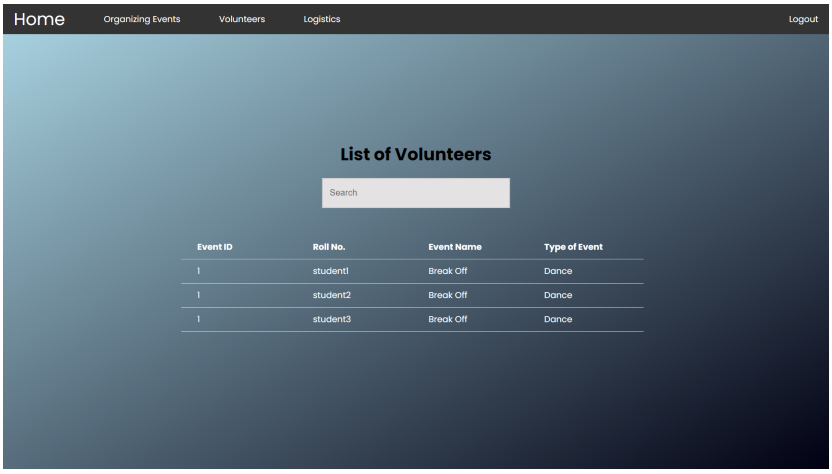
Home Page



List of Organizing Events

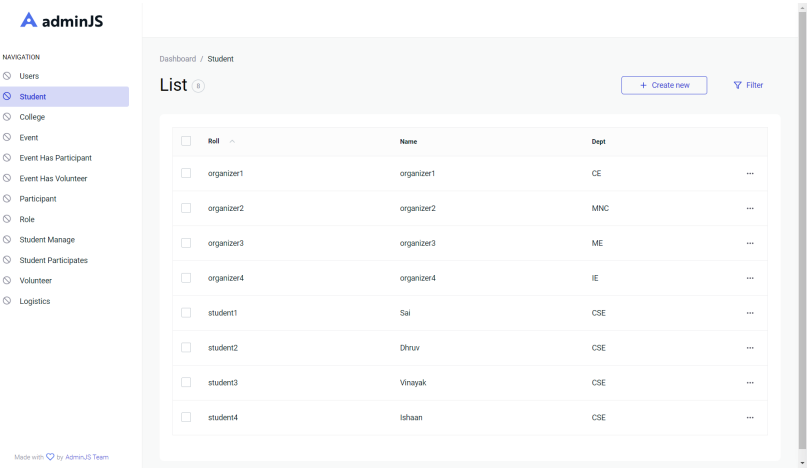


Logistics of external participants



List of Volunteers

6.6 Admin Interface



7. Additional Functionalities

7.1 Handling Encryption

```
const bcrypt = require('bcrypt');

const saltRounds = 10;

const encrypt = async (password) => {
    try {
        const hashedPassword = await bcrypt.hash(password, saltRounds);
        return hashedPassword;
    } catch (error) {
        throw new Error('Encryption failed');
    }
};

const decrypt = async (password, hashedPassword) => {
    try {
        const match = await bcrypt.compare(password, hashedPassword);
        return match;
    } catch (error) {
        throw new Error('Decryption failed');
    }
};

module.exports = { encrypt, decrypt };
```

The provided Node.js module utilizes the bcrypt library for password encryption and decryption. Here's the explanation of how bcrypt works within the context of the provided code:

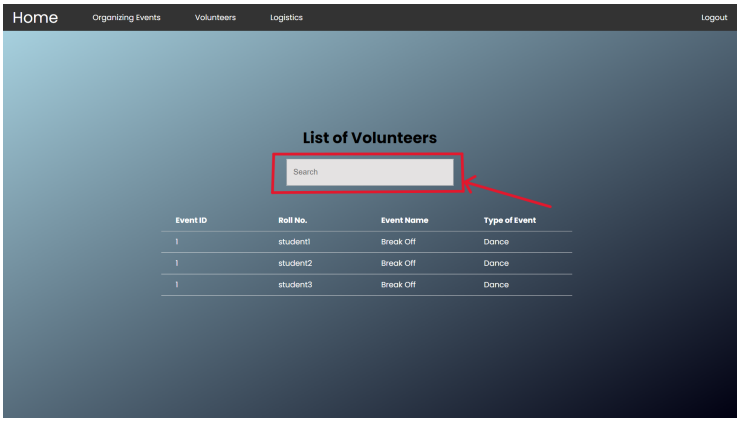
In contrast to symmetric encryption algorithms like AES, bcrypt is a one-way hashing function specifically designed for password hashing. It enhances security by incorporating features like salting and a cost factor to increase computational difficulty.

Hashing with Bcrypt:
The encrypt function takes a plain text password as input. Inside this function, the bcrypt.hash method is utilized to generate a secure hash of the password. The saltRounds variable determines the cost factor for the hashing algorithm, influencing the computational cost of generating the hash. A higher value increases the computational expense. The resulting hashed password is returned from the encrypt function.

Comparing Hashes with Bcrypt:
Unlike traditional encryption algorithms, bcrypt does not allow for the decryption of hashed passwords back to their original form. Instead, it compares a plain text password with a hashed password to verify if they match. The decrypt function accepts a plain text password and a hashed password as input. Inside the decrypt function, the bcrypt.compare method is employed to compare the plain text password with the hashed password. If the comparison succeeds, bcrypt returns true, indicating that the plain text password matches the hashed password. Otherwise, it returns false.

In summary, bcrypt offers a secure and efficient means of hashing passwords for storage and comparison. It employs a one-way hashing algorithm with salts and a cost factor to mitigate common password security vulnerabilities like rainbow table attacks and brute force cracking. The bcrypt library abstracts away many complexities associated with securely hashing passwords, making it a preferred choice for password hashing in web applications and other software systems.

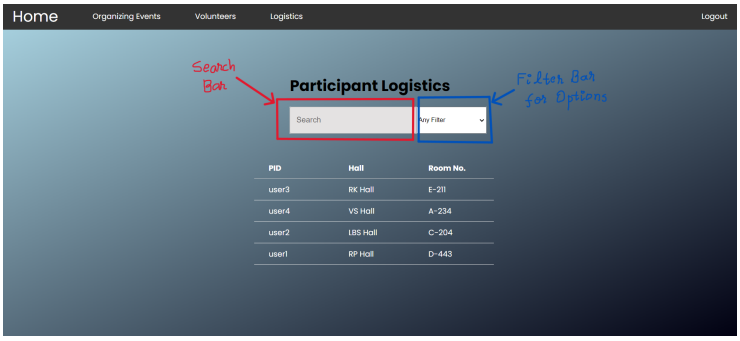
7.2 Implementing Search Function in List of Volunteers



List of Volunteers

This search bar searches any input field provided(in the search bar) among all attributes and even if the input field matches any substring of it then that volunteer will be displayed on the page. So this is one of the additional functionality, as now an organizer along with viewing the list of volunteers could also search a volunteer by giving input in the search bar provided here.

7.3 Implementing Search Function along with filters in logistics for an organizer



Logistics of External Participants

We have Implemented a search bar in logistics of External Participants but in this search bar we have given few options to the organizer that he/she may

search for participants either by applying no filter or by applying a suitable filter of their wish like PID, Hall or Room number. And the searching algorithm which we have implemented over here is a bit different than the searching algorithm in the volunteers page, here if an organizer is searching for an external participant by applying no field then whatever input he/she is giving in the search bar then that input will be checked for matching from starting of the string of the attributes not substring, and if a filter is given then the input is checked with only the attributes of the particular field.

THE END