



# CS 353 Database Systems

## Design Report

### Local Events Application

**Tuna Öğüt - 21803492**

**Erengazi Mutlu - 21803676**

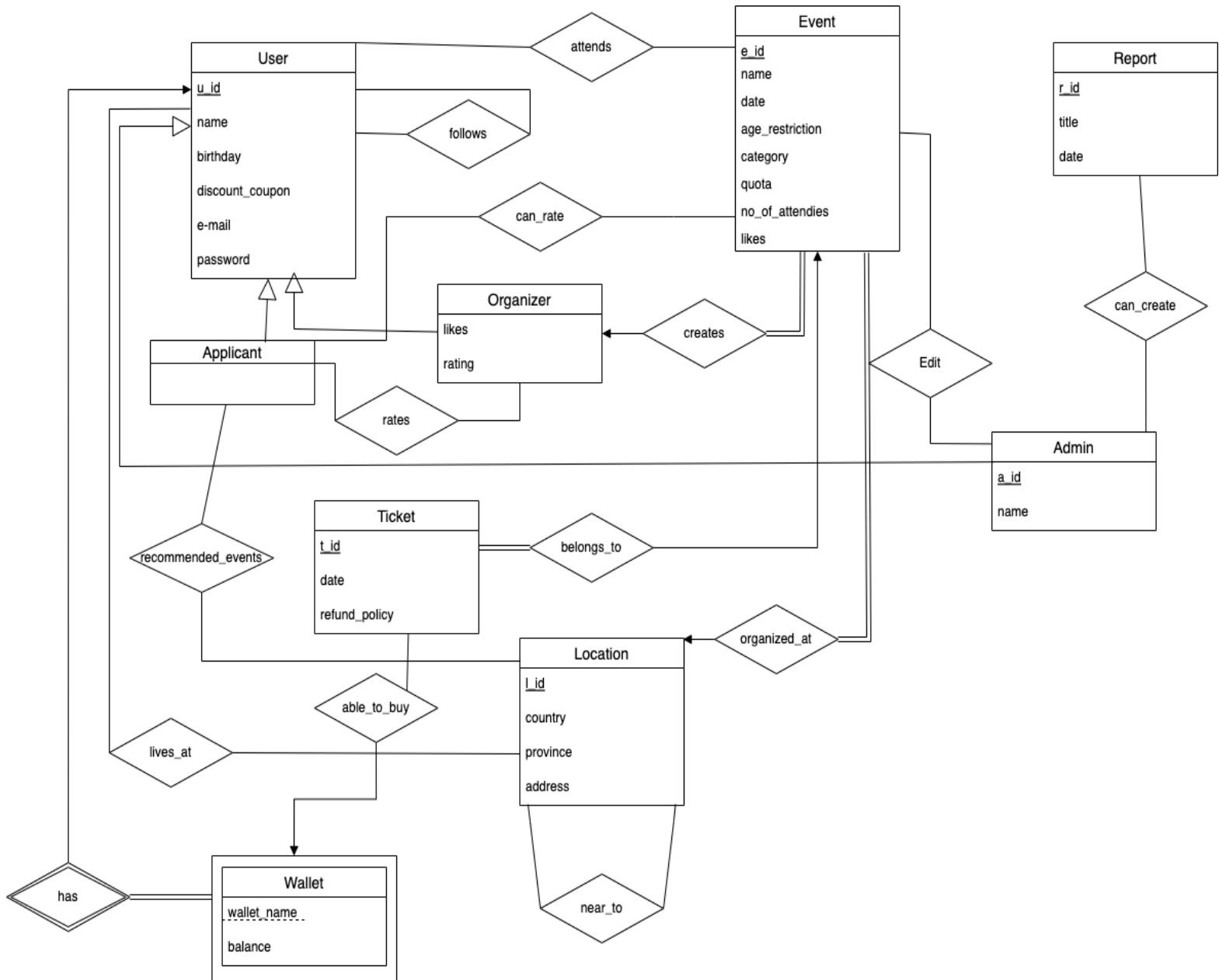
**Yiğit E. Dinç - 21704275**

**Arda Güven Çiftçi - 21801763**

## **Table Of Contents**

<b>1. Revised ER Diagram</b>	<b>3</b>
<b>2. Table Schemas</b>	<b>4</b>
<b>3. Functional Components</b>	<b>20</b>
3.1 Use case model	20
3.2 Use Case Scenarios	20
<b>4. UI and SQL Statements</b>	<b>29</b>
4.1 Wireframe & Representative SQL Queries	29
4.1.1 Login Screen	29
4.1.2 Sign Up Screen	30
4.1.3 Profile Screen	32
4.1.4 Event Options Screen	34
4.1.5 Friendlist Screen	35
4.1.6 Notifications Screen	36
4.1.7 Event Creation Screen	37
4.1.8 Event Join Screen	38
4.1.9 User Details	39
4.1.10 Event Details Page	40
4.1.11 Event Stats Page	41
<b>5. Advanced Database Components</b>	<b>42</b>
5.1 Reports	42
5.1.1 Probable Table View Reports	42
5.1.2 Probable Relation View Reports	42
5.1.3 Probable Complex View Reports	42
5.2 Views	43
5.3 Triggers	43
5.3.1 Probable Triggers	43
5.4 Constraints	44
5.4.1 Probable Constraints	44
5.5 Stored Procedures	45
5.5.1 Probable Procedures	45
<b>6. Implementation Details</b>	<b>46</b>
<b>7. Website</b>	<b>46</b>

# 1. Revised ER Diagram



## 2. Table Schemas

### 2.1 User

Relational Model

user(u-id, name, birthday, discount-coupon, e-mail, hashed-pw)

Functional Dependencies

u-id -> name birthday discount-coupon e-mail hashed-pw

Candidate Keys

{{u-id}}

Normal Form

3NF

Table Definition

```
CREATE TABLE user (  
    u-id UNIQUE NOT NULL INT AUTO_INCREMENT,  
    name NOT NULL VARCHAR(25),  
    birthday NOT NULL DATE,  
    discount-coupon VARCHAR(25),  
    e-mail NOT NULL VARCHAR(25),  
    hashed-pw NOT NULL VARCHAR(256),  
    PRIMARY KEY(u-id));
```

### 2.2 Event

Relational Model

event(e-id, name, date, age-res, category, quota, nof-attendies, likes)

Functional Dependencies

e-id -> name date age-res category quota nof-attendies likes

Candidate Keys

{(e-id)}

Normal Form

3NF

Table Definition

```
CREATE TABLE event(  
    e-id UNIQUE NOT NULL AUTO_INCREMENT,  
    name NOT NULL VARCHAR(35),  
    date NOT NULL DATE,  
    age-res BIT,  
    category VARCHAR(120),  
    quota INT,  
    nof-attendies INT,  
    likes INT  
    PRIMARY KEY (e-id));
```

## 2.3 Report

Relational Model

report(r-id, title, date)

Functional Dependencies

r-id -> title date

Candidate Keys

{{r-id}}

Normal Form

3NF

Table Definition

```
CREATE TABLE report(
    r-id UNIQUE NOT NULL INT AUTO_INCREMENT,
    title VARCHAR(70),
    date DATE,
    PRIMARY KEY (r-id));
```

## 2.4 Applicant

Relational Model

applicant(u-id)

FK: u-id REFERENCES user

Functional Dependencies

u-id ->

Candidate Keys

{{u-id}}

Normal Form

3NF

Table Definition

```
CREATE TABLE applicant(
    u-id UNIQUE NOT NULL INT AUTO_INCREMENT,
    FOREIGN KEY(u-id) REFERENCES user);
```

## 2.5 Organizer

### Relational Model

organizer(u-id, likes, rating)

### Functional Dependencies

u-id -> likes rating

### Candidate Keys

{(u-id)}

### Normal Form

3NF

### Table Definition

```
CREATE TABLE organizer(  
    u-id UNIQUE NOT NULL INT AUTO_INCREMENT,  
    likes INT,  
    rating INT,  
    FOREIGN KEY(u-id) REFERENCES user);
```

## 2.6 Admin

### Relational Model

admin(u-id)

