CSE 596 CSE Capstone Design Personal Report

TheRealCapstone: Event Ticketing Page

April 28, 2022

Student Name: Elijah Moppins

Team Members:

Abu Abukar, Caty Battjes, Steven Johnson, Elijah Moppins, Lam Nguyen, Jose Ramos

Customers:

TheRealSeat

Overview of task assignment and project activities

Week 1-2: Read and Setup AWS S3 Documentation, Go Through React App Tutorial, AWS SAM Tutorial

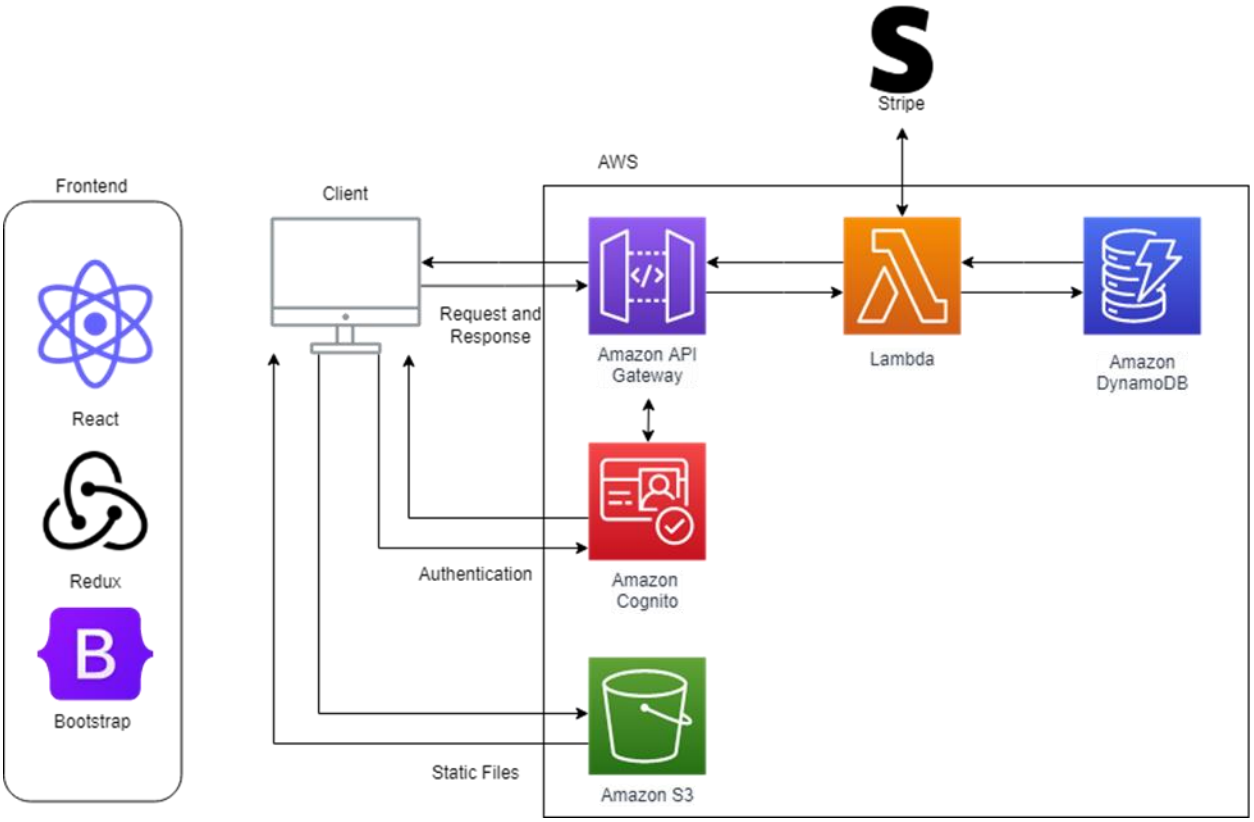
Week 3-5: Mock Event/Product Page, Event Page Header/Navbar and Footer, Event Page Hero Section, Make STG Bucket and Branch

Week 5-6: Redux Tutorial, Implement Redux in your own Branch, Update Hero Section,

Week 6-8: AWS SAM Implementation, Lambda Implementation, Dynamo Implementation

Week 9-10: Create Stripe Account, Implement Stripe (test key only)

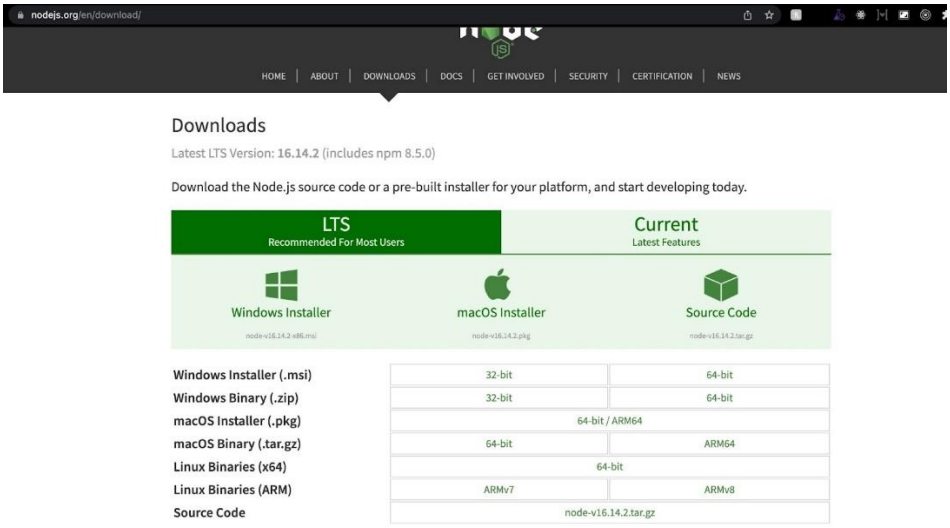
Week 10-11 Test Application



Overview of task assignment and project activities.

