Bethe OPS Project

System & Program Design

May 16, 2019

Table of Contents

1 Overview

The purpose of this document is to outline the system design and program design for the Bethe OPS Project. Sections 2 to 7 cover the system design; the system has been broken down into components and each section describes the design of a component in detail. Section 8 covers the program design, describing the project folder structure and what implementation each part of the project repository accounts for.

2 Required Technologies

The Bethe OPS system is a web application built primarily using **Javascript**, more specifically, the framework of Bethe OPS is supported by **Node.js**. Version control for the system is maintained via **Github**. Packages, libraries, and other dependencies required for the app are installed using **npm**. The technologies required for the system are broken down into two categories: front-end and back-end.

2.1 Front-End Technologies

The following technologies are required for the front-end side (user interface) of the system:

- EJS (Embedded Javascript templating)
- CSS
    - Materialize CSS
- Javascript
- jQuery

2.2 Back-End Technologies

The following technologies are required for the back-end side (server, database) of the system:

- Node.js
    - Express
    - Passport
- MySQL
- AWS
    - EC2
    - RDS
- PM2
- Shibboleth

- SAML

3 Routing

The root route, */*, redirects the user to the CUWebLogin page first to log into the system. Upon login, if the user is new and not in the system, (s)he is then redirected to the sign-up page route, */signup*. Otherwise, if the user is already part of the system, (s)he is redirected to the appropriate landing page, of which there are three types: */students* for student users, */eventleader* for event leader users, and */admin* for admin users. These three routes are discussed in further detail in sections 3.1 to 3.3.

In addition, if an error occurs, the user is redirected to */error* in which an error page is rendered, informing the user to contact the house assistant dean (the client) for assistance.

3.1 /students

/students loads the event landing page for student users. Students are able to edit their personal profiles and sign up for events; each of these actions is handled by a separate route that processes the POST request sent when each action is submitted.

/students/editprofile processes POST requests made when students make changes to their personal profiles and submit those changes. The system is then redirected to */students* upon successful processing of such a POST request.

/students/signup processes POST requests made when students sign up for events. The system is then redirected to */students* upon successful processing of such a POST request.

3.2 /eventleader

/eventleader loads the event landing page for event leader users. Event leaders are able to edit their personal profiles, sign up for events, create events, edit events, delete events, add attendees to the sign-up list of an event, remove attendees from the sign-up list of an event, add students to the system, and remove students currently in the system from the system; each of these actions is handled by a separate route that processes the POST request sent when each action is submitted.

/eventleader/editprofile processes POST requests made when event leaders make changes to their personal profiles and submit those changes. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/signup processes POST requests made when event leaders sign up for events. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/event/create processes POST requests made when event leaders create an event. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/event/edit processes POST requests made when event leaders edit an event. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/deleteevent processes POST requests made when event leaders delete an event. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/addattendee processes POST requests made when event leaders add someone to the sign-up list of an event. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/removeattendee processes POST requests made when event leaders remove someone from the sign-up list of an event. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/addstudent processes POST requests made when event leaders add a new student to the system. The system is then redirected to */eventleader* upon successful processing of such a POST request.

/eventleader/removestudent processes POST requests made when event leaders remove a student from the system. The student removed must currently be in the system. The system is then redirected to */eventleader* upon successful processing of such a POST request.

3.3 /admin

/admin loads the event landing page for admin users. Admins are able to edit their personal profiles, sign up for events, create events, edit events, delete events, add attendees to the sign-up list of an event, remove attendees from the sign-up list of an event, add users to the system, and remove users currently in the system from the system;

each of these actions is handled by a separate route that processes the POST request sent when each action is submitted.

/admin/editprofile processes POST requests made when admins make changes to their personal profiles and submit those changes. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/signup processes POST requests made when admins sign up for events. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/event/create processes POST requests made when admins create an event. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/event/edit processes POST requests made when admins edit an event. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/deleteevent processes POST requests made when admins delete an event. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/addattendee processes POST requests made when admins add someone to the sign-up list of an event. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/removeattendee processes POST requests made when admins remove someone from the sign-up list of an event. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/adduser processes POST requests made when event leaders add a new user (student, event leader, or admin) to the system. The system is then redirected to */admin* upon successful processing of such a POST request.

/admin/removeuser processes POST requests made when event leaders remove a user (student, event leader, or admin) from the system. The user removed must currently be in the system. The system is then redirected to */admin* upon successful processing of such a POST request.

4 Hosting

The Bethe OPS system is hosted via AWS. Amazon EC2 running on an Ubuntu instance (i.e., a virtual machine) is used to host the app. The database for the system is loaded on Amazon RDS. These two components (the app and the database) are put into the same security group, which allows them to communicate with one another.

