**MySeatTime**

**backend env variable is defined in nodemon.json**

*API*

**Club: /api/clubs/**

POST: / 🡺 create a club. Create a club event table. club user table

GET: / 🡺 get all clubs

GET: /:id 🡺 get a club

PUT: /:id 🡺 update a club

DELETE: /:id 🡺 delete a club

**Event:**

POST: /events/ 🡺 create an event.

1. Create a user table for the event
2. add the event to club event table.
3. add the event to big event table

GET: /events/ 🡺 get all events for this club

PUT: /events/:event\_id 🡺 update a specific event in this club

DELETE: /events/:event\_id 🡺 delete an event from this club

**Event Participants: (Need to figure out authentication to determine whether to put this under club or event)**

POST: /event/user/:id 🡺add a user to an event. NOT SUPPORT. user needs to sign up.

PUT: /event/users/:id 🡺update a specific user in an event. Paid or unpaid

DELETE: /event/user/:id 🡺delete a specific user in an event

**User:**

POST: /user/:user\_id => add a user to the club. NOT SUPPORT. User need to join.

GET: /users/ 🡺 get all users in the club

GET: /users/:user\_id 🡺 get a specific user in the club from club user table

PUT: /users/:user\_id 🡺 update a user in the club. such as CAR, Race #

DELETE: /users/:user\_id 🡺 delete a user from club

**Events: /api/events/**

GET: / 🡺 get all events

GET: /:id 🡺 get a specific event

GET: /users/ 🡺get all the users in an event

**User: /api/users**

POST: / 🡺 create a user, create user’s event table, club table, garage table

GET: / 🡺 get all users

GET: /users/:id 🡺 get a specific user

UPDATE: /user/:id 🡺 update a user

DELETE: /user/:id 🡺 delete a user

Event

POST: /event/:id 🡺 sign up an event

GET: /event/ 🡺 get all the events

GET: /event/:id 🡺 get an event

DELETE: /event/:id 🡺 cancel an event

Club

POST:/club 🡺 sing up a club

Garage

POST:/garage/ 🡺 add a car

GET: /garage/ 🡺 get garage cars

GET: /garage/:id 🡺 get a specific car in the garage

UPDATE: /garage/:id 🡺 edit a car

DELETE: /garage/:id 🡺 delete a car

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

**React Structure**

**App.js**

**MainNavigation.js -> MainHeader.js -> NavLinks.js**

**-> SideDrawer.js ->**

index.js

↓

App.js (defines paths)

↓

Events.js (get the values from backend then use

EventList.js and EventItem.js to format the return list)

↓

EventList.js

↓

EventItem.js

**Event Table:**

id: 'u1',

clubId:

name: 'SCCA - San Francisco Region - Solo 1',

title: 'SCCA - San Francisco Region - Solo 1',

eventImage:

startDate: '06/25/2020 Sat',

endDate: '06/26/2020 Sun',

venue: 'NASA Crows Landing Airport and Test Facility',

address: 'Crows Landing, CA',

coordinate: { lat: 37.4015069, lng: -121.1059222 },

description: "SCCA - San Francisco Region - Solo. Reminder: You have to work! We keep a running list of those of you have skipped out on work. Check it out HERE and make sure you aren't on it.",

courseMap:'https://www.bmwautocross.com/wp-content/uploads/2019/10/20191019-ggcautoxCourseMap-FINAL.png',

**Login/Logout mechanism**

When club login, Backend generates jwt and set expiration time in 24h

clubsController.js loginClub()

token = jwt.sign(

{ clubId: existingClub.id, email: existingClub.email },

JWT\_PRIVATE\_KEY,

{ expiresIn: '24h' }

);

Backend returns response that has jwt to Frontend.

Frontend stores jwt in browser LocalStorage.

flow in Frontend:

ClubAuth.js clubSubmitHandler() gets the response from Backend then call clubLogin() to save jwt

clubAuthContext.clubLogin(

responseData.clubId,

responseData.name,

responseData.token

);

App.js clubLogin() saves to localStorage

It saves clubId, clubToken, and expiration.

localStorage.setItem(

'userData',

JSON.stringify({

clubId: cid,

clubToken: ctoken,

expiration: tokenExp

})

);

After logging in, requests that needs authentication will embed token into the request sending to Backend. Backend will match it with the token re-generated by private key.

If token matches, we need to further make sure the requests coming from the owner of data, club or event owner, to prevent cross-site hacking.

clubId in the token is used for that verification. For example, inside updateEvent(), Backend uses it to validate if the clubId comes with the token matches the event owner clubId.

// we added userData in check-auth after verifying jwt

if (event.clubId.toString() !== req.userData.clubId) {

const error = new HttpError('Unauthorized operation!!!', 401);

return next(error);

}

**Auto Login**

When user login, at Backend, we generate jwt with expiresIn: ‘24h’. At front end, we create a logoutTimer in App.js. We use useEffect to keep track of state of clubAuthContext.clubLogin().

Whenever app.js been rendered the very first time, **starting the app or page refresh**, clubLogin() will be instantiated and only once for the entire app life span because it’s useCallback() function. So when it gets instantiated, the state changes, useEffect will be notified and work through our Auto Login logic:

useEffect(() => {

console.log('I am in auto login');

const storageData = JSON.parse(localStorage.getItem('userData'));

if (

storageData &&

storageData.clubToken &&

moment(storageData.expiration) > moment()

) {

clubLogin(

storageData.clubId,

null,

storageData.clubToken,

moment(storageData.expiration)

);

}

}, [clubLogin]);

To note: When app starts or page refreshing, when clubLogin() gets instantiated, login will fail because all the values are null. When auto login useEffect calls clubLogin(), we will have all the information get from localStorage ‘userData’. If they are all valid, we will be able to login.