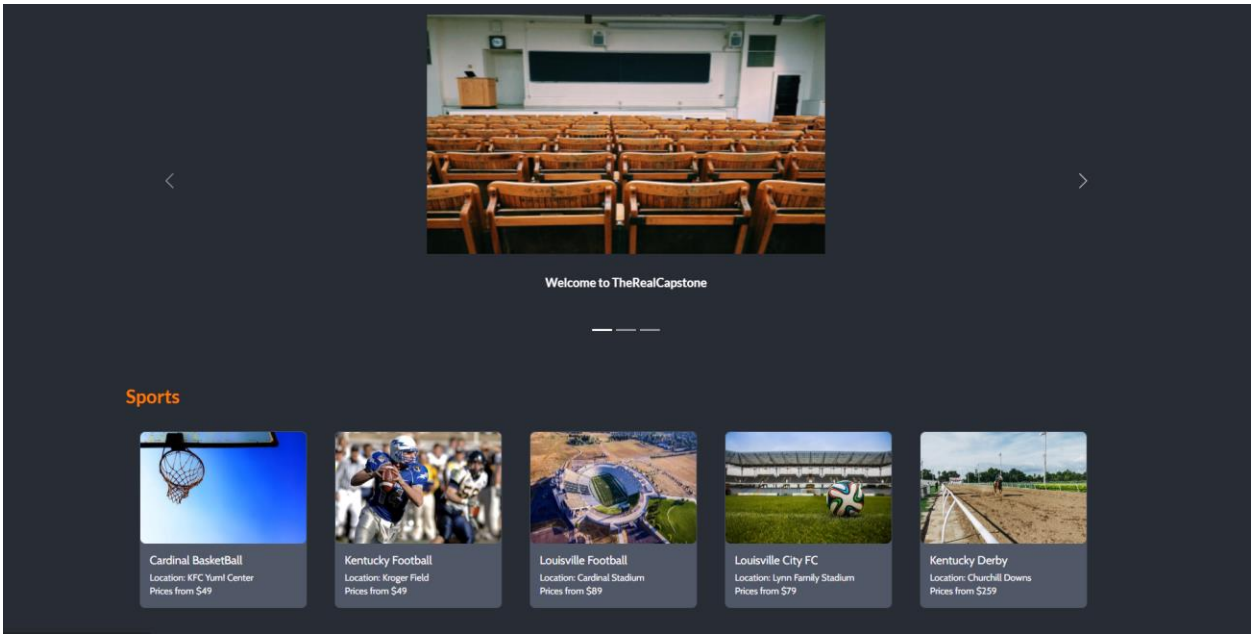
Week 1-2: Read and Setup AWS S3 Documentation, Go Through React App Tutorial, AWS SAM Tutorial

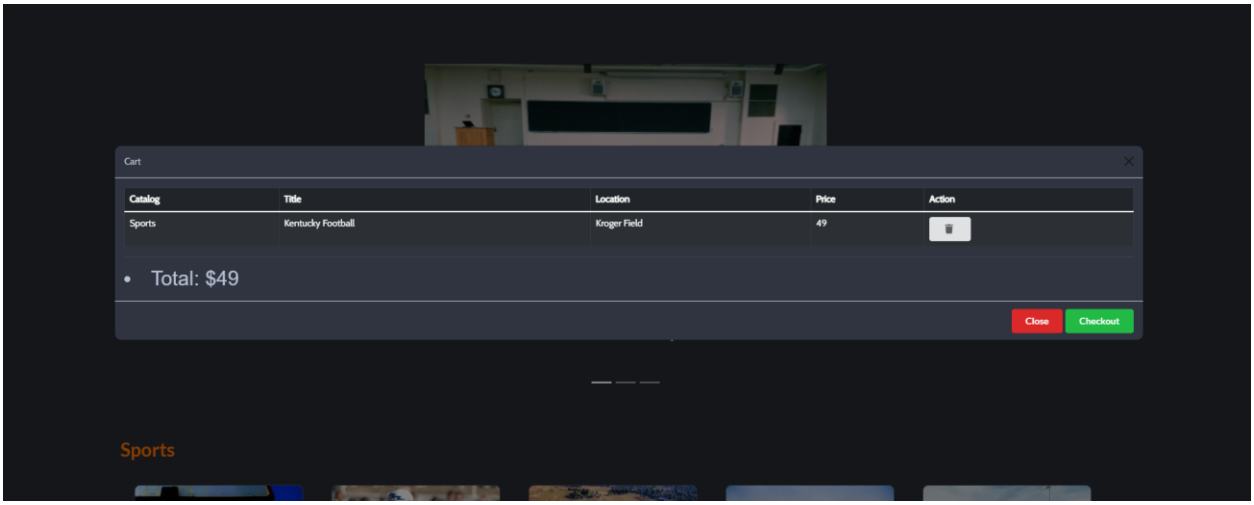
In Week One we went over goals and the reach of the project and also what objects we wanted to complete. My tasks we as outlined in Week 1 to get familiar with AWS technologies and to read on how Reach Web Apps operate. And to set up my computer to work with these technologies. We worked with our coordinator to set these things up.



