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.

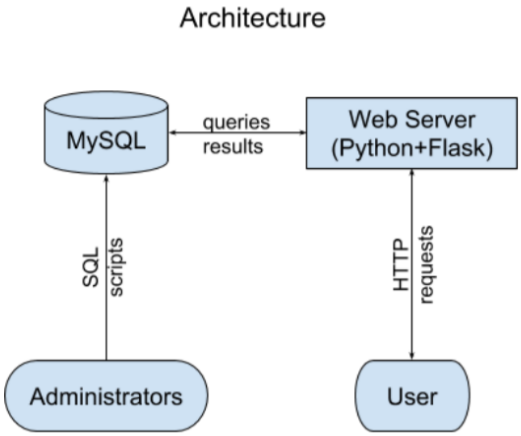


Figure 1: Overview of application's architecture.

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. This section also introduced some new constraints and generalizes project requirements to make the application more realistic.

There are multiple valid ways to design the database, and my final design reflects my own personal experiences and preferences in software engineering. As much as possible, I design the database schema to be easy to mutate without introducing anomalies and write views to make querying the database easier for the client.

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:

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 they 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.

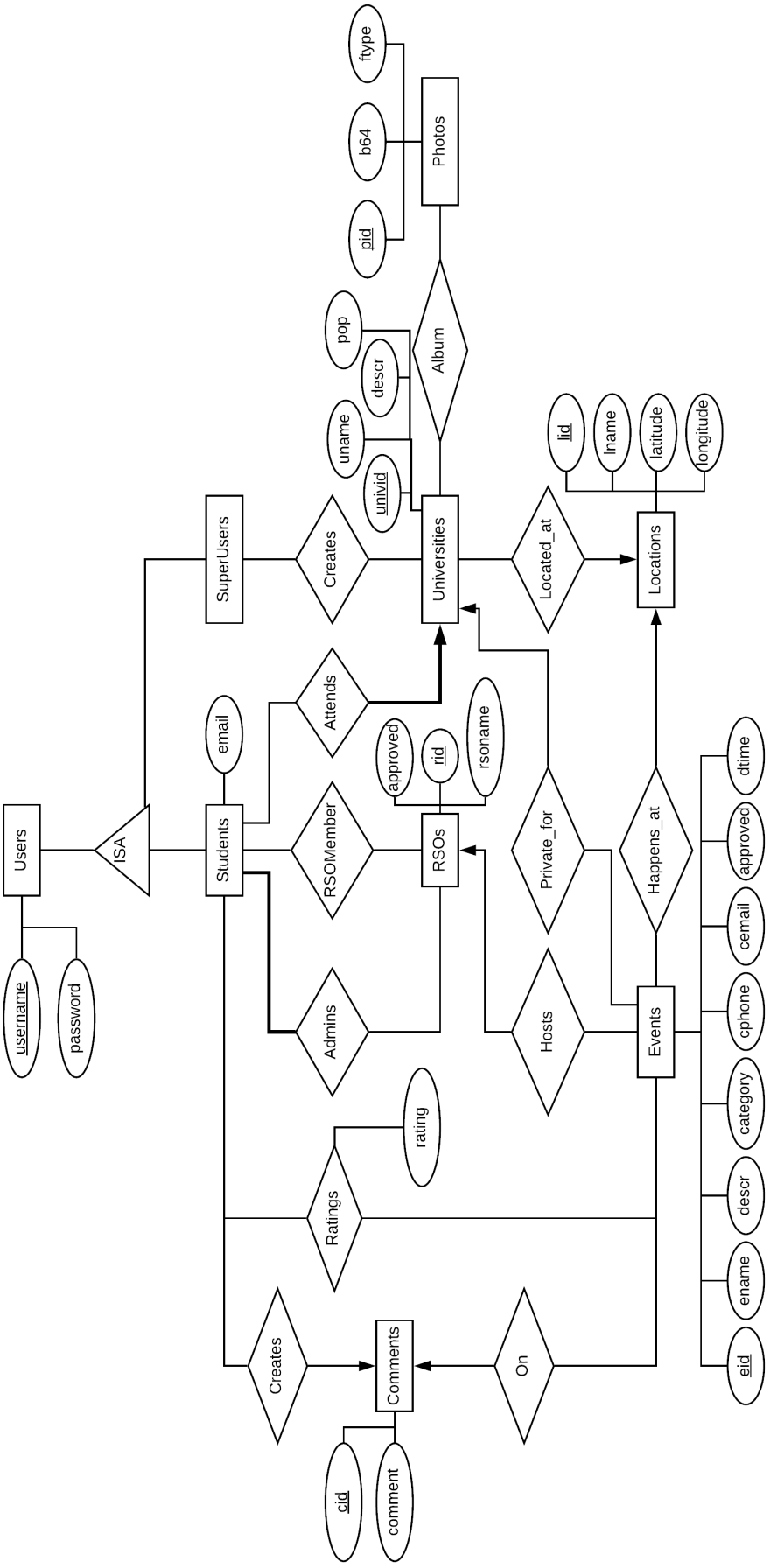


Figure 2: ER Model for events website. (rotated)

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).

2.2 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);
```

2.3.3 Activating and Deactivating RSOs

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)
```

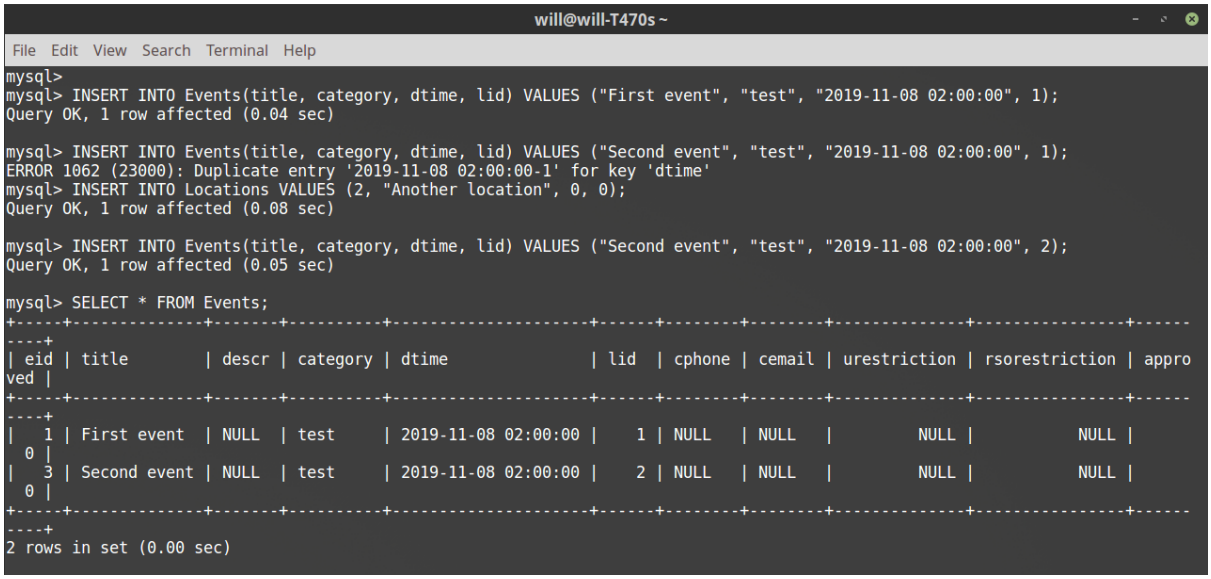


Figure 3: Attempting to insert two events at the same place and time fails, but two events at the same time and different places is allowed

```
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.

