ClubHub

A College Event Website

Course Info: COP4710, Section 1, Spring 2024

Group Members: Boston Childs, Juni Yeom, John Boyd

GitHub Link: https://github.com/junidy/COP4710Project

Table of Contents

Description

ClubHub is a website for colleges to be able to organize events and clubs and to make it easier for students to stay up-to-date and aware of what is going on at their campus. Any college can register themselves and can designate super admins and admins to oversee RSO creation, event creation, and users registering for their respective university.

GUI

The GUI for the project is a website created using React and Vite. For the underlying language we chose to use JavaScript with React. We debated using TypeScript in the beginning but decided upon JavaScript since the scale of the project doesn't require it and makes development easier. For the database we decided to use MySQL to manage it due to its simplicity.

Landing Page

For the landing page of the website we wanted to keep it relatively simple. The top of the page has the navbar to move between the main page, events page, and club page. Additionally the navbar also has the login and register buttons. After the navbar is the landing page's hero element. For the hero we went with a dimmed image of the UCF Reflection Pond with some basic text over it. Lastly there is a footer at the bottom of the page.

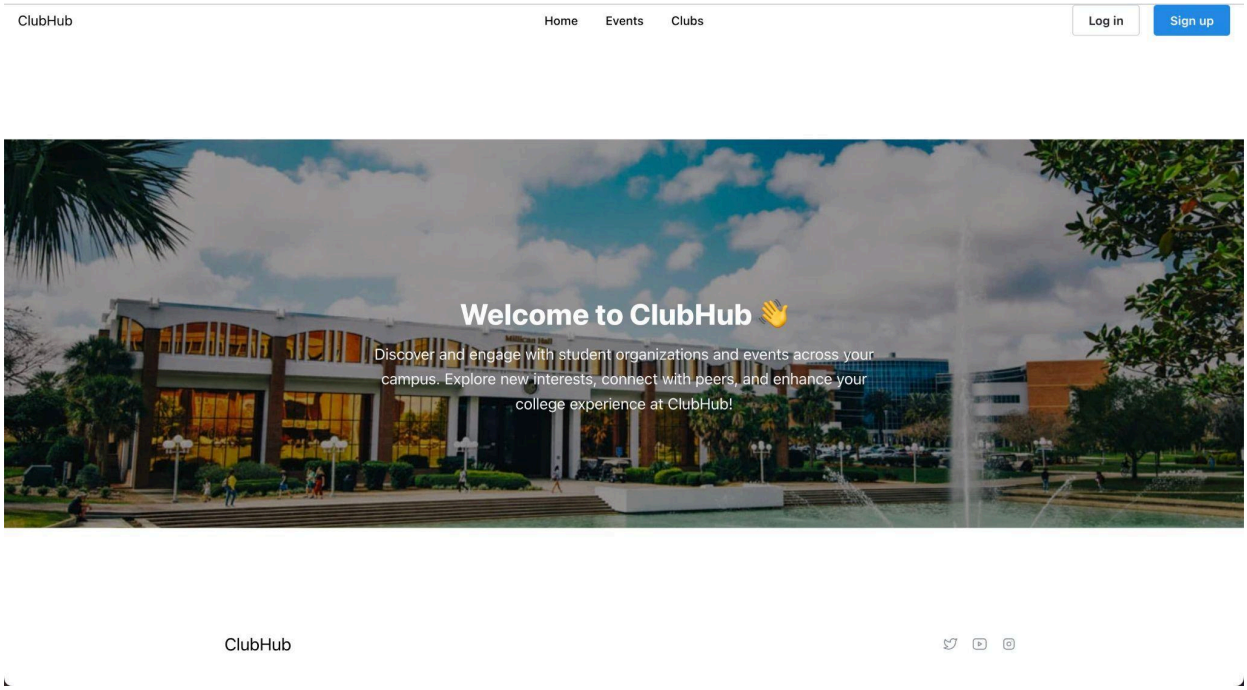


Image 1: The ClubHub landing page

Authentication Page

Logging in

The login page is kept relatively simple as well. It simply asks for the users email and password and if put in correctly takes them back to the home page signed in. If the user hasn't created an account there there is a helpful link that takes the user to the registration page. Similarly the registration only asks for the necessary information to create an account and sends the user back to the home page once an account is created. In the registration page, a dropdown menu provides the option of creating a regular student account or a club administrator account.