Week 3-5: Mock Event/Product Page, Event Page Header/Navbar and Footer, Event Page Hero

In these weeks I was able to work on the Product page, the event page, and the hero section of the product page to display events. During these weeks, the team held talks to understand what the functionality of the application and how we would sell the tickets to the user. We did some research into website design because we had to design the entire UI from scratch. The product page looks like this:





Week 5-6: Redux Tutorial, Implement Redux in your own Branch, Update Hero Section,

In these weeks we researched the uses and benefits of implement Redux state tech in our application and implemented it in the application. And I had to update the hero section to match a new layout the team decided on. The hero section is the main events that we were displaying on the top of the website, and these could be used to feature ads or promoted events to users. This was an important feature because catering to the needs of the user is a major part of the web design process. Here are some of the code snippets for the redux implementation.

```
const cart = createSlice({
    name: 'cart',
    initialState ,
    reducers: {
        addToCart(state, action) {
            console.log(action.payload);
            state.cartEvents.push(action.payload);
            state.cartTotalQuantity += 1;
            state.cartTotalAmount += parseFloat(action.payload.event_pricesFrom);
        },
        removeFromCart(state, action) {
            state.cartTotalAmount -= parseFloat(state.cartEvents[action.payload].event_pricesFrom);
            state.cartEvents.splice(action.payload, 1)
            state.cartTotalQuantity -= 1;
        }
    }
})

export const { addToCart, removeFromCart } = cart.actions
```

```
export default function configureStore(){
   const store= createStore(rootReducer,composeWithDevTools(applyMiddleware(thunk)))


   return store;
}
```

```
const store = configureStore();

return (
  <Provider store={store}>
    <BrowserRouter>
      <Navbar />
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/disclaimer" element={<Disclaimer />} />
        <Route path="/events" element={<Events />} />
        <Route path="/createevent" element={<EventForm />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
      <Footer />
    </BrowserRouter>
  </Provider>
```

Week 6-8: AWS SAM Implementation, Lambda Implementation, Dynamo Implementation

In these weeks, the team had to learn the AWS technology and how making our application serverless would make development faster and our application more secure. I read over the documentation and linked our code to the AWS S3 buckets which held the code and its changes similar to GIT. It took weeks to go over how to use S3 , Lambda, Cloud Front , and DB but eventually we were able to send API calls and update DB entries for our application. Here is some code that shows how its connected to AWS SAM

```yaml
name: Production Build
on: push

env:
  STRIPE_PUBLISHABLE_KEY: ${{ secrets.TEST_STRIPE_PUBLISHABLE_KEY }}
  STRIPE_SECRET_KEY: ${{ secrets.TEST_STRIPE_SECRET_KEY }}

jobs:
  src:
    name: build src files
    if: always()
    runs-on: ubuntu-latest
    timeout-minutes: 20
    steps:
    - name: Check out repository
      uses: actions/checkout@v2

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: us-east-1

    - name: Map Branch to DEPLOYMENT
      id: check_branch
      run: |
        echo "DEPLOYMENT=$(if [[ ${GITHUB_REF##*/} = main ]];
            then echo "PRD";
        elif [[ ${GITHUB_REF##*/} = stg ]];
            then echo "STG";
        elif [[ ${GITHUB_REF##*/} = caty* ]];
            then echo "CATY";
        elif [[ ${GITHUB_REF##*/} = abu* ]];
            then echo "ABU";
        elif [[ ${GITHUB_REF##*/} = jose* ]];
            then echo "JOSE";
        elif [[ ${GITHUB_REF##*/} = steven* ]];
            then echo "STEVEN";
        elif [[ ${GITHUB_REF##*/} = lam* ]];
            then echo "LAM";
        elif [[ ${GITHUB_REF##*/} = elijah* ]];
            then echo "ELIJAH";
        elif [[ ${GITHUB_REF##*/} = * ]];
            then echo "DEV"; fi)" >> $GITHUB_ENV

    - name: Map Branch to Domain ENV VAR
      run: |
        echo "DOMAIN=$(if [[ $DEPLOYMENT = PRD ]];
          then echo "www.therealcapstone.com";
        else
          echo "${DEPLOYMENT,,}.therealcapstone.com"; fi)" >> $GITHUB_ENV
    - name: Yarn Install
      run: |
        yarn install
    - name: Production Build
      run: yarn build

    - name: Deploy to s3
      run: |
        aws s3 sync ./build s3://$DOMAIN --acl public-read --delete
```

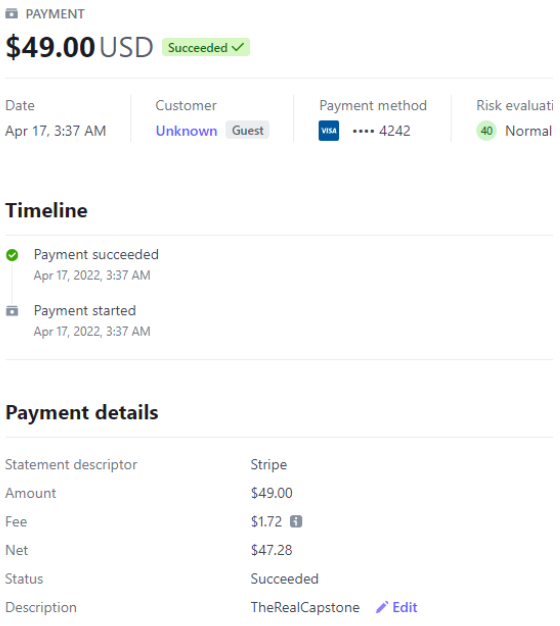
```yaml
EventAPI:
  Type: AWS::Serverless::Api
  Properties:
    Name: !Sub "EventAPI_${ENVIRONMENT}"
    StageName: Prod
    Cors:
      AllowMethods: "'*'"
      AllowHeaders: "'*'"
      AllowOrigin: "'*'"

UserTransactionAPI:
  Type: AWS::Serverless::Api
  Properties:
    Name: !Sub "UserTransactionAPI_${ENVIRONMENT}"
    StageName: Prod
    Cors:
      AllowMethods: "'*'"
      AllowHeaders: "'*'"
      AllowOrigin: "'*'"

PaymentIntentAPI:
  Type: AWS::Serverless::Api
  Properties:
    Name: !Sub "PaymentIntentAPI_${ENVIRONMENT}"
    StageName: Prod
    Cors:
      AllowMethods: "'*'"
      AllowHeaders: "'*'"
      AllowOrigin: "'*'"
```