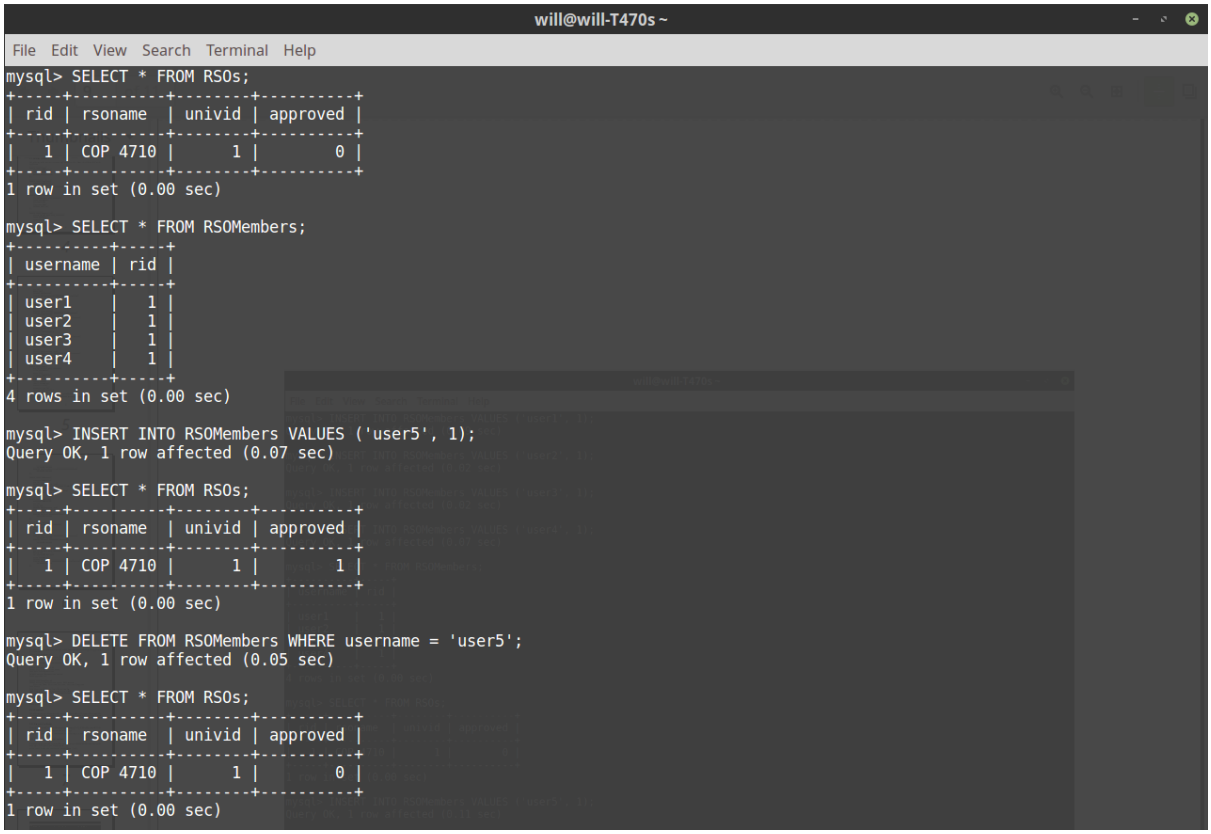


Figure 4: The RSO is activated when the fifth member is added and deactivated when they are removed.

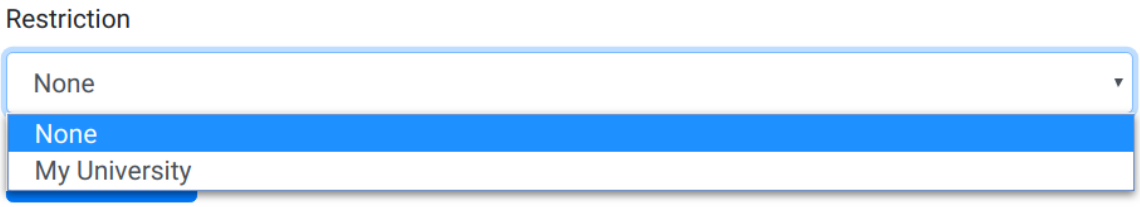


Figure 5: The current user does not administer any RSOs, so they cannot create an RSO event, but they can request to create a private event for their university.

2.4 SQL Examples

Example test data that can be used to initialize the database can be found in `scripts/init_data.sql`. The test data there is used for the example queries. The following figures show examples of SQL queries and results that demonstrate how the database is used in practice

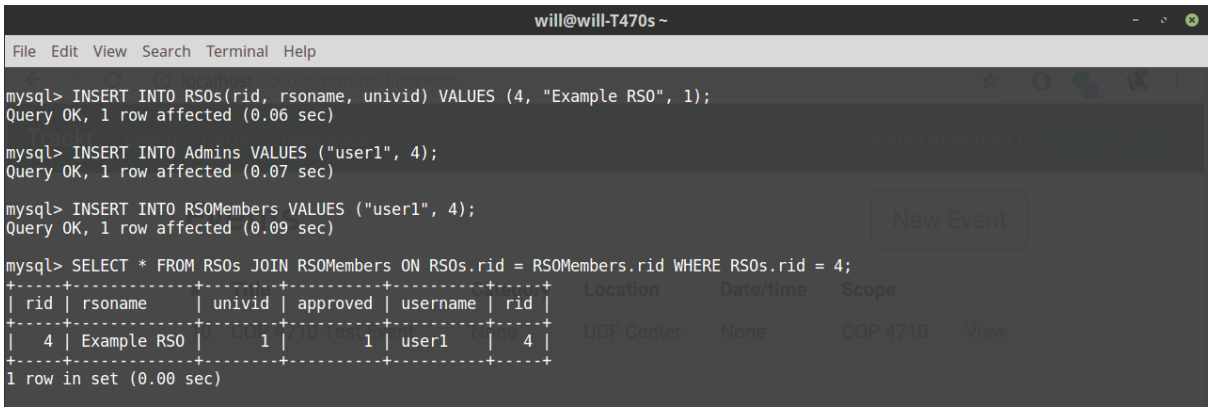


Figure 6: An example of creating a new RSO and adding a member and admin to it.

Figure 7: An example of creating a new event, adding a comment to it, and editing that comment. (rotated)

Figure 8: Examples of querying public, RSO, and private events (respectively) that are visible to `user1`. (rotated)

3 Graphical User Interface

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

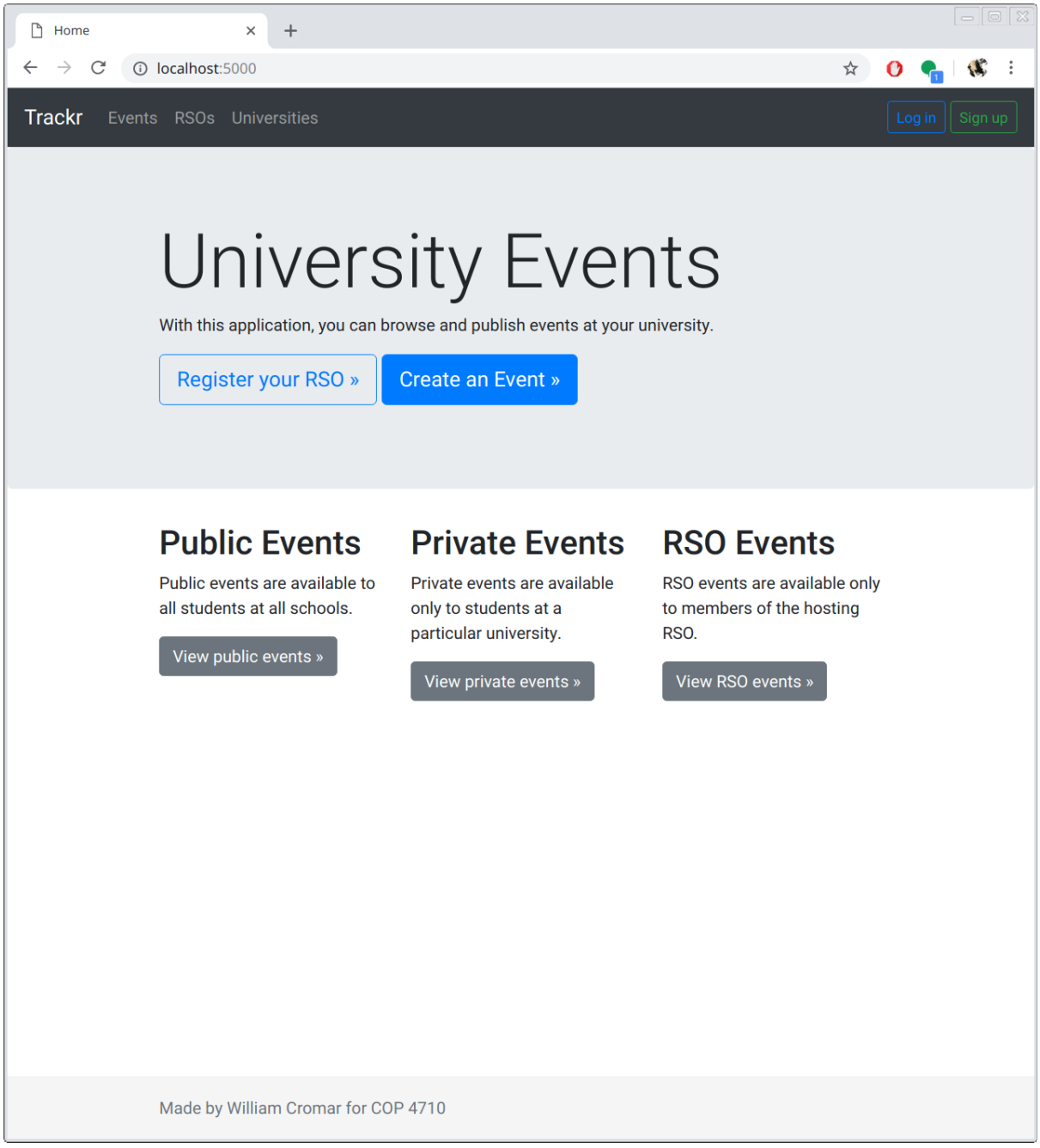


Figure 9: A new user starts on the home page of the application.