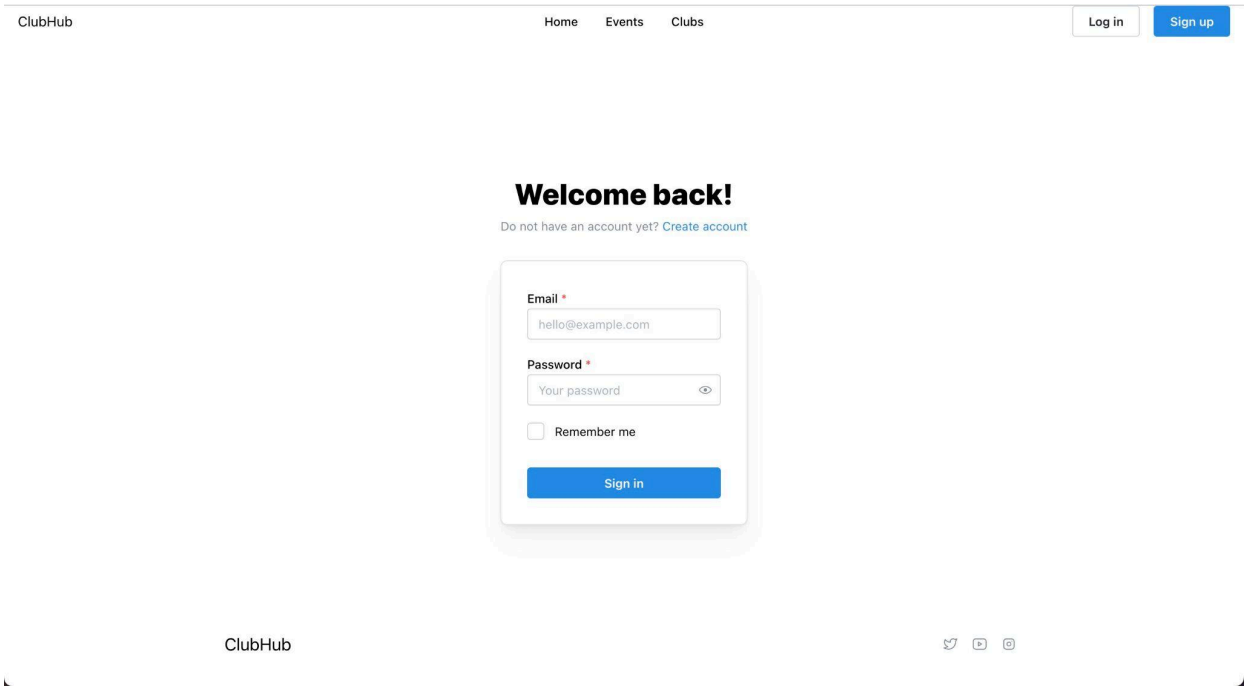


Image 2: The login page

Registering for a new account

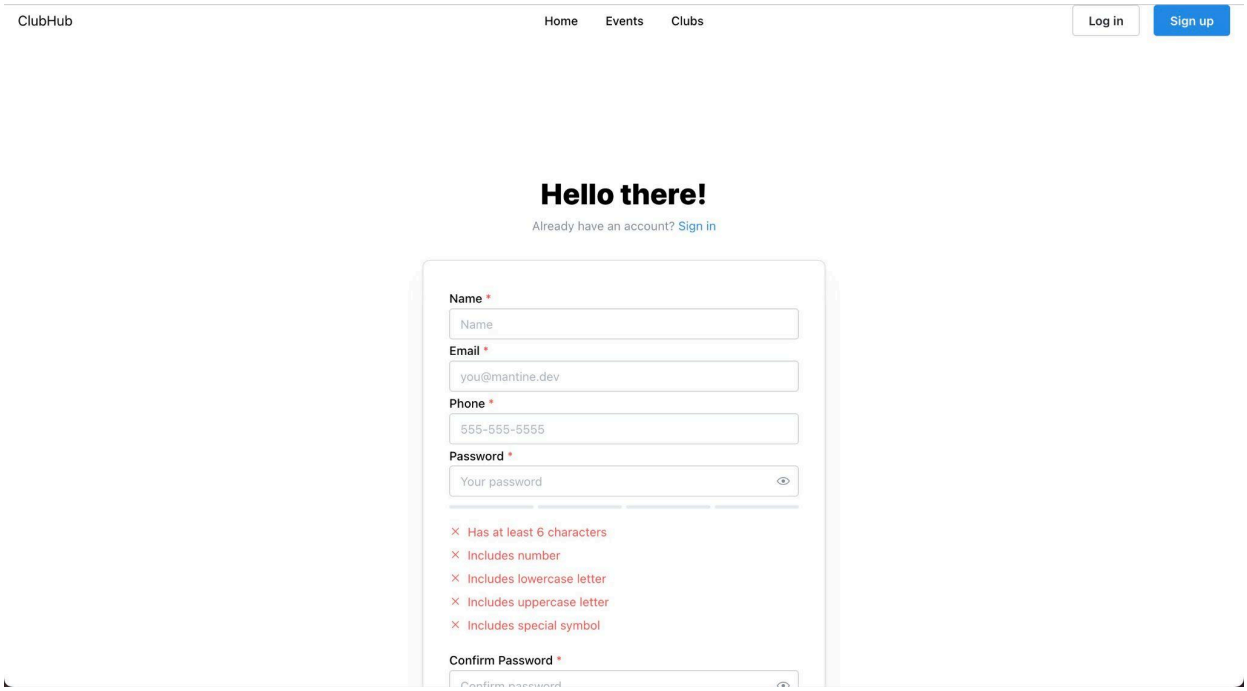


Image 3: The registration page

Club Management Page

Joining and leaving clubs

After creating an account a user will most likely want to join a club. Once the user navigates to the ‘Clubs’ page they can view all available clubs at the university they registered for. Joining the club is as easy as clicking the “Join” button underneath a club. Leaving is just as easy since the “Join” button will turn into a “Leave” button once clicked.

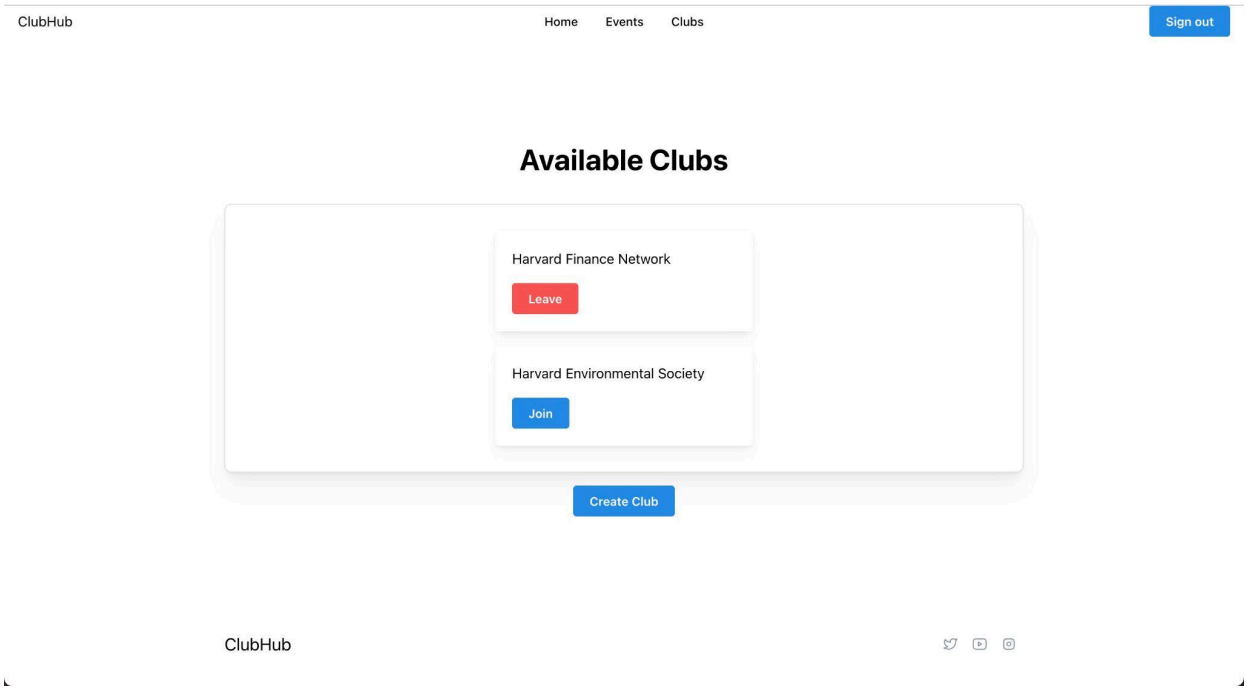


Image 4: The clubs page

Creating a new club

Creating a club on ClubHub is very simple. Once logged in users can navigate over to the ‘Clubs’ page and simply click create club. A pop up will ask for the name of the club and once a name is put in, click submit and the club will be placed under review to be created.

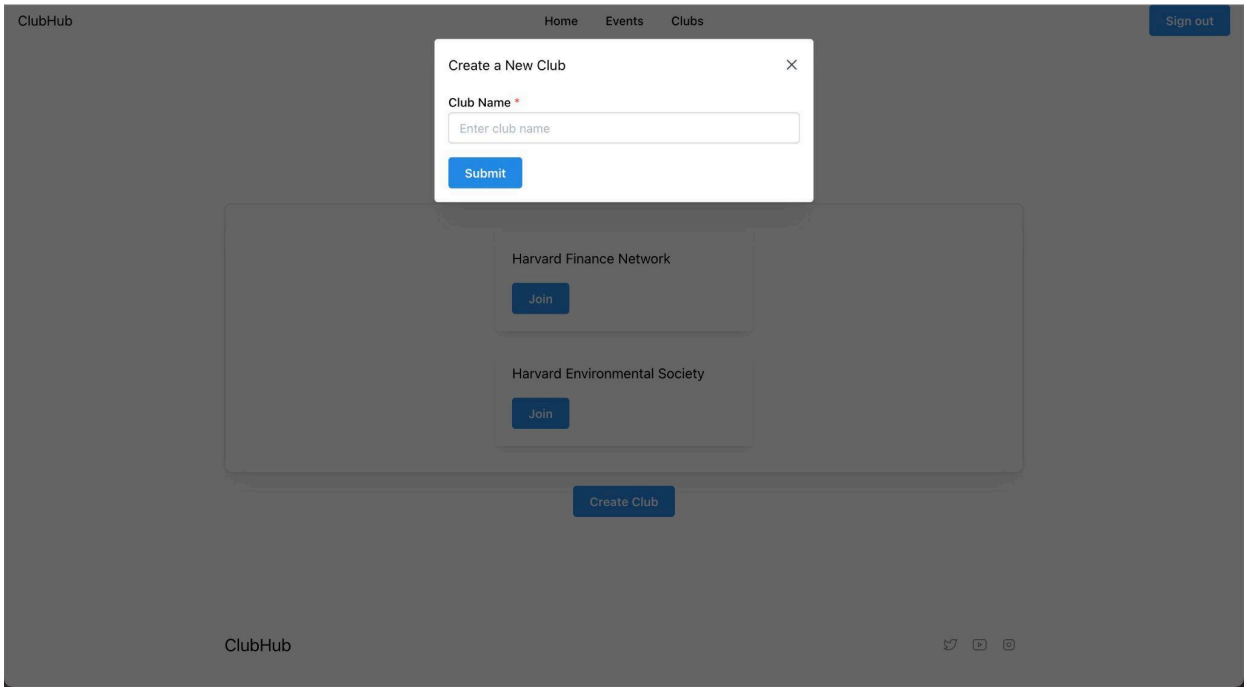


Image 5: Creation of a new RSO

Event Page

Creating a event

Creating an event is equally as simple. Over on the ‘Events’ page users can view all available events for RSOs at their university. If they would like to create an event simply click ‘Create Event’ and fill out the event information.