Week 9-10: Create Stripe Account, Implement Stripe (test key only)

This week the team worked on the Add to cart functionality and linked the functions to stripe so that we would charge card. I worked on the webhook which would notify the website if the transaction were successful. We began to prep our application for the showcase.




```
const CheckoutForm = ({ show, handleClose }) => {
  // const [Success,setSuccess] = useState(false);
  const stripe = useStripe();
  const elements = useElements();
  const cart = useSelector((state) => state.cart);
  const initialValue = {
    FirstName: "",
    LastName: "",
    Email: "",
  };
  const validationSchema = Yup.object({
    FirstName: Yup.string().required("You must enter your first name"),
    LastName: Yup.string().required("You must enter your last name"),
    Email: Yup.string().required("You must enter your email address"),
  });

  const handleSubmit = async (value) => {
    console.log(value);
    const { error, paymentMethod } = await stripe.createPaymentMethod({
      type: "card",
      card: elements.getElement(CardElement),
    });
    if (!error) {
      try {
        const { id } = paymentMethod;

        const response = await agent.PaymentIntent.create({
          amount: cart.cartTotalAmount * 100,
          currency: "usd",
          description: "TheRealCapstone",
          payment_method: id,
        });

        //id containers payment method
        console.log(response);
        if (response.data.success) {
          console.log("Successful payment");
          // setSuccess(true);
        }
      } catch (error) {
        console.log(error);
      }
    }
  };
```

Areas of self-development and new learning that were needed/attempted.

Here are areas of self-development I was able to work on Throughout the project.

Communication: Communication was the biggest part of the project because the tasks were not already predefined, so we basically had to meet to plan and meet to make sure that what were working on did not overstep someone else. Also keeping up with the PM and what he wanted us to do. We used Notion and Discord for communication.

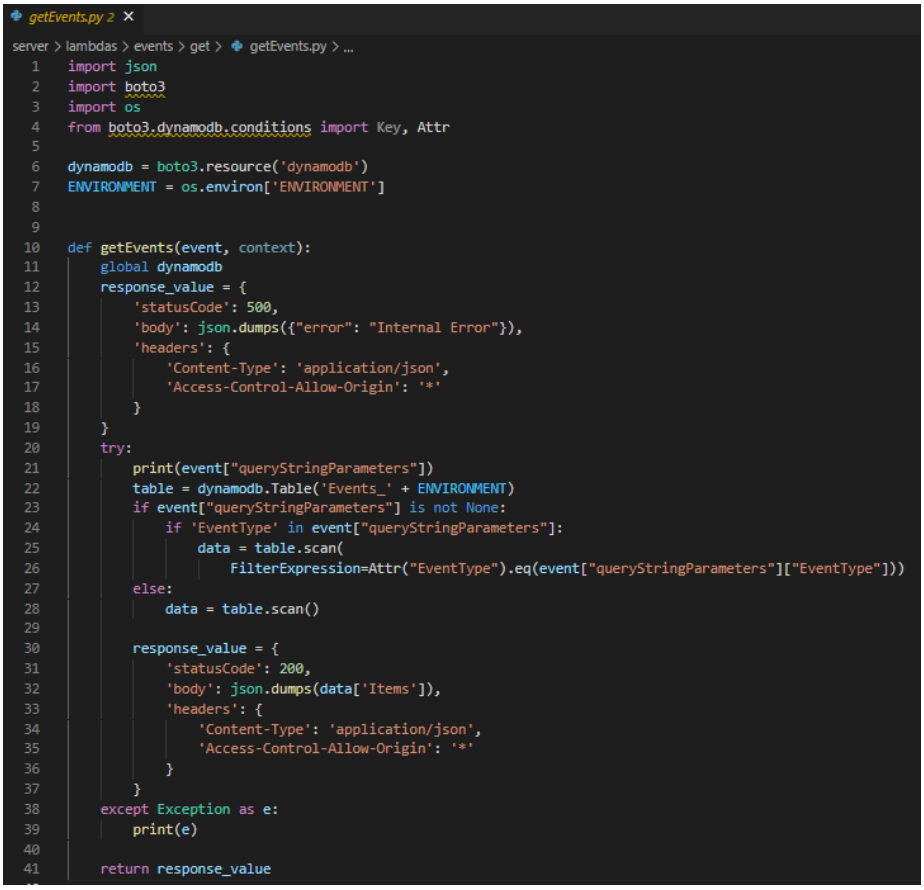
Development: One of the major things I learned about development these past years was that development = learning. So, software developer means software learner, learning new technologies and learn how to do the same thing in different ways are a pillar of this career. And I was able to develop software with a new stack of technologies for this capstone.

AWS Stack: This was a major learning point, the AWS stack is large and there are portions that have UI, CLI, and just plain JSON responses that you have to parse through to get information about what's happening within your application. I know that learning AWS stack is useful, so it was an enjoyable experience.