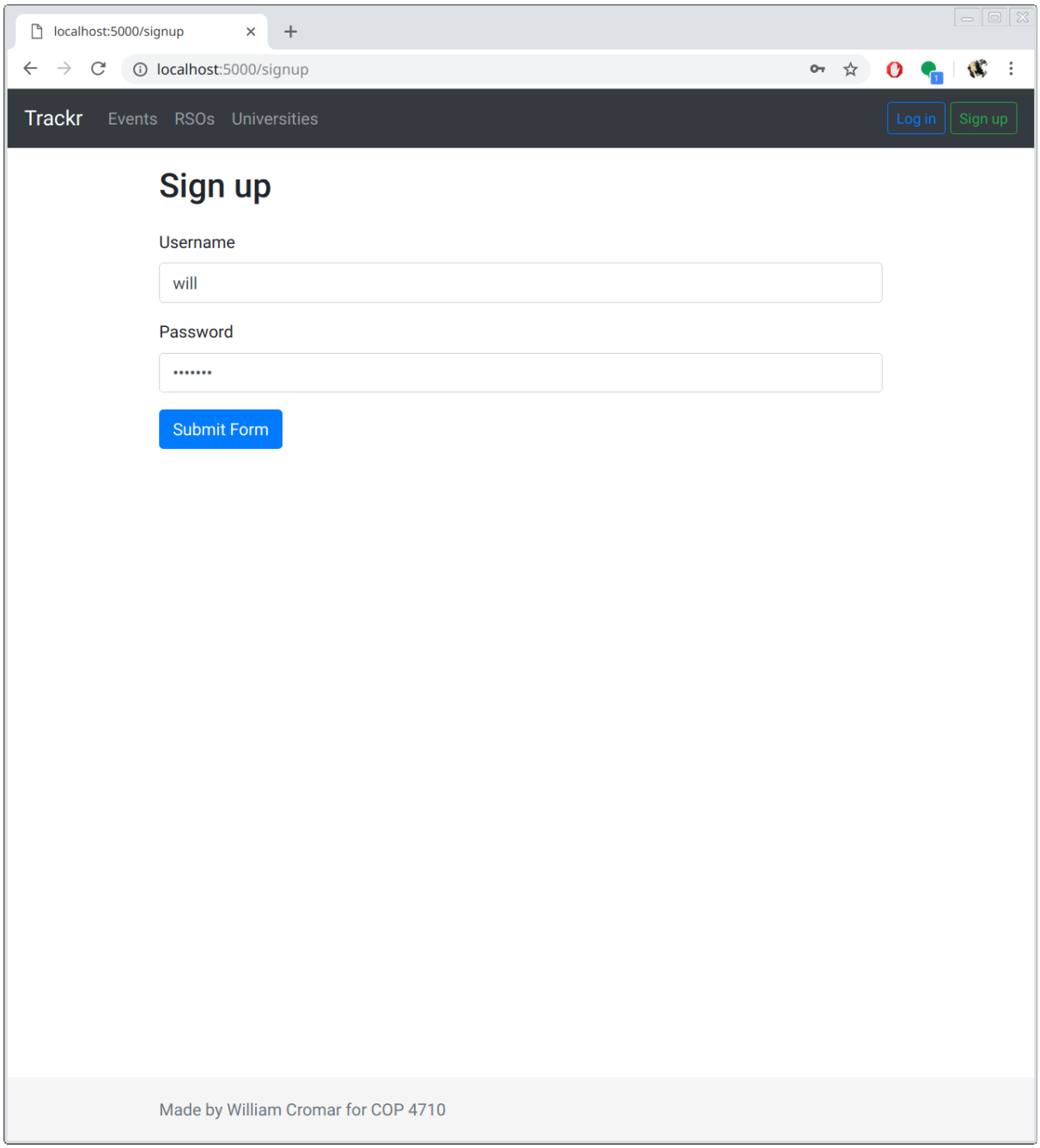


Figure 10: The new user registers with a username (`will`) and password.

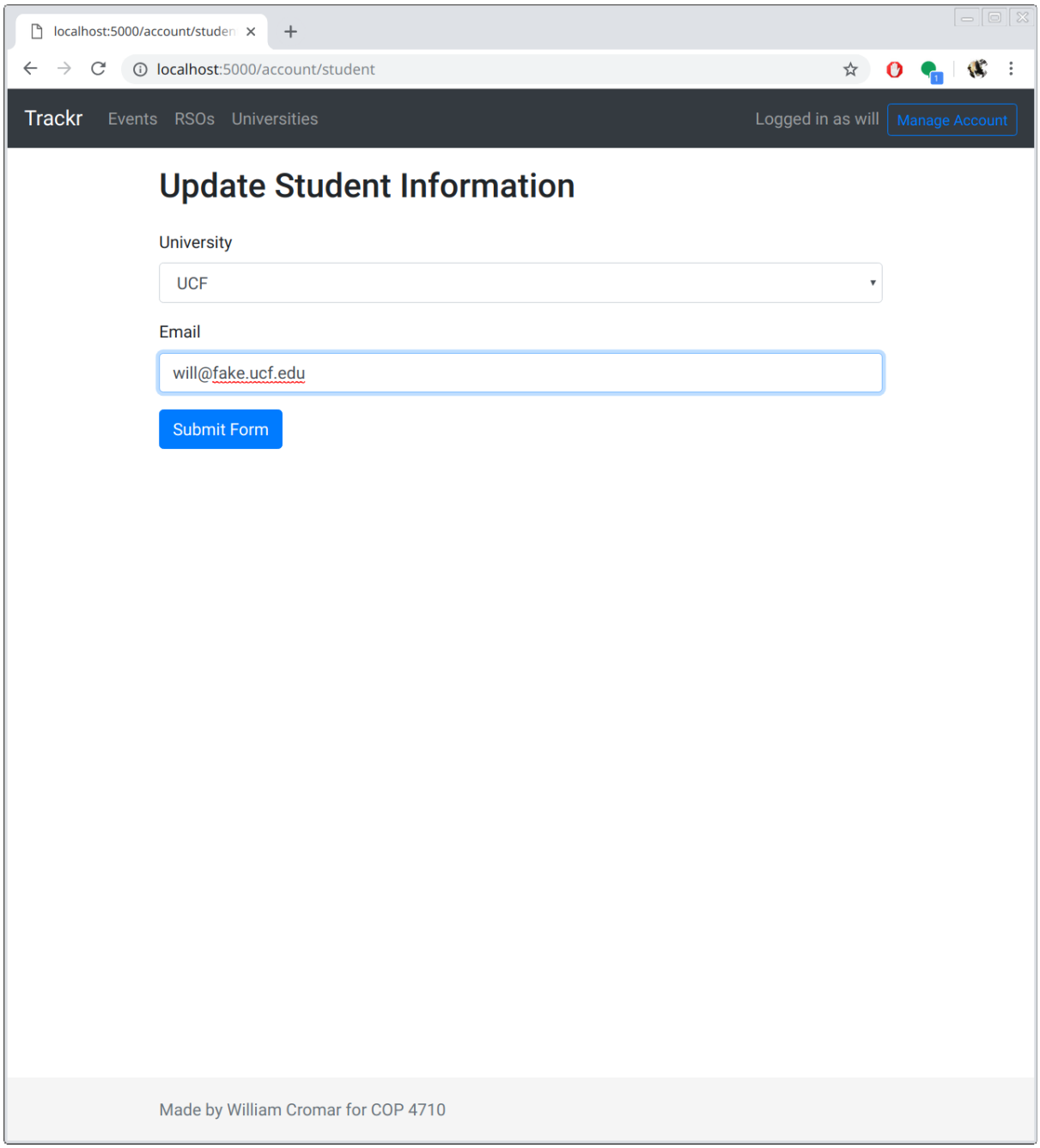


Figure 11: The user selects their school and university e-mail address.