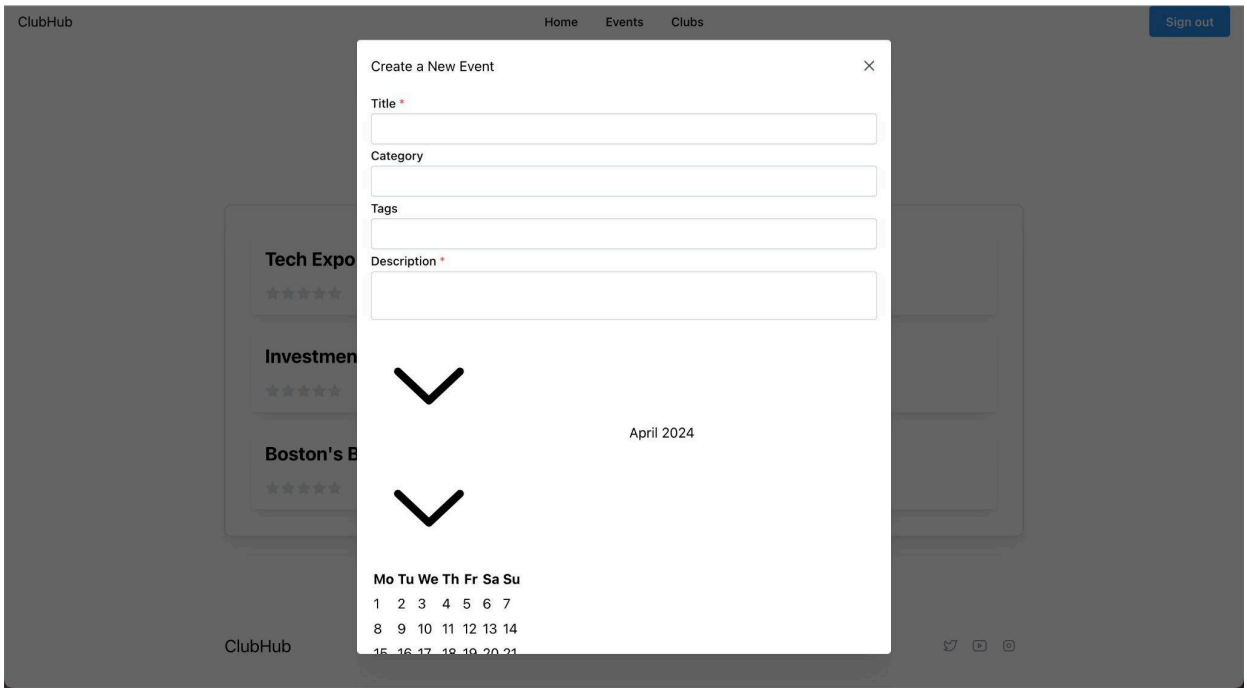


Image 6: Creation of a new RSO event

Event feed for a logged-in user

Once the event is created it will appear in the list of events for the user's university. If there is a high enough amount the window the events are located in become scrollable so as to not overflow onto the rest of the page. The details within each event entry in the feed include its category, and its tags.

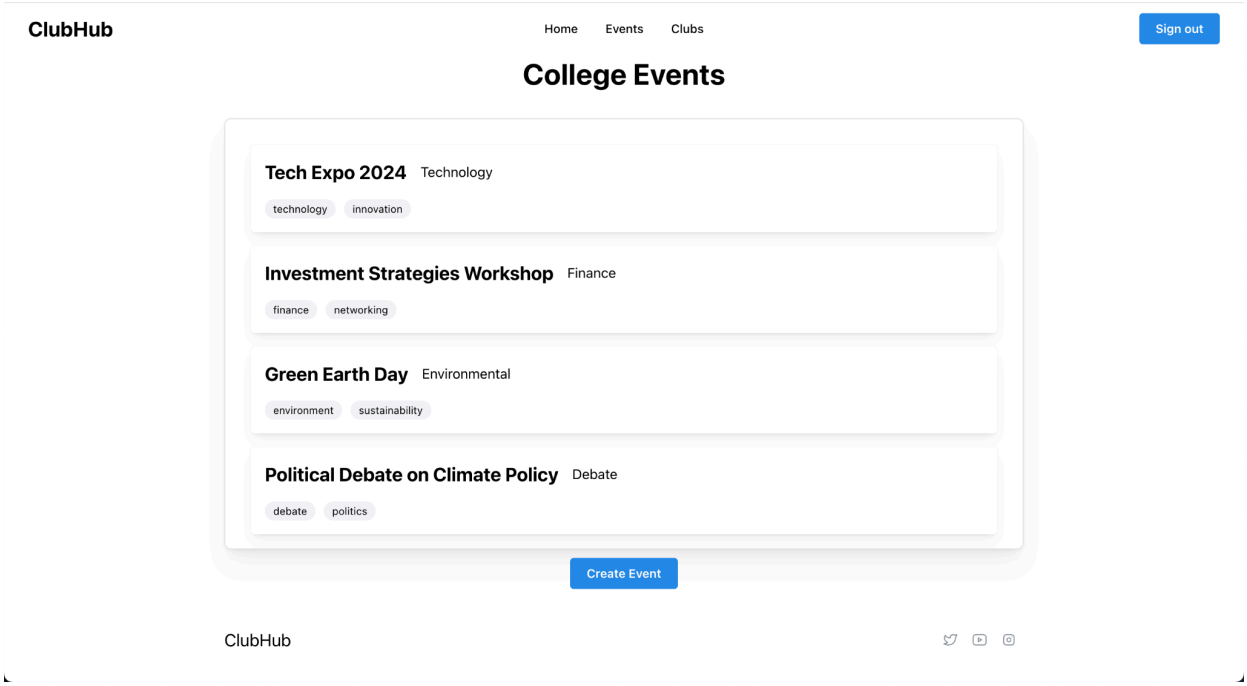


Image 7: All the available events for this user

Event details

An event can be clicked on from the event feed to reveal a modal window, displayings its details and comments. The event details include the description of the event, its date, the interval of time when it is occurring, contact information for the event, and its location. The comments section allows users to see other users' comments, and leave one of their own.

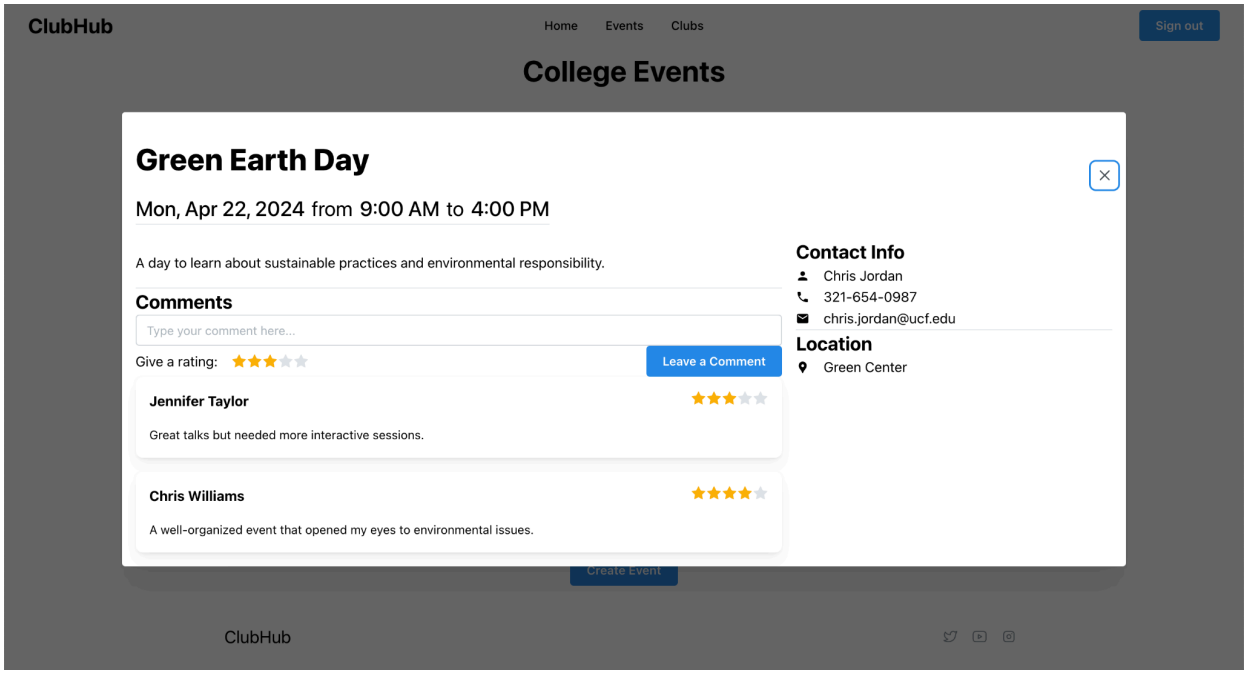


Image 8: A detailed view of an event

Comments

Leaving a comment

The contents of a comment are typed in the text box, and a rating (1 to 5 stars) can be left by clicking on the appropriate star rating.