FK: u-id REFERENCES user

### Functional Dependencies

u-id ->

### Candidate Keys

{(u-id)}

### Normal Form

3NF

### Table Definition

```
CREATE TABLE admin(
    u-id UNIQUE NOT NULL INT AUTO_INCREMENT,
    FOREIGN KEY(u-id) REFERENCES user);
```

## 2.7 Ticket

Relational Model

ticket(t-id, date, refund-policy)

Functional Dependencies

t-id -> date refund-policy

Candidate Keys

{(t-id)}

Normal Form

3NF

Table Definition

```
CREATE TABLE ticket(
    t-id UNIQUE NOT NULL INT AUTO_INCREMENT,
    date DATE,
    refund-policy TINYINT,
    PRIMARY KEY(t-id));
```

## 2.8 Location

Relational Model

location(l-id, country, province, address)

Functional Dependencies

l-id -> country province address

Candidate Keys



{{l-id}}

Normal Form

3NF

Table Definition

```
CREATE TABLE location(
    l-id UNIQUE NOT NULL INT AUTO_INCREMENT,
    countrY CHAR(2),
    province VARCHAR(35),
    address VARCHAR(70),
    PRIMARY KEY(l-id));
```

## 2.9 Wallet

Relational Model

wallet(w-id, balance)

Functional Dependencies

w-id -> balance

Candidate Keys

{{w-id}}

Normal Form

3NF

Table Definition

```
CREATE TABLE wallet(
    w-id UNIQUE NOT NULL INT,
    balance int,
    PRIMARY KEY(w-id));
```

## 2.10 Attends

### Relational Model

attends(u-id,e-id)

FK: u-id REFERENCES user

FK: e-id REFERENCES event

### Functional Dependencies

u-id e-id ->

### Candidate Keys

{(u-id),(e-id)}

### Normal Form

3NF

### Table Definition

```
CREATE TABLE attends(  
    u-id INT NOT NULL,  
    e-id INT NOT NULL  
    PRIMARY KEY(u-id, e-id),  
    FOREIGN KEY(u-id) REFERENCES user,  
    FOREIGN KEY(e-id) REFERENCES event);
```

## 2.11 Follows

### Relational Model

follows(follower, following)

### Functional Dependencies

-> follower following

### Candidate Keys

{()}

Normal Form

3NF

Table Definition

```
CREATE TABLE follows(
    follower int,
    follows int);
```

## 2.12 ae-rates

Relational Model

ae-creates(u-id,e-id, rate)

FK: u-id REFERENCES user

FK: e-id REFERENCES event

Functional Dependencies

u-id e-id -> rate

Candidate Keys

{(u-id),(e-id)}

Normal Form

3NF

Table Definition

```
CREATE TABLE ae-rates(
    u-id INT,
    e-id INT,
    rate INT,
    FOREIGN KEY(u-id) REFERENCES user,
    FOREIGN KEY(e-id) REFERENCES event);
```

### 2.13 ao-rates

Relational Model

ao-rates(rater, rated, rate)

Functional Dependencies

-> rater rated rate

Candidate Keys

{()}

Normal Form

3NF

Table Definition

```
CREATE TABLE ao-rates(
    rater INT,
    rates INT,
    rate INT);
```

### 2.14 ar-creates

Relational Model

ar-creates(u-id,r-id)

FK: u-id REFERENCES user

FK: r-id REFERENCES report

Functional Dependencies

u-id r-id ->

Candidate Keys

{(u-id),(r-id)}

Normal Form

3NF

### Table Definition

```
CREATE TABLE ar-creates(
    u-id INT NOT NULL,
    r-id INT NOT NULL,
    FOREIGN KEY (u-id) REFERENCES user,
    FOREIGN KEY (r-id) REFERENCES report);
```

## 2.15 Edits

### Relational Model

edits(u-id,e-id)

FK: u-id REFERENCES user

FK: e-id REFERENCES event

### Functional Dependencies

-> u-id e-id

### Candidate Keys

{(u-id),(e-id)}

### Normal Form

3NF

### Table Definition

```
CREATE TABLE edits(
    u-id INT NOT NULL,
    e-id INT NOT NULL,
    FOREIGN KEY(u-id) REFERENCES user,
    FOREIGN KEY(r-id) REFERENCES report);
```

## 2.16 oe-crates

### Relational Model

eo-crates(u-id,e-id)

FK: u-id REFERENCES user

FK: e-id REFERENCES event

### Functional Dependencies

u-id e-id ->

### Candidate Keys

{(u-id),(e-id)}

### Normal Form

3NF

### Table Definition

```
CREATE TABLE oe-creates(
    u-id INT NOT NULL,
    e-id INT NOT NULL,
    FOREIGN KEY(u-id) REFERENCES user,
    FOREIGN KEY(e-id) REFERENCES event);
```

## 2.17 r-events

### Relational Model

r-events(u-id, e-id)

### Functional Dependencies

u-id e-id ->

### Candidate Keys

{(u-id),(e-id)}

### Normal Form

3NF

## Table Definition

```
CREATE TABLE r-events(
    u-id INT NOT NULL,
    e-id INT NOT NULL,
    FOREIGN KEY(u-id) REFERENCES user,
    FOREIGN KEY(e-id) REFERENCES event);
```

## 2.18 Able-To-Buy

## Relational Model

able2buy(w-id,t-id)

FK: w-id REFERENCES wallet

FK: t-id REFERENCES ticket

## Functional Dependencies

w-id t-id ->

## Candidate Keys

{(w-id),(t-id)}

## Normal Form

3NF

## Table Definition

```
CREATE TABLE able2buy(
    w-id INT NOT NULL,
    t-id INT NOT NULL,
    FOREIGN KEY(w-id) REFERENCES wallet,
    FOREIGN KEY(t-id) REFERENCES ticket);
```

## 2.19 Belongs

### Relational Model

belongs(t-id,e-id)

FK: t-id REFERENCES ticket

FK: e-id REFERENCES event

### Functional Dependencies

t-id e-id ->

### Candidate Keys

{{t-id),(e-id}}

### Normal Form

3NF

### Table Definition

```
CREATE TABLE belongs(
    t-id NOT NULL INT,
    e-id NOT NULL INT,
    FOREIGN KEY(t-id) REFERENCES ticket,
    FOREIGN KEY(e-id) REFERENCES event);
```

## 2.20 Organized-At

### Relational Model

organized-at(e-id,l-id)

### Functional Dependencies

e-id l-id ->

### Candidate Keys

{{e-id),(l-id}}

### Normal Form



3NF

Table Definition

```
CREATE TABLE organized-at(
    e-id NOT NULL INT,
    t-id NOT NULL INT,
    FOREIGN KEY (e-id) REFERENCES event,
    FOREIGN KEY (t-id) REFERENCES ticket);
```

## 2.21 Lives-At

Relational Model

lives-at (u-id,l-id)

FK: u-id REFERENCES user

FK: l-id REFERENCES location

Functional Dependencies

u-id l-id ->

Candidate Keys

{{u-id),(l-id}}

Normal Form

3NF

Table Definition

```
CREATE TABLE lives-at(
    u-id INT NOT NULL,
    l-id INT NOT NULL,
    FOREIGN KEY(u-id) REFERENCES user,
    FOREIGN KEY(l-id) REFERENCES location);
```

## 2.22 Has

Relational Model

has(u-id,w-id)

FK: u-id REFERENCES user

FK w-id REFERENCES wallet

Functional Dependencies

u-id w-id ->

Candidate Keys

{(u-id),(w-id)}

Normal Form

3NF

Table Definition

```
CREATE TABLE has(
    u-id NOT NULL INT,
    w-id NOT NULL INT,
    FOREIGN KEY u-id REFERENCES user,
    FOREIGN KEY w-is REFERENCES wallet);
```

## 2.23 Near-To

Relational Model

near-to(loca, locb)

