ABSTRACT

The Travel Corporation project is a groundbreaking endeavor poised to revolutionize the travel landscape by introducing a comprehensive platform that seamlessly integrates user-generated and professionally curated travel experiences, thereby fostering a dynamic global community of adventurers. At its core, the project aims to provide individuals with a flexible and immersive environment to both host and join travels, enabling them to explore new destinations with ease. Leveraging state-of-the-art technology and a user-centric approach, the platform prioritizes intuitive navigation and customization options, empowering users to tailor their journeys according to their preferences and interests. Through the integration of interactive maps, real-time communication channels, and secure payment gateways, the Travel Corporation ensures a seamless and secure booking process for hosts and travelers alike. Furthermore, the project seeks to cultivate a vibrant online community by offering forums, social networking features, and collaborative tools that facilitate knowledge-sharing, experience exchange, and the formation of meaningful connections among like-minded individuals. By bridging the gap between adventure seekers and travel enthusiasts worldwide, the Travel Corporation aspires to not only redefine the traditional travel experience but also to promote cultural exchange, foster global understanding, and inspire a spirit of exploration and discovery on a scale never seen before. In essence, it represents a transformative force in the travel industry, poised to shape the future of travel in an increasingly interconnected world.

PROBLEM STATEMENT

In today's rapidly evolving travel landscape, traditional methods of planning and experiencing journeys often lack the flexibility, personalization, and community engagement desired by modern adventurers. Existing travel platforms typically offer limited options for customization, lack integration between user-generated and professionally curated experiences, and fail to foster a cohesive global community of travelers. This fragmentation results in a disjointed and often unsatisfactory travel experience for both hosts and travelers, hindering the exploration of new destinations and the formation of meaningful connections with fellow adventurers. Thus, there is a pressing need for a comprehensive travel platform that seamlessly integrates user-generated and professionally curated travel opportunities, prioritizes customization and community engagement, and empowers individuals to embark on unforgettable journeys tailored to their interests and preferences.

TABLE OF CONTENTS

1. PROBLEM UNDERSTANDING, IDENTIFICATION OF ENTITY AND RELATIONSHIPS, CONSTRUCTION OF DB USING ER MODEL FOR THE PROJECT

PROBLEM UNDERSTANDING:

The travel industry faces several challenges, including fragmented travel planning processes, limited options for personalization, and a lack of cohesive community engagement among travelers. Traditional travel platforms often fail to meet the evolving needs and preferences of modern adventurers, resulting in a disjointed and unsatisfactory travel experience. There is a growing demand for a comprehensive solution that seamlessly integrates user-generated and professionally curated travel opportunities, prioritizes customization, and fosters a vibrant global community of travelers.

ER DIAGRAM:

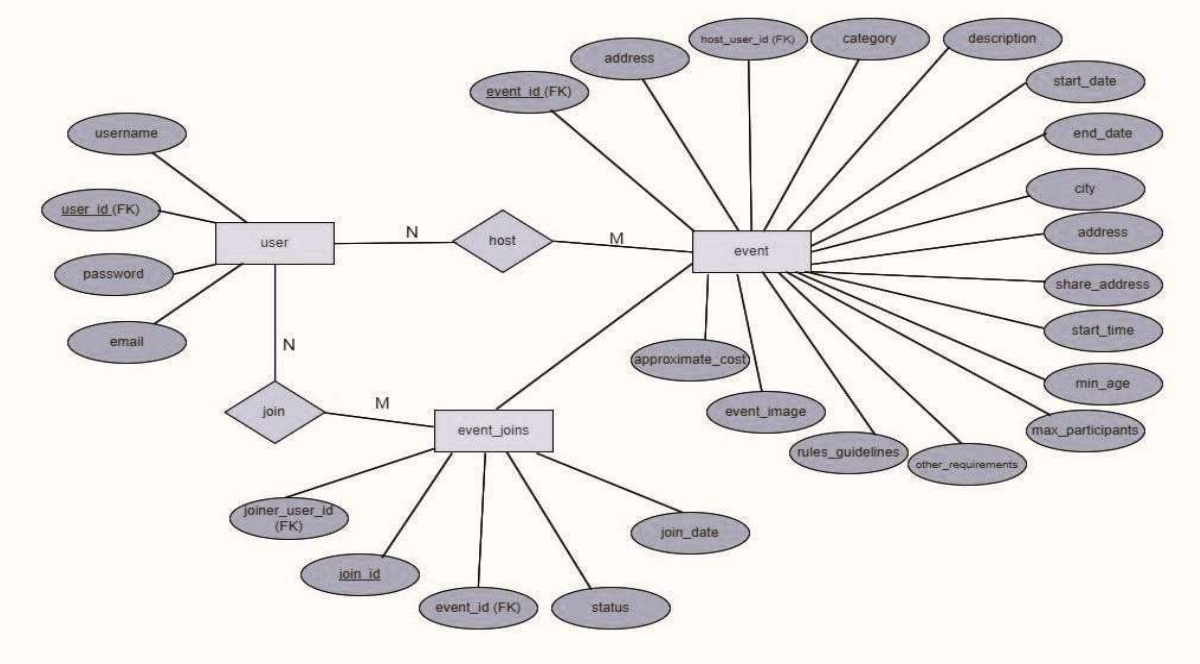

Figure 1.1

2. DESIGN OF RELATIONAL SCHEMAS, CREATION OF DATABASE TABLES FOR THE PROJECT

DESIGN OF RELATIONAL SCHEMA:

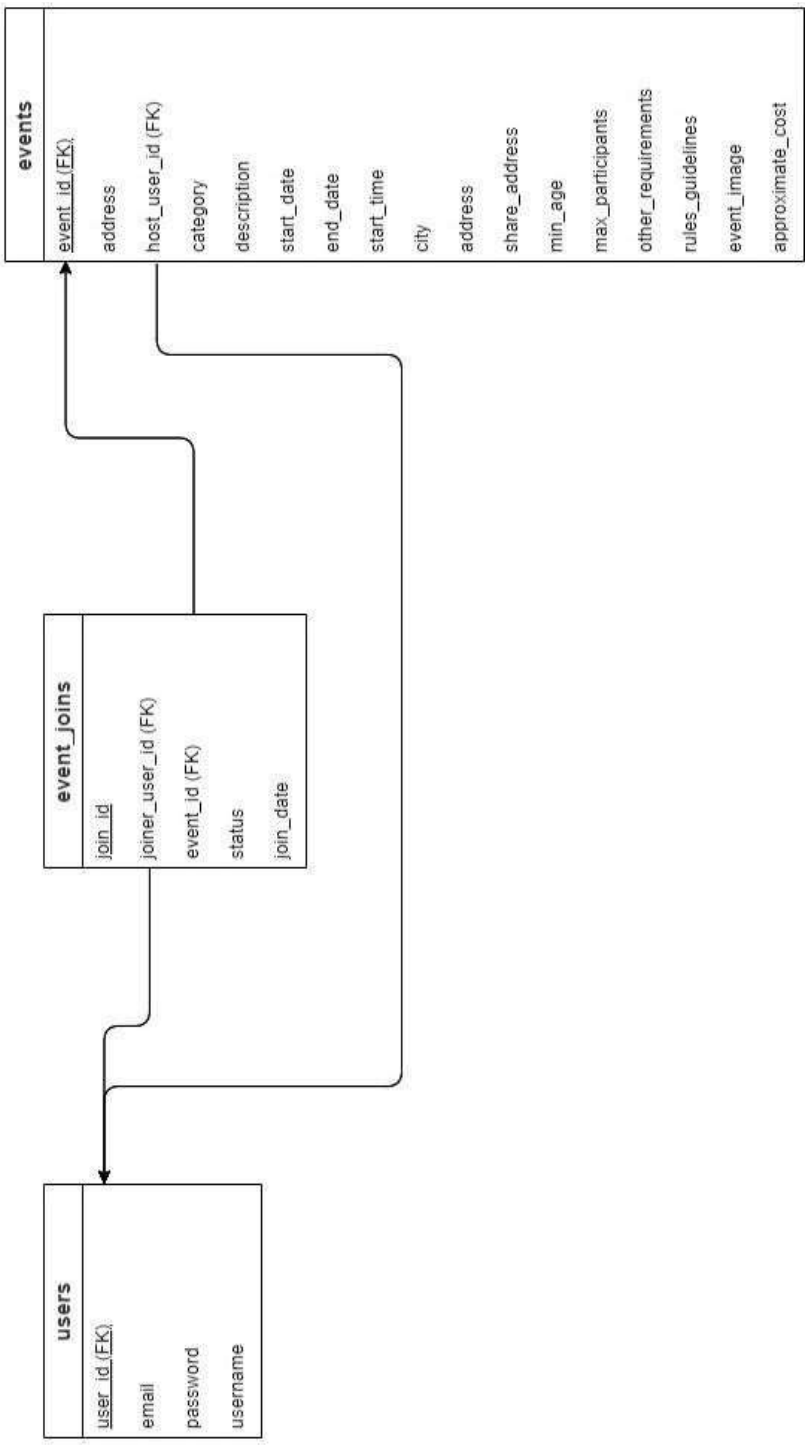

Figure 2.1

CREATION OF DATABASE TABLES:

user

column	Type
user_id (Primary)	INT
username	VARCHAR
email	VARCHAR
password	VARCHAR

Figure 2.2

event_joins

column	Type
join_id (Primary)	INT
event_id	INT
joiner_user_id	INT
join_date	TIMESTAMP
status	ENUM

Figure 2.3

events

column	Type
event_id (Primary)	INT
host_user_id	INT
title	VARCHAR
category	VARCHAR
description	TEXT
start_date	DATE
end_date	DATE
start_time	TIME
city	VARCHAR
address	VARCHAR
share_adddress	BOOLEAN
max_participants	INT
min_age	INT
other_requirements	TEXT
approximate_cost	DECIMAL
rules_guidelines	TEXT
event_image	VARCHAR

Figure 2.4

3. COMPLEX QUERIES BASED ON THE CONCEPTS OF CONSTRAINTS, SETS, JOINS, VIEWS, TRIGGERS AND CURSORS.

TRIGGERS:

Triggers are actions automatically performed in response to certain events. In this code, triggers are used to update event display status based on participant counts. Triggers are utilized to automate the management of event display status based on participant numbers. For instance, when a new record is inserted into the event_joins table, the update_event_display trigger activates, checking if the current number of participants for a specific event exceeds the maximum limit defined in the corresponding events table. If the limit is surpassed, the trigger updates the display status of the event to false, indicating it is full.

```
DELIMITER $$
CREATE TRIGGER update_event_display AFTER INSERT ON event_joins
FOR EACH ROW
BEGIN
    DECLARE current_participants INT;

    -- Get the current number of participants for the event
    SELECT COUNT(*) INTO current_participants
    FROM event_joins
    WHERE event_id = NEW.event_id;
```

Figure 3.1

```
    -- Update the display status based on the current number of participants
    IF current_participants >= (
        SELECT max_participants
        FROM events
        WHERE event_id = NEW.event_id
    ) THEN
        -- If max participants reached, set display status to false
        UPDATE events
        SET display_status = FALSE
        WHERE event_id = NEW.event_id;
    END IF;
END$$
CREATE TRIGGER update_display_on_remove AFTER DELETE ON event_joins
```

Figure 3.2

VIEWS:

The View all_events_details combine data from multiple tables (events and event_joins) to provide a comprehensive view of all events along with details of the hosts and participants. This view aggregates information such as event titles, categories, start and end dates, host usernames, and joiner usernames, effectively simplifying querying and data retrieval for users.

```
CREATE VIEW event_details AS
SELECT
    e.event_id,
    e.title,
    e.category,
    e.start_date,
    e.end_date,
    u.username AS host_username,
    GROUP_CONCAT(u_joiner.username) AS joiner_usernames,
    GROUP_CONCAT(u_joiner.email) AS joiner_emails, -- Adding the joiner emails
    GROUP_CONCAT(u_joiner.user_id) AS joiner_ids -- Adding the joiner IDs
```

Figure 3.3

```
e.event_id,
e.title,
e.category,
e.start_date,
e.end_date,
u.username;
select * from event_details;
```

Figure 3.4

event_id	title	category	start_date	end_date	host_username	joiner_username	join_date	entity_type
18	MADRAS	travel	2024-04-18	2024-04-20	Ashutosh Jha	NULL	NULL	Event
19	bangalorer	travel	2024-04-18	2024-04-24	Ashutosh Jha	NULL	NULL	Event
20	bangalorer	travel	2024-04-18	2024-04-24	Ashutosh Jha	NULL	NULL	Event

Figure 3.5

PROCEDURE & CURSOR:

The fetch_user_details stored procedure is created to retrieve event details from the events table. Inside the procedure, a cursor is used to select title, start_date, and end_date from the events table. The cursor iterates through each row of the result set, fetching event details into variables. Within the loop, the fetched event details are printed using the SELECT statement. Finally, the cursor is closed, and the stored procedure is called using CALL fetch_user_details () to execute the cursor and retrieve event details.

```
CREATE PROCEDURE fetch_user_details()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE event_title VARCHAR(255);
    DECLARE event_start_date DATE;
    DECLARE event_end_date DATE;
    DECLARE cur CURSOR FOR SELECT title, start_date, end_date FROM events;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN cur;

    read_loop: LOOP
        FETCH cur INTO event_title, event_start_date, event_end_date;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SELECT CONCAT('Event Title: ', event_title, ', Start Date: ', event_start_date, ', End Date: ', event_end_date);
    END LOOP;
    CLOSE cur;
END$$
```

Figure 3.6

OTHER COMPLEX QUERIES:

The JOIN operation is used in the provided SQL code to combine data from two tables, events, and users, based on a related column. Specifically, the JOIN is used to fetch event details along with the username of the event host.

```
SELECT e.id, e.title, e.start_date, e.end_date, u.username AS host_username
FROM events e
JOIN users u ON e.host_user_id = u.user_id;
```

Figure 3.7

```
LEFT JOIN
    information_schema.KEY_COLUMN_USAGE ON COLUMNS.TABLE_SCHEMA = KEY_COLUMN_USAGE.TABLE_SCHEMA
    AND COLUMNS.TABLE_NAME = KEY_COLUMN_USAGE.TABLE_NAME
    AND COLUMNS.COLUMN_NAME = KEY_COLUMN_USAGE.COLUMN_NAME
LEFT JOIN
    information_schema.TABLE_CONSTRAINTS AS CONSTRAINTS ON KEY_COLUMN_USAGE.CONSTRAINT_NAME = CONSTRAINTS.CONSTRAINT_NAME
WHERE
    COLUMNS.TABLE_SCHEMA = 'event_management'
    AND COLUMNS.TABLE_NAME = 'events';
```

Figure 3.8

UNION ALL is used to combine the results of two queries into a single result set. Unlike the UNION operator, which removes duplicate rows from the result set, UNION ALL retains all rows, including duplicates.

```
UNION ALL
SELECT ej.event_id, NULL AS title, NULL AS category, NULL AS start_date, NULL AS end_date, u.username AS joiner_username, ej.joiner_user_id, ej.join_date, 'Join' AS entity_type
FROM event_joins ej
```

Figure 3.9

4. ANALYZING THE PITFALLS, IDENTIFYING THE DEPENDENCIES, AND APPLYING NORMALIZATIONS

PITFALLS:

- Redundancy in Storing User Information: Both the users table and the event_joins table store user information (username, email) redundantly.
- Inefficient Data Retrieval: The fetch_user_details procedure uses a cursor to fetch event details, which might be inefficient for large datasets.
- Lack of Data Consistency: There is a potential inconsistency issue with the display_status column in the events table. This column is updated based on the number of participants, but there is no mechanism to ensure consistency if records are directly manipulated in the event_joins table without using the provided triggers.