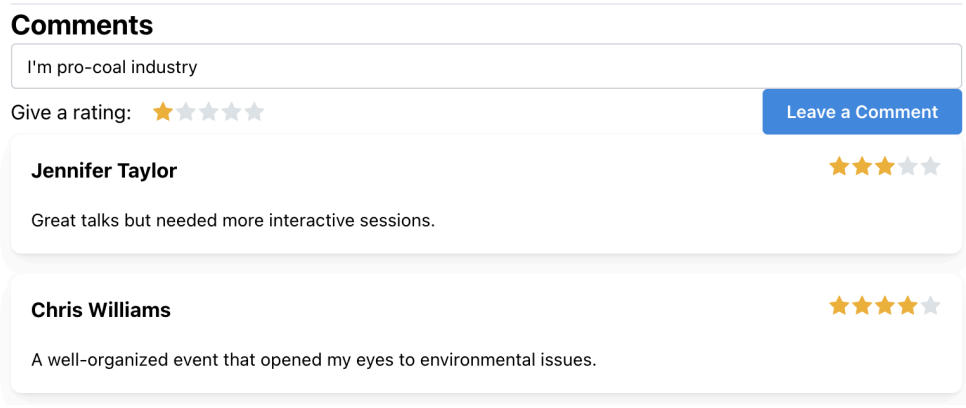


Image 9: About to submit a new comment

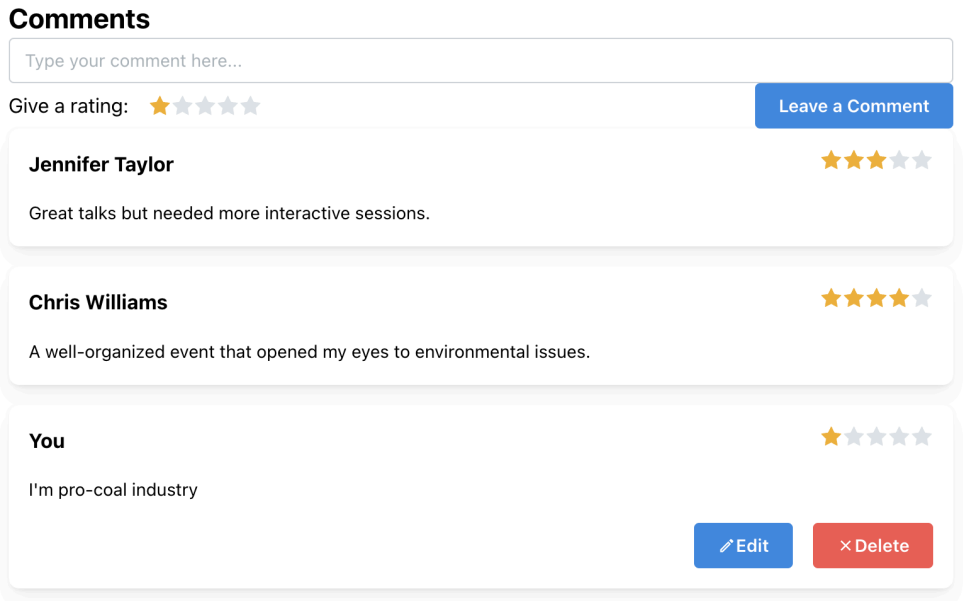


Image 10: New comment has been submitted

Editing and deleting comments

A user can edit or delete their own comments. Upon clicking the edit button, the comment's content and rating can be edited in place:

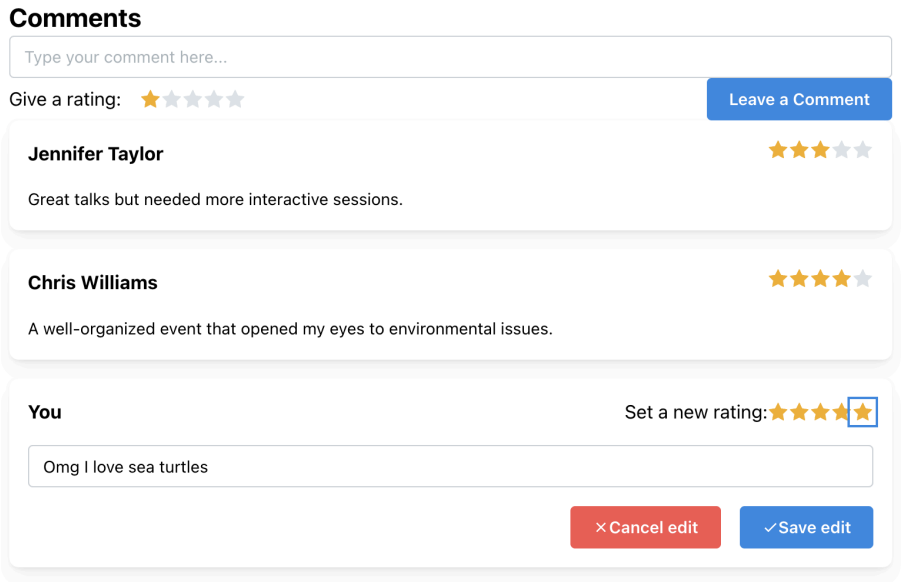


Image 11: Editing a comment

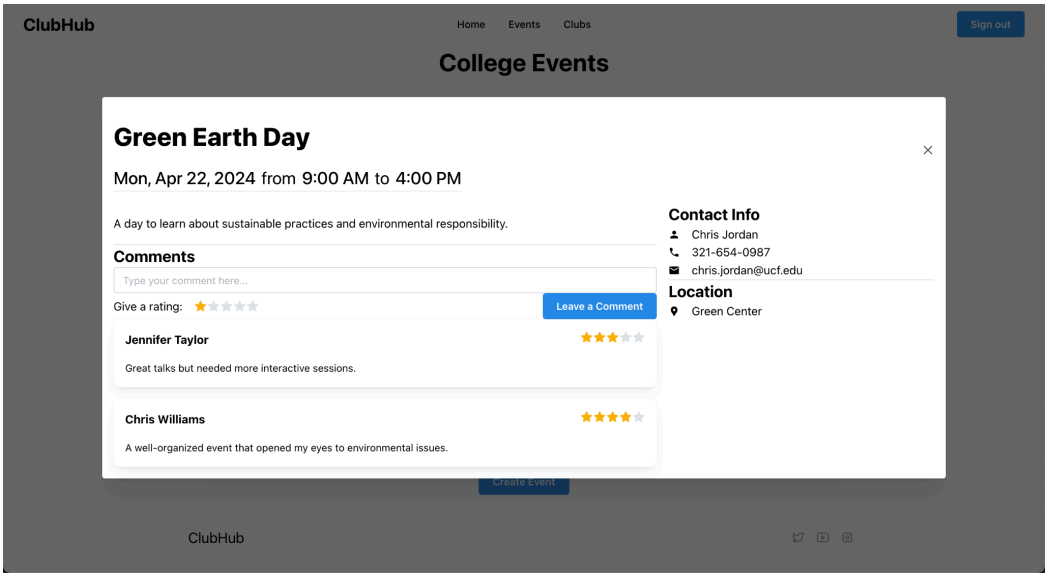


Image 12: After deletion of a comment

ER-Model

Here is the Entity Relation Diagram for our project. As it can be seen there tables for users (all kinds), universities, RSOs, events, and more. Users belong to a university and can belong to an RSO. RSOs have their own table for members which have foreign keys relating back to individual members. RSOs also have a table for RSO events which have a foreign key for items in the “Events” table that happen to be RSO events instead of public events.

