Group Report: University Events

COP 4710 Group 32

William Cromar

Contents

1 Overview

The overall architecture of the project is extremely simple in order to lighten the workload, since I'm working on the project alone. The backend is designed as a monolithic web server written in Python with Flask and a MySQL database. The web frontend is almost all static (written with just HTML and CSS) content generated from the backend and displayed in the browser, although there are some frontend components written in JavaScript to support advanced features.

The database accessed from the backend server through the official MySQL connection library for Python, which safely templates queries. As such, almost all SQL is embedded in strings in the Python application, with the exception of initial startup scripts. There are a few scripts written in SQL (saved in `scripts/` directory) to initialize the database schema and populate with some basic test data. These are expected to be run once by the administrator before starting the web server for the first time.

Figure 1 shows an overview of the application's architecture.

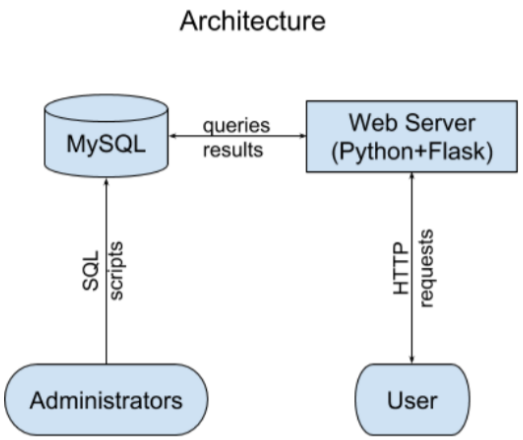

Figure 1: Overview of application's architecture

2 Database

2.1 Design

The relational model used for the database is shown in Figure 2. Most of the design of the data model flowed naturally from the requirements of the project, but there are a few key areas where trade-offs had to be analyzed and important design decisions had to be made. These decisions are covered in the following sections.

There are multiple valid ways to design the database, and my final design reflects my own personal experiences and preferences in software engineering. In order to better demonstrate why I made the decisions and trade-offs that I did, I provide an outline of my design philosophy:

1. Design the database schema to be easy to mutate without introducing anomalies.
2. Write views to make querying the database easier for the client when following (1) makes the data harder to read.
3. Don't take (1) to an extreme: views should be simple, understandable, and easy to update for schema changes.

2.1.1 Event

There are three event types given in the project requirements: public, private, and RSO events. Nearly all attributes of the events are common across all three types.These arguably form an inheritance hierarchy, where there's a supertype Event and three subtypes that add attributes. There are three conventional ways to implement such a hierarchy:

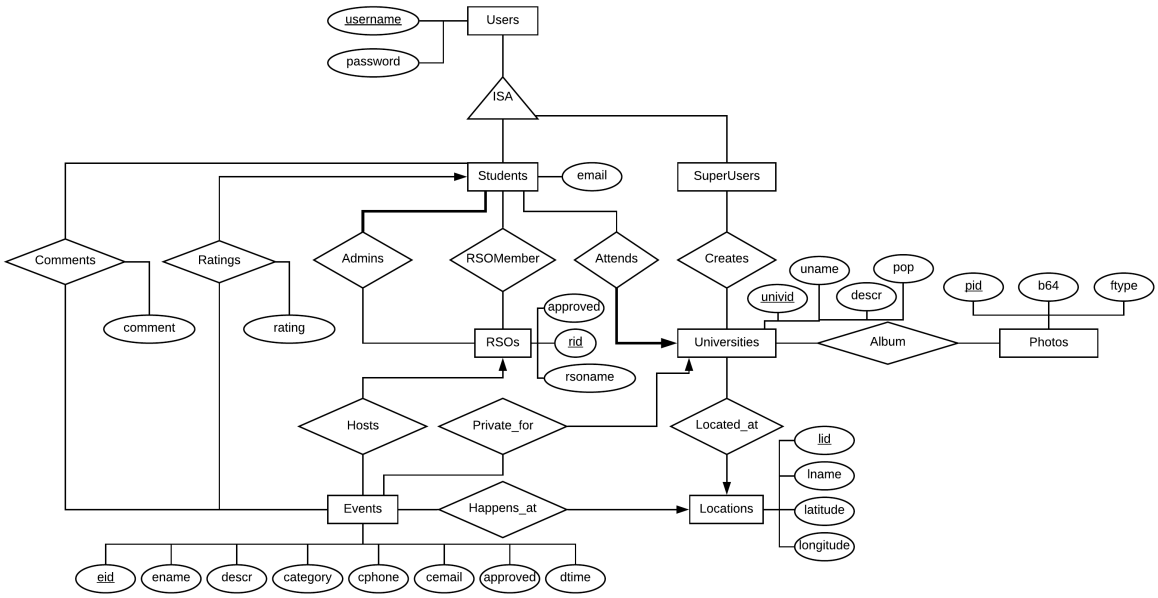

Figure 2: ER Model for events website.

1. Store each type of event in its own separate table. This makes querying all events significantly more difficult.
2. Store all three types in the same table, and only use some attributes for certain types of events. This imposes a risk of setting conflicting options (i.e. making an event that's both RSO and private type), but makes querying all events quite easy.
3. Store all common attributes in one main table Events and have tables for each type that reference it. This makes querying common attributes of events easy, but make determining the scope of an event given just its common attributes harder.

A separate decision to display the scope of an event in the user interface informed a decision to select option (2). In this design, all events are assumed to be public unless the set either a *university restriction* or *rso restriction*. In order to mitigate the risk of erroneously creating an event that sets both restrictions, a simple check was introduced to the table to ensure only zero or one restriction is set (see Section 2.3). Then, the scope of an event can be determine just by querying the one table.

2.1.2 Users

A similar decision is required to accurately represent the three user types: student, admin, and super-admin. A slightly different approach is taken to solve it, though:

1. Anyone can sign up for a username and password and are added to the `Users` table.
2. A user can be promoted to super admin by the DBA, who can add their name directly to the `SuperUsers` table.
3. `Users` can enter their university name and student e-mail to become a *student* in the Students table.
4. A student can request to create an RSO, and will be added to the Admins table. Note that this design permits multiple admins for one RSO and allows a student to administer multiple RSOs. This new RSO is approved by a super admin.

The information in these tables is combined into one view, that has the users credentials, nullable

student information, and a flag indicating whether they are a *super admin*.

2.1.3 Location

The location of an event can either be stored separately in some table, or all of that information can be embedded in both Events and Universities. I chose the former option to facilitate easier location selection, where a user can create an event by choosing the name of the location from a list rather than having to select off of a map every time. However, an interactive map is still available to add new location entries (see Section 4.4).

2.1.4 Other Decisions

A number of other minor design decisions are listed here:

- The range of a rating (1 to 5) is enforced by a CHECK constraint.
- RSOs have an approved field, and only move to the `ApprovedRSOs` view when the *Super-Admin* presses the "Approved" button.
- Images of universities can be uploaded by super admins, which are encoded as base 64 in the database.
- A student seeking to create an RSO must provide the names of 5 other students at the time of creation in the interface.
- All ID values that act as primary keys are auto-incremented integers.

