ucfeventtracker

COP 4710 Spring 2021

Group 1 - Josh Kraftchick & Joseph Mansy

Project Description

Our project is hosted on an Express server on Node.js and a MongoDB database. Node.js is a javascript runtime environment that allows for the back-end of the website to use the same language of the frontend and opens the door to the majority of the modern web frameworks and libraries. Our backend utilizes mongoose to interface with our database. Mongoose allows for easy object modeling and data validation making it a useful tool for our backend. We can define a schema, similar to constraints in SQL that is checked and enforced as well as running complex searches on the data.

The frontend is using React and Material-UI library for the frontend and axios for making HTML requests. React is a powerful framework that allows us to write UI components and create single-page applications. Material-UI is a component library that allows for wiring of UI components that follow the Material design language Google has developed. Finally we used leaflet to create our interactive map components

Overall, we used these technologies and libraries because we were already familiar with them and they are powerful tools to make a clean website.

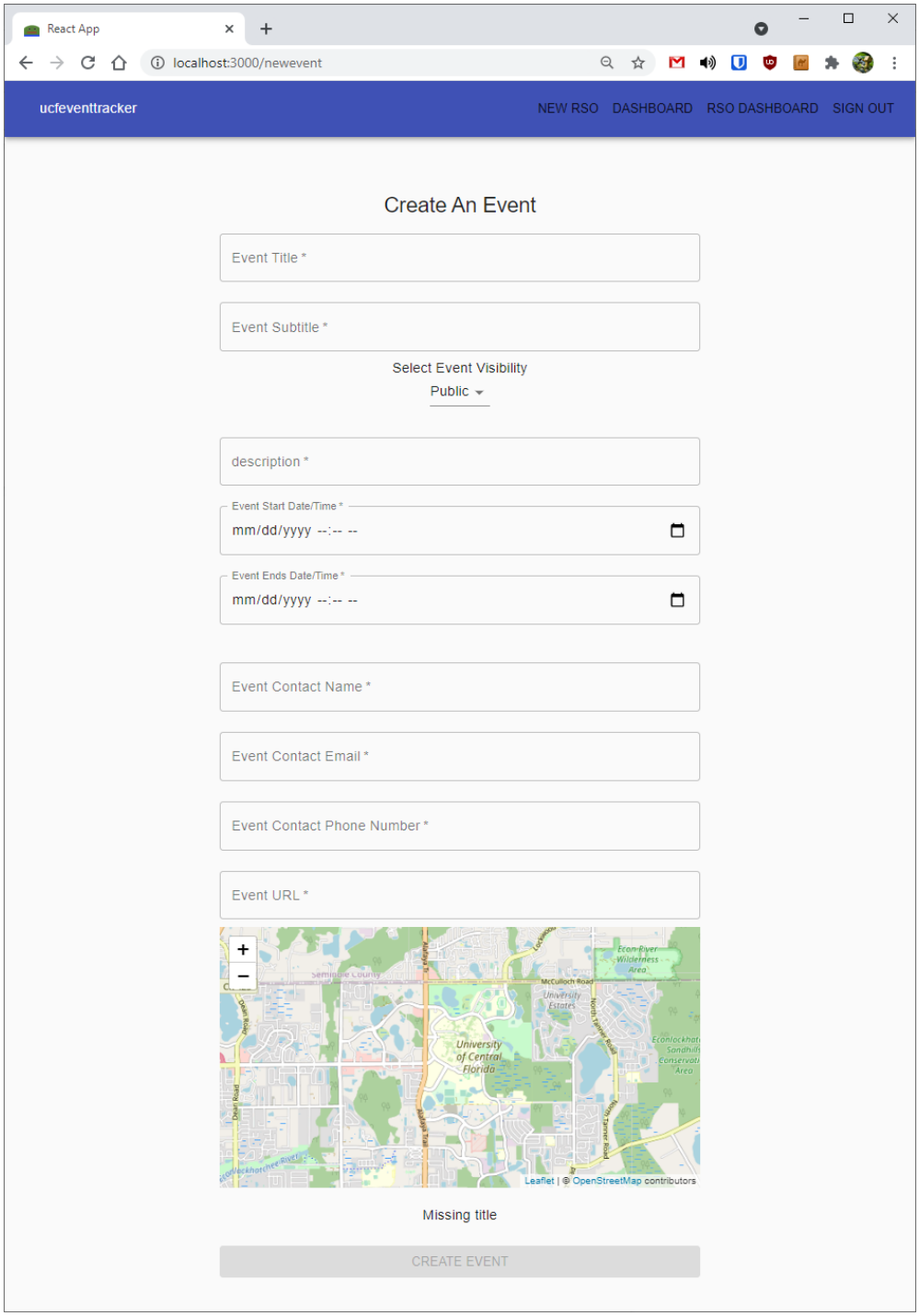
Though our website is not actually hosted, our database is. MongoDB offers free basic hosting of a database through MongoDB Atlas. Though we could have hosted our database ourselves hosting through a provider simplifies the setup process and makes the database accessible from anywhere.

Finally, our assumptions. One of the assumptions we took was valid data inside of the database. Our backend does not currently handle stale data. Should we run into a reference to a nonexistent event then that data is ignored. Normal usage of this website should not run into this problem since. One other assumption is that a user would not change schools, as in the real world this is rare. Once a user signs up and selects the school they belong to, they remain in that school. Finally, we did not implement comment moderation. Users are able to create and manage their own comments but nobody else can.