Functional Dependencies

-> loca locb

Candidate Keys

{()}

Normal Form

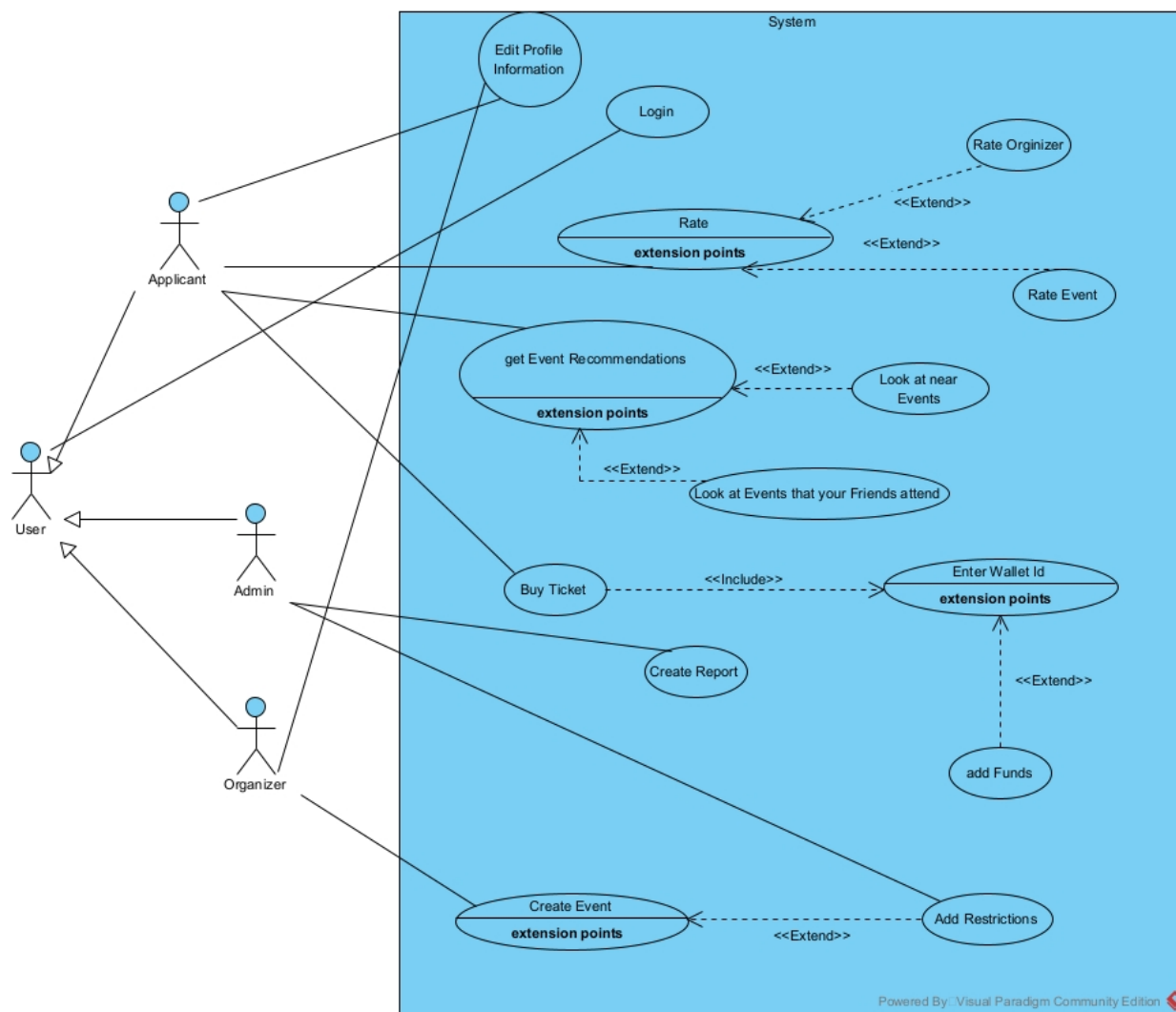
3NF

Table Definition

```
CREATE TABLE near-to(  
    loca INT NOT NULL,  
    locb INT NOT NULL);
```

### 3. Functional Components

#### 3.1 Use case model



#### 3.2 Use Case Scenarios

**Use Case Name:** Log In

**Participating Actor:** User

<b>Entry Condition:</b>	User has an account
<b>Flow Of Events:</b>	-User enters login information
<b>Exit Condition:</b>	User enters incorrect login info.
<b>Post Condition:</b>	User is logged in

<b>Use Case Name:</b>	<b>Edit profile Information</b>
<b>Participating Actor:</b>	Organizer, Applicant
<b>Entry Condition:</b>	User must have an account User must be logged in into his/her account
<b>Flow Of Events:</b>	-User enters new information -User clicks change info button
<b>Exit Condition:</b>	-
<b>Post Condition:</b>	User info is updated

**Use Case Name: Rate**

**Participating Actor:** Applicant

**Entry Condition:**

- User must be applicant
- User must log in to his/her account
- User must participate in that specific

**Flow Of Events:** -User opens up event history

- Rates the specific event

**Exit Condition:** User hits the close button

**Post Condition:** Event/Organizer rate updated

**Use Case Name: Rate Organizer**

**Participating Actor:** Applicant

**Entry Condition:** User must be applicant

- User must log in to his/her account
- User must participate in that specific event

**Flow Of Events:** -User opens up event history

- Clicks on the organizer
- Rates the organizer

**Exit Condition:** User hits the close button

**Post Condition:** Organizer rate updated

**Use Case Name: Rate Event**

**Participating Actor:** Applicant

**Entry Condition:** User must be applicant

- User must log in to his/her account

User must participate in that specific event

**Flow Of Events:** -User opens up event history

-Clicks on the event

- Rates the event

**Exit Condition:** User hits the close button

**Post Condition:** Event rate updated

**Use Case Name:**

**Create Event**

**Participating Actor:**

Organizer,

**Entry Condition:**

-User must have an account

-User must have an organizer account

-User must be logged in into his/her account

**Flow Of Events:**

-User enters all of the required event information

**Exit Condition:**

Clicks on either create event button or exit button

**Post Condition:**

Event is created

<b>Use Case Name:</b>	<b>Add Restrictions</b>
<b>Participating Actor:</b>	Organizer, Admin
<b>Entry Condition:</b>	<ul style="list-style-type: none"><li>-User must have an account</li><li>-User must have an organizer account or admin account</li><li>-User must be logged in into his/her account</li><li>-There must be a created event beforehand</li></ul>
<b>Flow Of Events:</b>	<ul style="list-style-type: none"><li>-User selects the event that he/she wants to edit.</li><li>-User enters restrictions</li></ul>
<b>Exit Condition:</b>	Clicks on either done button or exit button
<b>Post Condition:</b>	Restrictions added



<b>Use Case Name:</b>	<b>Buy tickets</b>
<b>Participating Actor:</b>	Applicant
<b>Entry Condition:</b>	<ul style="list-style-type: none"><li>-User must have an account</li><li>-User must have an applicant</li><li>-User must be logged in into his/her account</li><li>-There must be a created event beforehand</li><li>-User profile information must satisfy the rules of the event</li></ul>
<b>Flow Of Events:</b>	<ul style="list-style-type: none"><li>-User selects the event that he/she wants to buy the tickets for.</li><li>-If not added previously user have to enter wallet info</li><li>-User clicks to buy the ticket for the selected event</li><li>-If there are enough credits user buys the ticket</li></ul>
<b>Exit Condition:</b>	Clicks on either buy button or exit button