2.2 Relational Schema

The following is the contents of `schema.sql`, which is used to define the tables needed for the application.

```
CREATE TABLE Users(
    username VARCHAR(32),
    passwd VARCHAR(32),
    PRIMARY KEY (username)
);

CREATE TABLE SuperUsers(
    username VARCHAR(32),
    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES Users(username)
        ON DELETE CASCADE
);

CREATE TABLE Locations(
    lid INT NOT NULL AUTO_INCREMENT,
    lname VARCHAR(64),
    latitude FLOAT,
    longitude FLOAT,
    PRIMARY KEY (lid)
);

CREATE TABLE Universities(
```

```
    univid INT NOT NULL AUTO_INCREMENT,
    uname VARCHAR(64),
    primarylid INT,
    pop INT,
    descr TEXT,
    PRIMARY KEY (univid),
    FOREIGN KEY (primarylid) REFERENCES Locations(lid)
);

CREATE TABLE Students(
    username VARCHAR(32),
    univid INT NOT NULL,
    email VARCHAR(64),
    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES Users(username)
        ON DELETE CASCADE,
    FOREIGN KEY (univid) REFERENCES Universities(univid)
);

CREATE TABLE RSOs(
    rid INT NOT NULL AUTO_INCREMENT,
    rsoname VARCHAR(64) UNIQUE,
    univid INT,
    approved BOOL DEFAULT 0,
    PRIMARY KEY (rid),
    FOREIGN KEY (univid) REFERENCES Universities(univid)
        ON DELETE CASCADE
);

CREATE TABLE Admins(
    username VARCHAR(32),
    rid INT,
    PRIMARY KEY (username, rid),
    FOREIGN KEY (username) REFERENCES Students(username)
        ON DELETE CASCADE,
    FOREIGN KEY (rid) REFERENCES RSOs(rid)
        ON DELETE CASCADE
);

CREATE TABLE Events(
    eid INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(64),
    descr TEXT,
    category VARCHAR(64),
    dtime DATETIME,
    lid INT,
    cphone CHAR(10),
    cemail VARCHAR(64),
    urestriction INT,
    rsorestriction INT,
    approved BOOL DEFAULT 0,
```

```
    PRIMARY KEY (eid),
    FOREIGN KEY (lid) REFERENCES Locations(lid),
    FOREIGN KEY (urestriction) REFERENCES Universities(univid)
        ON DELETE CASCADE,
    FOREIGN KEY (rsorestriction) REFERENCES RSOs(rid)
        ON DELETE CASCADE,
);

CREATE TABLE RSOMembers(
    username VARCHAR(32),
    rid INT,
    PRIMARY KEY (username, rid),
    FOREIGN KEY (username) REFERENCES Students(username)
        ON DELETE CASCADE,
    FOREIGN KEY (rid) REFERENCES RSOs(rid)
        ON DELETE CASCADE
);

CREATE TABLE UserRating(
    username VARCHAR(32),
    eid INT,
    rating INT,
    PRIMARY KEY (username, eid),
    FOREIGN KEY (username) REFERENCES Users(username)
        ON DELETE CASCADE,
    FOREIGN KEY (eid) REFERENCES Events(eid)
        ON DELETE CASCADE,
);

CREATE TABLE UserComment(
    cid INT NOT NULL AUTO_INCREMENT,
    username VARCHAR(32),
    eid INT,
    comment TEXT,
    PRIMARY KEY (cid),
    FOREIGN KEY (username) REFERENCES Users(username)
        ON DELETE CASCADE,
    FOREIGN KEY (eid) REFERENCES Events(eid)
        ON DELETE CASCADE
);

CREATE TABLE Photos(
    pid INT NOT NULL AUTO_INCREMENT,
    univid INT,
    b64 MEDIUMTEXT NOT NULL,
    ftype VARCHAR(16),
    PRIMARY KEY (pid),
    FOREIGN KEY (univid) REFERENCES Universities(univid)
        ON DELETE CASCADE
);
```

2.2.1 Views

The following is the contents of `views.sql`, which defines several views used by the application for convenience.

```
CREATE VIEW EasyUsers AS
SELECT Users.*, Students.univid, Students.email,
    Users.username IN (SELECT * FROM SuperUsers) AS super
FROM Users
LEFT JOIN Students ON Users.username = Students.username;

CREATE VIEW ApprovedRSOs AS
SELECT RSOs.*, Universities.uname
FROM RSOs JOIN Universities
ON RSOs.univid = Universities.univid
WHERE approved = 1;

CREATE VIEW EventsInfo AS
SELECT Events.*, Universities.uname, RSOs.rsoname,
    Locations.lname, Locations.latitude, Locations.longitude
FROM Events
JOIN Locations ON Events.lid = Locations.lid
LEFT JOIN RSOs ON Events.rsorestriction = RSOs.rid
LEFT JOIN Universities ON Events.urestriction = Universities.univid
ORDER BY eid;

CREATE VIEW ApprovedEvents AS
SELECT * FROM EventsInfo
WHERE approved = 1;
```

2.3 Constraints

The following sections demonstrate the constraints that could not be expressed as simple key constraints. The demonstrations of these constraints do not contain a full set of test data, in order to make the constraints easier to understand in isolation.

2.3.1 Range of User Ratings

A simple `CHECK` is used to enforce the range of user ratings (1 to 5). This is also enforced by the GUI, which does not allow ratings outside of this range.

```
ALTER TABLE UserRating ADD CONSTRAINT CHECK (rating > 0 AND rating < 6);
```

2.3.2 Overlapping Event Times/Locations

A `UNIQUE` constraint is used to ensure that no two events are scheduled at the same time at the same place. See Figure 3 for a demonstration.

```
ALTER TABLE Events ADD CONSTRAINT UNIQUE(dtime, lid);
```

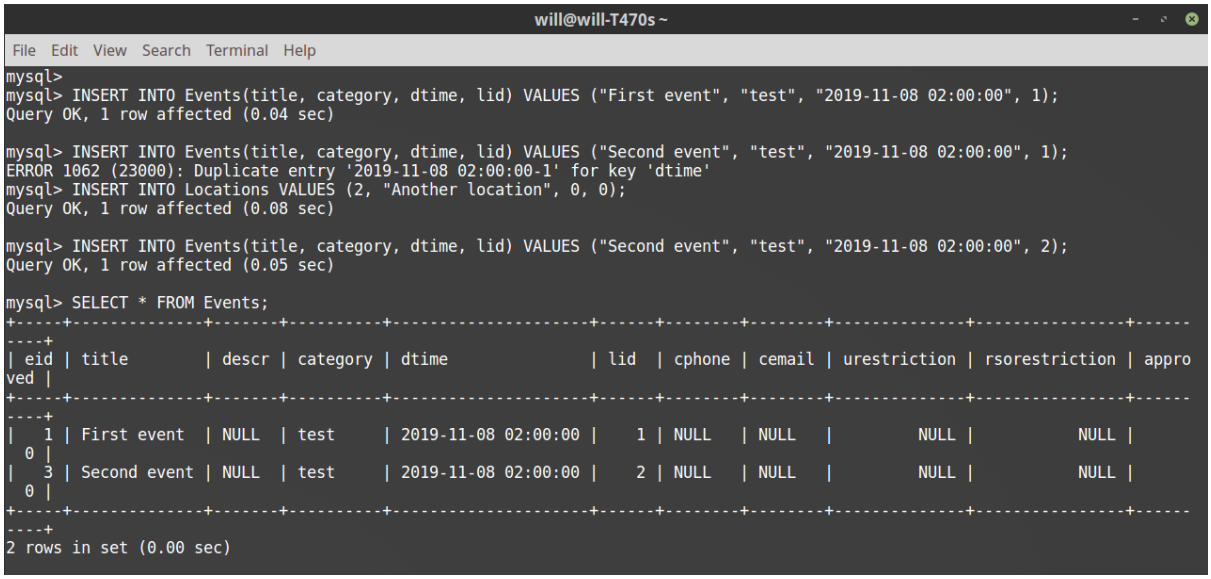

Figure 3: Attempting to insert two events at the same place and time fails, but two events at the same time and different places is allowed