In addition, the client and the team acquired the following domain name for Bethe OPS: http://events.hansbethehouse.cornell.edu/. This domain name points to the host name assigned to the EC2 instance for the app by Amazon EC2.

5 Login Authentication

The system makes use of CUWebLogin for login authentication in order to offer students a familiar interface for logging into the system. There are three main components of the implementation of login authentication.

5.1 Shibboleth and SAML

CUWebLogin is a service used to authenticate users at Cornell for access into several various websites. CUWebLogin shares the same single-sign as the Shibboleth system. Thus, by integrating Shibboleth into the system, CUWebLogin could be used to authenticate users. Below are a few links shared by Cornell IT for more information about Shibboleth at Cornell University:

- https://identity.cit.cornell.edu/authn/shibbolethintro.html
- https://confluence.cornell.edu/display/SHIBBOLETH/Shibboleth+at+Cornell+Page

In Shibboleth systems, there are entities known as Identity Providers (IdP) and Service Providers (SP). The IdP can be thought of as a system that does the authentication while SPs are the entities requiring authentication. The Bethe OPS System is the SP and the Cornell IdP provides the authentication. Currently, the system uses the “test” IdP provided by Cornell for simple authentication and testing purposes.

5.2 Passport and Passport-SAML

To integrate authentication and utilize Shibboleth, the *passport* module is used. Specifically, a variant called “passport-SAML” is used to allow the system to handle authentication with Shibboleth systems like CuWebLogin. To use the *passport* module

with Shibboleth, an extension of *passport* called *passport-saml* is also used. By passing it info about the SP, IdP, certificates, it allows for authentication using CUWebLogin.

5.3 Cookies

Once users are logged in, the system needs a way to track the users. To handle this tracking, the system makes use of internet cookies. Internet cookies are simply small amounts of data stored in the browser about the user of the browser. For the Bethe OPS system, the cookies will only store the netID of the logged in user as well as a numerical identifier used by the system. With this small amount of information, the system will know what privileges to grant the user as well as what information to show (or withhold) from the user.

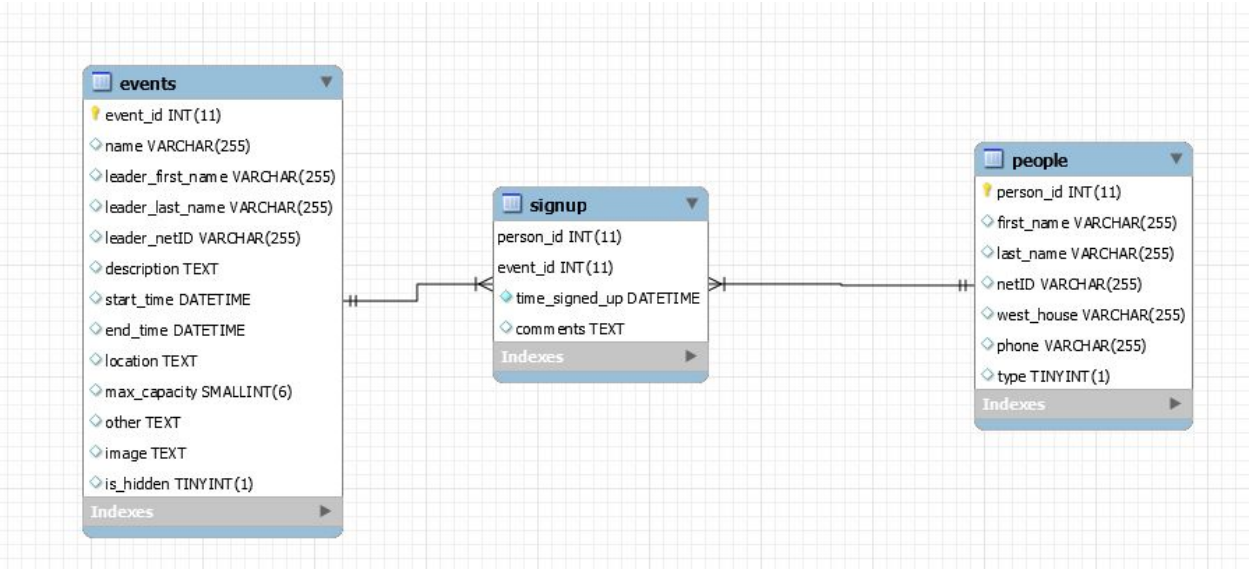
6 Email Notifications

The system sends the following kinds of email notifications:

- Email confirming a user's sign-up for an event
- Email confirming a user's removal from the roster for an event (s)he is signed up for
- Email confirming a user's removal from the waitlist and spot on the roster of an event (s)he is signed up for
- Emails reminding a user of an event (s)he is signed up for
    - One sent **one day** before the event
    - One sent **two hours** before the event

Nodemailer, a module for Node.js that allows email sending, was used to implement the email notification system within the Bethe OPS system. Email notifications are constructed and sent on the server side as described in section 3. Bethe OPS uses Gmail as the service and sends emails with the email address, betheops@gmail.com, as approved by the client.