FUNCTIONAL DEPENDENCIES:

1. Users Table:
    - The user ID uniquely determines the user's username, email, and password.
    - Each email address uniquely determines the associated username and password.
    - Each username uniquely determines the associated email and password.
2. Events Table:
    - Each event ID uniquely determines all details of the event.
    - The host user ID uniquely determines all details of the event.
3. Event Joins Table:
    - Each join ID uniquely determines the event ID, joiner user ID, join date, and status.
    - Each event ID uniquely determines the joiner user ID, join date, and status.
    - Each joiner user ID uniquely determines the event ID, join date, and status.

NORMALIZATION:

Step 1: First Normal Form (1NF)

The code already seems to satisfy the requirements of 1NF as it represents tabular data with atomic values in each cell.

Step 2: Second Normal Form (2NF)

To achieve 2NF, we need to ensure that each non-key attribute is fully functionally dependent on the entire primary key.

Both users and events tables have a single primary key, and all non-key attributes are dependent on the entire primary key. Thus, they satisfy 2NF.

Step 3: Third Normal Form (3NF)

To achieve 3NF, we need to ensure that there are no transitive dependencies; that is, non-key attributes should not depend on other non-key attributes.

The events table contains several attributes such as city, address, share_address, max_participants, min_age, other_requirements, approximate_cost, and rules_guidelines that could be related to the event itself rather than the primary key. To resolve this, we can create separate tables for these attributes.

```
CREATE TABLE IF NOT EXISTS event_location (
    location_id INT AUTO_INCREMENT PRIMARY KEY,
    event_id INT NOT NULL,
    city VARCHAR(100),
    address VARCHAR(255),
    share_address BOOLEAN,
    FOREIGN KEY (event_id) REFERENCES events(event_id)
);
CREATE TABLE IF NOT EXISTS event_additional_details (
    event_id INT PRIMARY KEY,
    max_participants INT,
    min_age INT,
    other_requirements TEXT,
    approximate_cost DECIMAL(10,2),
    rules_guidelines TEXT,
    event_image VARCHAR(255),
    FOREIGN KEY (event_id) REFERENCES events(event_id)
);
```

5. IMPLEMENTATION OF CONCURRENCY CONTROL AND RECOVERY MECHANISMS

1. Concurrency Control:

- Locking Mechanisms: Implement locking mechanisms to control access to database resources and prevent conflicts between concurrent transactions.

- Transaction Isolation Levels: Set appropriate transaction isolation levels (e.g., Read Uncommitted, Read Committed, Repeatable Read, Serializable) to control the visibility of changes made by concurrent transactions.

- Optimistic Concurrency Control: Use techniques like optimistic concurrency control where transactions proceed without locking resources but are validated for conflicts before committing.

2. Recovery Mechanisms:

- Transaction Logging: Implement transaction logging to record changes made by transactions. In case of a system failure, the logged transactions can be replayed to recover the database to a consistent state.

- Checkpoints: Periodically create checkpoints where all modified data is flushed to disk. Checkpoints provide recovery points that can be used to restore the database to a consistent state in case of failure.

- Redo and Undo Logs: Maintain redo and undo logs to facilitate database recovery. Redo logs contain changes that were made but not yet saved to disk, while undo logs contain information to rollback transactions if needed.

3. Implementation Steps:

- Integrate database management system features for locking, transaction isolation, and logging into the code.

- Configure transaction isolation levels according to the application's requirements.

- Implement locking mechanisms to prevent conflicts between concurrent transactions.