2.3.3 Disabling RSOs with Fewer than 5 Members

Two `TRIGGER` constraints are used to activate RSOs with enough members and deactivate those the too few members. See Figure 4 for a demonstration.

```
CREATE TRIGGER approverso
AFTER INSERT ON RSOMembers
FOR EACH ROW
UPDATE RSOs
SET approved = (SELECT COUNT(*) > 4
                FROM RSOMembers R
                WHERE R.rid = rid)
WHERE RSOs.rid = rid;

CREATE TRIGGER unapproverso
AFTER DELETE ON RSOMembers
FOR EACH ROW
UPDATE RSOs
SET approved = (SELECT COUNT(*) > 4
                FROM RSOMembers R
                WHERE R.rid = rid)
WHERE RSOs.rid = rid;
```

2.3.4 Non-admin Creating RSO Events

This constraint is enforced by the application rather than the database. The GUI presents a list containing only the current user's RSOs and Universities to restrict the scope of the event. So, a non-admin can't create an event on behalf of an RSO. See Figure 5 for an example.

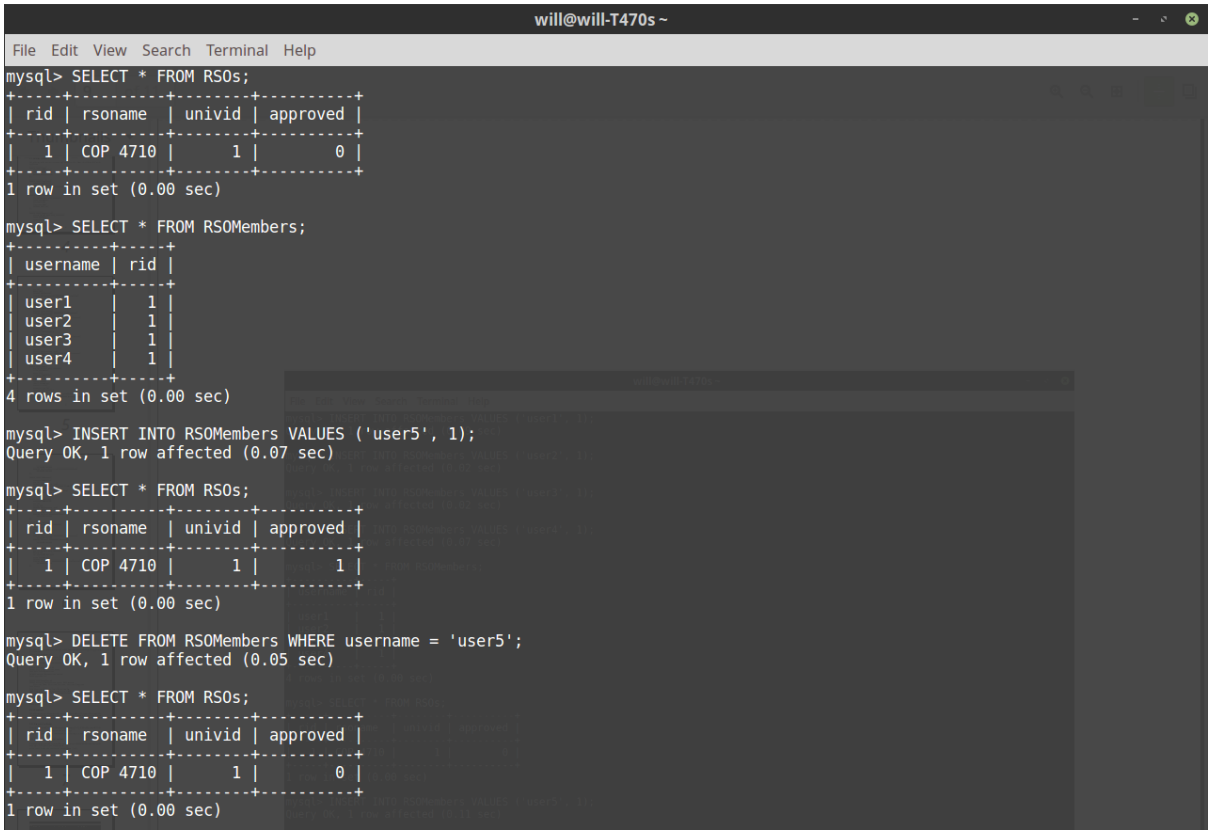

Figure 4: The RSO is activated when the fifth member is added and deactivated when he/she is removed.

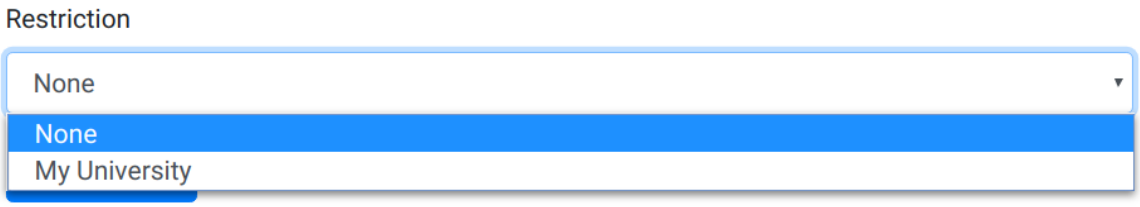

Figure 5: The current user does not administer any RSOs, so they cannot create an RSO event, but they can request to create a private event for their university.

2.4 Example Queries

In this section, I provide some examples of embedded queries from the backend application that demonstrate how the database is used in practice. The string `%s` indicates that the backend application will populate that field based on the current application state.

2.4.1 Fetch All RSO Events

```
SELECT * FROM ApprovedEvents
WHERE rsorestriction IN (
    SELECT rid FROM RSOMembers
    WHERE username = %s);
```

This query gets all events that have been created for an approved RSO where the current user has membership in that RSO. The design of the representation of Events lends itself to convenient querying of Events and filtering by access restrictions.

2.4.2 Create a New Location

```
INSERT INTO Locations(lname, latitude, longitude)
VALUES (%s, %s, %s)
```

Although the database enforces the uniqueness of location names, it automatically determines a unique primary key that identifies the location in the rest of the database. Representing locations in a separate tables allows the client to easy query (for example) all events at some location:

```
SELECT *
FROM ApprovedEvents
WHERE lid = %s;
```

2.4.3 Update and Access University Photo Albums

More detail on the visual design of this element is provided in Section 4.5.

```
INSERT INTO Photos(univid, b64, ftype)
VALUES (%s, %s, %s)
```

This query adds an uploaded photo (converted to base 64) to an album for a specific university. It is equally easy to fetch an entire album from the application:

```
SELECT *
FROM Photos
WHERE univid=%s;
```

3 Graphical User Interface

The following series of figures demonstrate a typical path through the application from a new user's perspective.