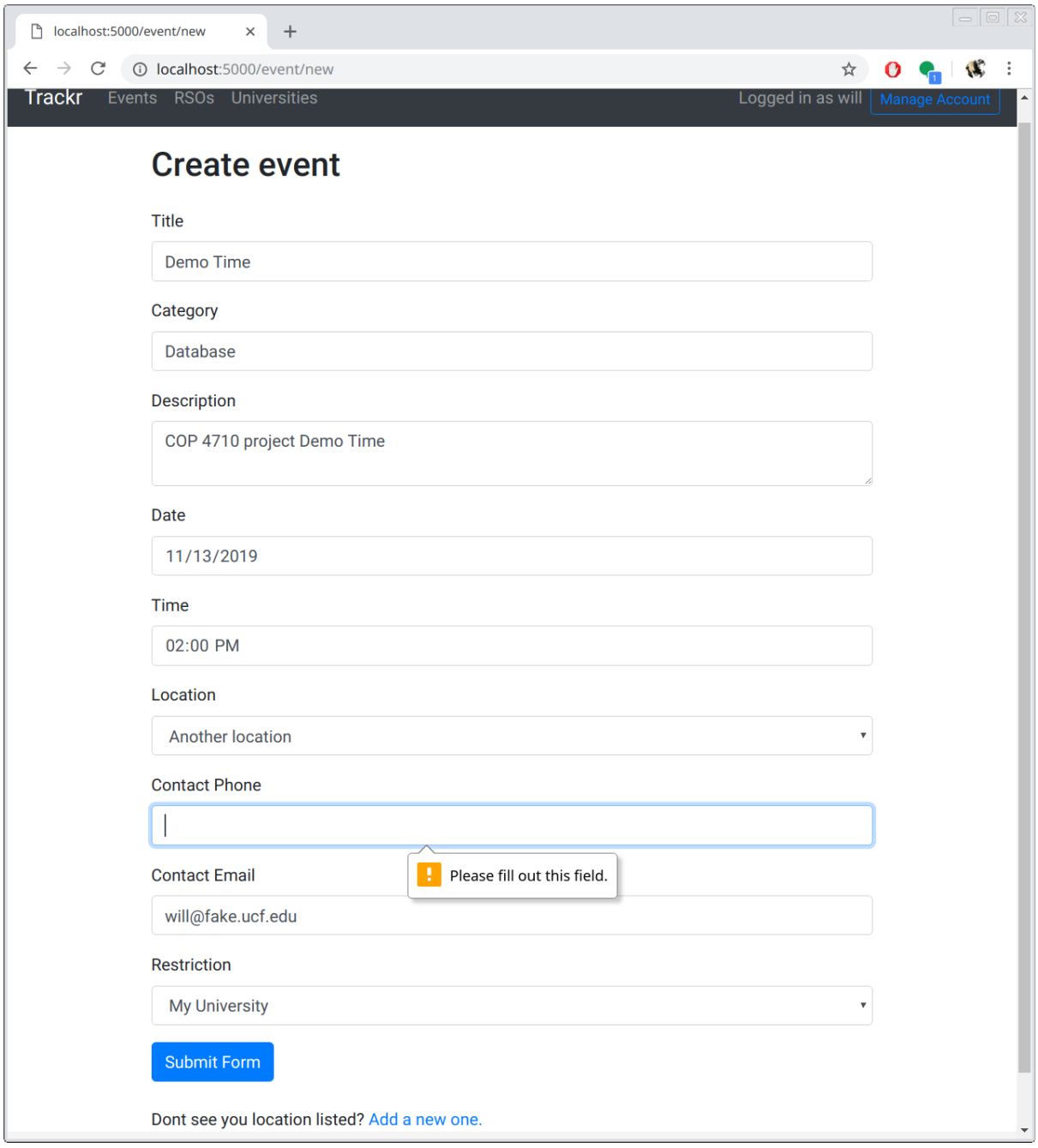


Figure 12: Any student can request to create an event private to their university. Note that the form also has client-side input validation.

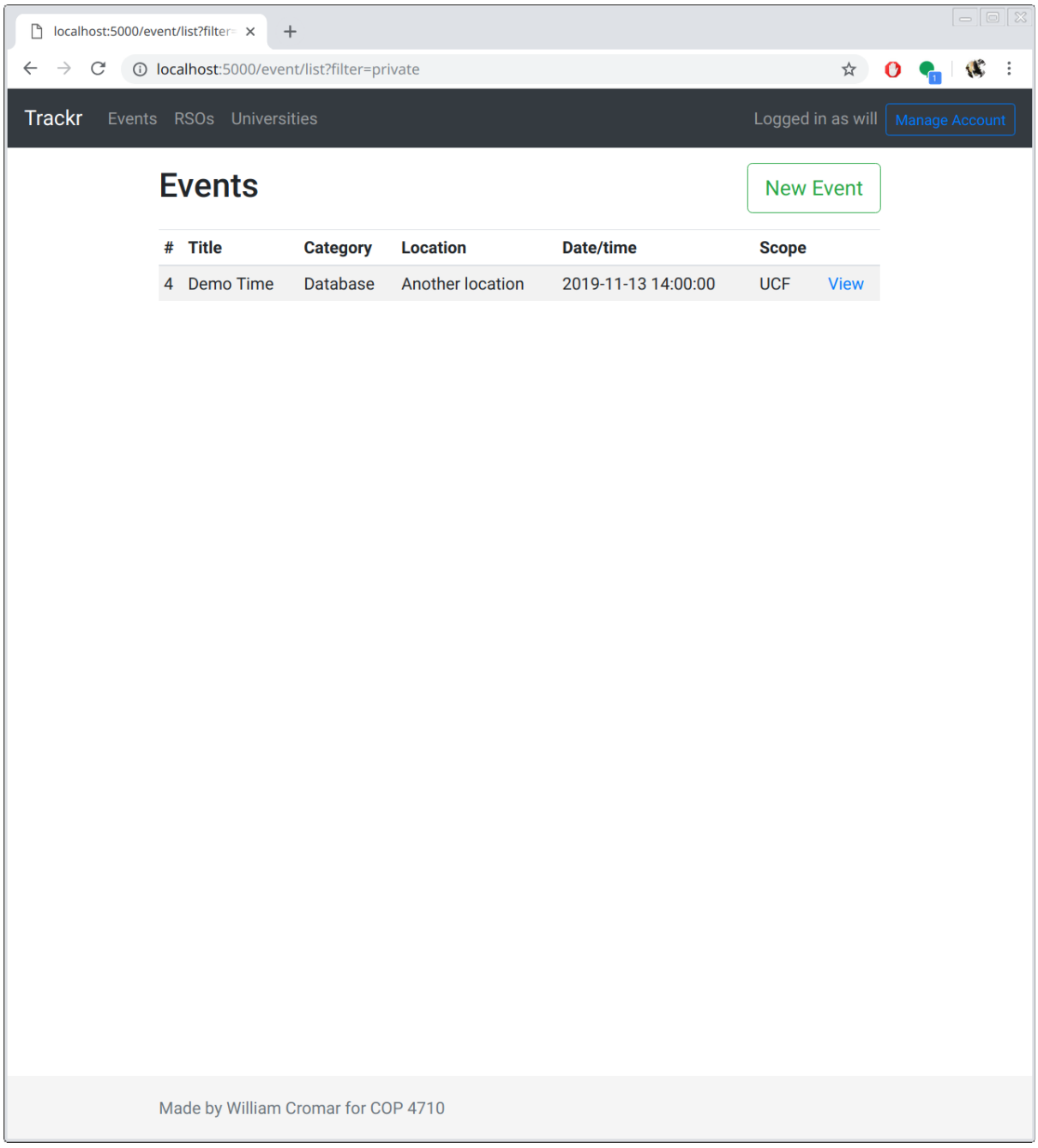


Figure 13: Once a super-admin approved the private event, the user can see it in their list of events.