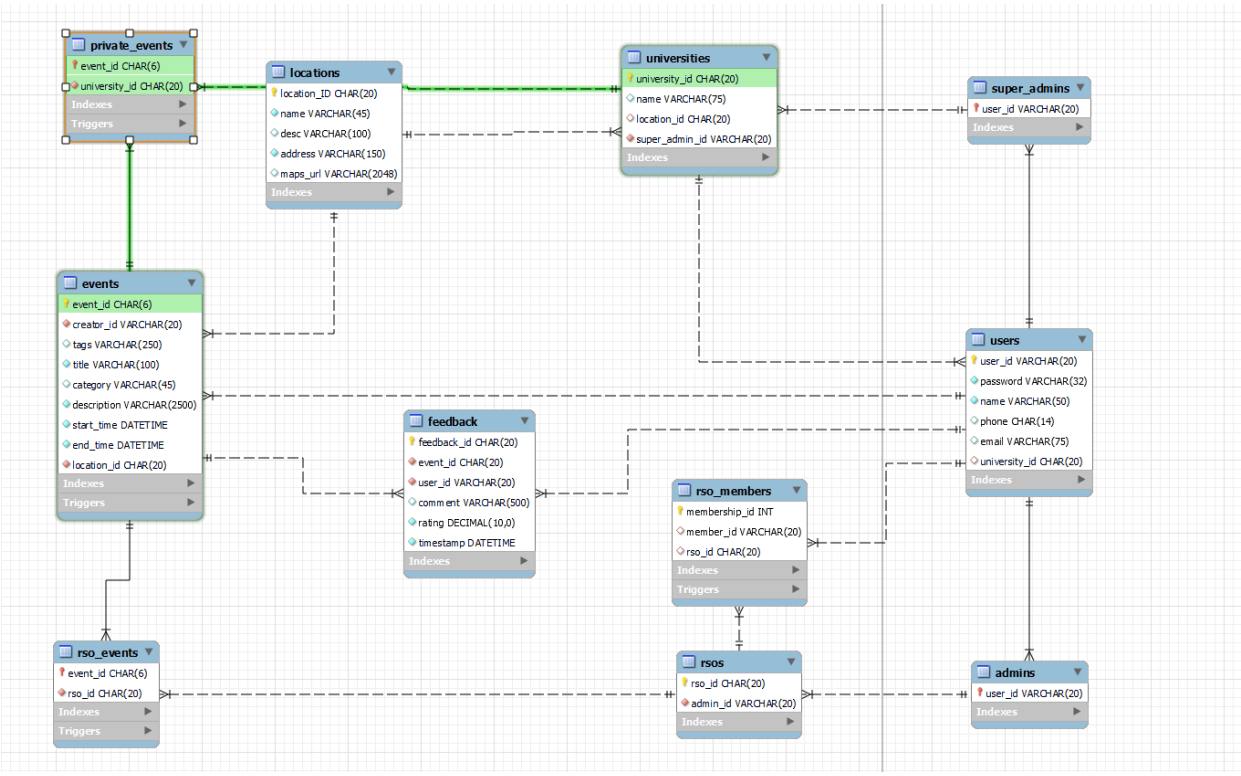


Image 13: The ERD for the project

Relational Data Model

To implement the relational data model in SQL, we used a SQL script to create all the tables to get the database setup. We used a couple different data types for the fields of tables (mostly varchars, dates, and ints). Additionally we used a lot of foreign keys to relate the tables to each other and to make sure that data can be correctly accessed by queries. We also have some constraints to make sure foreign keys reference the correct field from a table, like the ‘creator_id’ key in the ‘events’ table to the ‘user_id’ of the user who created the event (as pictured below).

```
CREATE TABLE `events` (
  `event_id` int NOT NULL AUTO_INCREMENT,
  `creator_id` int NOT NULL,
  `tags` varchar(250) DEFAULT NULL,
  `title` varchar(100) NOT NULL,
  `category` varchar(45) DEFAULT NULL,
  `description` varchar(2500) NOT NULL,
  `start_time` datetime NOT NULL,
  `end_time` datetime NOT NULL,
  `location_id` varchar(50) NOT NULL,
  `contact_name` varchar(50) NOT NULL,
  `contact_phone` varchar(14) NOT NULL,
  `contact_email` varchar(75) NOT NULL,
  PRIMARY KEY (`event_id`),
  UNIQUE KEY `event_id_UNIQUE` (`event_id`),
  KEY `fk_events_users_creator_id_idx` (`creator_id`),
  CONSTRAINT `fk_events_users_creator_id` FOREIGN KEY (`creator_id`) REFERENCES `users` (`user_id`),
  KEY `fk_events_locations_location_id_idx` (`location_id`),
  CONSTRAINT `fk_events_locations_location_id` FOREIGN KEY (`location_id`) REFERENCES `locations` (`location_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Image 14: The SQL statement to create the ‘events’ table

We also have checks as well for the creation of some tables. For the ‘feedback’ table, which represents the comments. There is a check to make sure the rating for a comment is between 1 and 5 since that is the range for the rating of a comment.

```
CREATE TABLE `feedback` (
  `feedback_id` int NOT NULL AUTO_INCREMENT,
  `event_id` int NOT NULL,
  `user_id` int NOT NULL,
  `comment` varchar(500) DEFAULT NULL,
  `rating` int NOT NULL,
  `timestamp` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`feedback_id`),
  UNIQUE KEY `feedback_id_UNIQUE` (`feedback_id`),
  KEY `fk_feedback_events_event_id_idx` (`event_id`),
  KEY `fk_feedback_users_user_id_idx` (`user_id`),
  CONSTRAINT `fk_feedback_events_event_id` FOREIGN KEY (`event_id`) REFERENCES `events` (`event_id`),
  CONSTRAINT `fk_feedback_users_user_id` FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`),
  CONSTRAINT `rating_range` check (
    `rating` BETWEEN 1 AND 5
  )
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Image 15: Creation of the comments table, noting the check for the rating at the bottom

Lastly we also have a few triggers in the creation script as well. If a new event is going to take place at the same location at the same time as a pre-existing event, there is a trigger put in place to prevent the insertion of the new event.

```
DELIMITER ;;
/*!50003 CREATE*/ /*!50017 DEFINER=`root`@`localhost`*/ /*!50003 TRIGGER `Overlapping_Event_Check` BEFORE INSERT ON `events` FOR EACH ROW BEGIN
    IF EXISTS (
        SELECT 1
        FROM `events` E
        WHERE E.location_id = NEW.location_id
        AND (NEW.end_time > E.start_time)
        AND (NEW.start_time < E.end_time)
    ) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot insert overlapping events';
    END IF;
END */;;
DELIMITER ;
```

Image 16: Trigger preventing overlapping events

```
/*!50003 CREATE*/ /*!50017 DEFINER=`root`@`localhost`*/ /*!50003 TRIGGER `RSO_Status_Update_A` AFTER INSERT ON `rso_members` FOR EACH ROW BEGIN
  IF (
        SELECT COUNT(*)
        FROM rso_members
        WHERE rso_id = NEW.rso_id
    ) >= 5 THEN
        UPDATE rsos
        SET active = TRUE
        WHERE rso_id = NEW.rso_id;
    END IF;
END */;;
```

Image 17: Trigger governing the “active” status of RSOs based on number of users

Sample Data

Here are some screenshots of the sample data we have put into the database. Firstly we have two universities in the database, one being UCF and the other is Harvard University.

```
mysql> select * from universities;
+---------------+-------------------------------+----------------+
| university_id | name                          | super_admin_id |
+---------------+-------------------------------+----------------+
|             1 | University of Central Florida |              1 |
|             2 | Harvard University            |              2 |
+---------------+-------------------------------+----------------+
2 rows in set (0.00 sec)
```

Image 18: All universities in the database

We have 20 sample students in our database currently. There are a variety of students for each university.