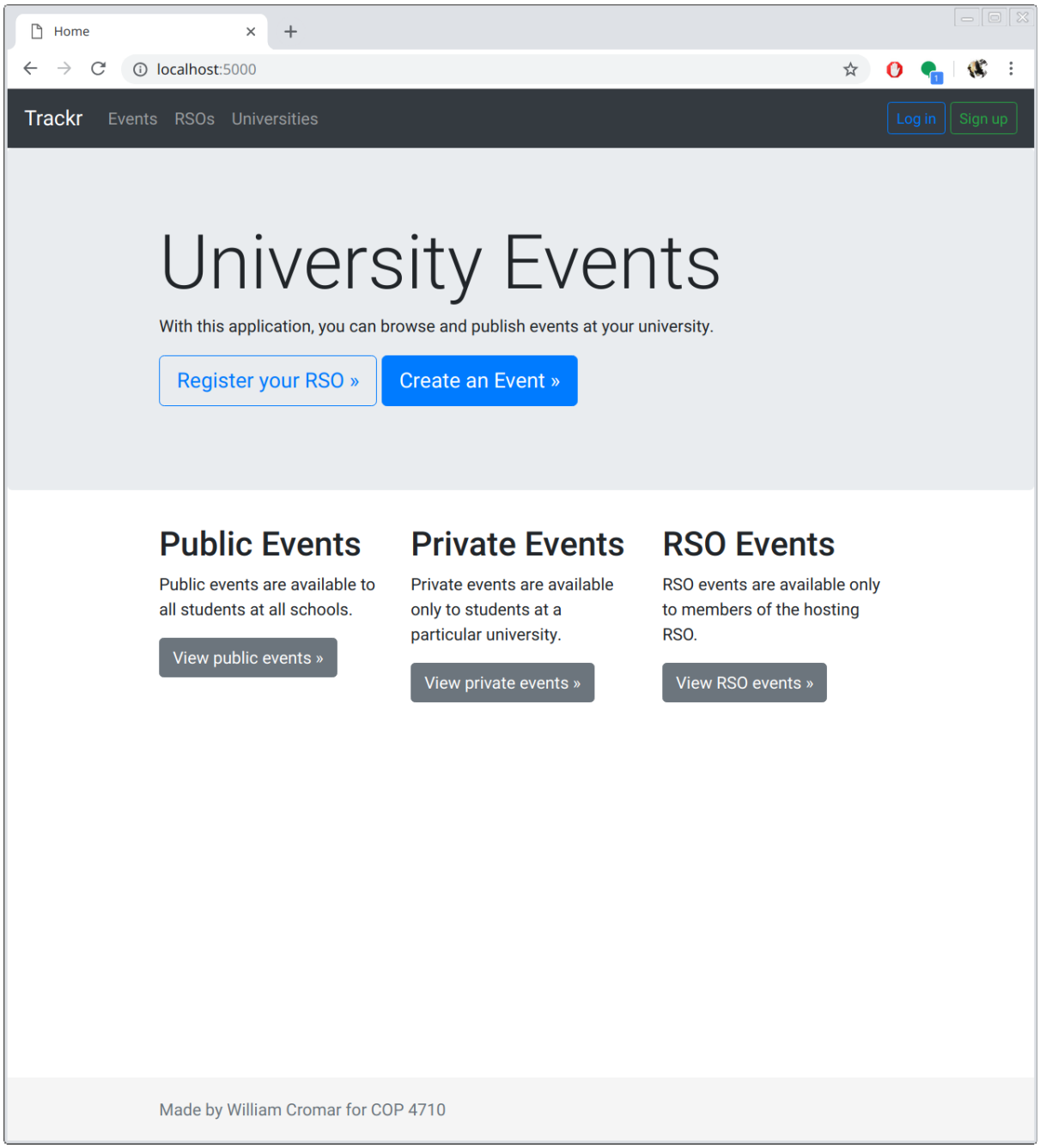

Figure 6: A new user starts on the home page of the application.

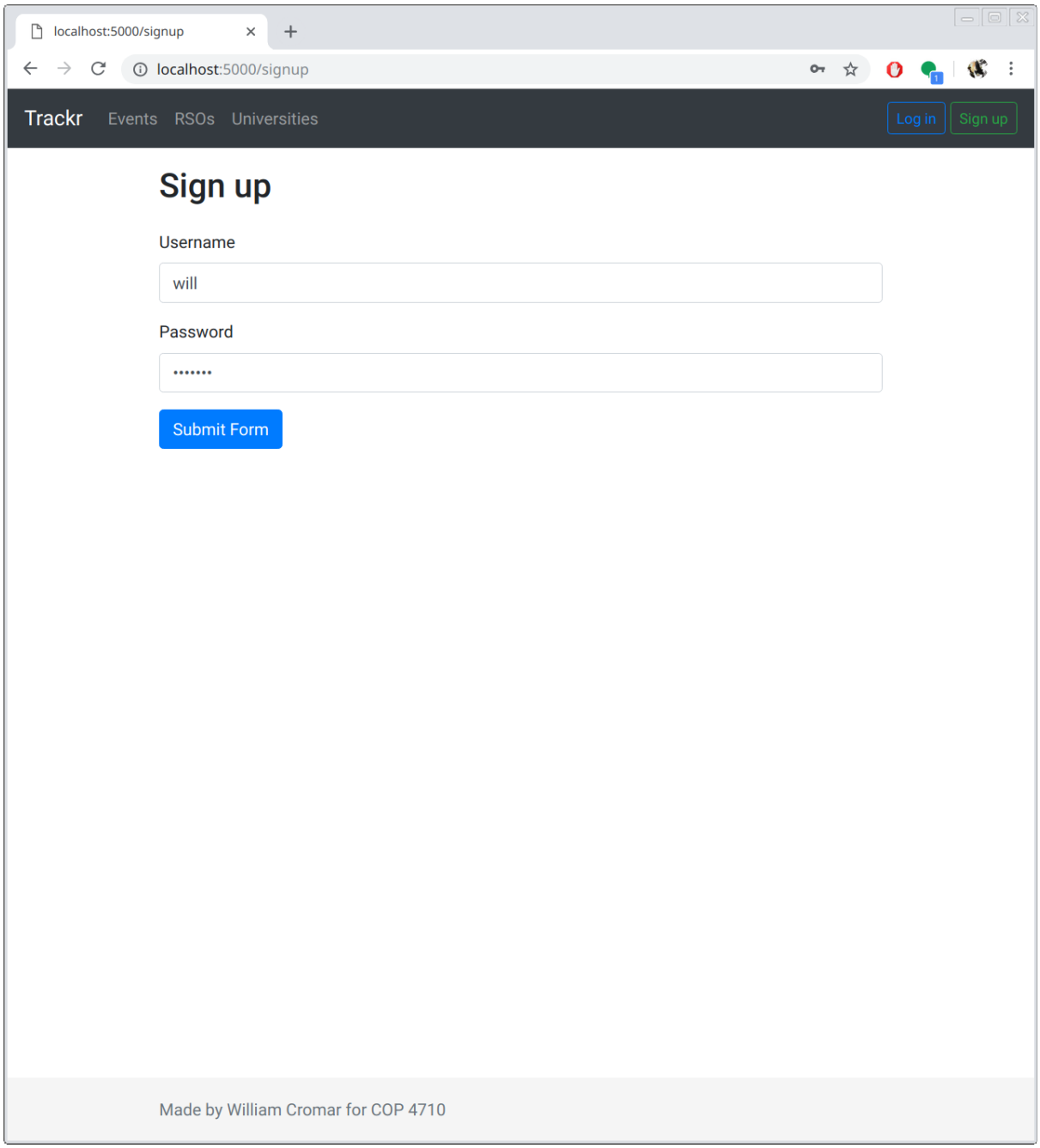

Figure 7: The new user registers with a username (`will`) and password.