Outline of specific contributions including portions of code and/or hardware if relevant

I worked on multiple parts of the application, and I will post the sections of code that I contributed to.

Code To pull events



```python
import json
import boto3
import os
from boto3.dynamodb.conditions import Key, Attr

dynamodb = boto3.resource('dynamodb')
ENVIRONMENT = os.environ['ENVIRONMENT']


def getEvents(event, context):
    global dynamodb
    response_value = {
        'statusCode': 500,
        'body': json.dumps({"error": "Internal Error"}),
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        }
    }
    try:
        print(event["queryStringParameters"])
        table = dynamodb.Table('Events_' + ENVIRONMENT)
        if event["queryStringParameters"] is not None:
            if 'EventType' in event["queryStringParameters"]:
                data = table.scan(
                    FilterExpression=Attr("EventType").eq(event["queryStringParameters"]["EventType"]))
        else:
            data = table.scan()

        response_value = {
            'statusCode': 200,
            'body': json.dumps(data['Items']),
            'headers': {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*'
            }
        }
    except Exception as e:
        print(e)

    return response_value
```

Code to update a DB entry with the webhook

```python
def postWebhook(event,context):

    dynamodb = boto3.resource('dynamodb')
    ENVIRONMENT = os.environ['ENVIRONMENT']
    eventBody = json.loads(event['body'])

    response_value = {
        'statusCode': 500,
        'body': json.dumps({"error": "Internal Error"}),
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        }
    }

    # Handle the event

    table = dynamodb.Table('User_Transactions_' + ENVIRONMENT)

    session = "NULL"

    if eventBody['object']['status'] == 'succeeded':
      session = "Succeeded"
    elif eventBody['object']['status'] == 'failed':
      session = "Failed"

      response = table.update_item(
        Key={
          'id': event['object']['payment_intent']
        },
        UpdateExpression='SET PaymentStatus = :newStatus',
        ExpressionAttributeValues={
            ':newStatus': session
        },
        ReturnValues="UPDATED_NEW"
)
    # ... handle other event types
    else:
      print('Unhandled event type {}'.format(event['type']))
    response_value = {
        'statusCode': 200,
        }

    return response_value
```

Here is the Hero Section

```jsx
function Hero() {
   return (
   <div className="Hero">
      <Carousel className='carousel'>
      <Carousel.Item className='carousel-item'>
         <img className="hero-img" alt="" src='https://images.pexels.com/photos/207691/pexels-photo-207691.jpeg?cs=srgb&dl=pexels-pixabay-207691.jpg&fm=jpg' />
         <Carousel.Caption>
         <h3>Welcome to TheRealCapstone</h3>
         </Carousel.Caption>
      </Carousel.Item>
      <Carousel.Item className='carousel-item'>
         <img className="hero-img" alt="" src="https://images.pexels.com/photos/11032963/pexels-photo-11032963.jpeg?cs=srgb&dl=pexels-josh-hild-11032963.jpg&fm=jpg"/>
         <Carousel.Caption>
         <h3>Louisville Concerts</h3>
         </Carousel.Caption>
      </Carousel.Item>
      <Carousel.Item className='carousel-item'>
         <img className="hero-img" alt="" src="https://images.pexels.com/photos/3722923/pexels-photo-3722923.jpeg?cs=srgb&dl=pexels-chait-goli-3722923.jpg&fm=jpg"/>
         <Carousel.Caption>
         <h3>Nashville Concerts</h3>
         </Carousel.Caption>
      </Carousel.Item>
      </Carousel>
   </div>
   )
}

export default Hero;
```

CSS throughout the site

```
    overflow-x: hidden;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content:safe;
    color: white;
    background: linear-gradient( rgba(0, 0, 0, 0.5), rgba(0, 0, 0,

    /* background-image: url('../../Assets/Images/hero_main.jpeg'); */
    background-position: center;
    background-repeat: no-repeat;
    background-size: cover;
  }

  .Form-header {
    overflow-x: hidden;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content:safe;
    color: white;
    background: linear-gradient( rgba(0, 0, 0, 0.5), rgba(0, 0, 0,

    /* background-image: url('../../Assets/Images/stadium_hero.jpeg');
    background-position: center;
    background-repeat: no-repeat;
    background-size: cover;
  }

  .inputText {
    height: 42px!important;
    margin-top: 20px!important;
    margin-bottom: 50px!important;
    padding-top: 10px!important;
    padding-bottom: 10px!important;
    border-width: 1px 1px 2px!important;
    border-color: #222!important;
    border-radius: 2px!important;
    background-color: rgba(255, 253, 253, 0.22);
    color: #fff!important;
    font-weight: 500!important;
  }

RMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE
```

Code from the navbar

```
return (
  <Segment inverted>
    <Menu inverted pointing secondary>
      <NavLink to="/">
        <Menu.Item
          name="Home"
          active={activeItem === "Home"}
          onClick={() => setActiveItem("Home")}
        />
      </NavLink>

      <NavLink to="/events">
        <Menu.Item
          name="Events"
          active={activeItem === "Events"}
          onClick={() => setActiveItem("Events")}
        />
      </NavLink>
      <NavLink to="/services">
        <Menu.Item
          name="Fans"
          active={activeItem === "Fans"}
          onClick={() => setActiveItem("Fans")}
        />
      </NavLink>
      <NavLink to="/Services">
        <Menu.Item
          name="Organizers"
          active={activeItem === "Organizers"}
          onClick={() => setActiveItem("Organizers")}
        />
      </NavLink>
      <NavLink to="/createEvent" >
      <Menu.Item    name="createEvent"
          active={activeItem === "createEvent"}
          onClick={() => setActiveItem("createEvent")}>
            <Button positive inverted content="Create Event"   />
          </Menu.Item>
      </NavLink>
      <div className="d-flex" style={{backgroundColor:'#576278',marginLeft:'6
        <Button style={{backgroundColor:'#576278',display:'flex'}} onClick={(

          <Badge badgeContent={cart.cartEvents.length} color="secondary">
```

How does your project relate to trends and emerging areas of CECS?