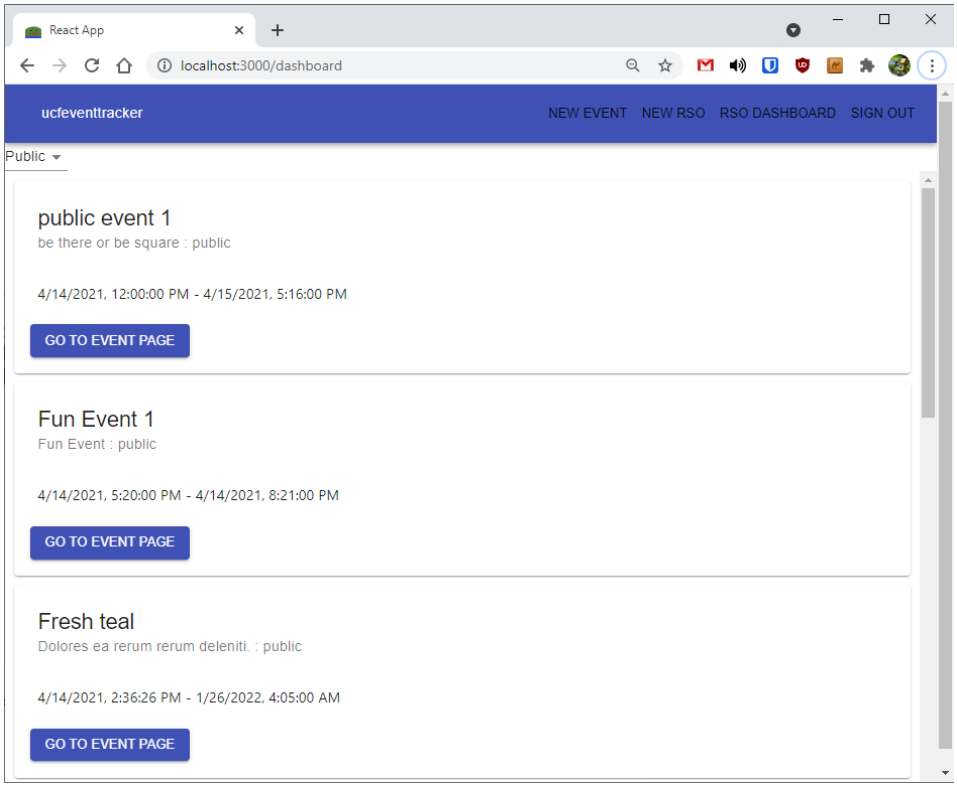
GUI

Platform: Node.js
Languages: JavaScript - Express & React
DBMS: MongoDB

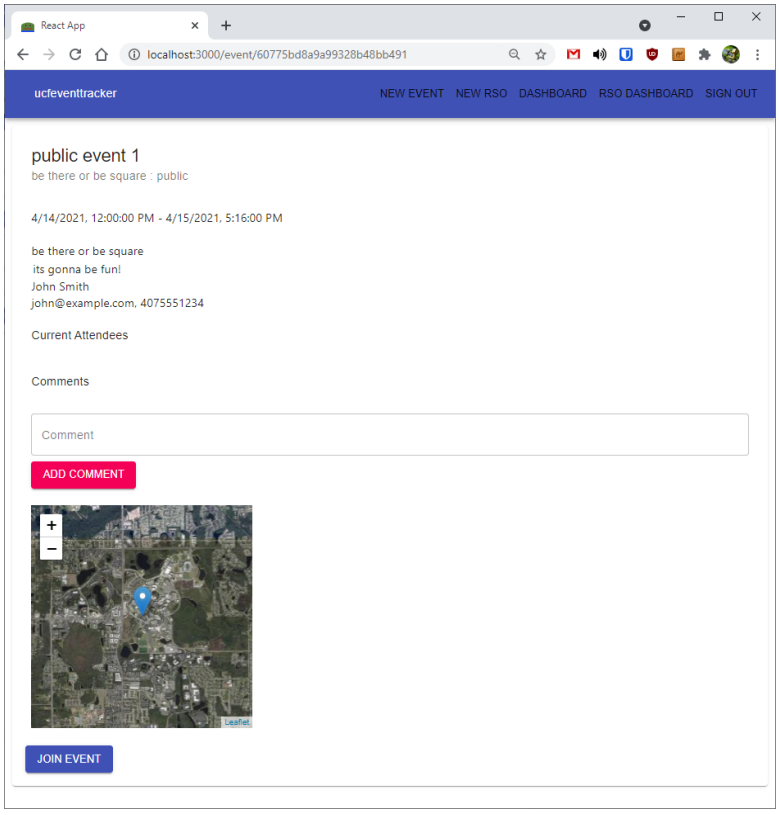
Create event



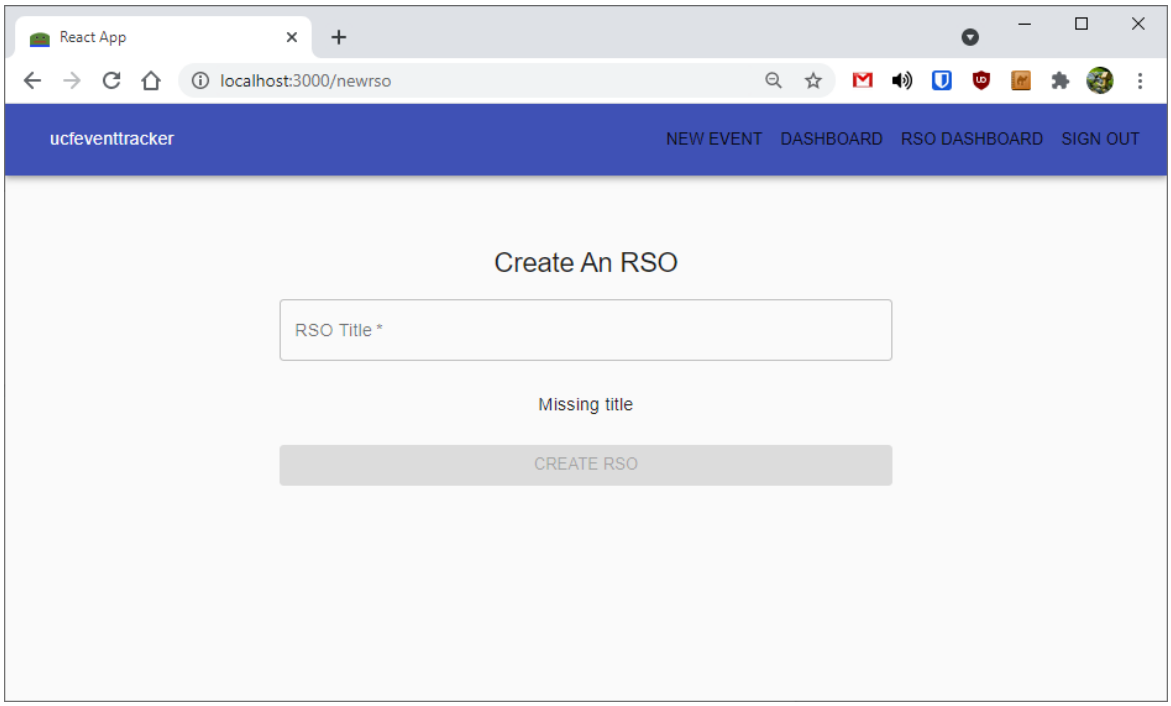
Event Dashboard



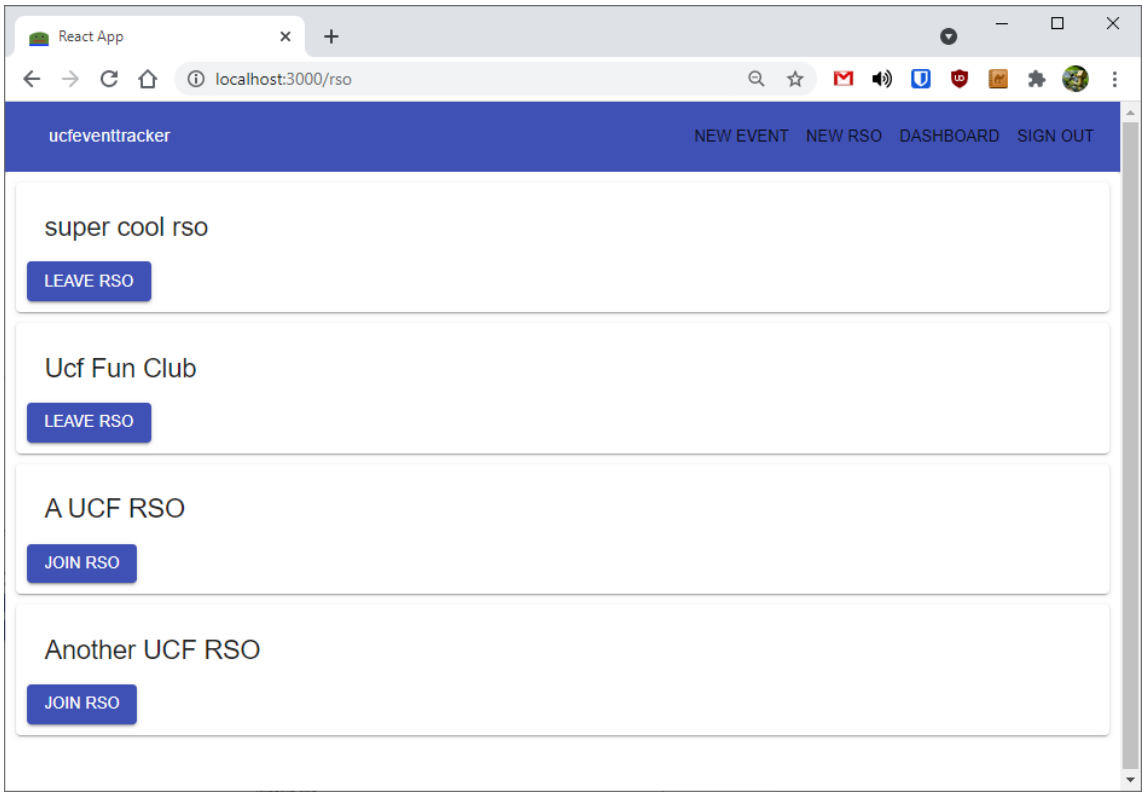
Event Page



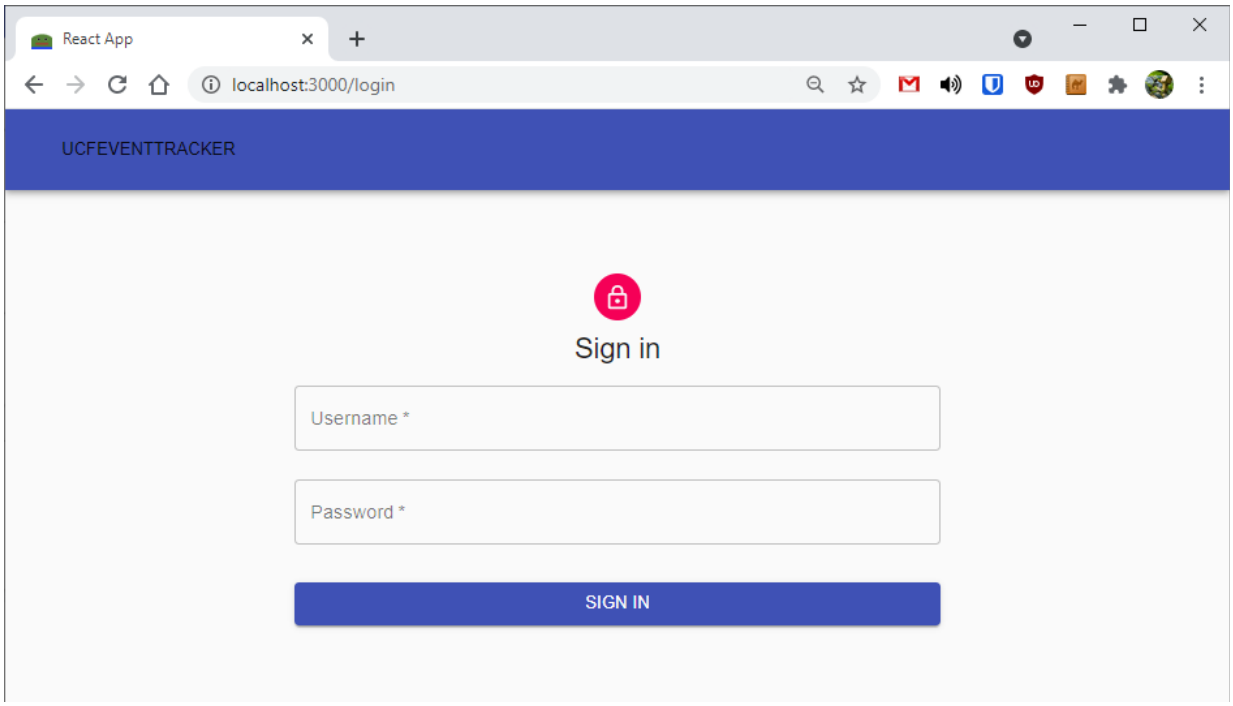