**Auto Logout:**

// Auto logout

// dependecies: clubToken state changes when clubLogin() or clubLogout()

useEffect(() => {

if (clubToken && clubTokenExpDate) {

const remainingTime = moment(clubTokenExpDate) - moment();

// if timeout gets triggered meaing clubToken expires, clubLogout will be called

logoutTimer = setTimeout(clubLogout, remainingTime);

} else {

clearTimeout(logoutTimer);

}

}, [clubToken, clubLogout, clubTokenExpDate]);

Dependency clubToken changes whenever clubLogin() been called (from app starting/refreshing page or login menu) or clubLogout(). if both clubToken && clubTokenExpDate exist, we will call setTimeout with clubLogout() and remainingTime to timeout. \*\*\*setTimeout runs in background to keep track of remainingTime, once remainingTime == 0, it gets triggered and call clubLogout() to auto logout.

Formik + Yup

// Yup validationSchema. Yup does not work well on multiple files validation.

// If one file has an error, the other files in different field will also get an error.

// We will use field level validation for image and courseMap

const validationSchema = () =>

Yup.object().shape({

name: Yup.string().required('Please enter an event name'),

venue: Yup.string().required('Please enter venue'),

address: Yup.string().required('Please enter address'),

description: Yup.string().required(

'Please enter event description'

),

instruction: Yup.string().required('Please enter instruction')

})

<ErrorMessage /> only works for onBlur. It doesn’t work for manual trigger validation.

{/\* <ErrorMessage name="name">

{msg => (

<div className="event-form\_\_field-error"> {msg}</div>

)}

</ErrorMessage> \*/}

A screenshot of a cell phone

Description automatically generated

**FormBuilder**

FormBuilder.js => protocol that connects MySeatTime to ReactFormBuiler

index.jsx => export ReactFormBuilder ( from index.jsx), club form builder

=> export RectFormGenerator(this is actually from form.jsx but export in index.jsx)

this is to generate form that is used by users

=> export store (from ./stores/store.js)

Add a new item to FormBuilder

1. form-elements.jsx add class for new item. Layout shows in Form Builder is controlled in this file. export it as BaseFormElements.

2. In sortable-form-elements, convert BaseFormElelments to sortableElement(defined in sortable-form-elements.jsx)

preview.jsx defines insertCard() moveCard()… move item from toolbar to form or vice versa. **It renders “<FormElementsEdit> if this.props.editElement !== null” or <SortableFormElement>**

Flow moving an item from toolbox to form:

sortable-element.jsx => preview.jsx insertCard => saveData => getElement => SortableFormElement => formElement => each component has rendering function that renders itself to display that component on the form.

mutable attribute will be set to false for displaying item on the form (rendering in form-element.jsx) because this is not editable. only set to true in form editing mode.

Flow of changing values in Editor:

1. form-elements-edit.jsx onEditStateChange() changes
   1. this.state contains: data, dirty, and element
      1. data: all the elements in the form
      2. dirty: indicating whether the data has been changed or not.
      3. element: the component we are currently editing
2. in render(), it gets “name”, “content”, or “label” from element object and use it as default.

let editorState;

if (this.props.element.hasOwnProperty('name')) {

console.log('name = ', this.props.element.name);

editorState = this.convertFromHTML(this.props.element.name);

}

if (this.props.element.hasOwnProperty('content')) {

console.log('content = ', this.props.element.content);

editorState = this.convertFromHTML(this.props.element.content);

}

if (this.props.element.hasOwnProperty('label')) {

console.log('label = ', this.props.element.label);

editorState = this.convertFromHTML(this.props.element.label);

<Editor defaultEditorState={editorState} …/>

1. Editor onEditStateChange calls onEditorStateChange to set the new value
2. Editor onBlur={this.updateElement.bind(this)} creates an updateElement function using “this” as argument. The function calls preview.updateElement(element) in which

/event/components/FormBuilder.js to add a new tool item that can be loaded at very beginning.

The original default toolbox items are defined in toobar.jsx but we want to load our own set of customized tools which is located in

in /event/components/EntryFormOptoins.js.

**form-elements-edit.jsx** is for clicking on “edit” icon to open up element for editing.

For example: Layout of Radio Button is defined in form-elements-edit.jsx

A screenshot of a cell phone

Description automatically generated

dynamic-option-group.jsx defines the layout of MultipleRadioButtonGroup in editing mode.

3 Ways to Copy Objects in JavaScript

https://www.javascripttutorial.net/object/3-ways-to-copy-objects-in-javascript/

Editor:

default value is set by “defaultEditorState”

<Editor

toolbar={toolbar}

defaultEditorState={editorState}

onBlur={this.updateElement.bind(this)}

onEditorStateChange={this.onEditorStateChange.bind(

this,

0,

'label'

)}

stripPastedStyles={true}

placeholder="Please enter label"

// editorClassName="rdw-editor-label"

/>

editorClassName=”rdw-editor-label” controls editor content window layout where “Day 1” is. Comment out will not display boundary as showing below.

A screenshot of a cell phone

Description automatically generated

form.jsx

validateForm 🡺 \_isInvalid(item) 🡺