The project related to the CECS trend of ecommerce. The real seat is a start up in this space that sponsored out capstone. The ecommerce trend is mostly about anyone becoming a seller of any good or item. Out capstone was the basis of an ecommerce site where you can buy and sell goods. Out team was close to adding functionality where anyone even non admins could add an event to sell tickets. If this feature was added our site would works as an ticketing exchange similar to eBay, where buyers and sellers can meet.

The second emerging area was cloud development and deployment. Our entire application was serverless and was deployed using AWS. This is a major industry shifter where companies' tech stack can be more dynamic, and they can easily create and pull analytics from their users. During my job search I see lots of Cloud developer roles because so many companies are still in the process of converting old tech into the cloud and working on designing newer implantations that have more features that utilize the serverless aspect.

CSE 596 CSE Capstone Design Team Report

TheRealCapstone: Event Ticketing Page

April 28, 2022

Team Members:

Abu Abukar, Caty Battjes, Steven Johnson, Elijah Moppins, Lam Nguyen, Jose Ramos

Customers:

TheRealSeat

Table Of Contents

Introduction

Online ticket service is not a new concept anymore, online ticketing has been on the online market for a long time now. For example Ticketmaster is a big company that handles online ticket service which was founded on October 2, 1976. TicketMaster sold 142 million+ tickets valued at $8 billion in 2007. This opened a lot of opportunities for ticketing online service, and this shows that a lot of people nowadays buy tickets through the Internet. TheRealCapstone is a project that is carried out by CSE 596's team. And our purpose is to provide a better experience for online ticket shopping. With our tool, clients can subscribe to our channel to receive the best tickets and deal with new events. Our client can decide whenever they want to buy tickets or sell their tickets to another customer. Our ticketing system is not limited to concerts only but other variety of events such as Sport, Musical, Art-Theater, or even Family events. With a lot of options for customers, we want to give our customers the best experience with online ticketing

System Description

Needs Assessment/System Requirements

The system requirements of a full stack event ticketing web application must satisfy a number of requirements and the specified audiences. Requirements of the web application must have a user interface that fetches persistent data that is safely stored, with the ability to add items to a shopping cart and make a payment. The audience that this developed for are users that want to search and purchase tickets to events. The size of the audience may vary largely so a scalable platform must be considered when developing the full stack event ticketing web application.

Initial System Specification

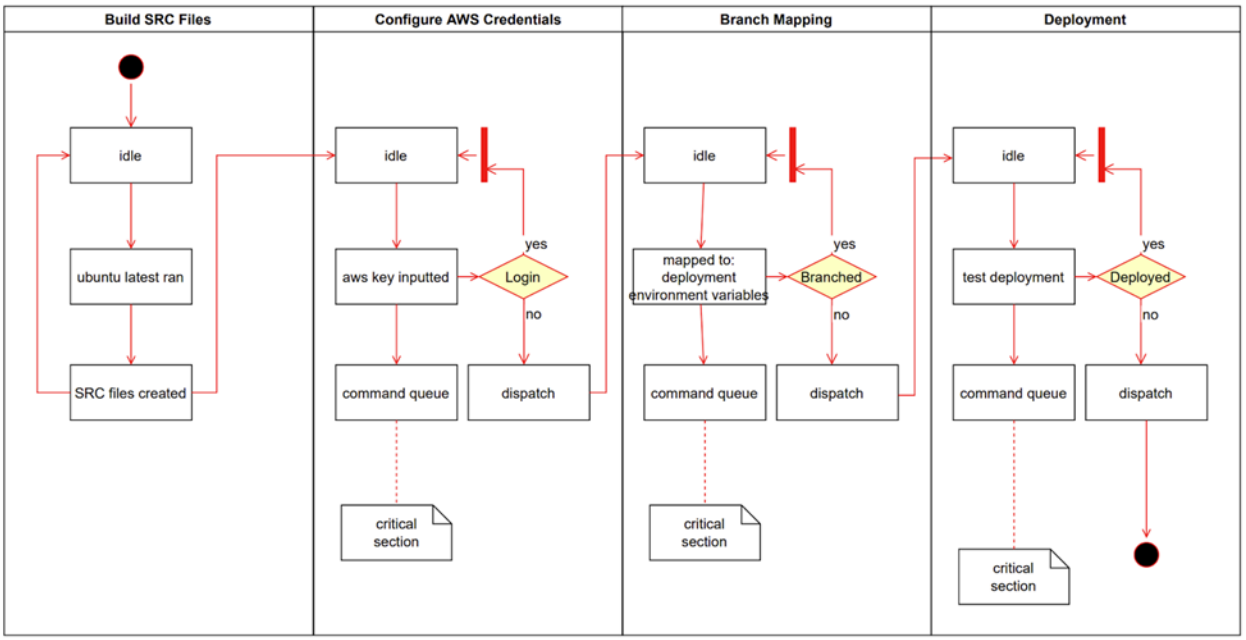
To develop a full stack web application for an Event Ticketing Platform an initial system specification was based on the needs of the project. The needs of the project consisted of viewing events that are stored in a database in a modern user interface design with payment integration for checkout. Therefore the web architecture consisted of React JS to develop the user interface and display events, and AWS Services such as AWS Lambda, API Gateway, and DynamoDB to fetch and store the Event Listing data that will be rendered to the user. Moreover, Stripe was used for payment integration as well as other necessary libraries such as Redux to develop the shopping cart and Axios to handle HTTP requests for the user.