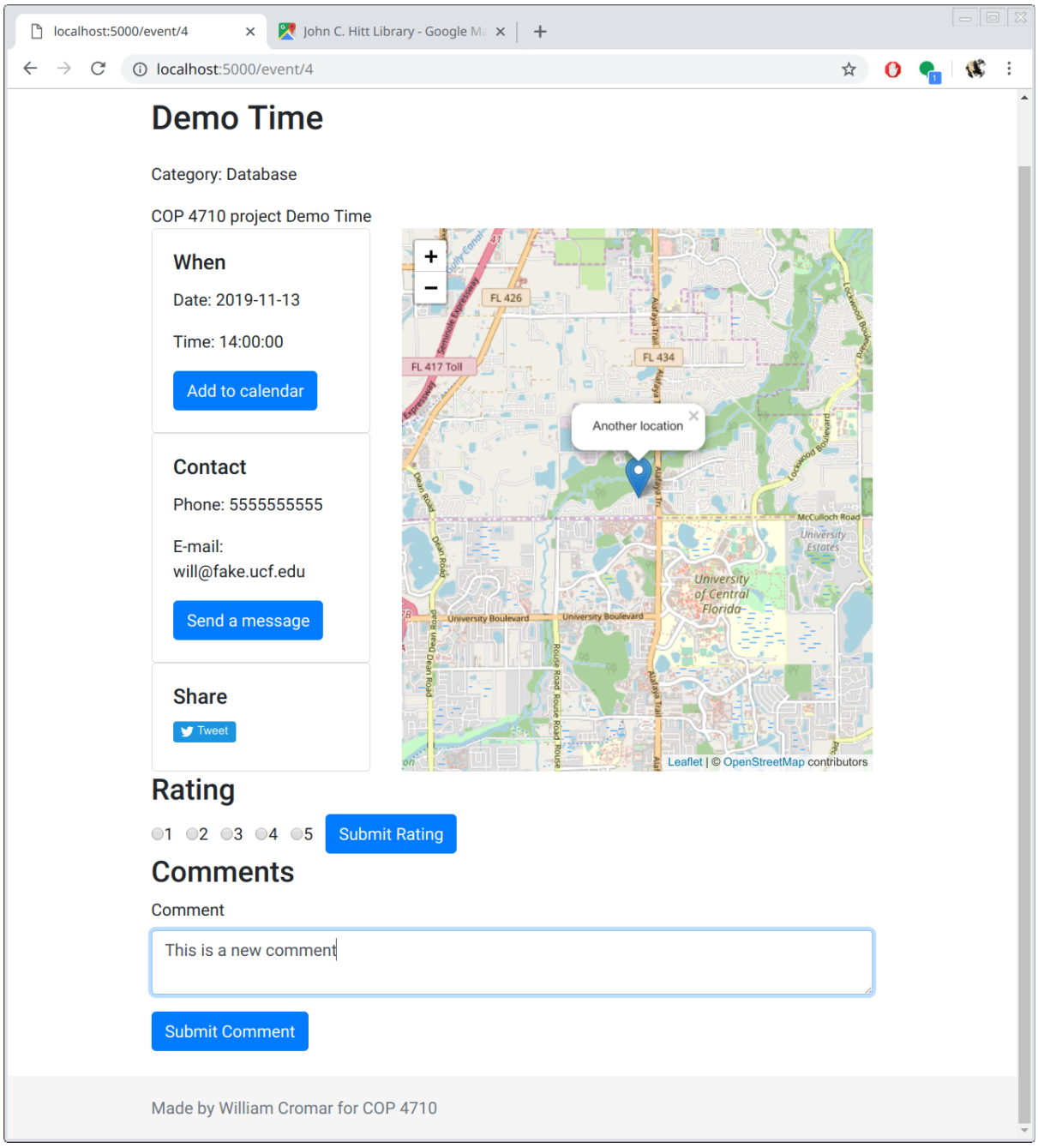


Figure 14: The user can click "View" to see all of the details of the event, as well as leave a comment.

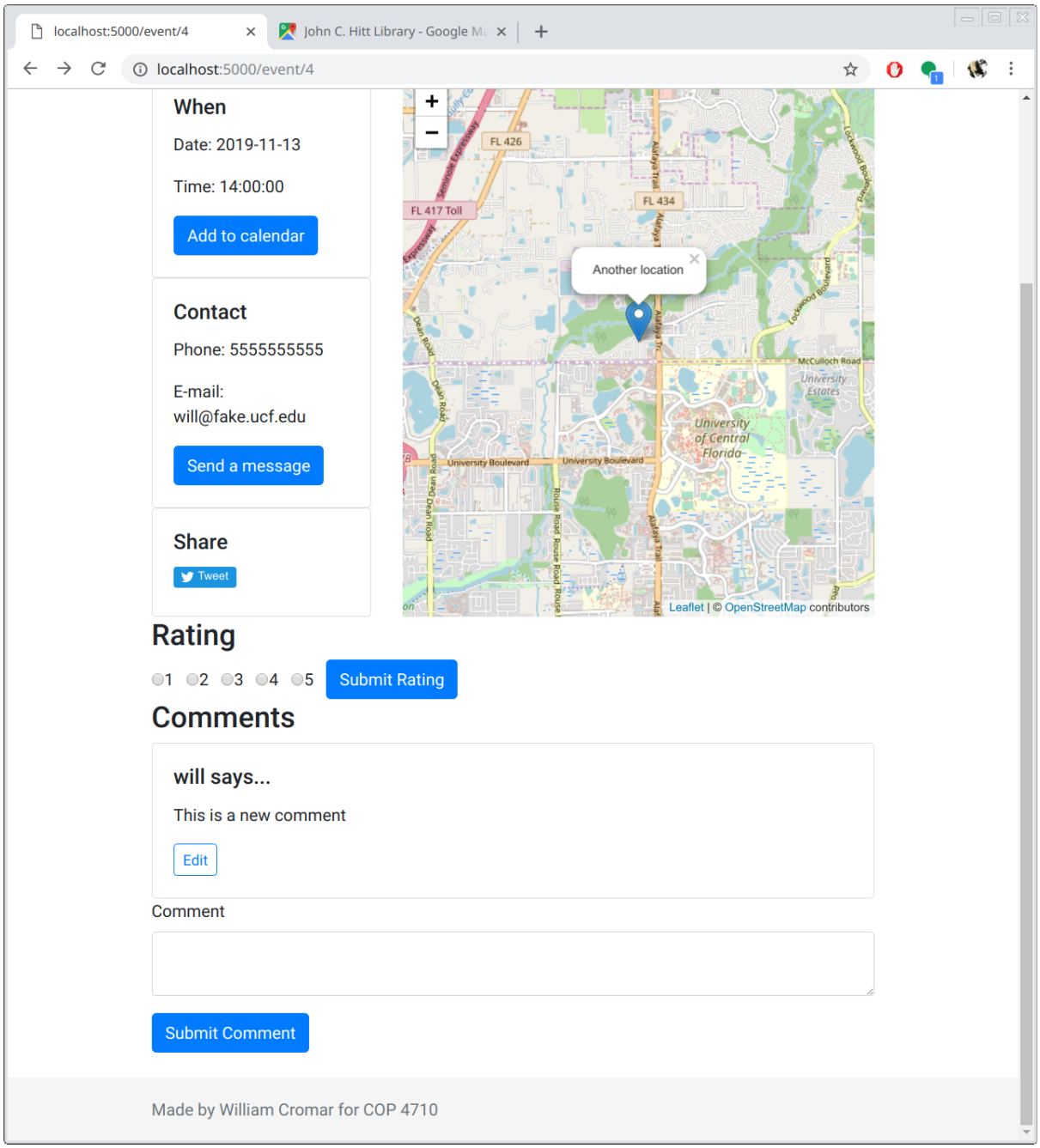


Figure 15: Users also have the option to edit their comments.