7 Database



As shown in the EER diagram above, the final database design has become much simpler compared to previous designs. There is now only a many-to-many relationship between *people* and *events*, which is indicated by the intermediary table called *signup*. By having this relationship between *people* and *events*, this allows any user, no matter the privilege level, to sign up to events. The team found rather than having separate tables for students, event leaders, and admins, it would be best to use a single *person* table since there were no fields of information that only applied to a subset of the users. The same type of information was stored for all users so it would be best to simply store all of it in a single table. While it may be good database design to split up entities into separate tables, it ended up causing many other complications that were avoided by having a single table that contained all the users. To distinguish between the different types of users, the *people* table also stores information about what privilege levels each user has.

Without a table specifically for event leaders and admins, the information about who led events are now stored in the event table itself. More information about the three tables shown above is below.

7.1 events

The *events* table stores the events created on Bethe OPS and information associated with each event. *events* includes the following fields:

- *event_id*: an integer representing the unique ID of an event

- *name*: a string representing the name of an event
- *leader_first_name*: a string representing the first name of the event leader for the corresponding event
- *leader_last_name*: a string representing the last name of the event leader for the corresponding event
- *leader_netID*: a string representing the net ID of the event leader for the corresponding event
- *description*: a string representing the description of an event
- *start_time*: a datetime representing the start time of an event
- *end_time*: a datetime representing the end time of an event
- *location*: a string representing the location of an event
- *max_capacity*: an integer representing the attendee quota of an event
- *other*: a string representing additional details regarding an event
- *image*: a string representing the image path of the poster image used for an event
- *is_hidden*: an integer with value 0 or 1 representing whether the attendees list is hidden for an event

7.2 people

The *people* table stores information regarding each person or user in the system. *people* includes the following fields:

- *person_id*: an integer representing the unique ID of a person or user in the system
- *first_name*: a string representing the first name of a user
- *last_name*: a string representing the last name of a user
- *netID*: a string representing the net ID of a user
- *west_house*: a string representing which West Campus house (or building if the user does not live on West Campus) a user lives in
- *phone*: a string representing the phone number of a user
- *type*: an integer with value 0 (student user), 1 (event leader user), or 2 (admin user) representing the type of user a user is
    - NOTE: There is one exception for *type* in which the house assistant dean, Erica Ostermann, has *type* value 3.

7.3 signup

The *signup* table stores all event sign-ups submitted on Bethe OPS. *signup* includes the following fields:

- *person_id*: an integer representing the unique ID of a person or user in the system
- *event_id*: an integer representing the unique ID of an event
- *time_signed_up*: a datetime representing when a user signed up for an event

- *comments*: a string representing any comments a user may include when signing up for an event

8 Program Design

As mentioned in section 2, the system is a web application implemented using mainly Node.js and Express, as well as templating via EJS. **npm** is a Node.js package manager required to install certain Javascript libraries and build the Node.js app. Each component of the project is described in the following sections. The directory, *documentation*, is not part of the program design but is included in the project directory to keep project materials (source code and documentation) in one place as requested by the client.

8.1 node_modules/

node_modules is a directory only for build tools. It contains packages and libraries installed when the command **npm install** is run. *node_modules* should not be manually modified.

8.2 package.json and package-lock.json

package.json defines what libraries will be installed into node_modules when **npm install** runs. It lists the packages the web application depends on.

package-lock.json is automatically generated for any operations where **npm** changes either the node_modules directory or package.json.

package.json and *package-lock.json* generally do not need to be manually modified unless scripts, authors, and dependencies information need to be manually added or revised.

8.3 app.js

app.js contains server-side code written in Node.js that starts the server for the app and calls the routes. *app.js* is also where packages and files are “required”; Node.js uses a module called **require** that requests modules needed for the app on the server-side.

8.4 exports/

exports contains other server-side code aside from *app.js*. Such code includes the implementation of routers, connection to the database, email notifications, and functions that render the pages on the front-end and execute queries to the database.

8.4.1 exports/db.js

db.js creates and exports the MySQL connection to the database used for the Bethe OPS system for the routes to use.

8.4.2 exports/makeHome.js

makeHome.js renders the event landing page (i.e., home page) for each type of user (student, event leader, and admin). This includes passing the profile for the user currently logged in and data concerning the events and users currently in the system to the front-end to render the data in other features of the app, such as the "Edit Profile" feature.