Final System Specification

The final system specification can be observed. During development the initial system specifications were revised slightly. The specifications that were maintained were the use of React JS to develop the user interface, axios to make HTTP requests, Redux for storing global user data such as a shopping cart, and the previously recorded AWS services to serverless deploy the website with its respective backend. The additions that were made were the use of another AWS service known as CloudFront to distribute the static content and tie everything to a web domain with the necessary SSL certificate. Moreover, styling frameworks and packages were added into the system including SCSS and React-Bootstrap.

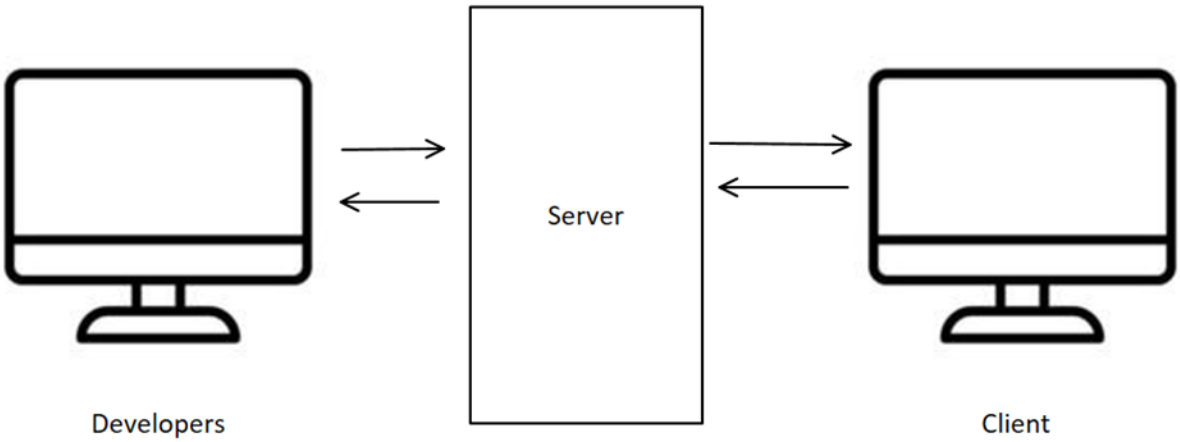
System Diagram

The system first builds the source files and then checks to see if there is an AWS key inputted. The files cannot be pushed to the S3 bucket without the key to know who is accessing AWS and where to send the files. Once logged in the code is dispatched and the code is branched to the developer's branch. After branching, the code is deployed and is reachable by the website.