**Post Condition:** Ticket is purchased

**Use Case Name: Get Event Recommendations**

**Participating Actor:** Applicant

**Entry Condition:** -User must have an account

-User must have an applicant

-User must be logged in into his/her account

**Flow Of Events:** User clicks on the add wallet button

User enters wallet information

If the information is correct wallet is added

**Exit Condition:** User hits the close button or done button

**Post Condition:** wallet and event information updated.

**Use Case Name: Look at near events**

**Participating Actor:** Applicant

**Entry Condition:** -User must have an account

-User must have an applicant

-User must be logged in into his/her account

**Flow Of Events:** User clicks on the add wallet button

User enters

**Exit Condition:** User hits the close button

**Post Condition:** Event information updated.

**Use Case Name:**

**Participating Actor:** Applicant

**Entry Condition:** -User must have an account

-User must have an applicant

-User must be logged in into his/her account

**Flow Of Events:** User clicks on the add wallet button

User enters

**Exit Condition:** User hits the close button

**Post Condition:** Event information updated.

**Use Case Name: Look at Events that your Friends attend**

**Participating Actor:** Applicant

**Entry Condition:** -User must have an account

-User must have an applicant

-User must be logged in into his/her account

-User must have friends

**Flow Of Events:** User clicks on events button

User clicks on events that my friends attend button

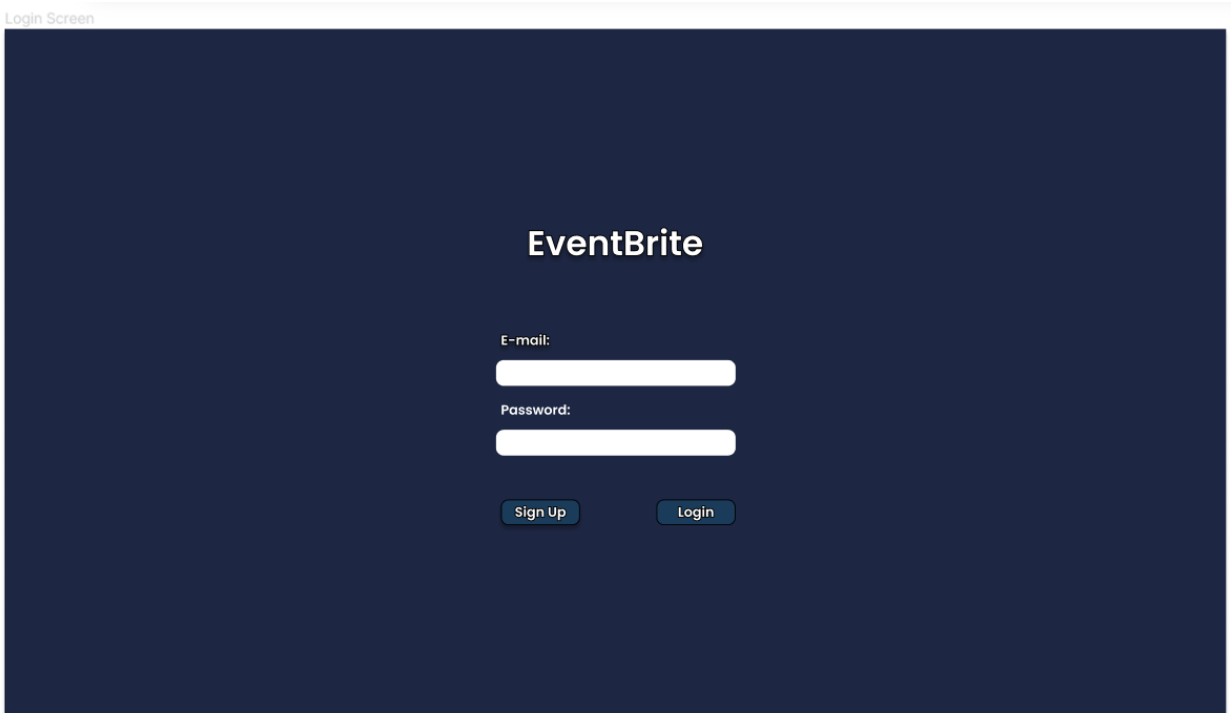
**Exit Condition:** User hits the close button

**Post Condition:** Event information monitored

## 4. UI and SQL Statements

### 4.1 Wireframe & Representative SQL Queries

#### 4.1.1 Login Screen



In the login screen it is required to check if the email exists in the database and then to compare its hashed password to the given password's hash. If they have the same value, login is correct else prompt an error message. Following are the representative of the SQL queries related to this issue and are planned to implement possibly.

To check if the email is at database

```
SELECT * FROM user WHERE email = @email;
```

If this returns at least one element (it is an unsaid constraint every email account can have one and only one account.) then we compare the returning view's password column to the

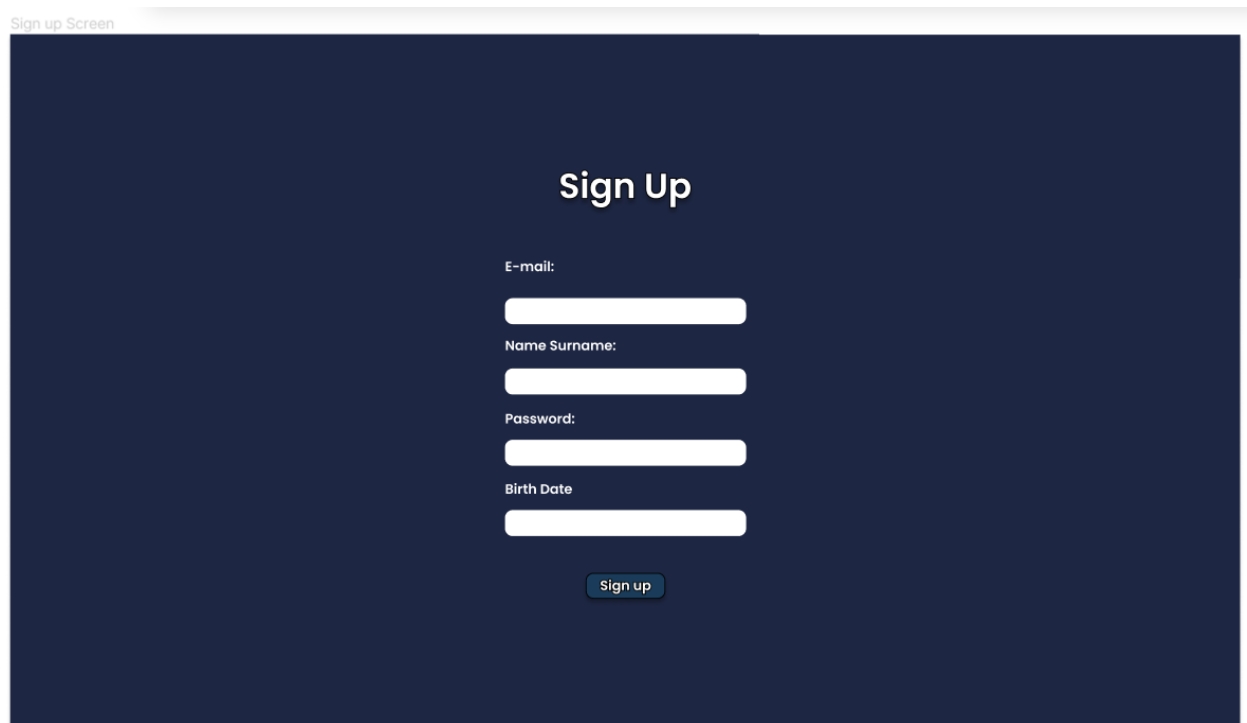
given @password value. Not to forget that after obtaining the password value from the user, the system is going to convert it to the hashed version of it. Thus only hashed passwords are kept and compared. This whole process can be handled in a single query as follows

```
SELECT * FROM user WHERE email = @email AND password =  
@password;
```

However for safety measures we will follow the first methodology.

#### 4.1.2 Sign Up Screen

Sign up Screen



The sign-up screen features a dark blue background with white text and input fields. The title 'Sign Up' is centered at the top. Below it, the labels 'E-mail:', 'Name Surname:', 'Password:', and 'Birth Date' are aligned to the left of their respective white input fields. A 'Sign up' button is centered at the bottom of the form area.