Create RSO



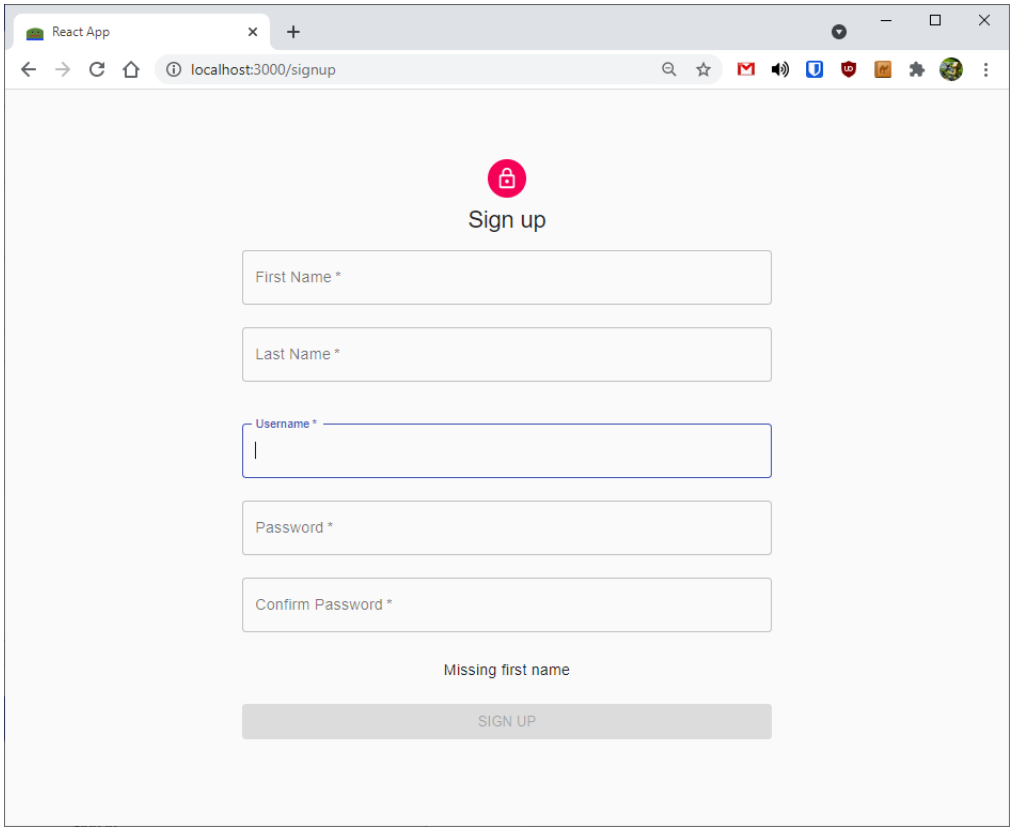
RSO Dashboard



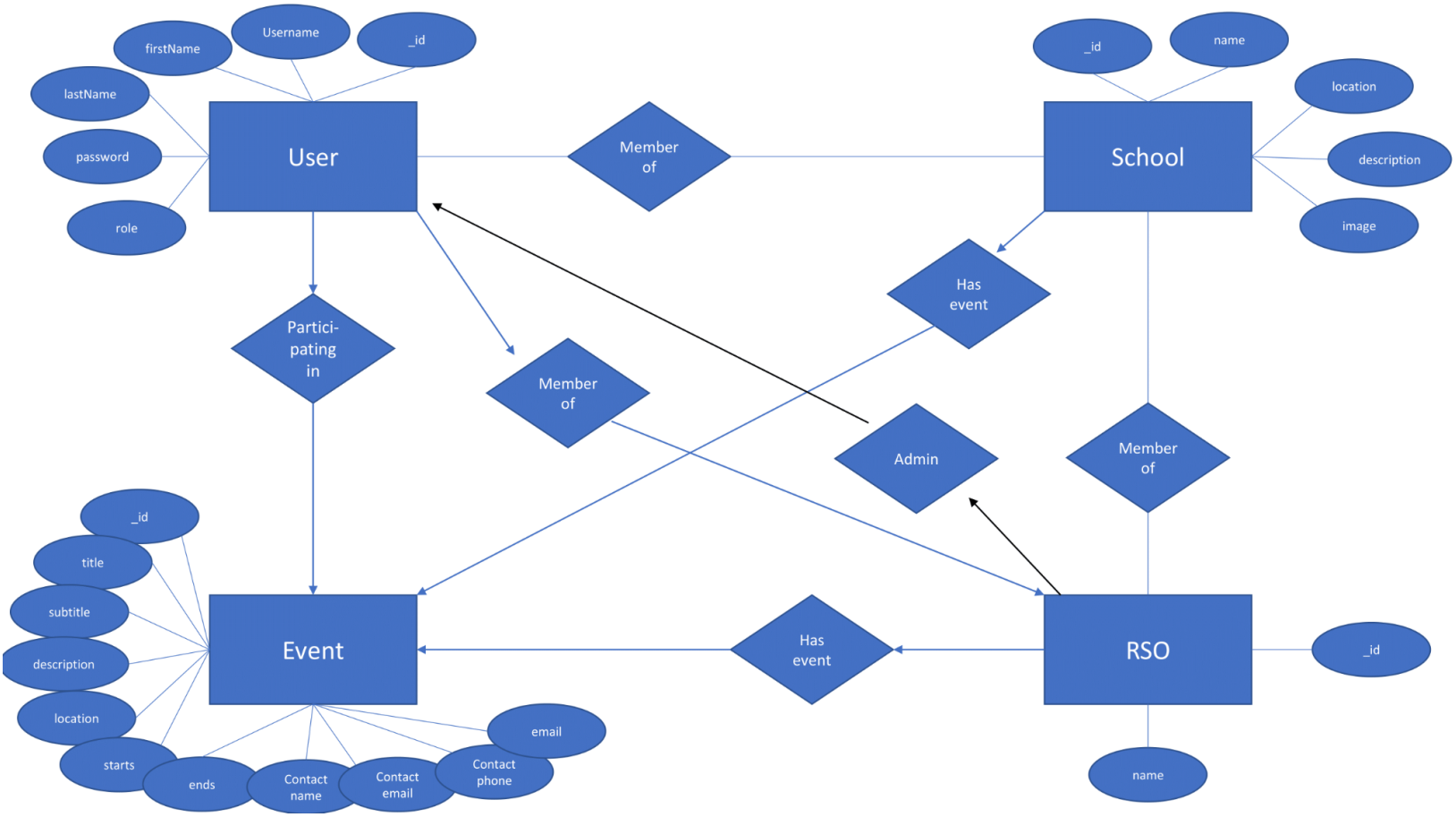
Sign in



Sign up



ER-Model

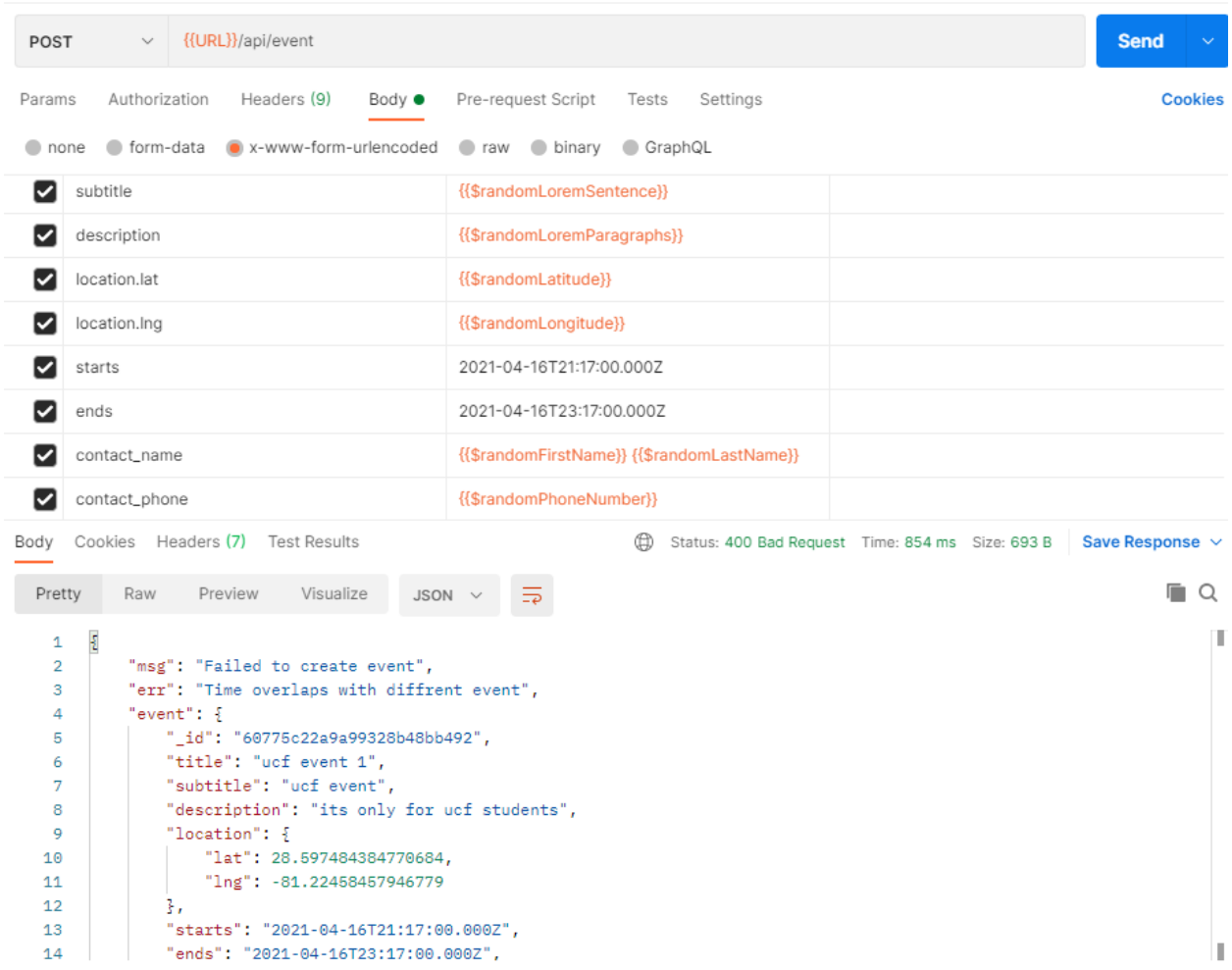

firstName
Username
_id
lastName
password
role
User
Member of
_id
name
location
School
description
image
Has event
Partici-pating in
Member of
Admin
Member of
