**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) 🡺

**BUILD REACT**

> npm run build

swap process.env inside .env.production

(.env is used for dev)

Heroku

use Heroku CLI

start from backend, no need to build backend

> heroku login

follow instructions on heroku page using Deployment Method: Heroku Git

Graphical user interface, application

Description automatically generated

Before push to heroku, make change in package.json

Graphical user interface, text

Description automatically generated



Change “nodemon app.js” to “node app.js”

Since we change to node, nodemon.json will not be used any more. We will need to add process.env to Heroku.

In nodemon.json

{

"env": {

"MONGO\_DB\_USER": "hung",

"MONGO\_DB\_PASSWORD": "hung",

"MONGO\_DB\_NAME": "MySeatTime",

"GOOGLE\_MAP\_API\_KEY": "AIzaSyACZSZtNh33iMJUXVaN4X-wOeNnlNjJajk",

"DUMMY\_CLUB\_ID": "5ef702c7ba7511499165e653",

"AWS\_ACCESSKEYID": "AKIAQ3RGE76SUKXZKJOU",

"AWS\_SECRETACCESSKEY": "S7zSD9d/fJOpK+SEcrZFc2WtogDJJv9jWc78WfcL",

"S3\_REGION": "us-west-1",

"S3\_ACL": "public-read",

"S3\_BUCKET\_NAME": "myseattime-dev",

"S3\_BUCKET\_NAME\_PROD": "myseattime-image",

"S3\_URL": "https://myseattime-dev.s3.us-west-1.amazonaws.com",

"S3\_URL\_PROD": "https://myseattime-image.s3.us-west-1.amazonaws.com",

"CLOUDFRONT\_URL": "https://dm6o7kx2v6zc3.cloudfront.net",

"CLOUDFRONT\_URL\_PROD": "https://d1139u9qxzy8k3.cloudfront.net"

}

}

Go to Heroku page => Setting => Config Vars (Reveal)

"AWS\_ACCESSKEYID": "AKIAQ3RGE76SUKXZKJOU",

"AWS\_SECRETACCESSKEY": "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",

"S3\_REGION": "us-west-1",

"S3\_ACL": "public-read",

"S3\_BUCKET\_NAME\_PROD": "myseattime-image",

"S3\_URL\_PROD": "https://myseattime-image.s3.us-west-1.amazonaws.com",

"CLOUDFRONT\_URL\_PROD": "https://d1139u9qxzy8k3.cloudfront.net"

Change "S3\_BUCKET\_NAME\_PROD" to “S3\_BUCKET\_NAME”.

Change "S3\_URL\_PROD" to “S3\_URL”.

Change “CLOUDFRONT\_URL\_PROD” to “CLOUDFRONT\_URL” in Heroku (not showing in the following pic)

Heroku git issue “Your branch is up to date with 'heroku/master'.”

in BackEnd/

// added a remote to local repository

> heroku git:remote -a myseattime

> git add .

> git commit -am “XXXXXXX”

> git push heroku master

MongoDB

add Heroku backend IP

for now, we don’t have a static IP so we will use 0.0.0.0/0. Even we allow anyone to connect with MongoDB, but under Database Access, we already configure user names to allow to connect with DB

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Collections on MongoDB

Graphical user interface, application

Description automatically generated

Deploy Front End to S3

S3

<https://docs.aws.amazon.com/AmazonS3/latest/dev/website-hosting-custom-domain-walkthrough.html#root-domain-walkthrough-create-buckets>

<https://dev.to/oyetoket/how-to-deploy-your-frontend-application-on-aws-s3-31m9>

CloudFront

<https://docs.aws.amazon.com/AmazonS3/latest/dev/website-hosting-cloudfront-walkthrough.html>

Stripe

Text

Description automatically generated

Tutorial Video:

<https://youtu.be/mI_-1tbIXQI?t=1831>

Design Idea:

Front End:

Use stripe React element

<https://stripe.com/docs/stripe-js> switch to React

Text

Description automatically generated

# Stripe Flow For Set up future payments

<https://stripe.com/docs/payments/save-and-reuse#use-payment-method>

Graphical user interface, text, application, email

Description automatically generated



**1.** **Create a customer in userController.createUser() – server side**

save stripeCustomerId: customer.id in USER

**2. When user submits entry, create setupIntent then collect card details:**

**both SetupIntent and PaymentIntent need to have on\_behalf\_of**

**Text

Description automatically generated**



**Relations between SetupIntent and payment\_method is we need to use SetupIntent to get client\_secret. With client\_secret, we can then create PaymentMethod using:**

Text

Description automatically generated

from result, we can get payment\_method

result.setupIntent.payment\_method

1. create a setupIntent on server-side – server side

A [SetupIntent](https://stripe.com/docs/api/setup_intents) is an object that represents your intent to set up a customer’s card for future payments.

We need client\_secret from setupIntent.

**const** intent = **await** stripe.setupIntents.**create**({

customer: user.stripeCustomerId,

});

Save setupIntent in each Payment

1. Collect card details on client-side

Render <CardSetupForm />

<CardSetupForm /> needs to be embedded by Elements

<**Elements***stripe*={**stripePromise**}>

<**CardSetupForm**/>

</**Elements**>

CardSetupForm uses

<CardSection /> that composes a credit card input page using <CardElement />

1. confirmCardSetup in CardSetupForm.js need extra work:

stripe.confirmCardPayment may take several seconds to complete. During that time, disable your form from being resubmitted and show a waiting indicator like a spinner. If you receive an error, show it to the customer, re-enable the form, and hide the waiting indicator. If the customer must perform additional steps to complete the payment, such as authentication, Stripe.js walks them through that process.

**3. Charge the saved card later – server side**

When you are ready to charge your customer off-session, use the Customer and PaymentMethod IDs to create a PaymentIntent.

Text

Description automatically generated



Error handling:

if (err.code === 'authentication\_required') {

// need customer to authenticate charge

} else if (err.code) {

// insufficient fund, ask customer to provide a different card

else {

// unknown error, ask club to try again later

}

Testing cards:

| NUMBER | DESCRIPTION |
| --- | --- |
| 4242424242424242 | Succeeds and immediately processes the payment. |
| 4000002500003155 | Requires authentication for the initial purchase, but succeeds for subsequent payments  (including off-session ones) as long as the card is setup with setup\_future\_usage. |
| 4000002760003184 | Requires authentication for the initial purchase, and fails for subsequent payments  (including off-session ones) with an authentication\_required decline code. |
| 4000008260003178 | Requires authentication for the initial purchase, but fails for subsequent payments  (including off-session ones) with an insufficient\_funds decline code. |
| 4000000000009995 | Always fails (including the initial purchase) with a decline code of insufficient\_funds. |

For the full list of test cards see our guide on [testing](https://stripe.com/docs/testing).