user_id	password	name	phone	email	university_id
1	$2b$10$9WpFyy0Qiqx3pIANNIhUXeQdZJlrkEI8b0QGc7xFL/SEZYwT1vjEC	Chris Smith	806-293-5333	chris.smith33@ucf.edu	1
2	$2b$10$kAeYSaUoSReQWMY30TYmS.mEg1K0eu2D.OmmBdamjtCOavQ393YjC	John Jones	935-442-8061	john.jones6@harvard.edu	2
3	$2b$10$mHXlKvfvhb4N5XtkSe/tRewrZ0VTK7UwL3DIvdlaCWQm5B6vhJGHK	Chris Wilson	325-672-6440	chris.wilson46@harvard.edu	2
4	$2b$10$y/ZSn.m/reCvtSbNEjgCVOee/aTapR/mKIZkjbQjfSxwWBU0T.X6u	Samantha Smith	642-827-7822	samantha.smith77@harvard.edu	2
5	$2b$10$AWCcmeUmtdyZquMq4zpHOee8aMUvPNr1Nx3xLu9YyJmClxfyJfcr6	John Davis	735-723-9867	john.davis21@ucf.edu	1
6	$2b$10$oqnU.8Xd.R50j3jakE.eOe7EzJ7vEEryPAf4QPznW0KYjm1axFDKa	Mike Smith	800-576-3253	mike.smith17@harvard.edu	2
7	$2b$10$wCjRK.oEqApMwSY.dlixCOd2mOxAgjRzHbqCPn5GRi6RXxSVSJKUa	Linda Davis	984-145-7310	linda.davis17@ucf.edu	1
8	$2b$10$VITVhLtN5YRkNJ7CqTsoH.TJc8bICdeDtiXcTlaS1DGK4bpJLrDMm	James Williams	580-297-7931	james.williams86@ucf.edu	1
9	$2b$10$J2.MVoJugm.fRf4dBiSzFOgX9pR7sG/.q1MhbmWBRFZCITlkL17Te	Chris Williams	924-766-4329	chris.williams72@ucf.edu	1
10	$2b$10$6OW8hJ992OeeMUfE.xWBxuzeAmkJEPbYWWSQEIVarmN1fp7VRSneS	Jennifer Taylor	769-737-7075	jennifer.taylor35@ucf.edu	1
11	$2b$10$JTI252HOTDya2/bgLDiK7OGWiAsI6ZaU0AT5PeiM8/szc1rD1GaTS	John Brown	192-715-4604	john.brown20@ucf.edu	1
12	$2b$10$uQtSaqnT.iNTau/9XTkNQOhEtmV8jedB5jmf8Tm5FVYuQGY6VNXAC	James Jones	545-820-4826	james.jones12@harvard.edu	2
13	$2b$10$ZYS8KBCCC2ghX0HOfFIPpOoQqU8t7tIPNUoUZQCwyoYvrALEir2/q	Linda Miller	390-763-7846	linda.miller82@ucf.edu	1
14	$2b$10$D59uSUVkNneTSpj2fdGBQuQM/k2SR5cun3SU.hyFCGE8nSM2f4Bv6	Laura Wilson	202-539-4685	laura.wilson61@harvard.edu	2
15	$2b$10$mSV7m6t22B5x/gtPnIjv1uaZbe3zhG1PyNeVBAaFMOI0Z3BQcrN5W	John Davis	538-705-6124	john.davis87@ucf.edu	1
16	$2b$10$BqeIRiH1nYg7msA.x7TRbewLMAGUNlw4iKvoyihaCeJhXCyHQ4pdy	Linda Miller	674-310-1680	linda.miller44@harvard.edu	2
17	$2b$10$kKJ2H4cfPdY05mQO5bXAF.q/pYDUYm4UXOcmhW7/AhEiJ/.PZWLMG	Mike Johnson	120-750-7828	mike.johnson68@ucf.edu	1
18	$2b$10$2xKofAg6mm5qz6TK/rjO2em1xCkboozY1.OE9JMnONESW8vi2b1wC	James Taylor	536-890-4856	james.taylor47@ucf.edu	1
19	$2b$10$6aLL1wFBYSa5Q6otAbnyPe4zAOs8C.TNk7ktRERHrXQjHT1R13Mu.	Jane Taylor	111-606-1929	jane.taylor45@ucf.edu	1
21	$2b$10$6aLL1wFBYSa5Q6otAbnyPe4zAOs8C.TNk7ktRERHrXQjHT1R13Mu.	Sussus Amongus	666-432-8421	sussus.amongus@ucf.edu	1

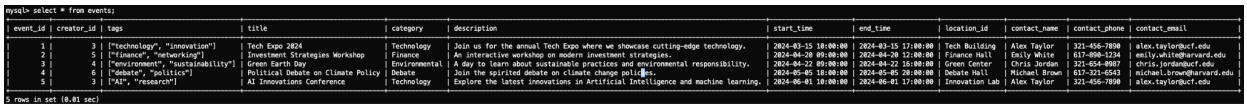
Image 19: All sample students in the database

For the RSOs we have four of them currently in the database. Two of them are for UCF and the other two are for Harvard.

```
mysql> select * from rsos;
+--------+----------+-----------------------------------+--------+
| rso_id | admin_id | name                              | active |
+--------+----------+-----------------------------------+--------+
|      1 |        3 | UCF Tech Club                     |      0 |
|      2 |        4 | UCF Sustainability Initiative     |      0 |
|      3 |        5 | Harvard Finance Network           |      0 |
|      4 |        6 | Harvard Environmental Society     |      0 |
+--------+----------+-----------------------------------+--------+
4 rows in set (0.01 sec)
```

Image 20: All RSOs in the database

We have 5 events currently in our database which have varying numbers of comments and for different RSOs.



```
mysql> select * from events;
+----------+------------+--------------------------------+---------------------+---------------+-----------------------------------------------------------------+---------------------+---------------------+-------------+--------------+--------------+-------------------------+
| event_id | creator_id | tags                           | title               | category      | description                                                     | start_time          | end_time            | location_id | contact_name | contact_phone | contact_email           |
+----------+------------+--------------------------------+---------------------+---------------+-----------------------------------------------------------------+---------------------+---------------------+-------------+--------------+--------------+-------------------------+
|        1 |          3 | ["technology", "innovation"]   | Tech Expo 2024      | Technology    | Join us for the annual Tech Expo where we showcase cutting-edge technology. | 2024-03-15 10:00:00 | 2024-03-15 17:00:00 | Tech Building | Alex Taylor | 321-456-7890 | alex.taylor@ucf.edu     |
|        2 |          5 | ["finance", "networking"]      | Investment Strategies Workshop | Finance | An interactive workshop on modern investment strategies.        | 2024-04-20 09:00:00 | 2024-04-20 12:00:00 | Finance Hall | Emily White | 617-890-1234 | emily.white@harvard.edu |
|        3 |          4 | ["environment", "sustainability"] | Green Earth Day  | Environmental | A day to learn about sustainable practices and environmental responsibility. | 2024-04-22 09:00:00 | 2024-04-22 16:00:00 | Green Center | Chris Jordan | 321-654-0987 | chris.jordan@ucf.edu    |
|        4 |          6 | ["debate", "politics"]         | Political Debate on Climate Policy | Debate | Join the spirited debate on climate change policies.            | 2024-05-05 18:00:00 | 2024-05-05 20:00:00 | Debate Hall | Michael Brown | 617-321-6543 | michael.brown@harvard.edu |
|        5 |          3 | ["AI", "research"]             | AI Innovations Conference | Technology | Explore the latest innovations in Artificial Intelligence and machine learning. | 2024-06-01 10:00:00 | 2024-06-01 17:00:00 | Innovation Lab | Alex Taylor | 321-456-7890 | alex.taylor@ucf.edu     |
+----------+------------+--------------------------------+---------------------+---------------+-----------------------------------------------------------------+---------------------+---------------------+-------------+--------------+--------------+-------------------------+
5 rows in set (0.01 sec)
```

Image 21: All the events in the database

For the comments in the database we have 3 that are left by 3 different users on 2 different events. All comments have a 5 star rating.

```
mysql> select * from feedback;
+-------------+----------+---------+-----------------------------------------+------
--+---------------------+
| feedback_id | event_id | user_id | comment                                 | ratin
g | timestamp           |
+-------------+----------+---------+-----------------------------------------+------
--+---------------------+
|           1 |        1 |       7 | Incredible event with lots of learning! |
5 | 2024-03-15 18:00:00 |
|           2 |        2 |       8 | Very informative workshop. Loved it!    |
5 | 2024-04-20 13:00:00 |
|           3 |        1 |       5 | Seems cool still wont go                |
5 | 2024-04-12 19:52:29 |
+-------------+----------+---------+-----------------------------------------+------
--+---------------------+
3 rows in set (0.00 sec)
```