**Sign Up**

E-mail:

Name Surname:

Password:

Birth Date

Account creation is made at this screen. Thus we are going to make sure that the user provided the necessary information. There exist 2 rules, the system is going to store the passwords in hashed form and an email address can only be given to one and only one account. Thus, converting password to hash is handled by the system itself and the existing email condition is handled before the account creation. Such that, when user presses the button it fires two SQL queries in the following order as follows

To control the email address exists in the system

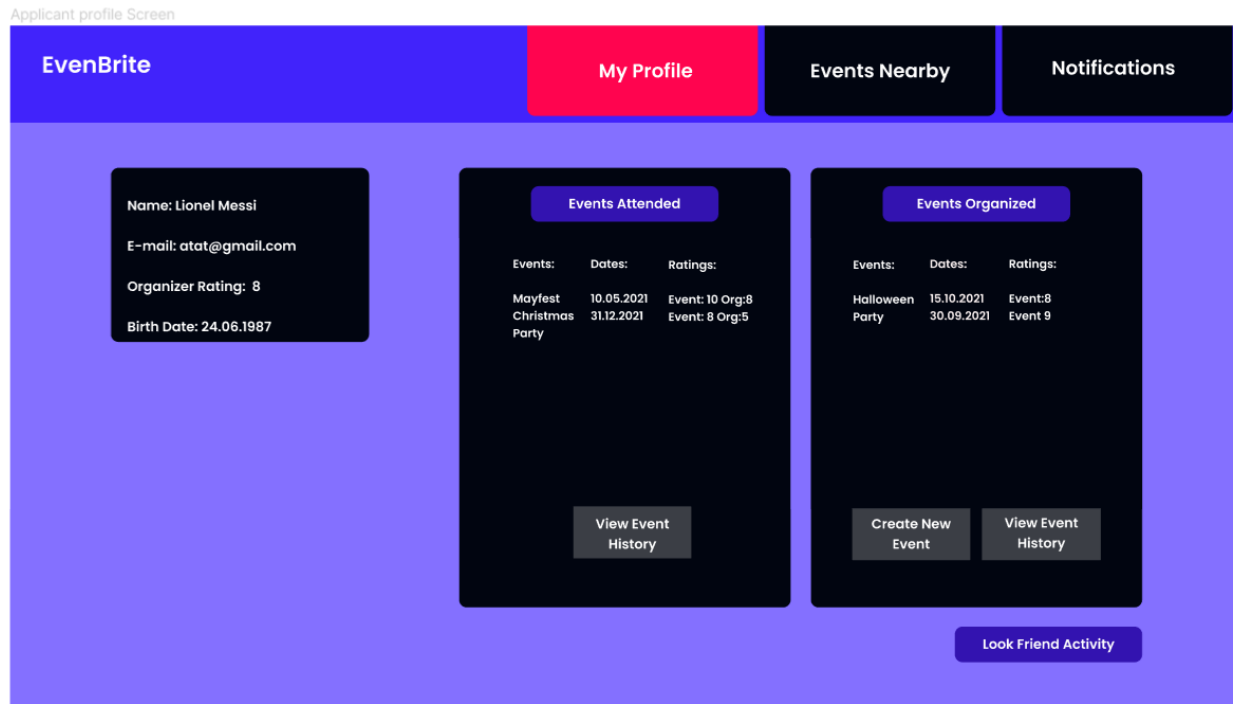
```
SELECT * FROM user WHERE email = @email;
```

If this query returns 0 rows then we can create the account. Take care that before sending the query there exists a case and format checking for the input email. That is to make sure that the given email address is a valid email address.

Then it is either prompting an error or firing the following query

```
INSERT INTO user (e-mail, name, password, bdate) VALUES (@email,  
@name, @password, @bdate);
```

### 4.1.3 Profile Screen



Left hand side black box has main values returned from the following SQL query (here there exists a catch such that after logging in the system keeps track of the u-id and adds it a session number, to create a session id. This session is passed through the pages for that user as long as they stay logged in)

```
SELECT name, email, birthdate, FROM user WHERE u-id=@u-id;
```

To obtain the organizer score, we call the ao-rate with the u-id given as organizer-id. Then an aggregate function to find the average is called. Hence we got the average value of the organizer score for that u-id.

```
SELECT AVG(rate) as OrganizerScore FROM ao-rate WHERE rated=@u-id;
```

In the middle section the user is able to see the attended events that are stored at the database. Attends relation is called.



```
SELECT e-id FROM attends WHERE attends.u-id=@u-id;
```

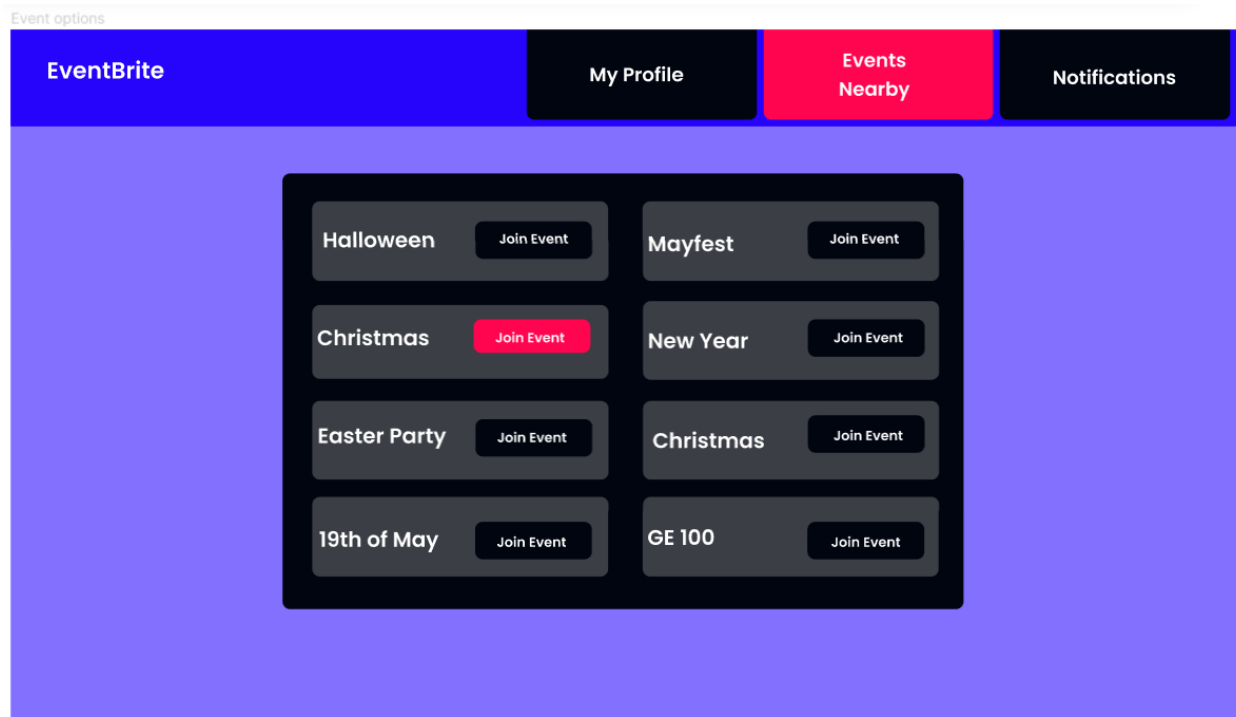
This will return the e-ids. Then with those e-ids we are looping through them to get the required information to be presented.

In the right hand side of the screen, the user is able to see the organized events. This follows the same structure as in the previous one.

```
SELECT e-id FROM ae-create WHERE ae-create.u-id=@u-id;
```

This will return the e-ids. Then with those e-ids we are looping through them to get the required information to be presented.

#### 4.1.4 Event Options Screen



This is where users can choose from the activity categories. This is handled by a form submission. That is in the end we can send SQL queries as follows to filter the events.