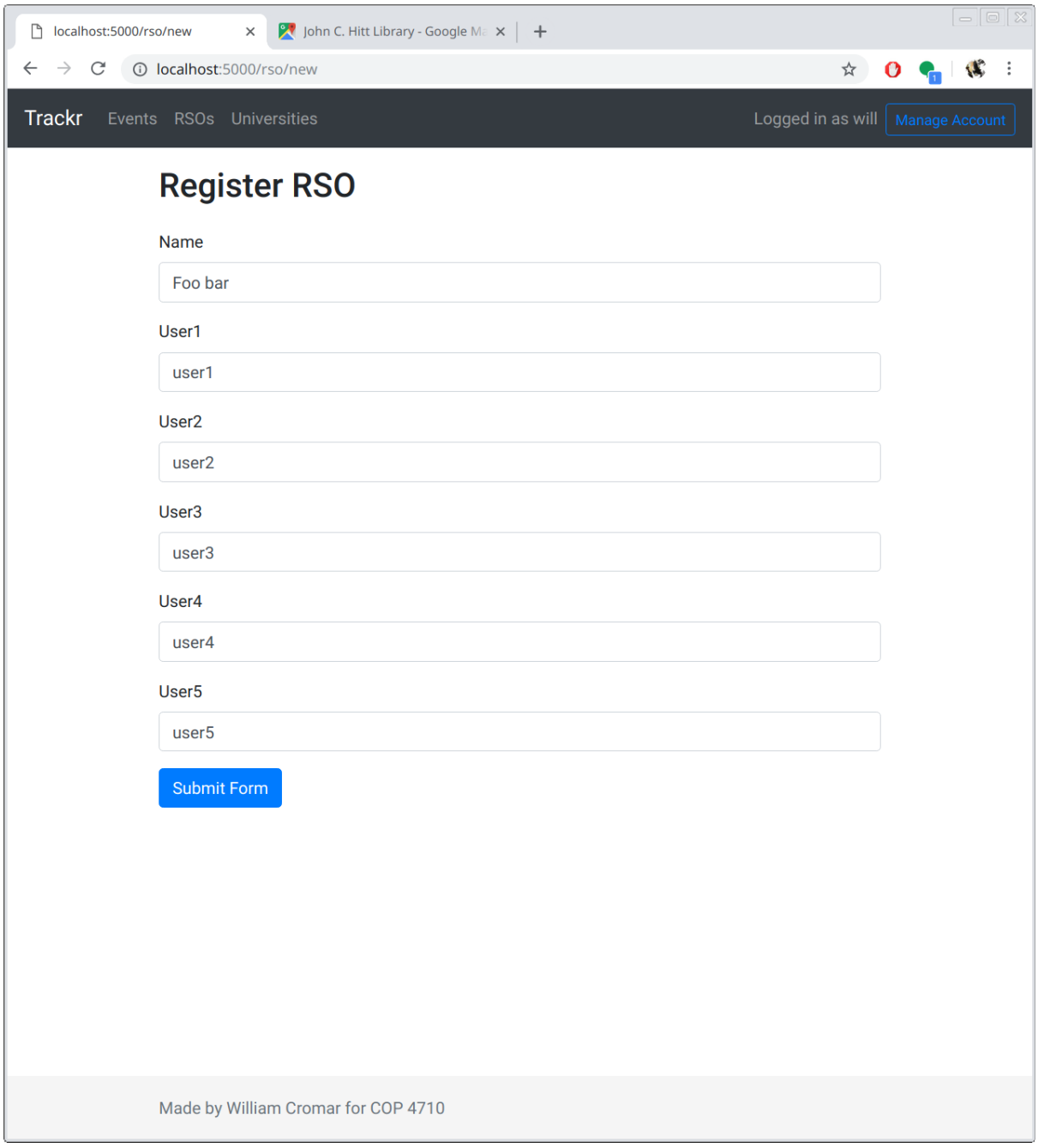


Figure 16: The new user can request to create an RSO with 5 of their friends and will become that RSO's first admin.

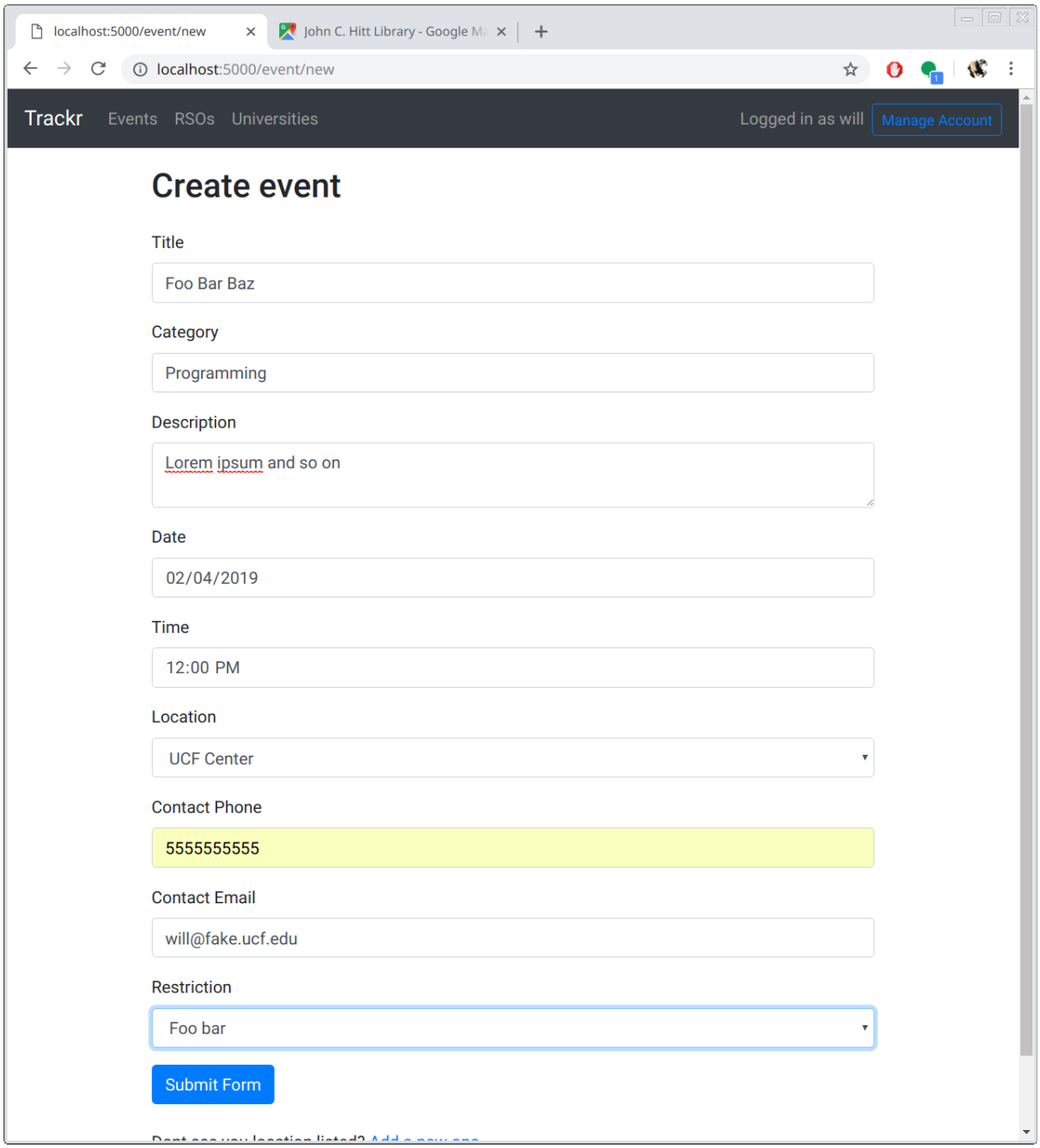


Figure 17: Now that the user is an admin of an RSO, they can create an RSO event for it.

4 Advanced Features

4.1 Responsive Interface Design

The graphical user interface is designed to reflow the contents of the page when the width of the screen is adjusted in order to be visually appealing on mobile. See Figure 18.

4.2 Social Networking

All event details pages include an option to share the event on Twitter. See Figure 18.

4.3 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 generates the `INSERT` statements to adapt the data to my database design. This can be used to populate the database with real test data. An list of real events is shown in Figure 19.

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.

An example of a map with a fixed in is shown in Figure 18. The graphical location picker is shown in Figure 20.

4.5 University Album

The application implements a University View page where users can see a list of all of the events for that university that also features an automatically scrolling list of photos, which are added by the super-admins. A screenshot is shown in Figure 21.