Image 22: All comments in the database

SQL Examples

The first example that will be shown is creating an RSO. As seen in the last section of screenshots there are only 4 RSOs, 2 for each university. We're going to run a query to create a new RSO for Harvard called the "Harvard Business Union" that will be created by the admin Jane Smith. Here is the query that will be run and a screenshot of the results:

```
INSERT INTO rsos (name, admin_id, active) VALUES ('Harvard Business
Union', 2, active);
```

```
mysql> INSERT INTO rsos (name, admin_id, active) VALUES ('Harvard Business Union', 2
, TRUE);
Query OK, 1 row affected (0.01 sec)
```

Image 23: Example of inserting an RSO into the database

For making a user a member of an RSO we will run a query for that. We are going to add the user "Alex Taylor" to the RSO "UCF Sustainability Initiative" which have the IDs 3 and 2 respectively.

```
mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (3, 2);
Query OK, 1 row affected (0.00 sec)
```

Image 24: A user joining an RSO query

The query to create an event is a little more complicated compared to the others since it has more fields to fill out and it involves dates. Here is a screenshot of the query that is run from the backend when the “create event” endpoint is called:

```
await query(`
  INSERT INTO events (creator_id, tags, title, category, description, start_time, end_time, location_id, contact_name, contact_email, contact_phone)
  VALUES (?, ?, ?, ?, ?, STR_TO_DATE(?, '%d %m %Y %H %i %S'), STR_TO_DATE(?, '%d %m %Y %H %i %S'), ?, ?, ?, ?)
`, [creator_id, tagsString, title, category, description, startString, endString, location_id, contact_name, contact_email, contact_phone]);
```

Image 25: The query to create an event

As you can see there are many fields to insert data into and 2 of them are times which have to be converted to times via the SQL STR_TO_DATE function.

Luckily the query to add a comment to an event is much less complicated. Let’s insert a comment on the “Tech Expo 2024” event by the user “Chris Jordan”.

```
mysql> INSERT INTO feedback (event_id, user_id, comment, rating, timestamp) VALUES (
1, 4, 'Interesting expo! Very informative and fun!', 4, '2024-04-15 16:00:00');
Query OK, 1 row affected (0.00 sec)
```

Image 26: Example query to leave a comment

Lastly the query to show events is quite long and complicated. It looks through to see if it can pull private events (if the user making the query can see it), pull RSO events (if the user is a member of an RSO), and if all else fails, show the public events. Here is a screenshot of that query in the backend:

```
    const events = await query(`
    SELECT DISTINCT e.*
FROM events e
LEFT JOIN private_events pe ON e.event_id = pe.event_id
LEFT JOIN rso_events re ON e.event_id = re.event_id
LEFT JOIN rso_members rm ON re.rso_id = rm.rso_id AND rm.member_id = @user_id
LEFT JOIN users u ON u.user_id = @user_id
WHERE
    e.event_id IN (
        SELECT event_id FROM events
        WHERE event_id NOT IN (SELECT event_id FROM private_events)
        AND event_id NOT IN (SELECT event_id FROM rso_events)
    )
    OR (
        pe.university_id = u.university_id
    )
    OR (
        rm.member_id = @user_id
    )
;

    `, [user_id]);
```

Image 27: The query to get events

Running the query on the user “Michael Brown”, who attends Harvard University and has a user_id of 6 yields the following results:

```
mysql> select distinct e.* from events e left join private_events pe on e.event_id = pe.event_id left join rso_events re on e.event_id = re.event_id left join rso_members rm on re.rso_id = rm.rso_id and rm.member_id = 6 left join users u on u.user_id = 6 where e.event_id in (select event_id from events where event_id not in (select event_id from private_events) and event_id not in (select event_id from rso_events)) or (pe.university_id = u.university_id) or (rm.member_id = 6);
```

event_id	creator_id	tags	title	category	description	start_time	end_time	location_id	contact_name	contact_phone	contact_email
1	3	["technology", "innovation"]	Tech Expo 2024	Technology	Join us for the annual Tech Expo where we showcase cutting-edge technology.	2024-03-15 10:00:00	2024-03-15 17:00:00	Tech Building	Alex Taylor	321-456-7890	alex.taylor@ucf.edu
2	5	["finance", "networking"]	Investment Strategies Workshop	Finance	An interactive workshop on modern investment strategies.	2024-04-20 09:00:00	2024-04-20 12:00:00	Finance Hall	Emily White	617-890-1234	emily.white@harvard.edu
3	5	"Celebration"	Boston's Birthday Party	Party	It is Boston's Birthday! Come celebrate!	2024-04-17 14:30:00	2024-04-17 22:00:00	UCF	Boston	1234567890	hello@example.com

3 rows in set (0.00 sec)

Image 28: Running the “get events” query with the user id of 6

Constraint Enforcement

In order to keep the integrity of the data we have a few constraints to prevent errors. One of these is making sure events can’t be held at the same “location_id” with the same start and end times. Here is a screenshot showing that overlapping events cannot be created:

```
mysql> INSERT INTO `events` (`creator_id`, `tags`, `title`, `category`, `description`, `start_time`
, `end_time`, `location_id`, `contact_name`, `contact_phone`, `contact_email`) VALUES
    -> ('5', '["environment", "conservation"]', 'Conservation Workshop', 'Environmental', 'A worksh
op focused on conservation techniques that overlap with Earth Day activities.', '2024-04-22 14:00:0
0', '2024-04-22 18:00:00', 'Green Center', 'Emily White', '617-890-1234', 'emily.white@harvard.edu'
);
ERROR 1644 (45000): Cannot insert overlapping events
```

Image 29: “Event overlap” constraint preventing the insertion of an invalid event entry

Additionally we also have a trigger that when an RSO goes from 4 to 5 members it becomes active as well as being set to inactive when it falls below 5 members. Here are some screenshots showing that process:

```
mysql> select * from rsos;
+--------+----------+-------------------------------+--------+
| rso_id | admin_id | name                          | active |
+--------+----------+-------------------------------+--------+
|      1 |        3 | UCF Tech Club                 |      0 |
|      2 |        4 | UCF Sustainability Initiative |      0 |
|      3 |        5 | Harvard Finance Network       |      0 |
|      4 |        6 | Harvard Environmental Society |      0 |
|      5 |        2 | Harvard Business Union        |      1 |
+--------+----------+-------------------------------+--------+
5 rows in set (0.00 sec)
```

Image 30: Harvard Business Union is the only active RSO

```
mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (1, 1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (2, 1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (3, 1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (5, 1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (6, 1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO rso_members (member_id, rso_id) VALUES (7, 1);
Query OK, 1 row affected (0.01 sec)
```

Image 31: Having multiple users “join” the UCF Tech Club

```
mysql> select * from rsos;
+--------+----------+-------------------------------+--------+
| rso_id | admin_id | name                          | active |
+--------+----------+-------------------------------+--------+
|      1 |        3 | UCF Tech Club                 |      1 |
|      2 |        4 | UCF Sustainability Initiative |      0 |
|      3 |        5 | Harvard Finance Network       |      0 |
|      4 |        6 | Harvard Environmental Society |      0 |
|      5 |        2 | Harvard Business Union        |      1 |
+--------+----------+-------------------------------+--------+
5 rows in set (0.00 sec)
```

Image 32: UCF Tech Club is now active

```
mysql> DELETE FROM rso_members WHERE member_id = 7 AND rso_id = 1;
Query OK, 1 row affected (0.00 sec)

mysql> DELETE FROM rso_members WHERE member_id = 6 AND rso_id = 1;
Query OK, 1 row affected (0.00 sec)

mysql> DELETE FROM rso_members WHERE member_id = 3 AND rso_id = 1;
Query OK, 1 row affected (0.00 sec)
```

Image 33: Removing members to set the club inactive

```
mysql> select * from rsos;
+--------+----------+--------------------------------+--------+
| rso_id | admin_id | name                           | active |
+--------+----------+--------------------------------+--------+
|      1 |        3 | UCF Tech Club                  |      0 |
|      2 |        4 | UCF Sustainability Initiative  |      0 |
|      3 |        5 | Harvard Finance Network        |      0 |
|      4 |        6 | Harvard Environmental Society  |      0 |
|      5 |        2 | Harvard Business Union         |      1 |
+--------+----------+--------------------------------+--------+
5 rows in set (0.01 sec)
```

Image 34: The UCF Tech Club is now inactive due to having < 5 members

Advanced Features

We have implemented industry-standard security into our authentication protocols and services. All endpoints that require a user's identity to be authenticated use JWT tokens, which the client obtains upon login and/or registration. Passwords are never stored in plaintext: instead, we hash passwords using the *bcrypt* hash function, then salt it before adding it to the database.