6. CODE FOR THE PROJECT

DATABASE:

```
-- Create the database if it doesn't exist and switch to it
CREATE DATABASE IF NOT EXISTS event_management;
USE event_management;

-- Create the users table with constraints
CREATE TABLE IF NOT EXISTS users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
        password VARCHAR(255), -- Adding the password column
    CONSTRAINT uc_email UNIQUE (email) -- Constraint to ensure
uniqueness of email
);
ALTER TABLE users
ADD COLUMN password VARCHAR(255) NOT NULL;
SELECT * FROM users;
-- Create the events table with constraints
CREATE TABLE IF NOT EXISTS events (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    host_user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    category VARCHAR(50) NOT NULL,
    description TEXT,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    start_time TIME,
    city VARCHAR(100),
    address VARCHAR(255),
    share_address BOOLEAN,
    max_participants INT,
    min_age INT,
    other_requirements TEXT,
    approximate_cost DECIMAL(10,2),
    rules_guidelines TEXT,
    event_image VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (host_user_id) REFERENCES users(user_id)
);
ALTER TABLE events MODIFY COLUMN host_user_id INT NOT NULL
DEFAULT 0;
```

```
-- Add the 'upi' column to the 'events' table
-- Add a column for display status to the events table
ALTER TABLE events
ADD COLUMN display_status BOOLEAN NOT NULL DEFAULT TRUE;

SELECT * from events;

Drop TABLE if Exists event_joins;
-- Create the event_joins table with constraints
-- Create the event_joins table with constraints
CREATE TABLE IF NOT EXISTS event_joins (
    join_id INT AUTO_INCREMENT PRIMARY KEY,
    event_id INT NOT NULL,
    joiner_user_id INT NOT NULL,
    join_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('pending', 'accepted', 'rejected') DEFAULT 'pending', --
Added status column
    FOREIGN KEY (event_id) REFERENCES events(event_id),
    FOREIGN KEY (joiner_user_id) REFERENCES users(user_id)
);

DROP VIEW IF EXISTS events;

-- Create a trigger to update event count when a user joins an event
-- Drop the existing trigger if it exists
DROP TRIGGER IF EXISTS update_display_on_remove;

-- Create a trigger to update event display status when max participants are
reached
DELIMITER $$
CREATE TRIGGER update_event_display AFTER INSERT ON event_joins
FOR EACH ROW
BEGIN
    DECLARE current_participants INT;

    -- Get the current number of participants for the event
    SELECT COUNT(*) INTO current_participants
    FROM event_joins
    WHERE event_id = NEW.event_id;

    -- Update the display status based on the current number of participants
            IF current_participants >= (
                SELECT max_participants
                FROM events
```

```
        WHERE event_id = NEW.event_id
    ) THEN
        -- If max participants reached, set display status to false
        UPDATE events
        SET display_status = FALSE
        WHERE event_id = NEW.event_id;
    END IF;
END$$
CREATE TRIGGER update_display_on_remove AFTER DELETE ON
event_joins
FOR EACH ROW
BEGIN
    DECLARE current_participants INT;
    DECLARE max_part INT;
    SELECT COUNT(*) INTO current_participants FROM event_joins
WHERE event_id = OLD.event_id;
    SELECT max_participants INTO max_part FROM events WHERE
event_id = OLD.event_id;

    IF current_participants < max_part THEN
        UPDATE events SET display_status = TRUE WHERE event_id =
OLD.event_id;
        INSERT INTO trigger_logs (event_id, current_participants,
max_participants, action_taken) VALUES (OLD.event_id,
current_participants, max_part, 'Updated display_status to TRUE');
    ELSE
        INSERT INTO trigger_logs (event_id, current_participants,
max_participants, action_taken) VALUES (OLD.event_id,
current_participants, max_part, 'No update needed');
    END IF;
END$$
DELIMITER ;
CREATE TABLE IF NOT EXISTS trigger_logs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    event_id INT,
    current_participants INT,
    max_participants INT,
    action_taken VARCHAR(255),
    log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


SHOW Triggers;
-- Sample usage of cursor to fetch user details
```

```
-- Drop the existing stored procebdure if it exists
DROP PROCEDURE IF EXISTS fetch_user_details;

-- Create the updated fetch_user_details stored procedure
DELIMITER $$
CREATE PROCEDURE fetch_user_details()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE event_title VARCHAR(255);
    DECLARE event_start_date DATE;
    DECLARE event_end_date DATE;
    DECLARE cur CURSOR FOR SELECT title, start_date, end_date FROM
events;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done =
TRUE;

    OPEN cur;

    read_loop: LOOP
        FETCH cur INTO event_title, event_start_date, event_end_date;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SELECT CONCAT('Event Title: ', event_title, ', Start Date: ',
event_start_date, ', End Date: ', event_end_date);
    END LOOP;

    CLOSE cur;
END$$
DELIMITER ;

-- Call the stored procedure to fetch user details
CALL fetch_user_details();

-- Sample usage of join to fetch event details along with host username
SELECT e.id, e.title, e.start_date, e.end_date, u.username AS host_username
FROM events e
JOIN users u ON e.host_user_id = u.user_id;
SHOW TRIGGERS;
SHOW INDEX FROM events;


DROP VIEW IF EXISTS all_events_details;
```

```
CREATE VIEW all_events_details AS
SELECT e.event_id, e.title, e.category, e.start_date, e.end_date,
e.host_username, NULL AS joiner_username, NULL AS join_date, 'Event' AS
entity_type
FROM event_details e
UNION ALL
SELECT ej.event_id, NULL AS title, NULL AS category, NULL AS
start_date, NULL AS end_date, u.username AS joiner_username,
ej.joiner_user_id, ej.join_date, 'Join' AS entity_type
FROM event_joins ej
JOIN users u ON ej.joiner_user_id = u.user_id;

-- View the merged events with details
SELECT * FROM all_events_details;
SELECT DISTINCT
    COLUMNS.COLUMN_NAME AS column_name,
    COLUMNS.DATA_TYPE AS data_type,
    COLUMNS.CHARACTER_MAXIMUM_LENGTH AS max_length,
    COLUMNS.IS_NULLABLE AS nullable,
    CONSTRAINTS.CONSTRAINT_NAME AS constraint_name,
    CONSTRAINTS.CONSTRAINT_TYPE AS constraint_type
FROM
    information_schema.COLUMNS
LEFT JOIN
    information_schema.KEY_COLUMN_USAGE ON
COLUMNS.TABLE_SCHEMA =
KEY_COLUMN_USAGE.TABLE_SCHEMA
    AND COLUMNS.TABLE_NAME =
KEY_COLUMN_USAGE.TABLE_NAME
    AND COLUMNS.COLUMN_NAME =
KEY_COLUMN_USAGE.COLUMN_NAME
LEFT JOIN
    information_schema.TABLE_CONSTRAINTS AS CONSTRAINTS ON
KEY_COLUMN_USAGE.CONSTRAINT_NAME =
CONSTRAINTS.CONSTRAINT_NAME
WHERE
    COLUMNS.TABLE_SCHEMA = 'event_management'
    AND COLUMNS.TABLE_NAME = 'events';

SHOW CREATE TABLE events;
SELECT * FROM event_joins;
SELECT @@hostname AS 'Host', USER() AS 'Current User', DATABASE()
AS 'Current Database';
SHOW DATABASES LIKE 'event_management';
```

```
SELECT user, host, plugin FROM mysql.user WHERE user = 'root';
ALTER USER 'root'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'ashutosh';
DROP VIEW IF EXISTS event_details;

CREATE VIEW event_details AS
SELECT
    e.event_id,
    e.title,
    e.category,
    e.start_date,
    e.end_date,
    u.username AS host_username,
    GROUP_CONCAT(u_joiner.username) AS joiner_usernames,
    GROUP_CONCAT(u_joiner.email) AS joiner_emails, -- Adding the joiner
emails
    GROUP_CONCAT(u_joiner.user_id) AS joiner_ids -- Adding the joiner
IDs
FROM
    events e
JOIN
    users u ON e.host_user_id = u.user_id
LEFT JOIN
    event_joins ej ON e.event_id = ej.event_id
LEFT JOIN
    users u_joiner ON ej.joiner_user_id = u_joiner.user_id
GROUP BY
    e.event_id,
    e.title,
    e.category,
    e.start_date,
    e.end_date,
    u.username;
    select * from event_details;


    select * from event_joins;
```

FRONTEND:

index.html

```
<script>
  function redirectToPage() {
    // Redirect to the host an event page URL
    window.location.href = 'http://localhost:3000/host.html?'; // Change the URL to your actual host event page URL
  }
  // Fetch events from the server
  fetch('http://localhost:3000/events')
    .then(response => response.json())
    .then(data => {
      const eventsList = document.getElementById('eventsList');
      data.forEach(event => {
        const eventBox = document.createElement('div');
        eventBox.classList.add('event-box');
        const imageUrl = `path/to/images/${encodeURIComponent(event.event_image)}`;

        eventBox.innerHTML = `
          <div class="event-details">
            <img src="${imageUrl}" alt="Event Image" style="max-width: 100%; height: auto;">
            <h2 style="margin-bottom: 5px;">${event.title}</h2>
            <p><strong>Category:</strong> ${event.category}</p>
            <p><strong>Description:</strong> ${event.description}</p>
            <p><strong>Date:</strong> ${new Date(event.start_date).toDateString()}</p>
            <p><strong>Time:</strong> ${event.start_time}</p>
            <p><strong>Location:</strong> ${event.city}, ${event.address}</p>
            <p><strong>Max Participants:</strong> ${event.max_participants}</p>
            <p><strong>Min Age:</strong> ${event.min_age}</p>
            <p><strong>Cost:</strong> $${event.approximate_cost}</p>
          </div>
        `;
        // Check if the event status is pending or not registered, show the "Request Registration" button
        if (!event.status || event.status === 'pending') {
          eventBox.innerHTML += `
            <button class="request-registration" onclick="requestRegistration(${event.event_id})">Request Registration</button>
          `;
        }
```

```
      eventsList.appendChild(eventBox);
    });
  })
  .catch(error => console.error('Error fetching events:', error));

// Function to request registration for an event
function requestRegistration(eventId) {
  // Extract userId from the URL or replace it with your actual userId retrieval method
  const userId = getUserId();

  // Send a POST request to register for the event
  fetch(`http://localhost:3000/event/${eventId}/register`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        userId: userId
      })
    })
    .then(response => {
      if (response.ok) {
        // Registration successful, show alert
        alert("You have successfully registered for the event!");
      } else if (response.status === 400) {
        // User is already registered, show alert
        alert("You are already registered for this event.");
      } else {
        // Registration failed, show error message
        alert("Failed to register for the event. Please try again later.");
      }
    })
    .catch(error => {
      console.error('Error registering for event:', error);
      alert("An error occurred while registering for the event. Please try again later.");
    });
}

// Function to extract userId from URL query params or replace it with your actual userId retrieval method
```

```
  function getUserId() {
    const urlParams = new URLSearchParams(window.location.search);
    return urlParams.get('userId');
  }

</script>
```