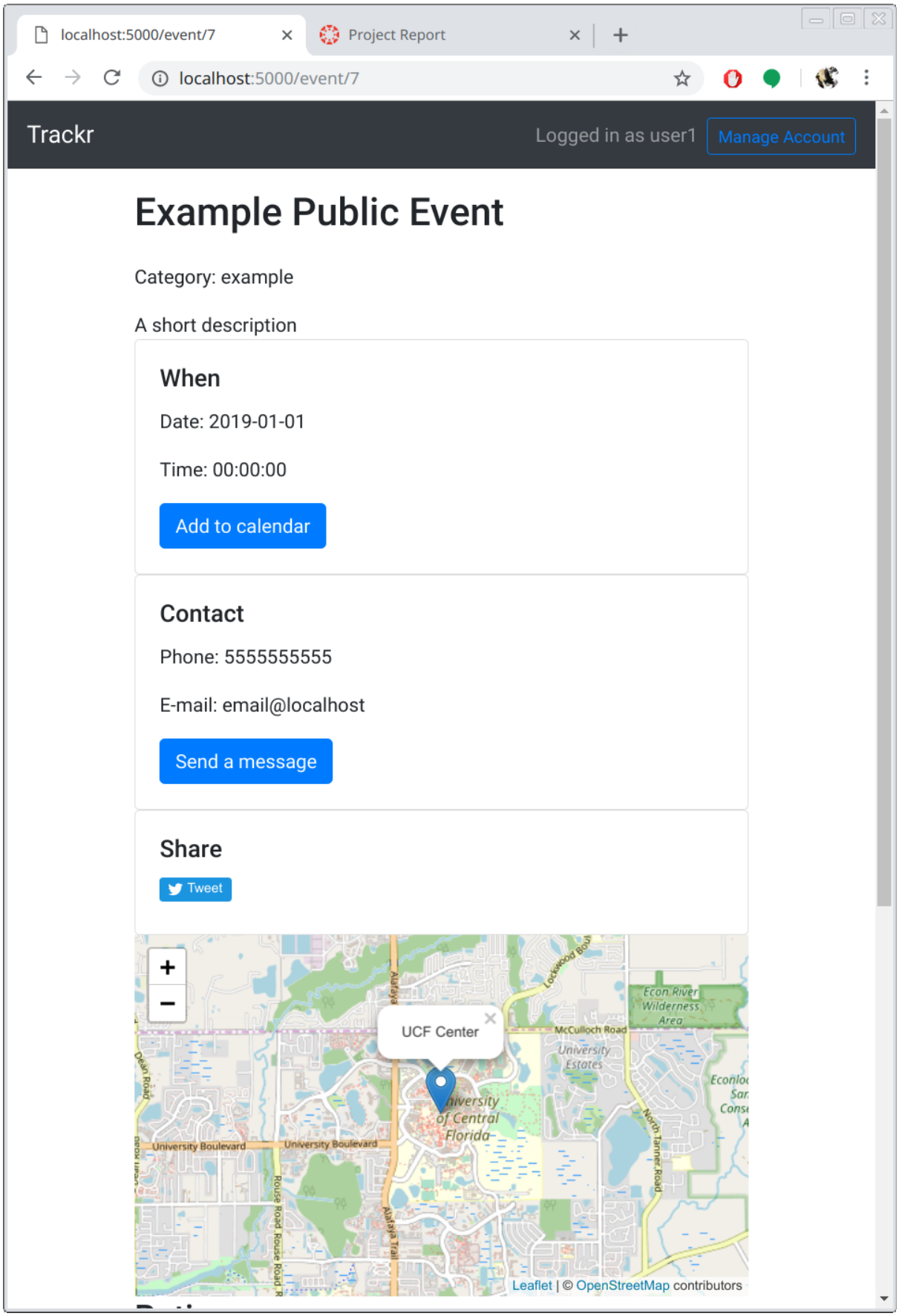
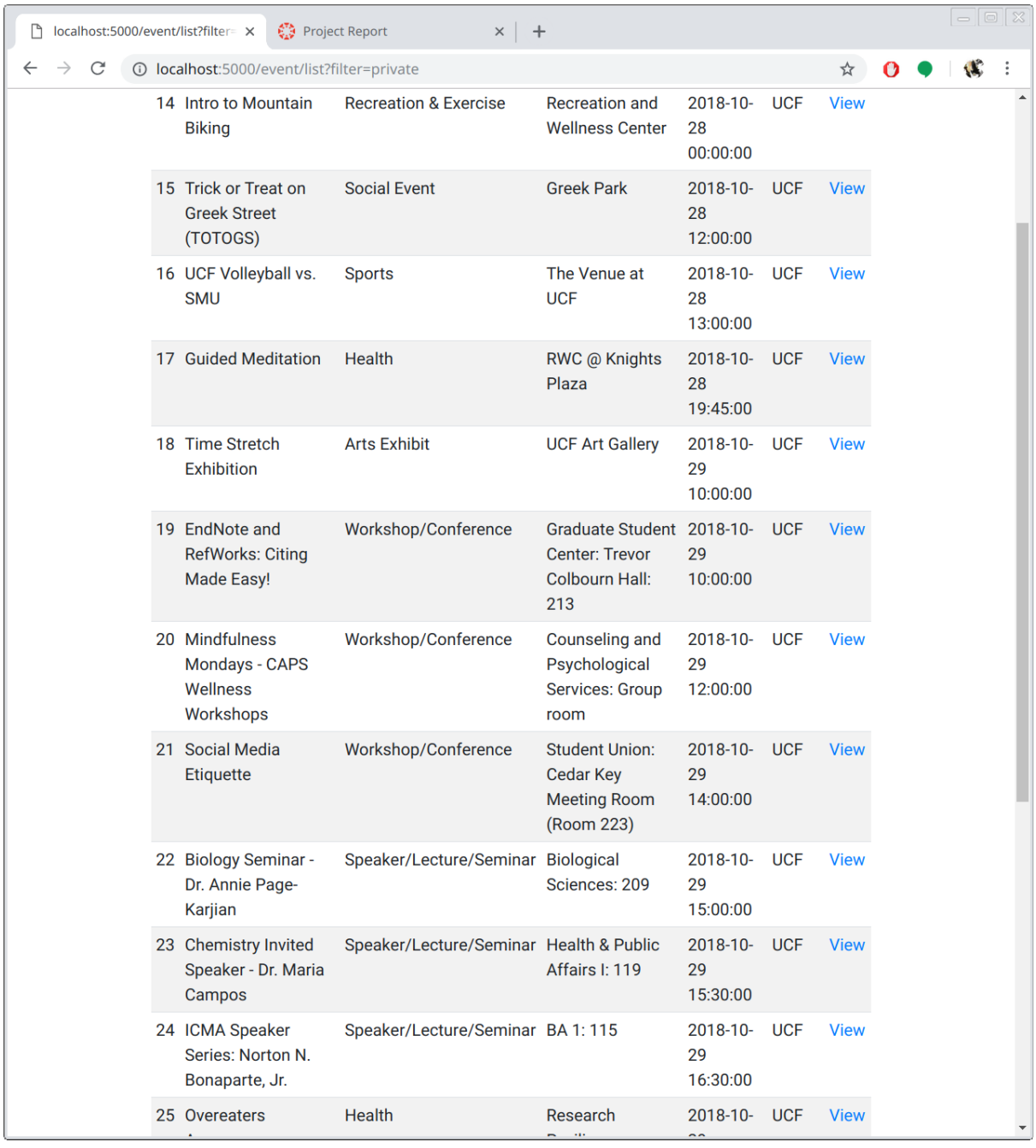


Figure 18: The event details page automatically reflows content to fit the screen size.



14	Intro to Mountain Biking	Recreation & Exercise	Recreation and Wellness Center	2018-10-28 00:00:00	UCF	View
15	Trick or Treat on Greek Street (TOTOGS)	Social Event	Greek Park	2018-10-28 12:00:00	UCF	View
16	UCF Volleyball vs. SMU	Sports	The Venue at UCF	2018-10-28 13:00:00	UCF	View
17	Guided Meditation	Health	RWC @ Knights Plaza	2018-10-28 19:45:00	UCF	View
18	Time Stretch Exhibition	Arts Exhibit	UCF Art Gallery	2018-10-29 10:00:00	UCF	View
19	EndNote and RefWorks: Citing Made Easy!	Workshop/Conference	Graduate Student Center: Trevor Colbourn Hall: 213	2018-10-29 10:00:00	UCF	View
20	Mindfulness Mondays - CAPS Wellness Workshops	Workshop/Conference	Counseling and Psychological Services: Group room	2018-10-29 12:00:00	UCF	View
21	Social Media Etiquette	Workshop/Conference	Student Union: Cedar Key Meeting Room (Room 223)	2018-10-29 14:00:00	UCF	View
22	Biology Seminar - Dr. Annie Page-Karjian	Speaker/Lecture/Seminar	Biological Sciences: 209	2018-10-29 15:00:00	UCF	View
23	Chemistry Invited Speaker - Dr. Maria Campos	Speaker/Lecture/Seminar	Health & Public Affairs I: 119	2018-10-29 15:30:00	UCF	View
24	ICMA Speaker Series: Norton N. Bonaparte, Jr.	Speaker/Lecture/Seminar	BA 1: 115	2018-10-29 16:30:00	UCF	View
25	Overeaters	Health	Research	2018-10-	UCF	View

Figure 19: Real events from UCF can be scraped from the website and added to this application.