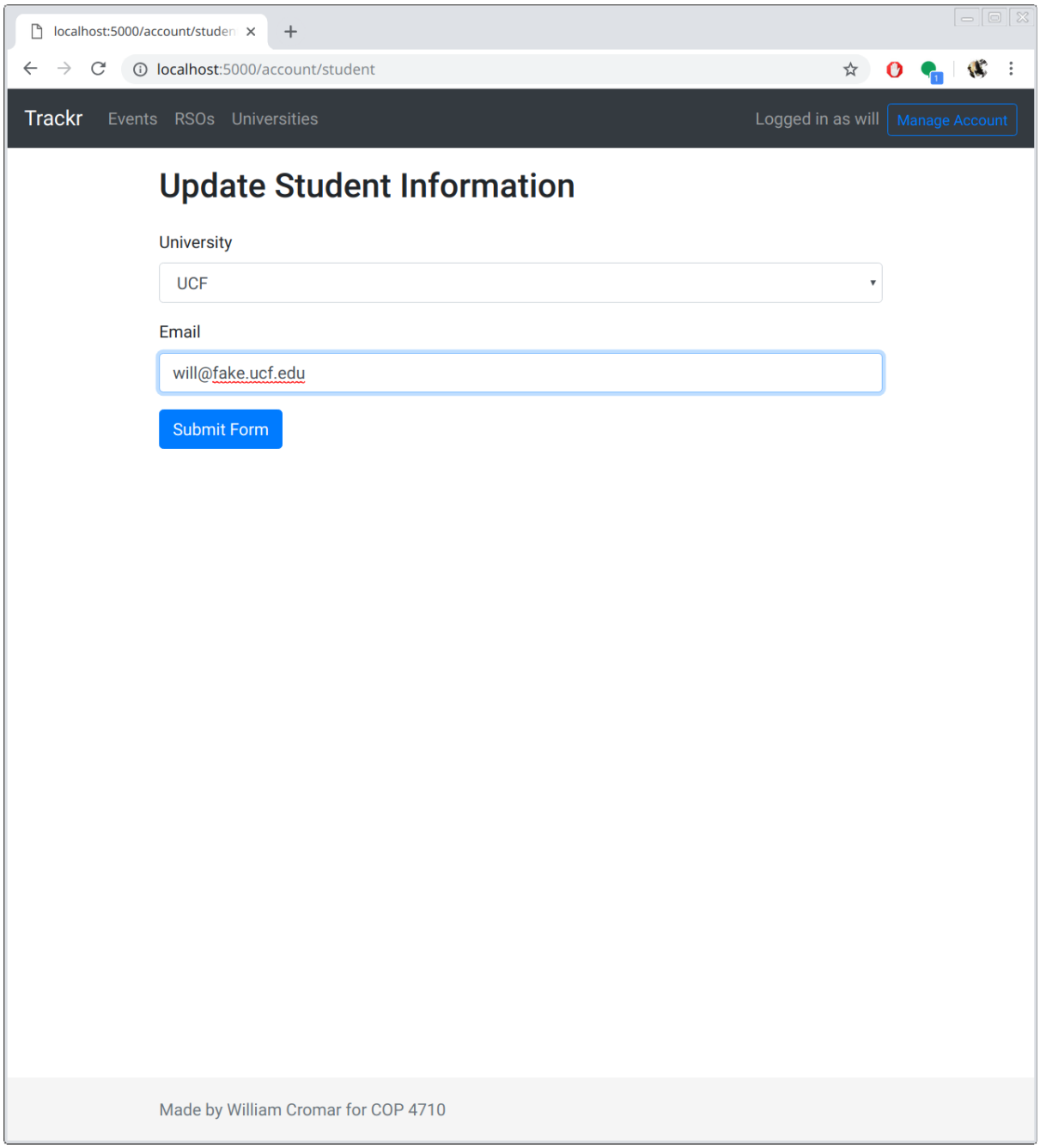

Figure 8: The user selects their school and university e-mail address.