_id
title
subtitle
description
location
starts
ends
Contact name
Contact email
Contact phone
email
Event
Has event
RSO
_id
name

Constraints

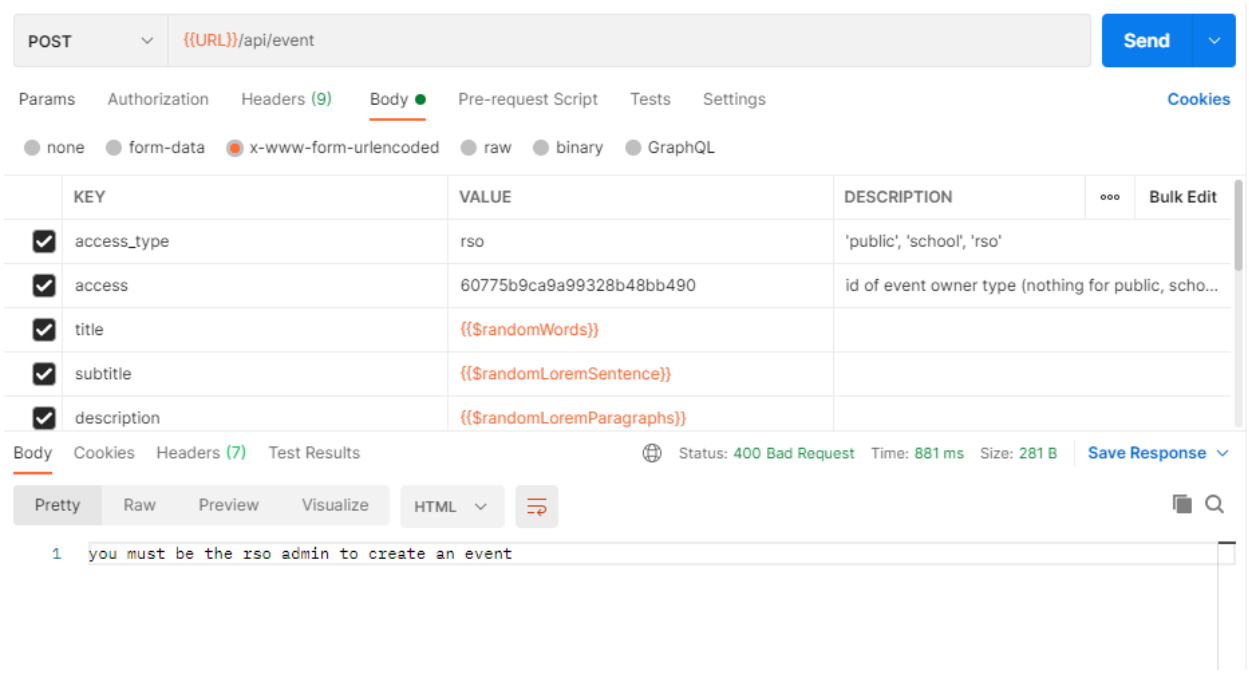
Since we used mongoDB and mongoose, our constraints were defined in our schemas. Once defined, the data is checked by the database and it returns errors if the data does not match.

Here we used postman, an api endpoint tester to send requests to the server based on the examples presented in the requirements

A new event to be held at the same location and overlapping times with an existing event: Show error message with enough detail such as the conflicting event, time, location, etc.