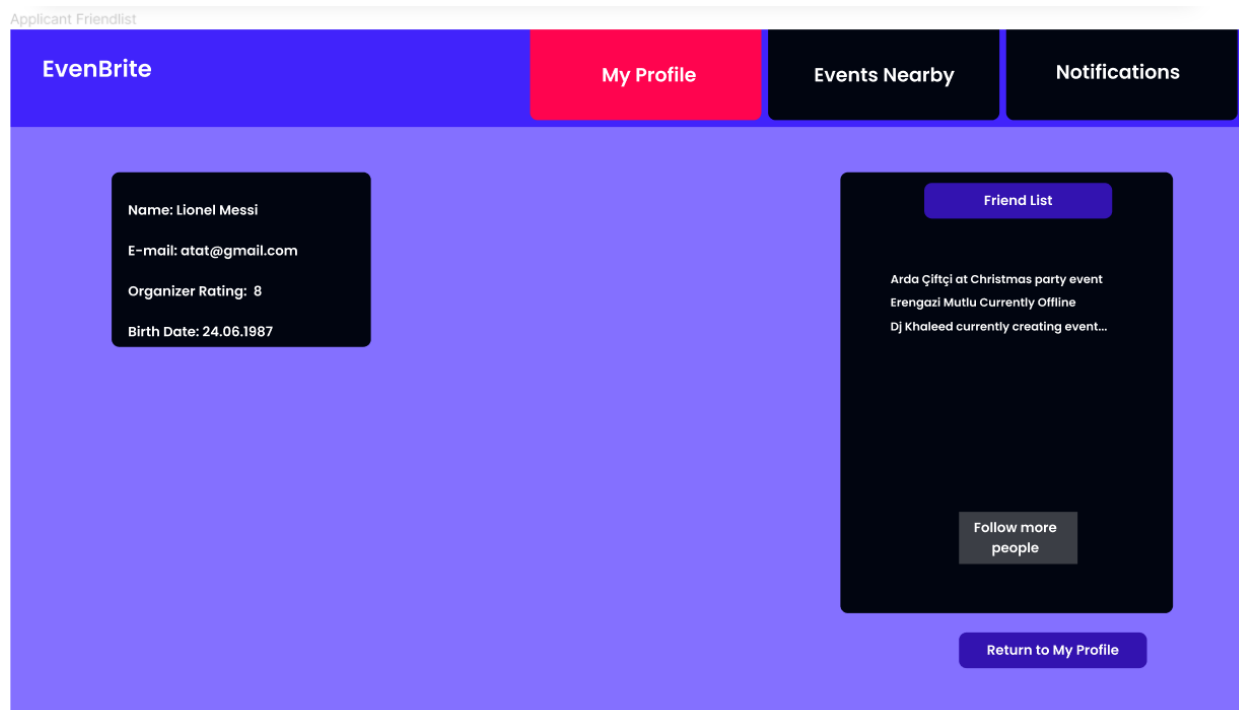
```
SELECT * FROM event WHERE category=@input;
```

After that, users can join the event. Which add a new line to the attends relation as in the following format

```
INSERT INTO attends (u-id, e-id) VALUES(u-id, e-id);
```

The values u-id is sliced from the session id and the e-id is hidden in the join button.

#### 4.1.5 Friendlist Screen



Although it is called friendlist screen it represents the same principles for a followers list and a following list. If a follower is being followed then they are called friends. This is to increase the overall interaction with users though, it can only bring small to zero more interaction. The following are the SQL queries that the system would call while getting the data required.

```
SELECT follower FROM follows WHERE follower=@u-id;
```

The @u-id represents the user. This query is going to call the users that are followed by this user.

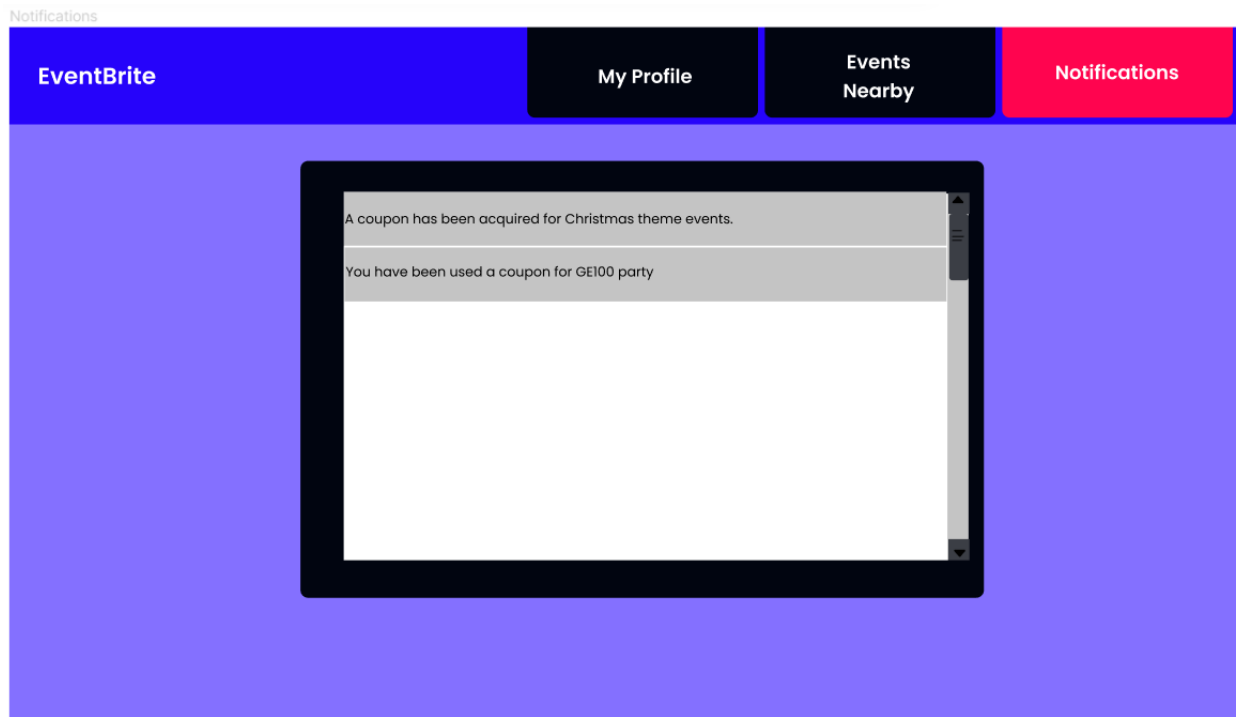
```
SELECT following FROM follows WHERE follower=@u-id;
```

The @u-id represents the user. This query is going to call the users that follow this user. In the end their intersection is the friends list. The notifications about friends are obtained by looping through friends' u-ids and their attends tables.

```
SELECT * FROM attends WHERE u-id=@u-id;
```

Where @u-id is a representative of every single friend in the user's friend list.

#### 4.1.6 Notifications Screen



This part follows the same exact part for the previous screens friend notifications box. However, this includes self notifications that are sent through the system. Those system notifications are handled by the system itself and presented on the coupon-coded column at the user table. However, this reach of the system is not constructed very well yet. Thus, quite prone to changing. Additionally, regardless of this tendency on the change, the following (or somewhat similar code) is going to be executed for the user.

To get the user the events that they participated in.

```
SELECT e-id FROM attends WHERE u-id=@u-id;
```