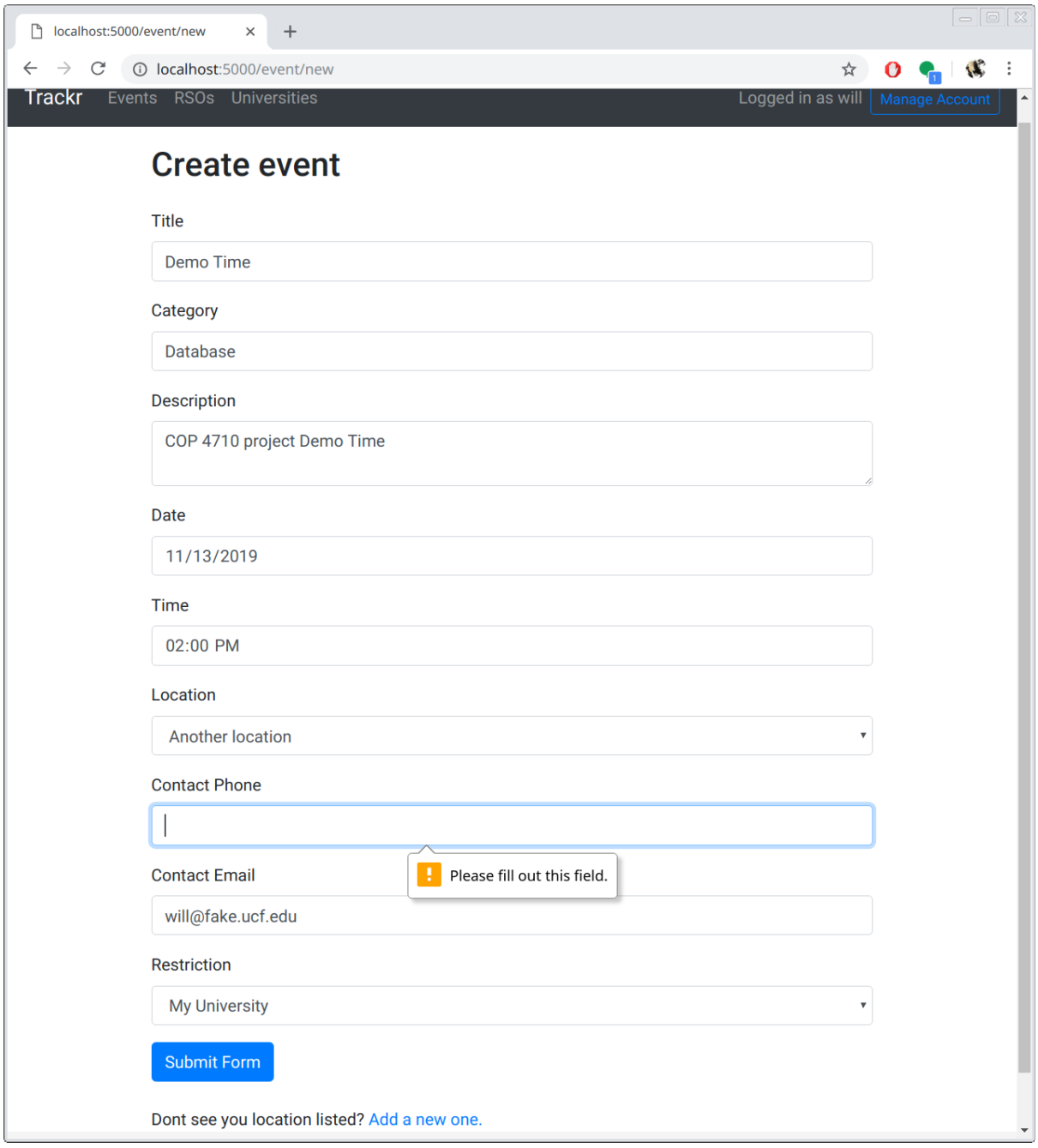

Figure 9: Any student can request to create an event private to their university. Note that the form also has client-side input validation.

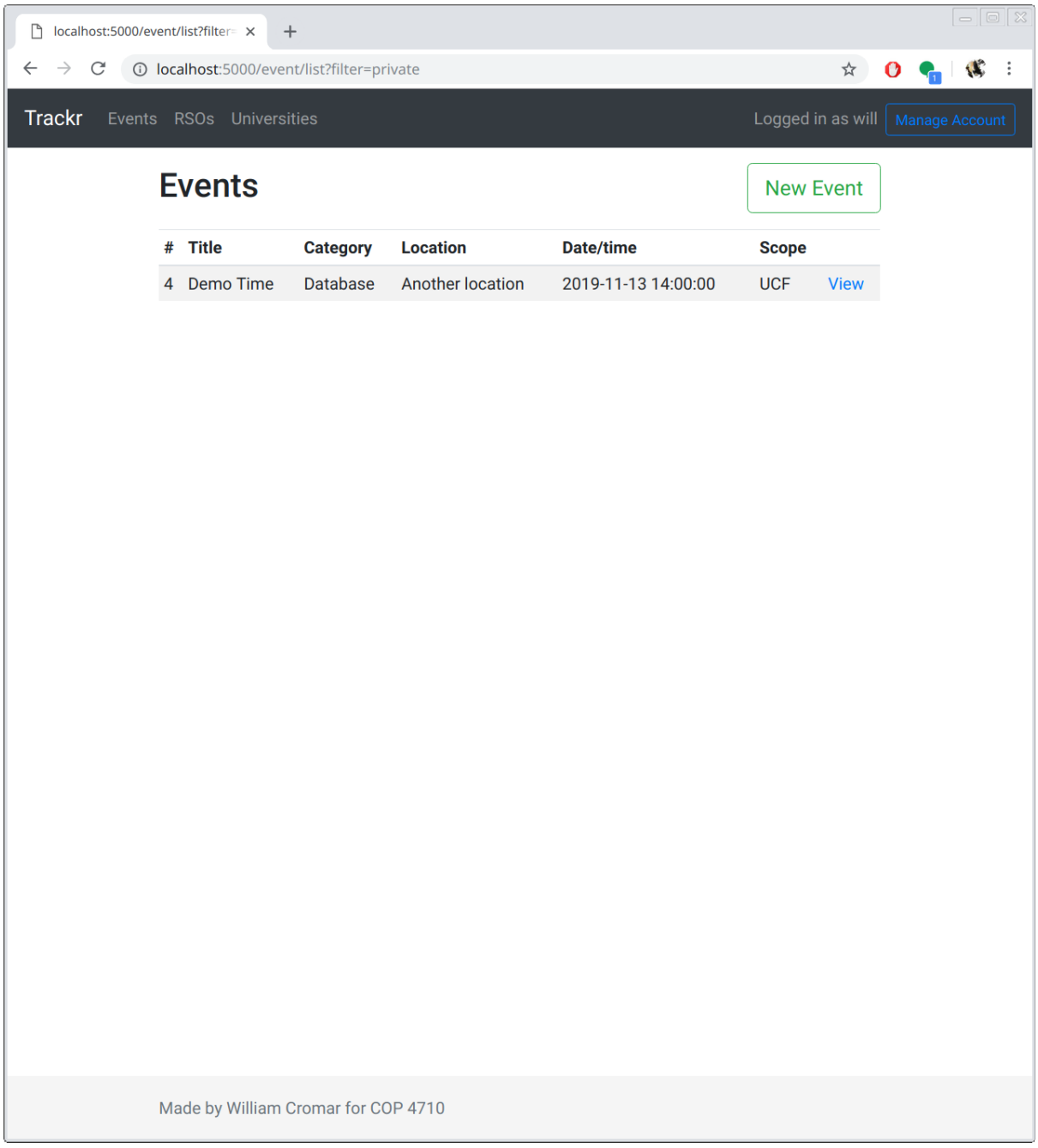

Figure 10: Once a super-admin approved the private event, the user can see it in their list of events