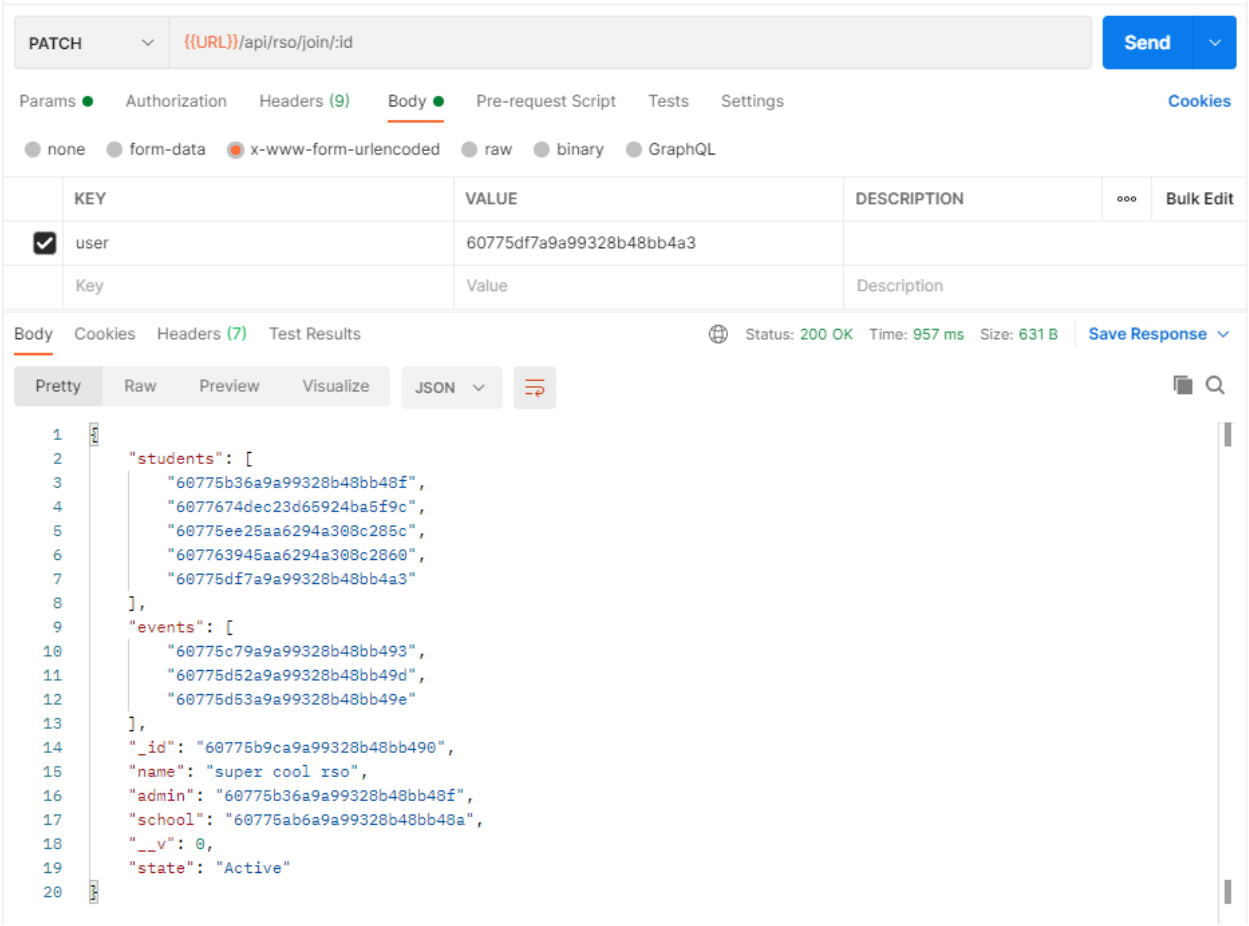


An admin who is not the Admin of the RSO attempts to create an event for that RSO: Show an error message.

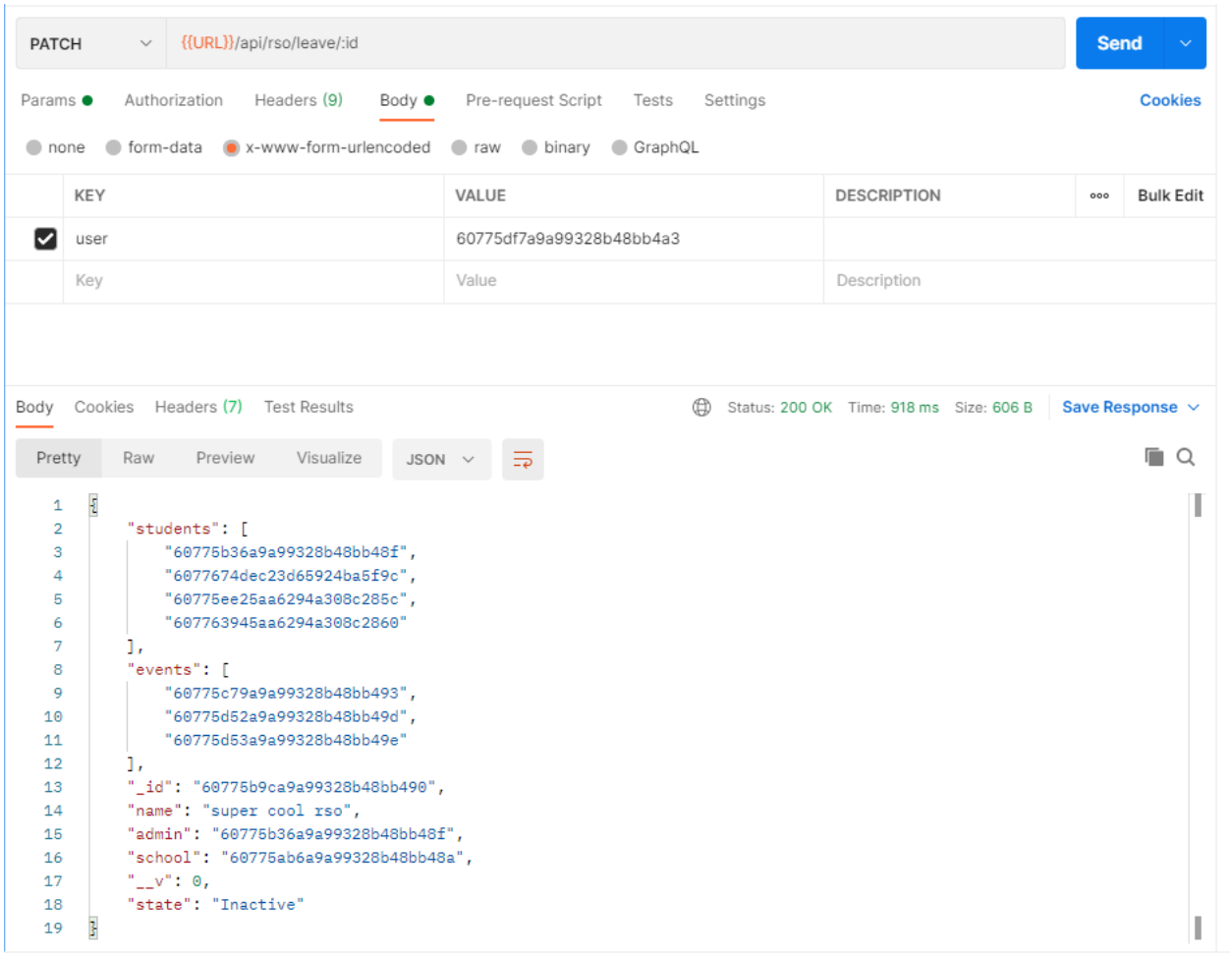


An INSERT of a member of an RSO with 4 members: Show the status of the RSO changing to ‘Active.’ A DELETE of a member of an RSO with 5 members: Show the status of the RSO changing to ‘Inactive.’