#### 4.1.7 Event Creation Screen

Event Creation

EventBrite
My Profile
Events Nearby
Notifications

Event Name: New Year Party
Capacity: 250

Event Rule: 1  
Minimum age of participants must be 20

Event Rule: 2  
Food or Drinks from outside of event will not be taken inside

Event Rule 3  
People should bring present due to concept of party

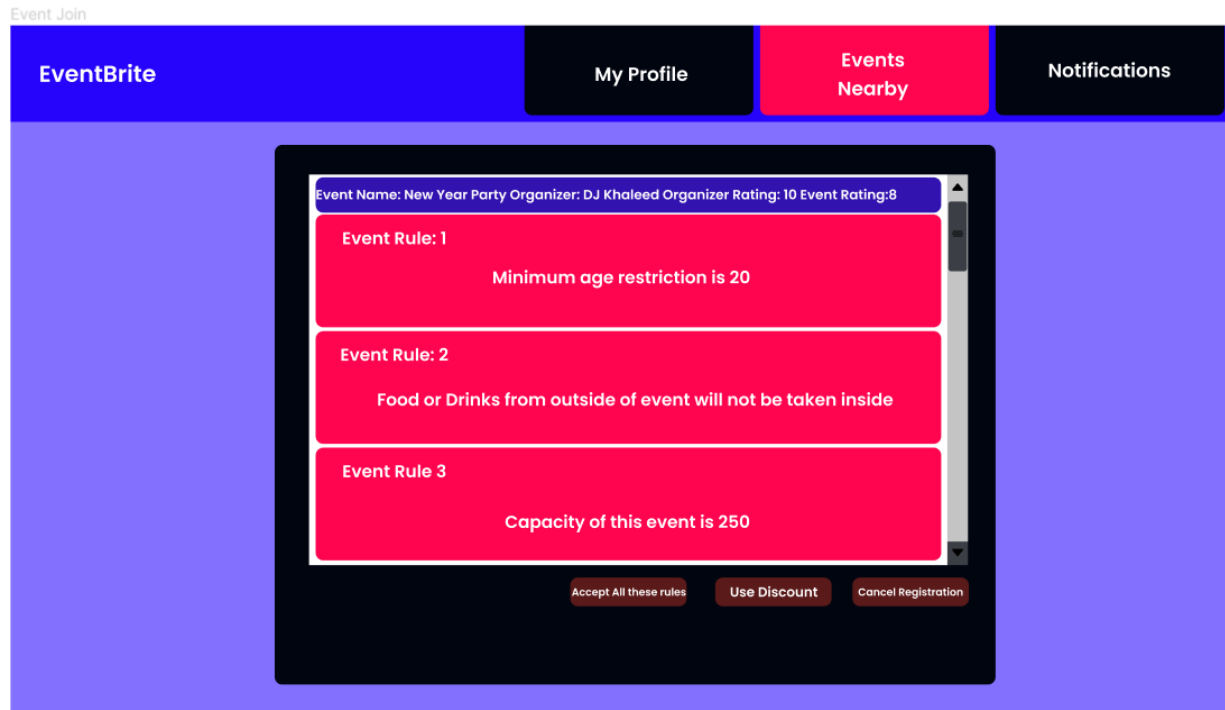
Finish Creating Event
Delete rule
Add New Rule

This page is an equivalent of the registration page for the events. Thus a SQL query to insert the new values is required.

```
INSERT INTO event(name, date, age-restriction, category, quota, nof-attendies, likes)
VALUES(@name, @date, @age-restriction, @category, @quota, @nof-attendies, @likes);
```

No control for duplicate events is provided.

#### 4.1.8 Event Join Screen



This is where the user joins itself to the event chosen. When a user chooses an event then e-id is also added to the session id to pass between the correct data. Main SQL dependent requirements are for this page as follows.

To get the event rules and details

```
SELECT * FROM event WHERE e-id=@e-id;
```

When user presses the accept button event's quota is checked through the following SQL queries

```
SELECT quota FROM event WHERE e-id=@e-id;
```

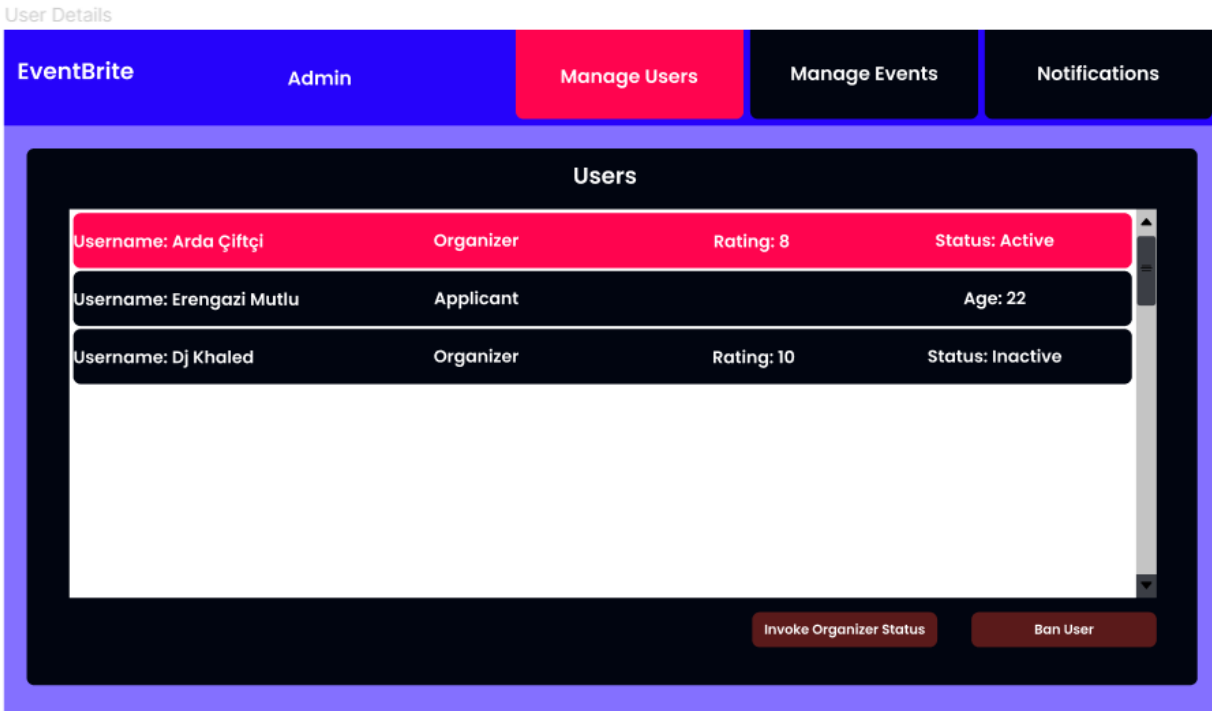
If there exist empty places then the user is going to be signed in. Else an error message is going to be prompted.

To sign in the user

```
INSERT INTO attends(u-id, e-id) VALUES(@u-id, @e-id);
```

When this code is fired, the system should increment the nof-attendies and decrement the quota value at the event table.

#### 4.1.9 User Details

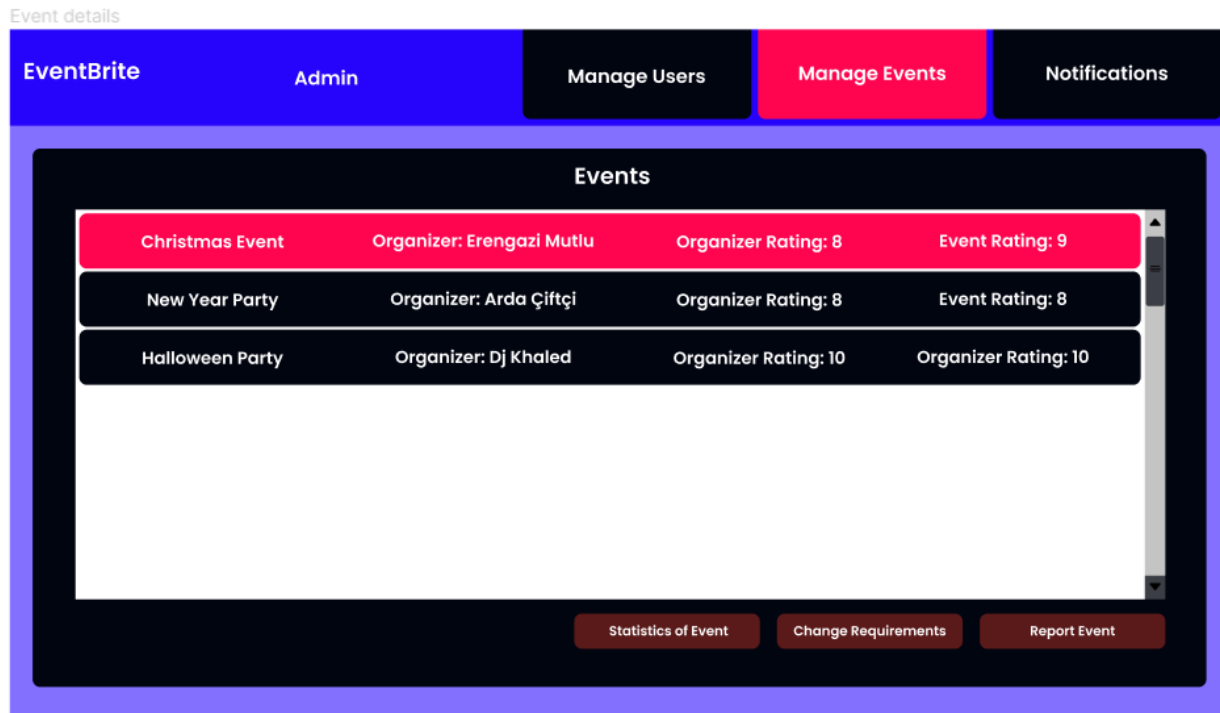


User details page shows the all users name and their ratings. This is done by following SQL queries

```
SELECT name FROM user;
```

```
SELECT rating FROM organizer;
```