host.html

```
<script>
    document.addEventListener('DOMContentLoaded', function() {
        const eventForm = document.getElementById('eventForm');
        const userId = getUserId(); // Call getUserId() to get the user ID

        if (userId) {
            const formAction = '/submit-form?userId=' + userId;
            eventForm.action = formAction; // Set the form action with user ID
        }
    });

    function getUserId() {
        const urlParams = new URLSearchParams(window.location.search);
        const userId = urlParams.get('userId');
        console.log(userId + " ")
        return userId;
    }
</script>
```

host_selection.html

```
<script>
    // Function to get query parameter value from URL
    function getParameterByName(name, url) {
        if (!url) url = window.location.href;
        name = name.replace(/[\[\]]/g, '\\$&');
        var regex = new RegExp('[?&]' + name + '(=([^&#]*)|&|#|$)'),
            results = regex.exec(url);
        if (!results) return null;
```

```
        if (!results[2]) return '';

        return decodeURIComponent(results[2].replace(/\+/g, ' '));

    }

    // Get userId from the URL query parameters

    var userId = getParameterByName('userId');

    // Set the userId value to the hidden input field

    document.getElementById("userIdInput").value = userId;
</script>
```

front.html

```
<script>
    function showCreateUsernameForm() {
        var createUsernameForm = document.getElementById('create-username-form');
        var loginForm = document.getElementById('login-form');


        if (createUsernameForm.style.display === 'none') {
            createUsernameForm.style.display = 'block';
            loginForm.style.display = 'none';
        } else {
            createUsernameForm.style.display = 'none';
            loginForm.style.display = 'block';
        }
    }

    function showLoginForm() {
        var createUsernameForm = document.getElementById('create-username-form');
        var loginForm = document.getElementById('login-form');


        if (loginForm.style.display === 'none') {
            loginForm.style.display = 'block';
            createUsernameForm.style.display = 'none';
        } else {
            loginForm.style.display = 'none';
            createUsernameForm.style.display = 'block';
        }
    }
</script>
```

dashboard.html

```
<script>
```

```
// Function to display hosted events
function displayHostedEvents(hostedEvents) {
    const hostedEventsList = document.getElementById('hosted-events-list');
    hostedEvents.forEach(event => {
        const listItem = document.createElement('li');
        let removeEventButton = ''; // Define removeEventButton here

        let joinersHTML = '';

        let acceptAllButton = ''; // Define acceptAllButton here

        // Check if there are joiners
        if (event.joiner_usernames && event.joiner_emails && event.joiner_ids) {
            const joinerUsernames = event.joiner_usernames.split(',');
            const joinerEmails = event.joiner_emails.split(',');
            const joinerIds = event.joiner_ids.split(',');
            // Check if there are more than one joiners
            const multipleJoiners = joinerUsernames.length > 1;

            if (multipleJoiners) {
                acceptAllButton = `<button onclick="acceptAllJoiners(${event.event_id})">Accept All Joiners</button>`;
            }
            for (let i = 0; i < joinerUsernames.length; i++) {
                joinersHTML += `
                    <div>
                        <p><strong>ID:</strong> ${joinerIds[i]}</p>
                        <p><strong>Name:</strong> ${joinerUsernames[i]}</p>
                        <p><strong>Email:</strong> ${joinerEmails[i]}</p>
                        <button onclick="acceptJoiner(${event.event_id}, ${joinerIds[i]})">Accept</button>
                        <button onclick="removeJoiner(${event.event_id}, ${joinerIds[i]})">Remove</button>
                    </div>
                `;
            }
        }
```

```
        removeEventButton = `<button onclick="removeEvent(${event.event_id})">Remove Event</button>`;

        // Create HTML for event details including the accept all button if applicable
        listItem.innerHTML = `
            <strong>${event.title}</strong>
            <p>${event.description}</p>
            <p>Date: ${event.start_date} to ${event.end_date}</p>
            ${removeEventButton} <!-- Display the Remove Event button -->
            ${joinersHTML}
            ${acceptAllButton} <!-- Display the Accept All button only if there are multiple joiners -->
        `;

        // Append the list item to the hosted events list
        hostedEventsList.appendChild(listItem);
    });
}
// Function to remove an event
function removeEvent(eventId) {
    // Perform the logic to remove the event with the given eventId
    // Assuming you have an endpoint for removing events
    fetch('/remove-event', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            eventId: eventId
        })
    })
    .then(response => {
        if (response.ok) {
            // Event removed successfully
            console.log(`Event with ID ${eventId} removed successfully`);
            // Optionally, you can update the UI or take any additional actions
```

```
        } else {
            // Error occurred while removing event
            console.error(`Failed to remove event with ID ${eventId}`);
        }
    })
    .catch(error => {
        console.error('Error:', error);
    });
}
// Function to handle accepting all joiners for an event
function acceptAllJoiners(eventId) {
    fetch('/accept-all-joiners', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ eventId })
    })
    .then(response => {
        if (response.ok) {
            // Reload the page or update UI as needed
            location.reload(); // Reload the page to reflect the changes
        } else {
            console.error('Failed to accept all joiners');
        }
    })
    .catch(error => {
        console.error('Error accepting all joiners:', error);
    });
}
// Function to handle accepting a joiner
function acceptJoiner(eventId, joinerId) {
    fetch('/accept-joiner', {
        method: 'POST',
```

```
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ eventId, joinerId })
    })
    .then(response => {
        if (response.ok) {
            // Update UI to show only remove option
            const acceptButton = document.getElementById(`accept-button-${eventId}-${joinerId}`);
            const removeButton = document.getElementById(`remove-button-${eventId}-${joinerId}`);
            if (acceptButton && removeButton) {
                acceptButton.style.display = 'none'; // Hide accept button
                removeButton.style.display = 'inline-block'; // Show remove button
            }
            // Reload the page to reflect the changes
            location.reload();
        } else {
            console.error('Failed to accept joiner');
        }
    })
    .catch(error => {
        console.error('Error accepting joiner:', error);
    });
}
// Add event listener to accept joiner buttons
document.addEventListener('DOMContentLoaded', () => {
    const acceptButtons = document.querySelectorAll('.accept-button');
    acceptButtons.forEach(button => {
        button.addEventListener('click', () => {
            const eventId = button.dataset.eventId;
            const joinerId = button.dataset.joinerId;
            acceptJoiner(eventId, joinerId);
        });
    });
```

```
    });
</script>
```

Server.js

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

const cors = require('cors');

const mysql = require('mysql');

const path = require('path');


const app = express();

app.set('view engine', 'ejs');

const port = 3000;

app.use(express.static(path.join(__dirname, 'public')));

app.get('/', (req, res) => {

  res.sendFile(path.join(__dirname, 'front.html'));

});

app.use(cors());

let connection;

try {

  connection = mysql.createConnection({

    host: '127.0.0.1',

    user: 'root',

    password: 'ashutosh',

    database: 'event_management',

  });

  connection.connect((err) => {

    if (err) {

      console.error('Error connecting to MySQL: ', err);

      return;

    }

    console.log('Connected to MySQL');

  });