Joining RSO and making it Active with 5 Members



Leaving RSO making it inactive with only 4 Members



Relational data model

Here we show what the SQL would have looked like if we used an SQL language. Since we used MongoDB and mongoose, our schema was controlled by indexes and constraints configured in mongoose.

```
CREATE TABLE EVENTS (
      _id ObjectId NOT NULL,
      Title VARCHAR(50) NOT NULL,
      Subtitle VARCHAR(50),
      Description VARCHAR(256),
      Location.lat DOUBLE(25, 3),
      Location.lng DOUBLE(25, 3),
      Starts DATETIME,
      Ends DATETIME,
      Contact_name VARCHAR(50),
      Contact_phone VARCHAR(50),
      Contact_email VARCHAR(50),
      Url VARCHAR(256),
      Type ENUM('Public', 'School', 'Rso'),
      Users ObjectId[],
      Comments VARCHAR(50),
      PRIMARY KEY (_id),
      FOREIGN KEY (Users) REFERENCES USERS(_id),
      CONSTRAINT CHK_Lat CHECK (Location.lat>=-90 AND
Location.lat<=90),
      CONSTRAINT CHK_Lng CHECK (Location.lng>=-180 AND
Location.lng<=180),
      CONSTRAINT CHK_Time CHECK (Starts < Ends),
);

CREATE TABLE USERS (
      _id ObjectId NOT NULL,
      Username VARCHAR(50) NOT NULL,
      FirstName VARCHAR(50),
      LastName VARCHAR(50),
      Password VARCHAR(50),
      Role ENUM('Superadmin', 'Admin', 'User'),
      School ObjectId,
      Rsos ObjectId[],
      PRIMARY KEY (_id),
      FOREIGN KEY (School) REFERENCES SCHOOLS(_id),
```

```
      FOREIGN KEY (Rsos) REFERENCES RSOS(_id)
);
```

Here is the actual schema as made in mongoose

```
const EventSchema = new Schema({
    title: {
        type: String,
        required: true
    },
    subtitle: String,
    description: String,
    location: Schema.Types.Mixed,
    starts: Date,
    ends: Date,
    contact_name: String,
    contact_phone: String,
    contact_email: String,
    url: String,
    access_type: {
        type: String,
        required: true,
        enum: ['public', 'school', 'rso']
    },
    access: {
        type: Schema.Types.ObjectId,
        refPath: 'access_type'
    },

    users: [{ type: Schema.Types.ObjectId, ref: 'user' }],
    comments: [String]
});

const RsoSchema = new Schema({
    name: {
        type: String,
        required: true,
        unique: true
    },
    admin: {
```

```
        type: Schema.Types.ObjectId,
        ref: 'user',
        required: true
    },
    school: { type: Schema.Types.ObjectId, ref: 'school' },
    students: [{ type: Schema.Types.ObjectId, ref: 'user' }],
    events: [{ type: Schema.Types.ObjectId, ref: 'event' }]
});

const SchoolSchema = new Schema({
    name: {
        type: String,
        required: true,
        unique: true
    },
    location: {
        lat: Number,
        lng: Number
    },
    description: {
        type: String
    },
    image: {
        type: String
    },

    students: [{ type: Schema.Types.ObjectId, ref: 'user' }],
    rsos: [{ type: Schema.Types.ObjectId, ref: 'rso' }],
    events: [{ type: Schema.Types.ObjectId, ref: 'event' }]
});

const UserSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    },
    firstName: {
        type: String,
        required: true
```

```
    },
    lastName: {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
    },
    role: {
        type: String,
        required: true,
        enum: ["student", "admin", "superadmin"]
    },
    school: { type: Schema.Types.ObjectId, ref: 'school' },
    rsos: [{ type: Schema.Types.ObjectId, ref: 'rso' }]
});
```

```
CREATE TABLE SCHOOLS (
      _id ObjectId NOT NULL,
      Name VARCHAR(50) NOT NULL,
      Location.lat DOUBLE(25, 3),
      Location.lng DOUBLE(25, 3),
      Description VARCHAR(50),
      Image VARCHAR(50),
      Students ObjectId[],
      Rsos ObjectId[],
      Events ObjectId[]
      PRIMARY KEY (_id),
      FOREIGN KEY (Students) REFERENCES USERS(_id),
      FOREIGN KEY (Rsos) REFERENCES RSOS(_id),
      FOREIGN KEY (Events) REFERENCES EVENTS(_id)
);

CREATE TABLE RSOS(
      _id ObjectId NOT NULL,
      Name VARCHAR(50) NOT NULL,
      Admin ObjectId,
      School ObjectId,
      Students ObjectId[],
      Events ObjectId[]
      PRIMARY KEY (_id),
      FOREIGN KEY (Admin) REFERENCES USERS(_id),
      FOREIGN KEY (School) REFERENCES SCHOOLS(_id),
      FOREIGN KEY (Students) REFERENCES USERS(_id),
      FOREIGN KEY (Events) REFERENCES EVENTS(_id)
);
```