#### 4.1.10 Event Details Page



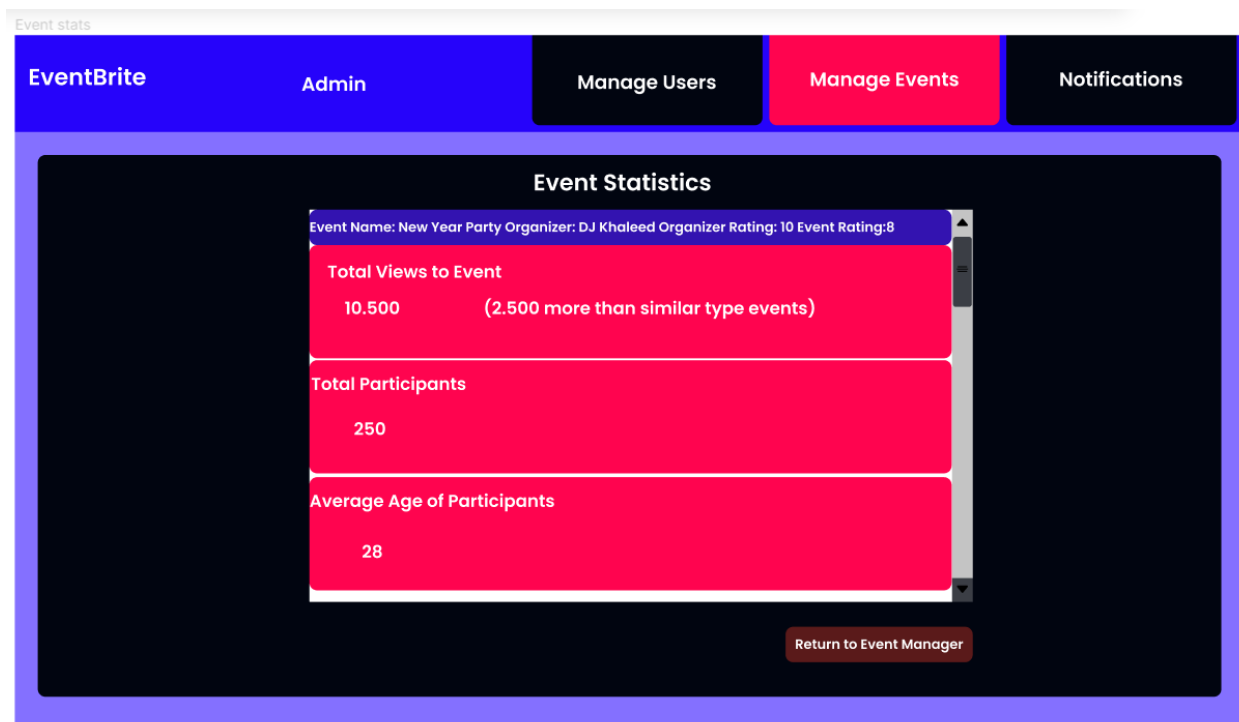
Same as the user details page, this page prints the main information for the events. Thus, following SQL queries are going to be executed.

```
SELECT name FROM event;
```

```
SELECT name, rating FROM organizer, user WHERE organizer.u-id=user.u-id;
```



#### 4.1.11 Event Stats Page



Data shown in this screen is based on systems functionality and to some extent what the system stores at the database. Thus, SQL queries are going to be needed. Following are some examples of the codes.

```
SELECT name, quota, nof-attendies FROM event;
```

## 5. Advanced Database Components

### 5.1 Reports

#### 5.1.1 Probable Table View Reports

```
CREATE VIEW LocationOverall AS SELECT * FROM location;
```

```
CREATE VIEW EventOverall AS SELECT * FROM event;
```

```
CREATE VIEW ReportOverall AS SELECT * FROM report;
```

```
CREATE VIEW UserOverall AS SELECT * FROM user;
```

So on

#### 5.1.2 Probable Relation View Reports

```
CREATE VIEW AttendsOverall AS SELECT * FROM attends;
```

```
CREATE VIEW EditsOverall AS SELECT * FROM edits;
```

```
CREATE VIEW BelongsOverall AS SELECT * FROM belongs;
```

```
CREATE VIEW FollowsOverall AS SELECT * FROM follows;
```

So on

#### 5.1.3 Probable Complex View Reports

```
CREATE VIEW UserEventAttendsOverall AS SELECT * FROM event, user,
attends, WHERE attends.u-id=user.u-id AND event.e-id=attends.e-id;
```

```
CREATE VIEW IDSpecificLocationSeach AS SELECT DISTINCT near-to.locb
FROM near-to WHERE near-to.loca=2107;
```

```
CREATE VIEW CloseEventsOverall AS SELECT DISTINCT event.name FROM
organized-at, event, location WHERE event.e-id=organized-at.e-id AND
organized-at.l-id=location.l-id;
```

## 5.2 Views

Probable views can be seen under the 5.1 Reports

## 5.3 Triggers

### 5.3.1 Probable Triggers

Account creation triggers a u-id increment signal. (i.e it increments it by 1, before insert)

Event creation triggers an e-id increment signal. (i.e it increments it by 1, before insert)

Report creation triggers the r-id increment signal. (i.e it increments it by 1, before insert)

Ticket creation triggers the t-id increment signal. (i.e it increments it by 1, before insert)

Location creation triggers the l-id increment signal. (i.e it increments it by 1, before insert)

Wallet creation triggers the w-id increment signal. (i.e it increments it by 1, before insert)

Following should trigger a new update on the follows relation. (after insert, before next update)

Rating an event should trigger a new update on the ae-rates relation. (after insert, before next update)

Rating an event owner should trigger a new update on the ao-rates relation. (after insert, before next update)

## 5.4 Constraints

### 5.4.1 Probable Constraints

System cannot be used but viewed without an account.

System cannot be used until logging in to the account.

There are size limitations on every possible user input. Those are as follows: for names it is limited by 25 characters, for coupons it is 35 characters, for every other input it is up to 256 characters. For passwords, users are expected to enter whatever they like for 256 characters but their passwords are going to be converted to md5 hashes.

Locations are assigned by the system.

Admins are assigned by the system.

Being an event owner is nothing but increased restrictions on the events hosted.

Reports are created by the admins and only available to admins.

Wallet balance cannot be a minus value.

Ticket's refund policy and Event's age restrictions are set with respect to the following look up table.

Refund policy is stored as a tinyint (0 to 255) however, its first 4 values are going to be used. This is to enable us to be flexible as much as possible for the possible distinct refund policies while letting us save more space.

Refund Policy	Description
0	No refund
1	Refunds available for 24 (but not exceeding the event start time) hours only after buying the ticket.
2	Refunds available until 1 hour is left to the start of the event.

3	Refunds are available until the start of the event.
Any other value	No meaning in the database and throws an error in the code.

Age restriction is stored as a bit (0,1,Null) however, it is possible to restructure it.

This is only to show some of the less common types.

Age Restriction	Description
0	No age restriction i.e +18 and -18 can come in.
1	Age restriction i.e + 18 can and -18 cannot come in.

## 5.5 Stored Procedures

### 5.5.1 Probable Procedures

Everything under the 5.5.1 is also suitable for this section as their view names being their procedure names and if exist their specific values as the parameters.

*Definition:*     CREATE PROCEDURE LocationOverall AS SELECT \* FROM location GO;

*Execution:*     EXEC LocationOverall;

or a slightly improved version

*Definition:*     CREATE PROCEDURE LocationOverall @l-id int AS SELECT \* FROM location;

*Execution:*     EXEC LocationOverall @l-id=2107;

## **6. Implementation Details**

We will use PostgreSQL as database management in our project. We will use NodeJS in the back-end and we will use javascript, html, css and React.js in the front-end.

## **7. Website**

The Reports of our project and the source code files can be found in the repository:

<https://github.com/CS353OCP/cs353ocp>