JSON Web Token Usage

JWT is an authentication and authorization standard for transmitting data in a manner that both parties can be assured of each other's identity, and that the payload within has not been tampered with. Here we use JWT tokens whenever an endpoint requires that a modification occurs on the state of the behalf of the requesting user. In this manner we can assure that hostile actors cannot impersonate other users and cause illegal modifications to the database.

```
// If password matches, generate a JWT token
const token = jwt.sign({ userId: user.user_id }, jwtSecretKey, { expiresIn: '24h' });

// Send the token in the response
res.json({ token });
```

Image 35: Generation of a JWT token in the /auth/login endpoint

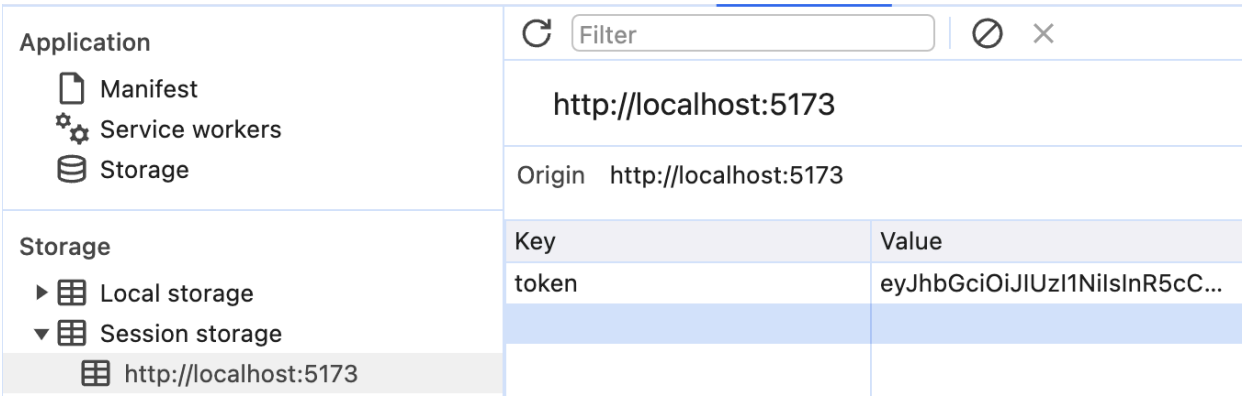


Image 36: Storage of the JWT token obtained from login endpoint in client browser session storage

Password hashing and salting

Hashing and salting passwords before their storage in a database is a ubiquitous security standard. A *hash function* is a one-way cryptographic function that takes in an input string and produces an output string (the *hash*), in a manner such that deriving the input string given the hash is, for all purposes and intents, impossible.

For the same input string, a given hash function will always produce the same hash from that string, which can pose a vulnerability – if an attacker knows a given user's password, and has access to our hashed passwords in our database, the attacker also knows the passwords of all the users that have the same hash. To get around this, we introduce a *salt* – a randomly generated string added to the password before hashing it, and which is stored alongside the hash in the database. With different salts, the same input password will have different hashes, closing off this vector of attack.

```
// Hash the password using bcrypt
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);

// Insert the new user into the database
const result = await pool.query(
  'INSERT INTO users (user_id, password, name, phone, email, university_id) VALUES (?, ?, ?, ?, ?, ?)',
  [user_id, hashedPassword, name, phone, email, university_id]
);
```

Image 37: Hashing and salting passwords before adding to database

user_id	password
1	$2b$10$9WpFyy0Qiqx3pIANNIhUXeQdZJlrkEI8b0QGc7xFL/SEZYwT1vjEC
2	$2b$10$kAeYSaUoSReQWMY30TYmS.mEg1K0eu2D.OmmBdamjtCOavQ393YjC
3	$2b$10$mHXlKvfvhb4N5XtkSe/tRewrZ0VTK7UwL3DIvdlaCWQm5B6vhJGHK
4	$2b$10$y/ZSn.m/reCvtSbNEjgCVOee/aTapR/mKIZkjbQjfSxwWBU0T.X6u
5	$2b$10$AWCcmeUmtdyZquMq4zpHOee8aMUvPNr1Nx3xLu9YyJmClxfyJfcr6
6	$2b$10$oqnU.8Xd.R50j3jakE.eOe7EzJ7vEEryPAf4QPznW0KYjm1axFDKa
7	$2b$10$wCjRK.oEqApMwSY.dlixCOd2mOxAgjRzHbqCPn5GRi6RXxSVSJKUa
8	$2b$10$VITVhLtN5YRkNJ7CqTsoH.TJc8bICdeDtiXcTlaS1DGK4bpJLrDMm
9	$2b$10$J2.MVoJugm.fRf4dBiSzFOgX9pR7sG/.q1MhbmWBRFZCITlkL17Te
10	$2b$10$6OW8hJ992OeeMUfE.xWBxuzeAmkJEPbYWWSQEIVarmN1fp7VRSneS
11	$2b$10$JTI252HOTDya2/bgLDiK7OGWiAsI6ZaU0AT5PeiM8/szc1rD1GaTS
12	$2b$10$uQtSaqnT.iNTau/9XTkNQOhEtmV8jedB5jmf8Tm5FVYuQGY6VNXAC
13	$2b$10$ZYS8KBCCC2ghX0HOfFIPpOoQqU8t7tIPNUoUZQCwyoYvrALEir2/q
14	$2b$10$D59uSUVkNneTSpj2fdGBQuQM/k2SR5cun3SU.hyFCGE8nSM2f4Bv6
15	$2b$10$mSV7m6t22B5x/gtPnIjv1uaZbe3zhG1PyNeVBAaFMOI0Z3BQcrN5W
16	$2b$10$BqeIRiH1nYg7msA.x7TRbewLMAGUNlw4iKvoyihaCeJhXCyHQ4pdy
17	$2b$10$kKJ2H4cfPdY05mQO5bXAF.q/pYDUYm4UXOcmhW7/AhEiJ/.PZWLMG
18	$2b$10$2xKofAg6mm5qz6TK/rjO2em1xCkboozY1.OE9JMnONESW8vi2b1wC
19	$2b$10$6aLL1wFBYSa5Q6otAbnyPe4zAOs8C.TNk7ktRERHrXQjHT1R13Mu.
21	$2b$10$6aLL1wFBYSa5Q6otAbnyPe4zAOs8C.TNk7ktRERHrXQjHT1R13Mu.

Image 38: *password* column in *users* contains hashed passwords

Conclusion

Database performance

We were surprised that despite our number of tables and how complex some of the queries to retrieve or insert data were, we consistently get < 0.02 sec for processing the queries. After looking at our SQL script to create the database there were quite a few things that helped that.

For starts we use quite a few primary key constraints in tables to help with faster lookup and retrieval. Indexes are also automatically created for primary keys which also helps with fast searches. We also have a few unique keys to make sure a good chunk of the data is unique so the search operations are more efficient.

Our schema is also normalized, with separate tables for different entities. We have proper relationships defined between them which helps reduce data redundancy and improves query efficiency. Our triggers, especially the one on preventing overlapping events, make things easier since they are validated at the database level and don’t have to be validated at the application level.

Desired features/functionalities

The most urgent feature that would need to be implemented were this application brought to production is the integration of events from external feeds, such as a JSON/XML feed from events.ucf.edu. This could be accomplished by a script in the backend that periodically fetches new events from specified feeds, and transforms the data into a format that can be added into the database (the database would include a constraint that prevents the addition of redundant events).

Problems encountered

There were several times where the database schema had to be altered during the implementation of the backend server and frontend interface, due to the late realization of requirements that were not captured on the initial iteration of the schema. This application is a CRUD app, and CRUD apps are at their essence merely a means for users to interface with a database. The raison d'etre for these types of data-intensive applications is the database, and therefore the entire application is an extension of the architecture of the database. Getting it right at the source can save significant amounts of developer time.