Populating sample data

Here we created what our SQL commands would have looked like if we used a SQL language. Since we used MongoDB, everything was done using the mongoDB driver through mongoose.

```
INSERT INTO SCHOOLS (_id, Name, Location.lat, Location.lng,
Description, Image)
VALUES (123, SchoolA, 50.795, 124.291, "Its SchoolA", IMAGE);
INSERT INTO SCHOOLS (_id, Name, Location.lat, Location.lng,
Description, Image)
VALUES (456, SchoolB, 28.664, -63.256, "Its SchoolB", IMAGE);

INSERT INTO USERS (_id, Username, FirstName, LastName, Password,
Role, School)
VALUES (1, user1, john, smith, password, SuperAdmin, 123);
INSERT INTO USERS (_id, Username, FirstName, LastName, Password,
Role, School)
VALUES (2, user2, sally, smith2, password, Admin, 123);
INSERT INTO USERS (_id, Username, FirstName, LastName, Password,
Role, School)
VALUES (3, user3, bob, smith, password, User, 123);
INSERT INTO USERS (_id, Username, FirstName, LastName, Password,
Role, School)
VALUES (4, user4, john, doe, password, SuperAdmin, 456);
INSERT INTO USERS (_id, Username, FirstName, LastName, Password,
Role, School)
VALUES (5, user5, sally, doe, password, Admin, 456);
INSERT INTO USERS (_id, Username, FirstName, LastName, Password,
Role, School)
VALUES (6, user6, bob, doe, password, User, 456);

INSERT INTO RSOS (_id, Name, Admin, School)
VALUES (1231, SchoolARso1, 1, 123);
INSERT INTO RSOS (_id, Name, Admin, School)
VALUES (1232, SchoolARso2, 2, 123);
INSERT INTO RSOS (_id, Name, Admin, School)
VALUES (4561, SchoolBRso1, 4, 456);
INSERT INTO RSOS (_id, Name, Admin, School)
VALUES (4562, SchoolBRso2, 5, 456);
```

```
INSERT INTO EVENTS (_id, Title, Subtitle, Description, Location.lat,
Location.lng, Starts, ends, Contant_name, Contact_Phone,
Contact_email, url, Type)
VALUES (1, event1, event subtitle, description, 12.345, 98.765,
“4/10/2021”, “4/11/2021”, “Public”);
INSERT INTO EVENTS (_id, Title, Subtitle, Description, Location.lat,
Location.lng, Starts, ends, Contant_name, Contact_Phone,
Contact_email, url, Type)
VALUES (2, event2, event subtitle, description, 22.345, 88.765,
“4/12/2021”, “4/13/2021”, “School”);
INSERT INTO EVENTS (_id, Title, Subtitle, Description, Location.lat,
Location.lng, Starts, ends, Contant_name, Contact_Phone,
Contact_email, url, Type)
VALUES (3, event3, event subtitle, description, 32.345, 78.765,
“4/14/2021”, “4/15/2021”, “Rso”);
INSERT INTO EVENTS (_id, Title, Subtitle, Description, Location.lat,
Location.lng, Starts, ends, Contant_name, Contact_Phone,
Contact_email, url, Type)
VALUES (4, event4, event subtitle, description, 32.345, 68.765,
“4/16/2021”, “4/17/2021”, “Public”);
INSERT INTO EVENTS (_id, Title, Subtitle, Description, Location.lat,
Location.lng, Starts, ends, Contant_name, Contact_Phone,
Contact_email, url, Type)
VALUES (5, event5, event subtitle, description, 42.345, 58.765,
“4/18/2021”, “4/19/2021”, “School”);
INSERT INTO EVENTS (_id, Title, Subtitle, Description, Location.lat,
Location.lng, Starts, ends, Contant_name, Contact_Phone,
Contact_email, url, Type)
VALUES (6, event6, event subtitle, description, 52.345, 48.765,
“4/20/2021”, “4/21/2021”, “Rso”);
```

DB Examples

Here is some examples from the api routes that shows some of the examples of inserting into the DB

This is the code within the api/routes folder in the server code that is takes the api requests, validates them, then writes to the database.

insert a new RSO (part of the processing of the ‘Create RSO’ form), show results

```
      let rso = new Rsos({
            name,
            admin,
            school: _school,
            students
        })

        rso.save();

        await Schools.findByIdAndUpdate(_school, { $push: { rsos: rso._id
} }, { useFindAndModify: false, new: true });
        await Users.findByIdAndUpdate(decoded._id, { $push: { rsos:
rso._id } }, { useFindAndModify: false, new: true });
```

SQL statement to insert a new student to an existing RSO (part of the processing of the ‘Join RSO’ form), show results

```
Rsos.findByIdAndUpdate(req.params.id, { $push: { students: req.body.user }
}, { useFindAndModify: false, new: true }, (dberr, dbres) => {
        if (dberr) return res.status(400).send(dberr);

        Users.findByIdAndUpdate(req.body.user, { $push: { rsos:
req.params.id } }, { useFindAndModify: false, new: true }, (dberr, dbres2)
=> {
            if (dberr) return res.status(400).send(dberr);
            //res.send(dbres);
            return res.send(dbres)
        })
    })
```

SQL statement to insert a new event (part of the processing of the ‘Create Event’ form), show results

```
let event = new Events({
```

```
                title,
                subtitle,
                description,
                location,
                starts,
                ends,
                contact_name,
                contact_phone,
                contact_email,
                url,
                access_type: 'rso',
                access: rso._id
            })

            event.save();

            await Rsos.findByIdAndUpdate(access, { $push: { events:
event._id } }, { useFindAndModify: false });
```

SQL statement to insert/update a (new) comment (part of the processing of the 'Create/Add/Modify Comment' form), show results

```
Events.findByIdAndUpdate(req.params.id, { $push: { comments:
req.body.comment } }, { useFindAndModify: false, new: true, populate:
{path:'users', select: 'firstName lastName _id'} }, (dberr, dbres) => {
        if (dberr) return res.status(400).send(dberr);
        res.send(dbres);
    })
```

Several SQL queries to display events—public, private, and RSO-- (part of the processing of the 'View Event' request by a user with a specific role), show results

```
let search = { access_type: "public" }

            if (level === "school") {
                //search = { $or: [{ access_type: "public" }, {
access_type: "school", access: user.school }] }
                search = { access_type: "school", access: user.school };
            }
            else if (level === "rso") {
                //search = { $or: [{ access_type: "public" }, {
access_type: "school", access: user.school }, { access_type: "rso",
access: { $in: user.rsos } }] }
```

```
                search = { access_type: "rso", access: { $in: user.rsos }
}
            }

            Events.find(search).populate('users', 'firstName lastName
_id').populate('access', 'name').exec((err, events) => {
                if (err) return res.status(500).send(err);

                return res.send(events);
            })
```

Conclusion

Database performance

Action	AVG DB Response Time
Login	33 ms
Sign Up	67 ms
Get all Events	94 ms
Get Event by ID	33 ms
Create Event	62 ms
Get all RSOs	64 ms
Join RSO	62 ms
Leave RSO	62 ms
Get all School	34 ms
Get School by ID	33 ms

Desired Functionality:
We implemented everything we wanted to implement. If we had to add more, the next step would probably be actually securing the app as passwords are stored in plain text and the website does not use http. After that would probably be reworking the frontend to be cleaner as we are not experienced in frontend development meaning we just added forms and boxes as it was required.

Problems Encountered:
We had problems setting up MySql and other SQL servers when we started the project leading us to want to use a different database. We chose MongoDB because we both have prior experience with it and use it for work related projects, meaning it was more useful to learn more database stuff on it than an SQL language that we aren't using for work. Another problem we encountered was using Express and React, they work well once configured but took some time to set up as neither of us had set up a react or express app, only worked on pre existing ones. This was another good opportunity to work on a tool both of us use for work instead of using PHP or a different less used language.