} catch (err) {

  console.error('Error establishing MySQL connection: ', err);

  process.exit(1);
```

```
}
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
app.use(express.static('public'));
app.post('/register', (req, res) => {
  const { name, email, password } = req.body;
  const query = 'INSERT INTO users (username, email, password) VALUES (?, ?, ?)';
  connection.query(query, [name, email, password], (err, results) => {
    if (err) {
      console.error('Error inserting data into MySQL: ', err);
      res.status(500).send('Internal Server Error');
      return;
    }
    console.log('User data inserted into MySQL');
    res.status(200).send('User registered successfully');
  });
});
app.post('/login', (req, res) => {
  const { email, password } = req.body;
  const query = 'SELECT * FROM users WHERE email = ? AND password = ?';
  connection.query(query, [email, password], (err, results) => {
    if (err) {
      console.error('Error querying database: ', err);
      res.status(500).send('Internal Server Error');
      return;
    }
    if (results.length === 0) {
      res.status(401).send('Invalid email or password');
      return;
    }
    const userId = results[0].user_id;
    res.redirect(`/host_selection.html?userId=${userId}`);
  });
});
```

```
app.post('/select_page', (req, res) => {
  const selectedPage = req.body.page;
  const userId = req.body.userId;

  if (selectedPage === 'dashboard') {
    res.redirect(`/dashboard.html?userId=${userId}`);
  } else if (selectedPage === 'index') {
    res.redirect(`/index.html?userId=${userId}`);
  } else if (selectedPage === 'host') {
    res.redirect(`/host.html?userId=${userId}`);
  } else {
    res.status(400).send('Invalid selection');
  }
});
app.get('/host.html', (req, res) => {
  const userId = req.query.userId;
  res.sendFile(path.join(__dirname, 'host.html'));
});
app.get('/index.html', (req, res) => {
  const userId = req.query.userId;
  res.sendFile(path.join(__dirname, 'index.html'));
});
app.get('/host_selection.html', (req, res) => {
  const userId = req.query.userId;
  res.sendFile(path.join(__dirname, 'host_selection.html'));
});
app.get('/dashboard.html', (req, res) => {
  const userId = req.query.userId;
  res.sendFile(path.join(__dirname, 'dashboard.html'));
});
app.post('/submit-form', (req, res) => {
  const formData = req.body;
  const userId = req.query.userId;
  const query = 'INSERT INTO events (host_user_id, title, category, description, start_date, end_date, start_time, city,
address, share_address, max_participants, min_age, other_requirements, approximate_cost, rules_guidelines, event_image)
```

```
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)';
  connection.query(query, [
    userId,
    formData['event-title'],
    formData.category,
    formData.desc,
    formData['start-d'],
    formData['end-d'],
    formData['start-t'],
    formData.city,
    formData.add,
    formData.showadd === 'on' ? 1 : 0,
    formData.quantity,
    formData.age,
    formData['other-req'],
    formData.cost,
    formData.rules,
    formData['event-img'],
  ], (err, results) => {
    if (err) {
      console.error('Error inserting data into MySQL: ', err);
      res.status(500).send('Internal Server Error');
      return;
    }
    console.log('Form data inserted into MySQL');
    res.status(200).send('Form data submitted successfully');
  });
});
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