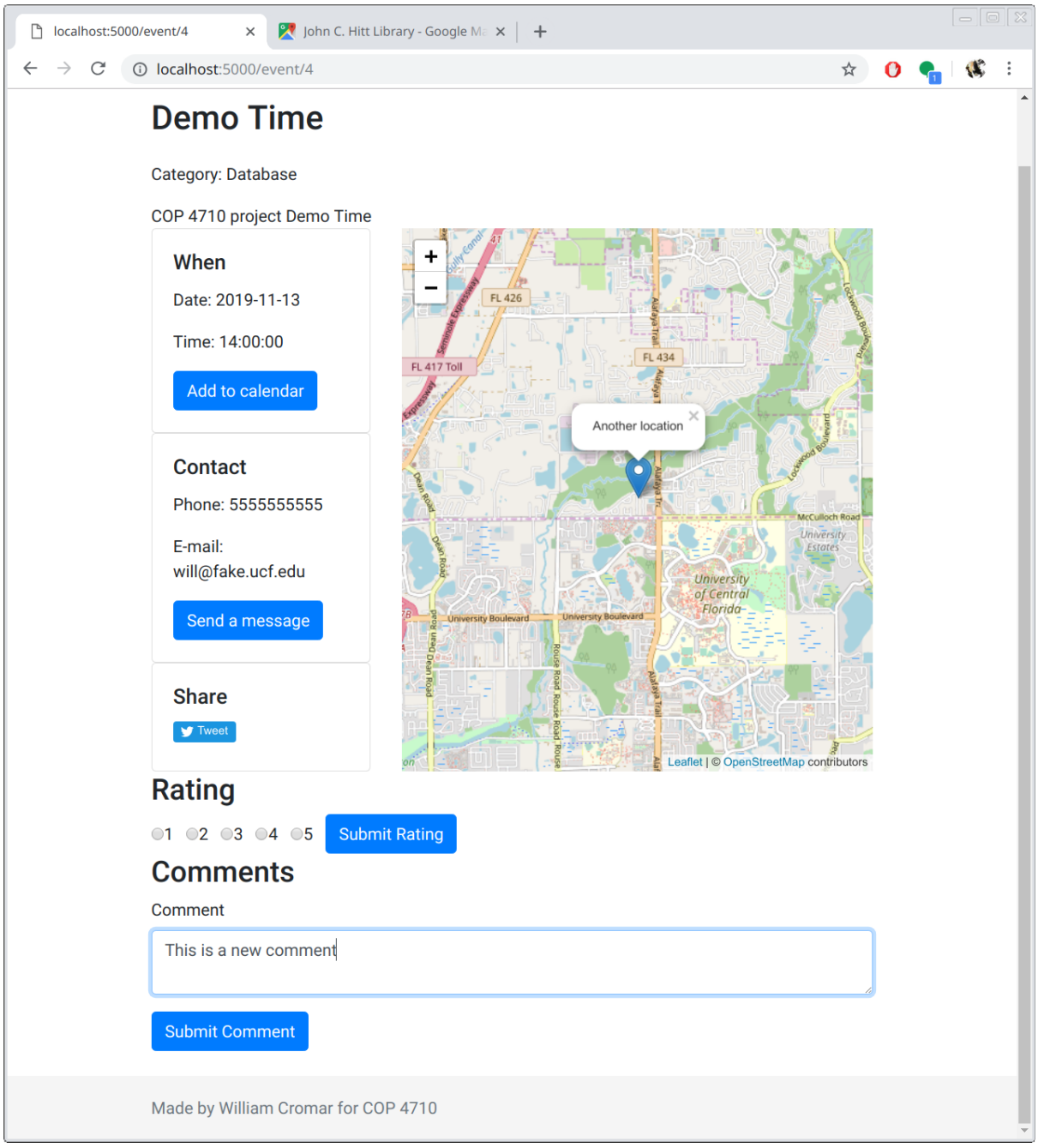

Figure 11: The user can click "View" to see all of the details of the event, as well as leave a comment

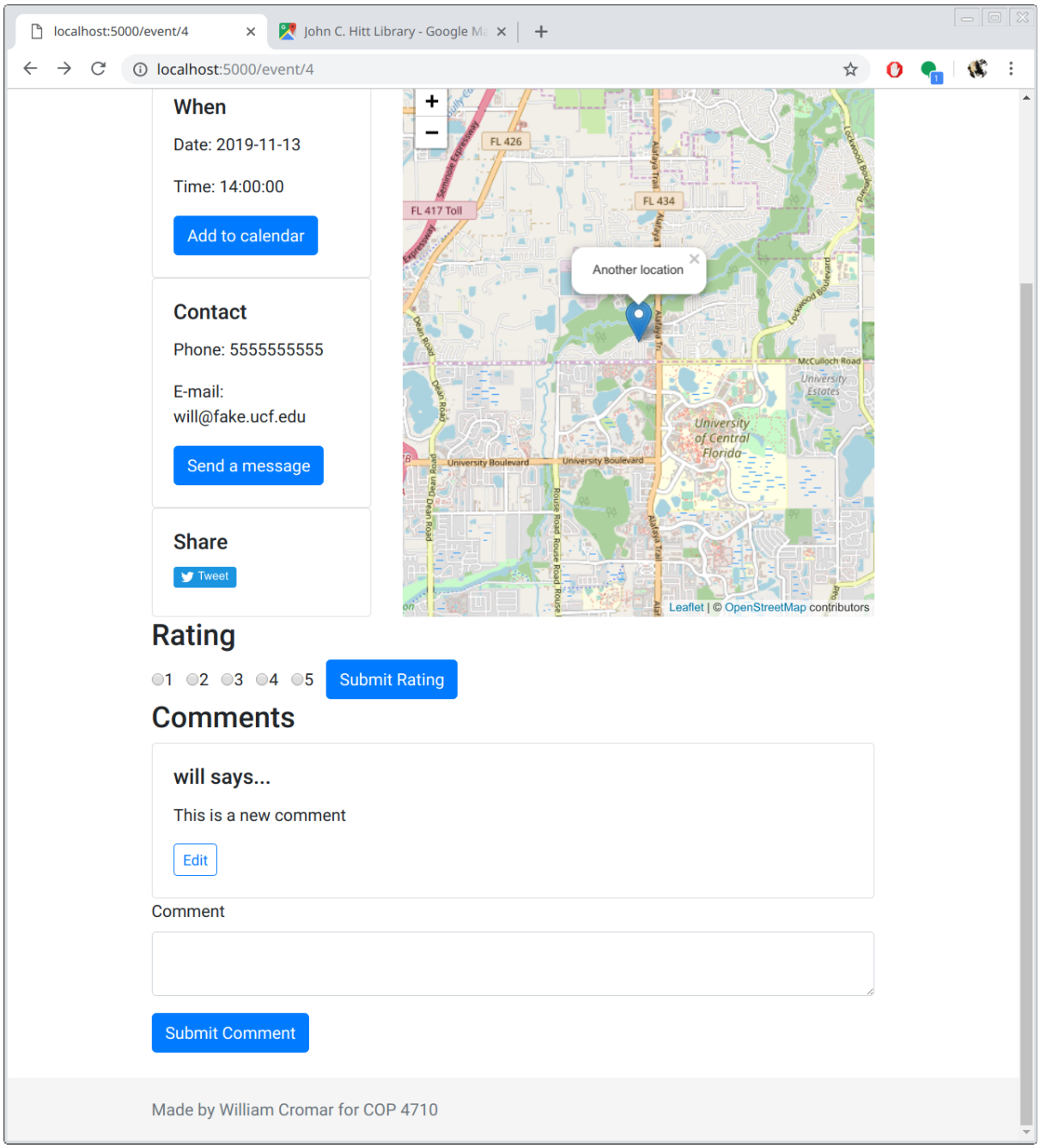

Figure 12: Users also have the option to edit their comments.