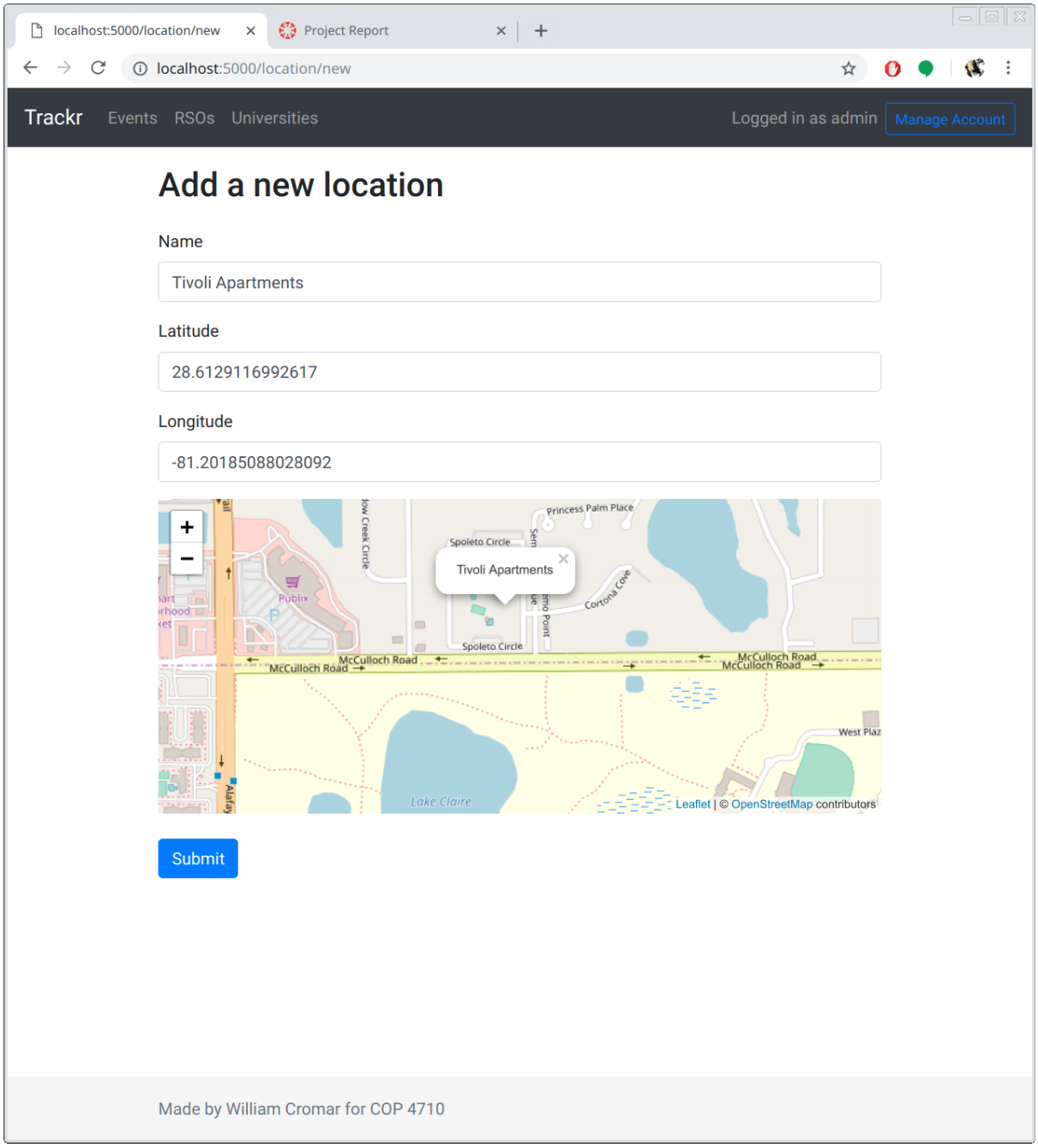


Figure 20: The location picker component lets the user either enter the coordinates of a location automatically or select a location directly on the map.

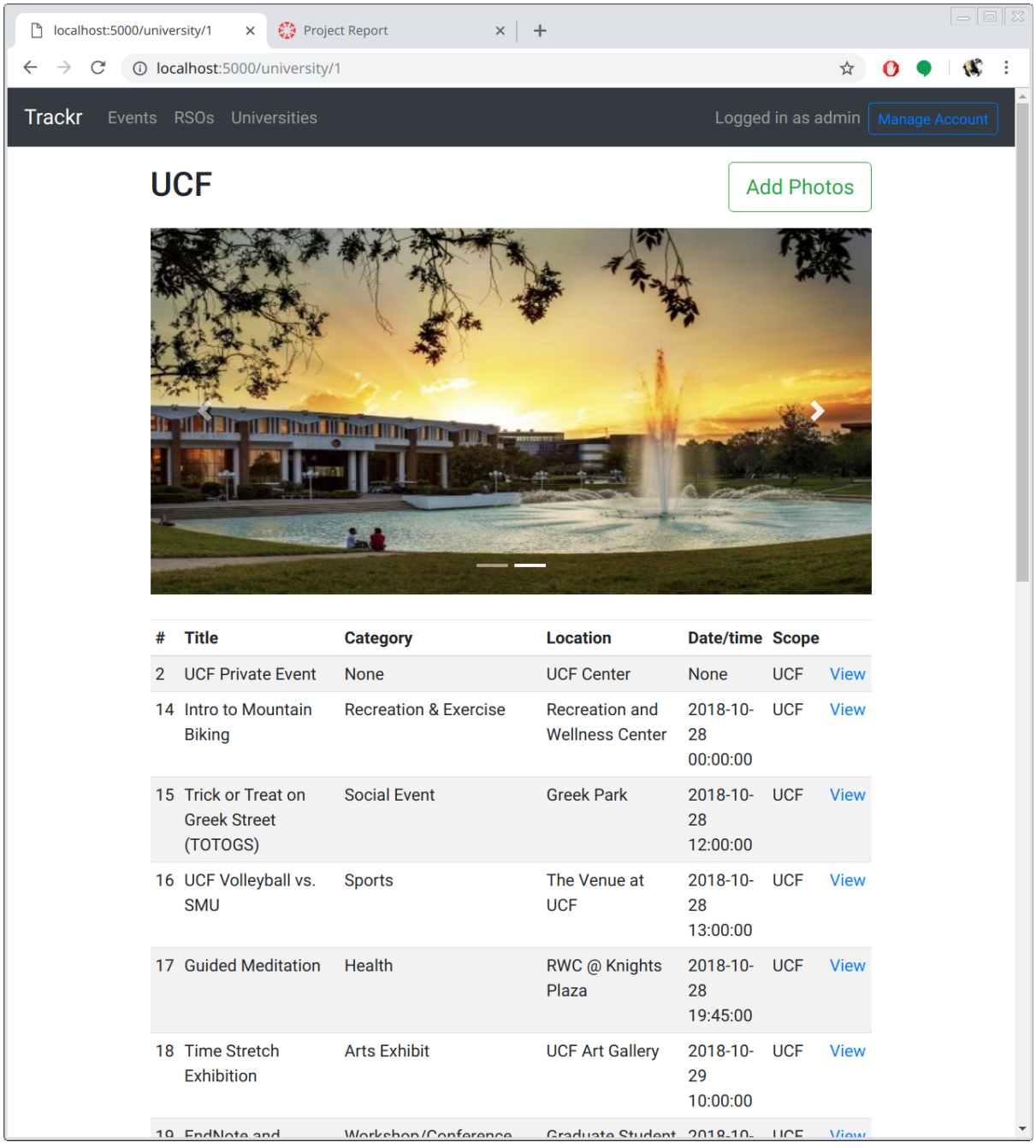


Figure 21: The university view page features a list of all events for that university and a slideshow.

5 Conclusion

5.1 Performance

The database is quite responsive, since critical views that involve joins are computed ahead of time. In order to speed up the database, it would probably be worthwhile to index events by their restrictions (university and RSO) to speed up lookups of Private and RSO events, which have to search on those attributes.

5.2 Areas for Improvement

If I were to redo the project, I would decompose the database differently. In particular I would flatten `Users` and `SuperUsers` such that there is an attribute on `Users` that indicates whether that user is a super-admin. Likewise, I would flatten `Admins` and `RSOMembers` such that there is an attribute on `RSOMembers` that indicates which members are also admins. Then, a view can be constructed on that table to filter for admins.

5.3 Future Work

Although the application is resistant to XSS and CSRF attacks, there remain some significant security vulnerabilities in the application. Most significantly, passwords are not hashed, so a compromised database would leak real passwords to the attacker. I would also integrate additional social sharing options beyond just Twitter.

5.4 Challenges

Building this application turned out to be considerably more work than I expected, although I found all of it to be pretty straightforward. I used embedded SQL queries everywhere to meet the project restrictions. However, I found this to be much more error-prone than necessary. Since all queries were written by hand and embedded in the application, typos in table or column names could not be caught with static analysis tools (e.g. linters) and always surfaced at runtime. In my industry experience, developers nearly always use ORM (object-relational mapper) libraries to define their tables (rather than directly using `CREATE TABLE`) and only explicitly write out queries in SQL for performance reasons. This lets them do most simple operations using constructs defined in the application language where static analysis tools can check their work. If I were to start a real-world project from scratch, I would certainly use an ORM and avoid embedded queries.