8.4.3 exports/common_functions.js

common_functions.js contains functions that are used to support actions common to the event landing pages of all three types of users. These actions include editing the user profile, signing up for an event, and removing an attendee from an event sign-up list (the user logged in or another user signed up for the event).

8.4.4 exports/admin_EL_functions.js

admin_EL_functions.js contains functions that are used to support actions exclusive to event leader users and admin users. These actions include adding an attendee to an event sign-up list, creating an event, editing an event, deleting an event, adding a user to the system, and removing a user from the system.

8.4.5 exports/storage.js

storage.js manages the storage of files uploaded via Bethe OPS. This specifically pertains to poster files that are uploaded when creating events.

8.4.6 exports/email.js

email.js contains server-side code that supports email notification functionalities. To send emails, *email.js* uses Nodemailer, as mentioned in section 6. The email functionalities supported include sending an email to:

- Confirm a user's event sign-up
- Notify a user that (s)he has been added to an event roster
- Notify a user that (s)he has been removed from an event roster
- Notify a user that (s)he has been removed from the waitlist and added to the roster of an event
- Remind a user one day before of an event (s)he is signed up for happening the next day

- Remind a user two hours before of an event (s)he is signed up for happening in two hours

8.5 routes/

routes is a directory that contains the routes of the system, which are organized into separate files.

8.5.1 routes/index.js

index.js sets up a router that calls the routes for the three event landing pages: */students*, */eventleader*, and */admin*.

8.5.2 routes/students.js

students.js contains all routes pertaining to student users.

8.5.3 routes/eventleader.js

eventleader.js contains all routes pertaining to event leader users.

8.5.4 routes/admin.js

admin.js contains all routes pertaining to admin users.

8.6 public/

public is a directory that contains all static files such as images, styles, and Javascript.

8.6.1 public/css

css contains all stylesheets for the app.

css/style.css is the main stylesheet that contains all custom styling implemented by the team.

css/materialize.css and css/*materialize.min.css* are stylesheets provided by the Materialize framework.

8.6.2 public/images

images stores all images used by the app. Images for chevrons used for the weekly calendar are stored in *images/chevrons*. Uploaded images used for the event poster graphics are stored in *images/posters*.

8.6.3 public/js

js stores Javascript files for the app.

js/materialize.js and *js*/*materialize.min.js* are scripts used by the Materialize framework.

js/init.js is a script that initializes and defines the behavior of several elements of the interface. Such initialization includes populating dropdowns for different form fields and that of elements provided by Materialize CSS.

js/sidebar.js supports a number of functionalities for the sidebar. These functionalities include:

- Displaying the weekly calendar
- Loading event posters in the sidebar
- Jumping to events for the date selected on the weekly calendar
- Loading the event profile for the event selected in the sidebar

8.7 views/

views is a directory that provides EJS templates that are rendered and served by the routes.

8.7.1 views/pages

pages contains the EJS templates for the various pages of the app.

pages/index.ejs renders the event landing page for student users. Similarly, *pages/eventleader.ejs* and *pages/admin.ejs* render the event landing pages for event leader users and for admins, respectively.

pages/signup.ejs renders the sign-up page for new student users signing up on the system and creating a profile.

8.7.2 views/partials

partials contains pieces of reusable code that are EJS templates and that represent components common across different pages of the app.

partials/header, *partials/footer*, and *partials/sidebar* are the header, footer, and sidebar, respectively, used in all pages.

partials/landingPageNavBar.ejs represents the navigation bar used for the student event landing page. *partials/landingPageNavBarEL.ejs* represents the navigation bar used for the event leader landing page. *partials/landingPageNavBarAdmin.ejs* represents the navigation bar used for the admin landing page.

8.8 database.sql

database.sql is the SQL script (written in MySQL) that creates and initializes the database for the Bethe OPS system.

8.9 BetheOPS.pem and BetheOPSConverted.ppk

BetheOPS.pem is the PEM file with the keypair associated with the EC2 instance for the app that AWS provides. Use this file when connecting to the EC2 instance via SSH.

BetheOPSConverted.ppk is the PPK file the original PEM file is converted to. Use this file when connecting to the EC2 instance via PuTTY.

8.10 IDP/

IDP contains the certificate of the test IdP instance, which is available here.

The production version of the certificate is also available, but since the system currently uses the test IdP, the test IdP certificate is needed.

8.10.1 IDP/cornell-idp-test.crt

cornell-idp-test.crt is the certificate that contains the public key that the SP, which in this scenario is the Bethe OPS system, uses to encrypt authentication messages to the IdP. The IdP then uses its private key to decrypt the encrypted message and proceeds to conduct authentication.