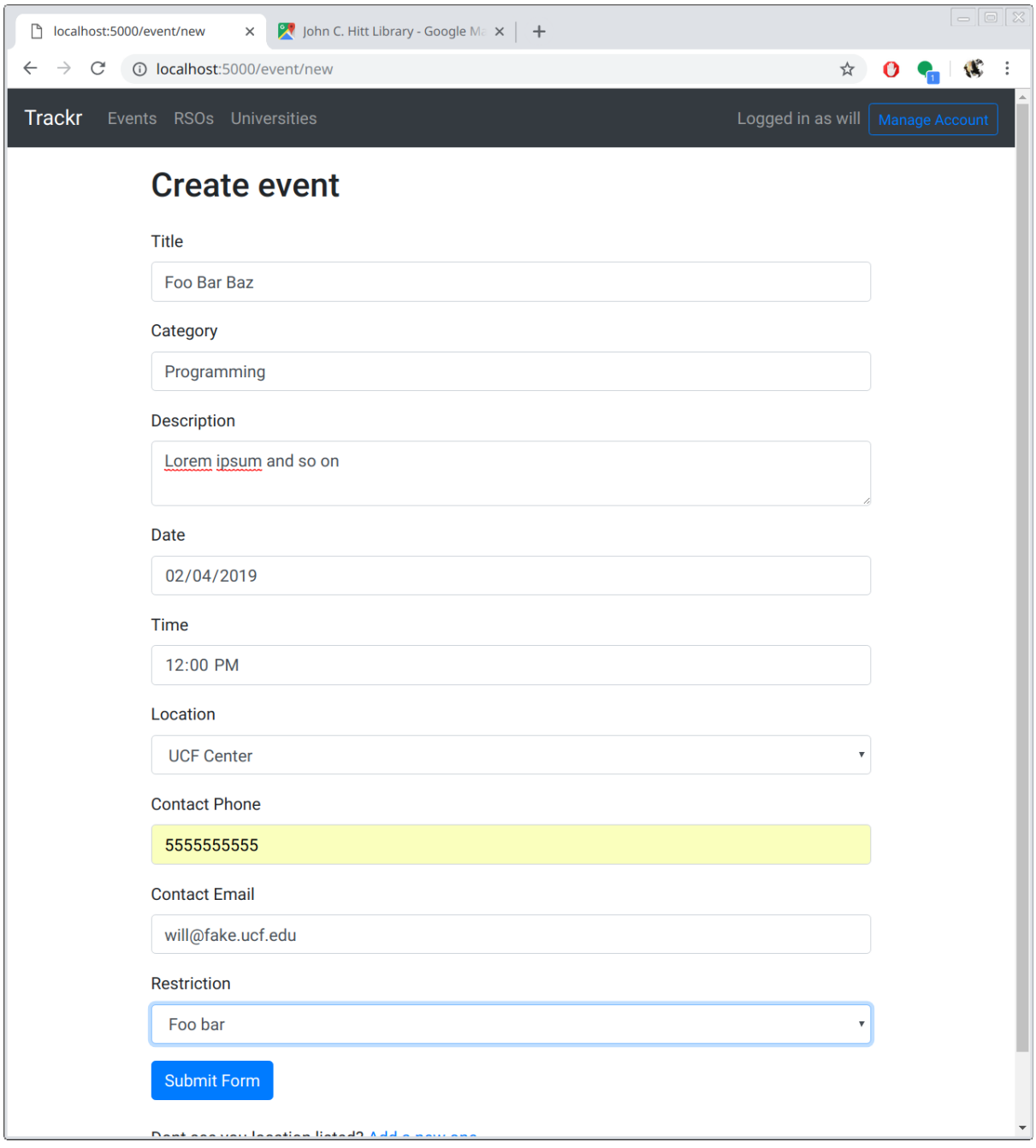

Figure 13: The new user can request to create an RSO with 5 of their friends and will become that RSO's first admin

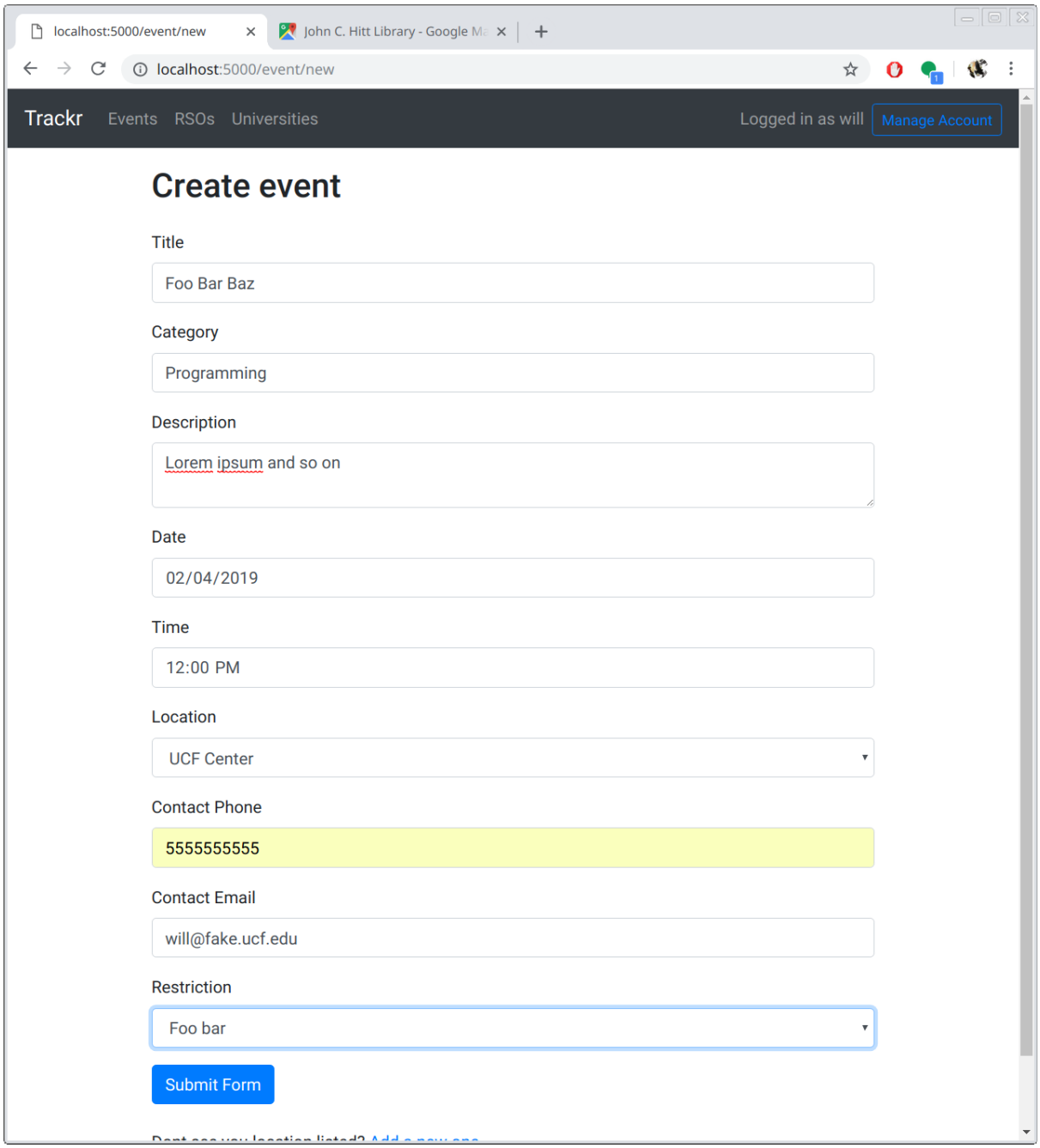

Figure 14: Now that the user is an admin of an RSO, they can create an RSO event for it.

4 Non-essential Features

The central part of the project is the database, and the backend application provides a thin layer that lets users securely access it. In this section, I describe features that go beyond the minimum requirements to make the database functional and accessible and make it easier to use the application.

4.1 Interface Design

I went out of my way to may the website intuitive and visually appealing. Some screenshots are provided below.

screenshots

4.2 UCF Event Feed Scraping

A Python script is included in the `scripts/` directory that fetches a week worth of events from the official UCF events website and generate the `INSERT` statements to adapt the data to my database design. This can be used to populate the database with real test data.

4.3 Social Networking

All event details pages include an option to share the event on Twitter.

screenshot

4.4 Interactive Maps

Event details pages all feature an interactive map that lets users see the area around where an event is scheduled. Additionally, new events can be created by selecting the name of an existing location from a list, but users can also add new locations to host their event at using an interactive map.

screenshots

4.5 University Album