7. RESULT AND DISCUSSION

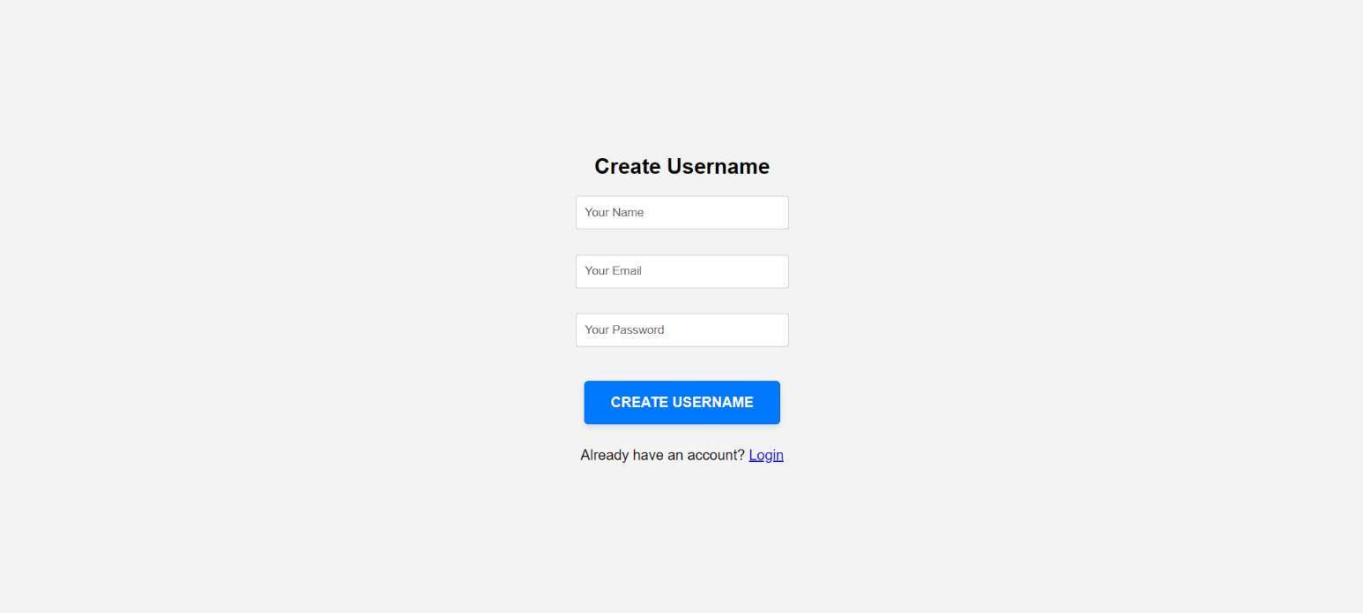

Figure 7.1

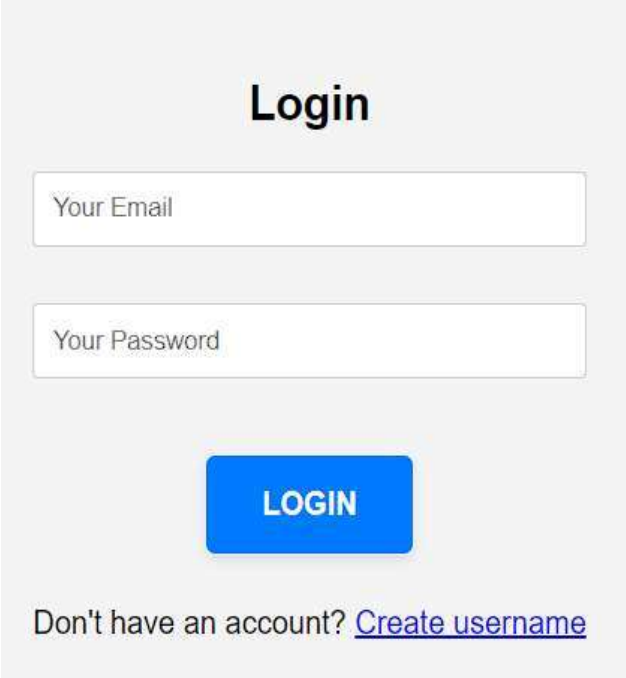

Figure 7.2

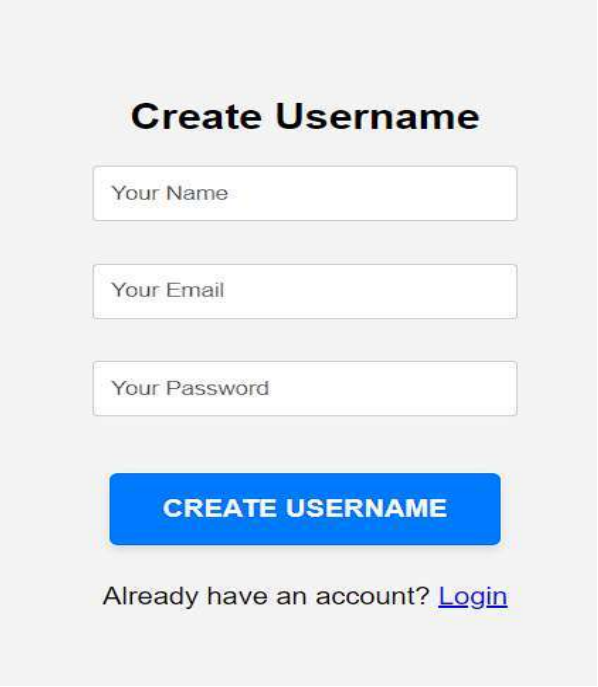

Figure 7.3

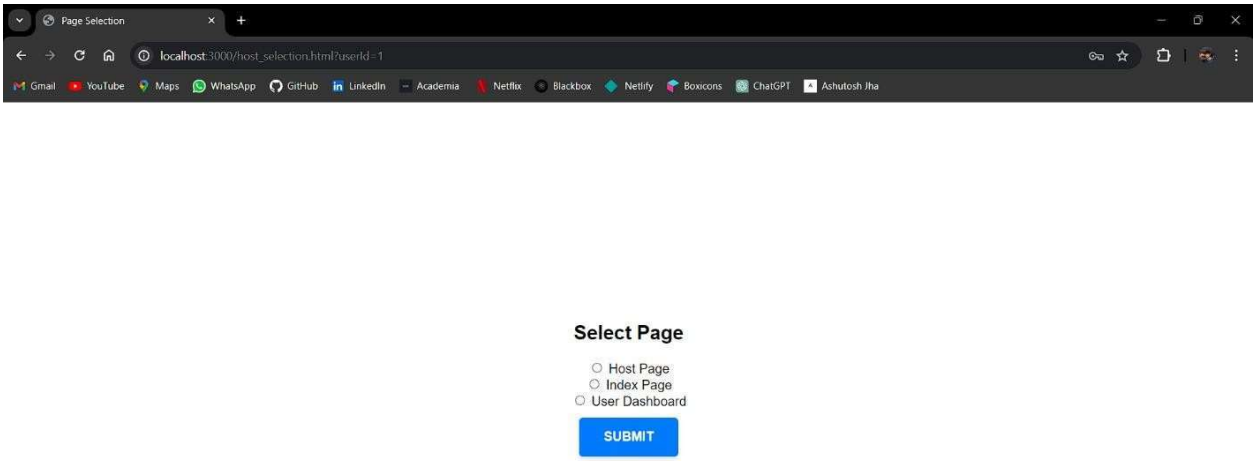

Figure 7.4

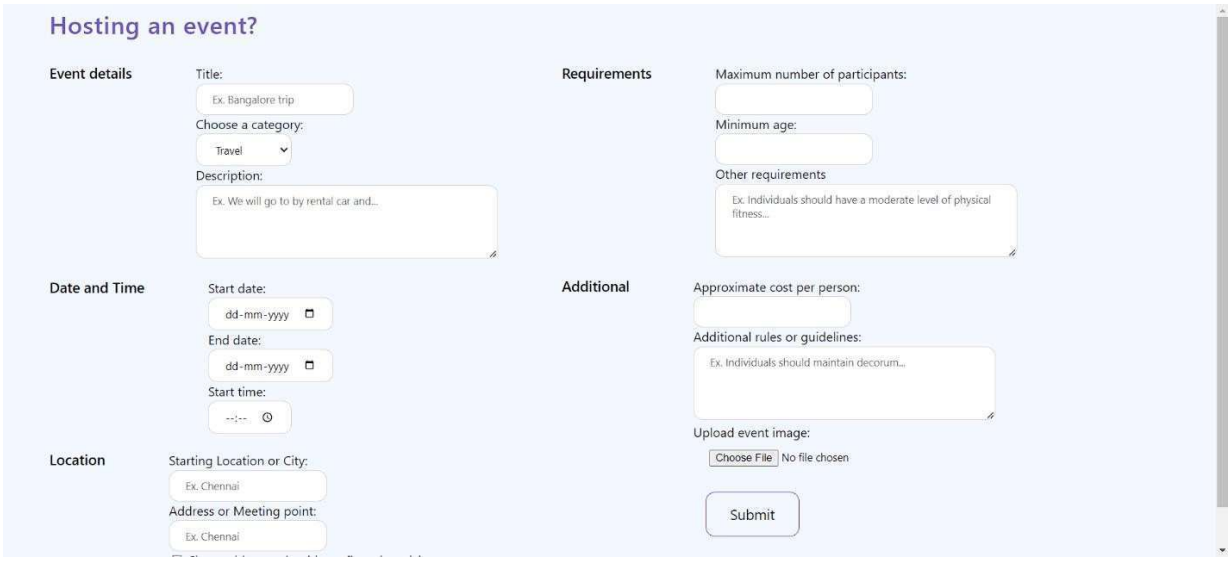

Figure 7.5

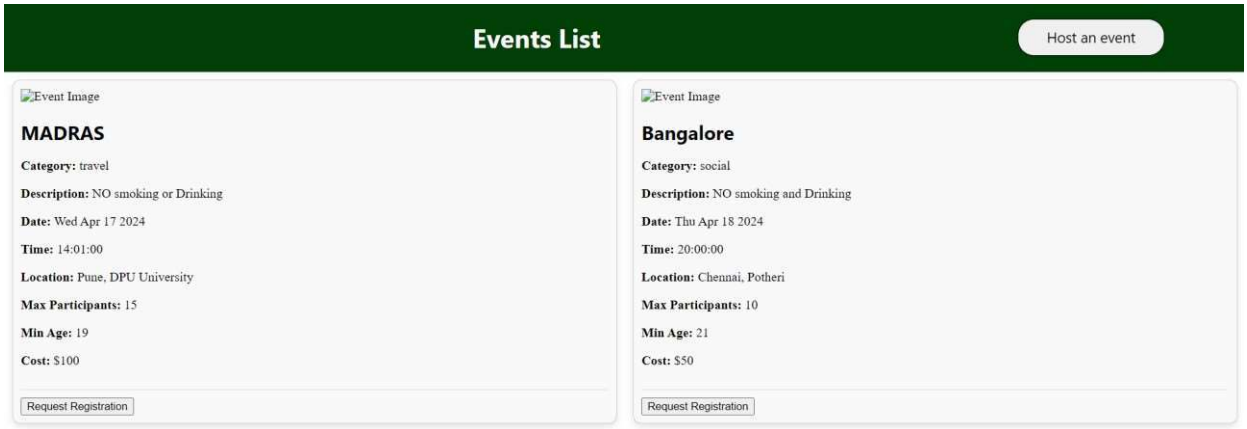

Figure 7.6

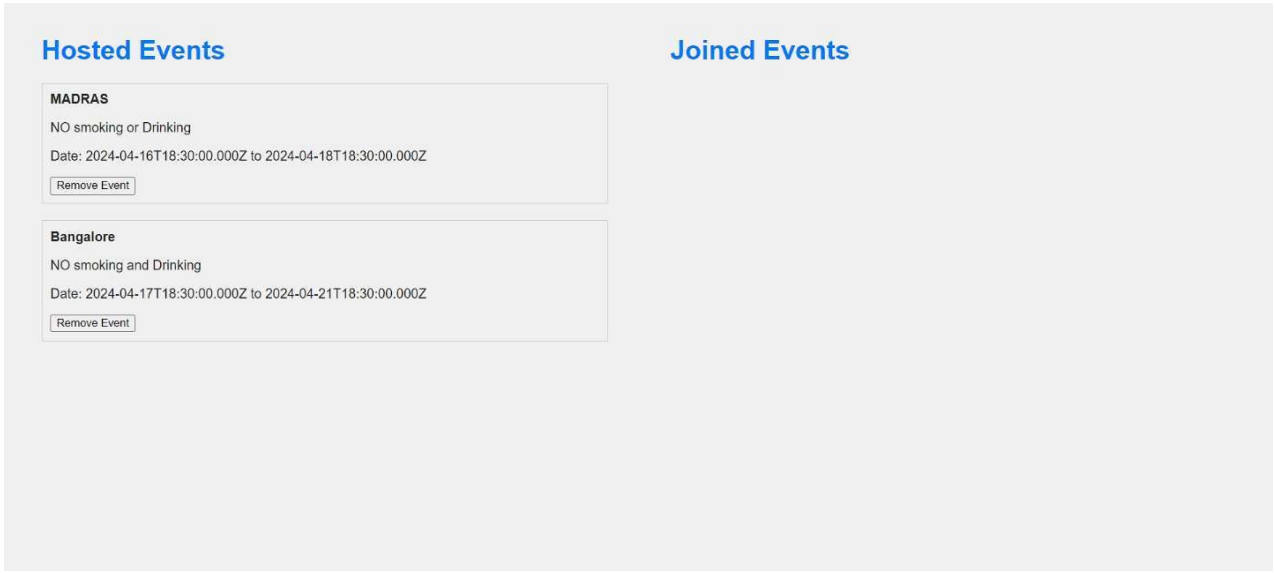

Figure 7.7

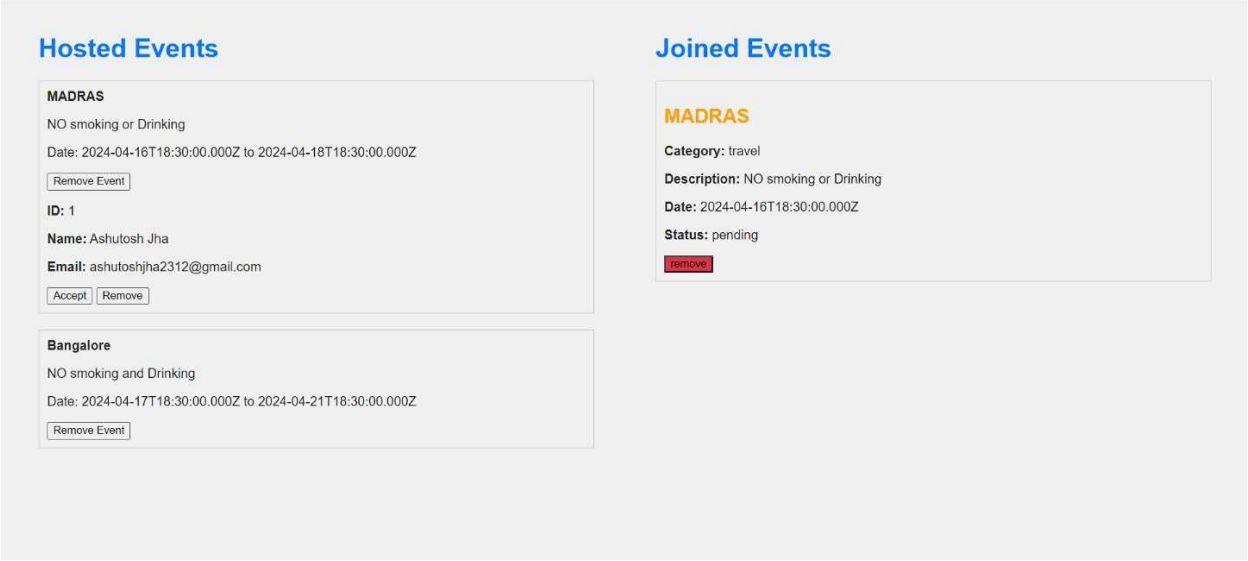

Figure 7.8

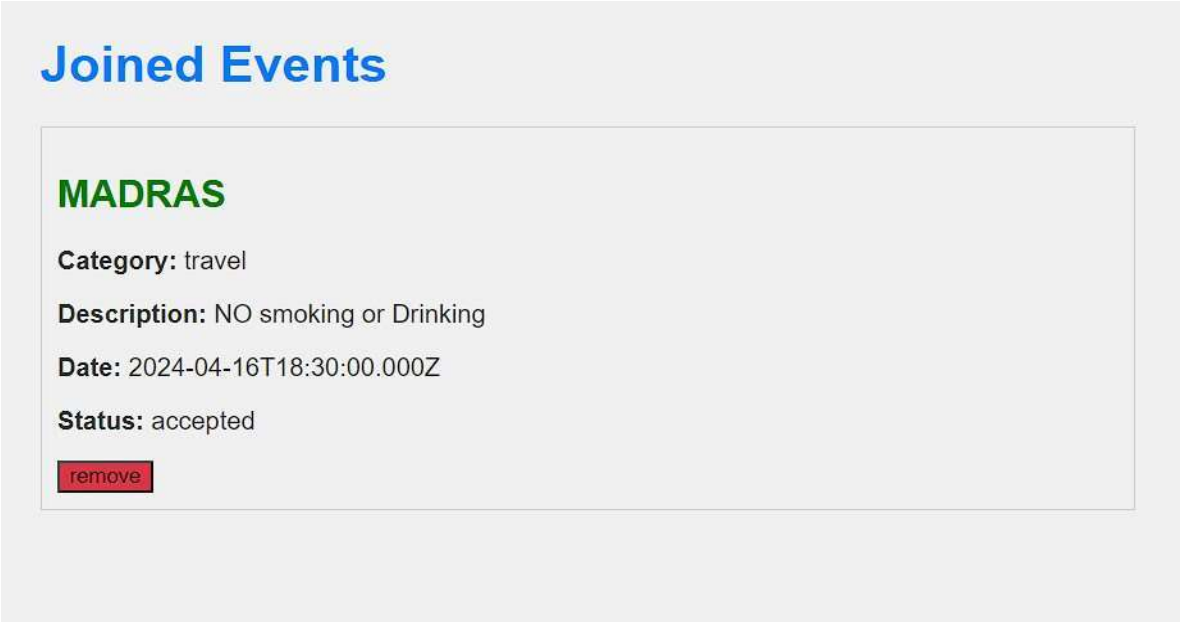

Figure 7.9

DISCUSSION

The Event Management System showcased here underscores the importance of concurrency control and recovery mechanisms in maintaining system reliability. Concurrency control ensures data consistency when multiple users interact with the system simultaneously, preventing conflicts and ensuring accurate updates. Meanwhile, robust recovery mechanisms safeguard against unexpected system failures, allowing for swift restoration of data integrity and system functionality. By incorporating these measures, the system can uphold reliability and user satisfaction even in the face of challenging operational scenarios.

Furthermore, the implementation of triggers, stored procedures, and views in the database schema facilitates efficient data management and retrieval. Triggers automate actions based on predefined conditions, such as updating event display status when maximum participants are reached. Stored procedures streamline complex database operations, enhancing performance and maintainability. Views provide simplified access to consolidated data, allowing for seamless integration into application logic. Together, these database features optimize system functionality and contribute to a smooth user experience in the Event Management System.