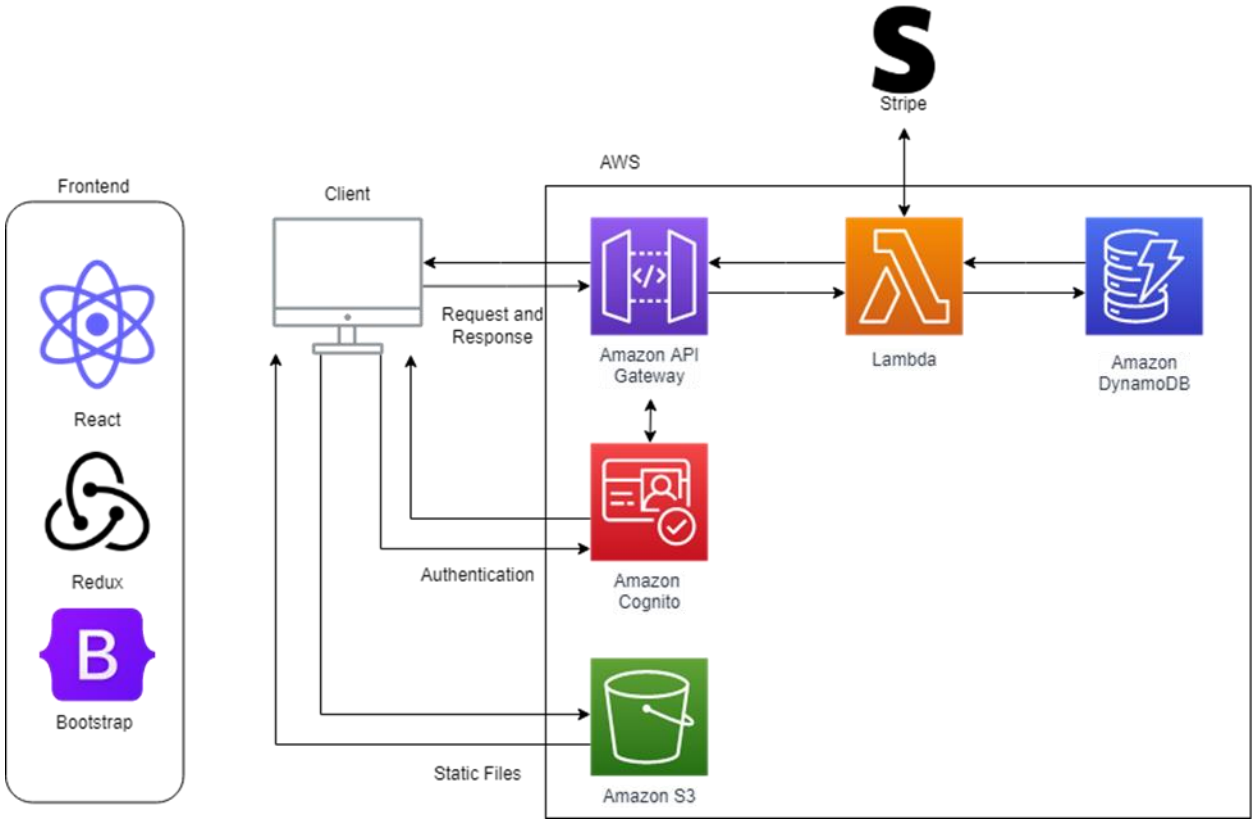
Hardware Overview Diagram

Since the website is deployed severlessly, the only hardware involved is the machines that the developers are using to create the website and the machines that users are using to access the website. All changes made by the developers run from their machine to the cloud to be deployed. Users can then see the deployed website by hitting the cloud-based website from their devices. Users will be able to clearly view the website on multiple devices, including laptops and phones.



Software Overview Diagram

TheRealCapstone software implementation contains software for the frontend and the backend. React, along with several extensions including Redux and Axios, and Bootstrap are the main frontend components for TheRealCapstone. These are used to create a good user experience by making the UI, or User Interface, legible and easy to use. AWS, or Amazon Web Services, is used to create the backend. S3 allows static files to be stored in buckets and is where the website is launched from. API Gateway works with Lambda and DynamoDB to pull and store user and event information. When a user adds an event, the information is stored to DynamoDB through a Lambda, Lambda also pulls from DynamoDB to display all events listed. When any transaction is made through Stripe, a Lambda pulls that information from Stripe and stores it to DynamoDB. API Gateway also works with Cognito to provide authentication and security.



Economical, technical and time constraints

Evaluation of a project's constraints are always important to consider and estimate for efficient project control. The team gathered and discussed among themselves and also with the project representative to discuss limitations. It was decided that by estimating where and when difficult encounters could happen, the team would be prepared beforehand. Among the members, there was no one assigned with the role of a project manager, however, the representative did organize the majority of the project's framework. These three constraints, economical, technical and time, are the key indicating factors that defined how to approach the project’s framework. Throughout the project's lifespan, adjustments and modifications to the project were made to fit within these limitations.

When managing the project, the manner of evaluation for each constraint was based on quality and risk. Quality is mutual among what is to be delivered for the representative and what the team will learn. A good, functional and expected product is what the company deemed as good quality. For the team, good quality is the task that achieves professionalism as a community and also demands high standards for the individual. For risk, this applies to how impacting a task would be in case it falls out of the bounds of said constraints. Tolerance dictates how the project needs to be addressed and managed. Quality and risk are both evaluated based on probability and expectancy. Both these characteristics share the manner in which the team can manage and deal with the outcome.

Economical constraints for this project were minimal. The nature of the project is set up to limit the spending on development and runtime to a bare minimum, almost non-existent. Even though the constraint was of little significance, it was still a constraint and it was taken into account. Through the setup of AWS, the project operated in a serverless manner. By operating serverlessly, there is no need for credits or fees to be applied. Lambda functions are operations that are called when specific actions occur. The runtime of lambda functions vary heavily depending on what is needed to be done, but for the team's project purposes, these functions are in the millisecond range. By having the runtime of lambda functions be so small, they have no real effect on the economic constraints. As a team, the economic constraint was heavily considered and it remained well managed throughout the project's life span.

Technical constraints were difficult to establish a plan for at first. Team members all have different experiences and manners of approach, the first approach was to have members work as a team. It quickly became evident that there was a lack of experience when it came to web development and e-commerce in specific. The project sponsor came to this realization and

changed the outline of the project into a more hands on experience, members now approached it independently. An independent method worked well for the first sprints where everything was introductory and rehearsal of known knowledge, but soon the latter sprints required more time and effort. Teammates discussed and an update on the approach was made, work was to be separated as groups. Group working meant that things would get done quicker, while also maintaining that quality assurance of the tasks involved within each sprint. The team sponsor would post the tasks required for each sprint at the commencement of each sprint respectively, display the level of difficulty, members recommended, brief description and resources of how to complete each task, members would now assign themselves to each task. Once the members were assigned, they began working on their tasks while assuring risk and quality given the time limit of the sprint. Sprints consisted of average 10 tasks and lasted two weeks. Upon completion of the sprint, team members gathered with the project sponsor and gave updates of the project, and what is to be expected for the next sprint. This method assured the project would be completed within the given time frame.

Time constraints were the main issue at the first quarter of the project's lifespan. The project sponsor was misinformed about the time constraints team members could each dedicate towards the project. Initially, he had been promised that members could provide more hours weekly than what he was now being informed about. Since the team members consist of students who are concurrently enrolled in other classes, there could not be as much time as the sponsor had originally planned for. The tasks of the first weeks were more demanding and required each individual to dedicate hours beyond what they were capable of. This led to the project sponsor having a meeting with the team and planning out what would be the best decision moving forward that would benefit all parties involved. The team as students still required a project to

learn and gain experience from, and the sponsor needed a quality project that might not have been to the degree initially hoped for, but was still close enough. A conclusion was reached that the e-commerce website was to be created, but it would refrain from the algorithmic approach that was initially described. Having made these changes, things moved more smoothly, members could dedicate the time required from them and the tasks were being completed with high quality. As mentioned in the economic constraints section, the new approach involved an agile process development. An agile approach assured quality remained high while risk remained low.

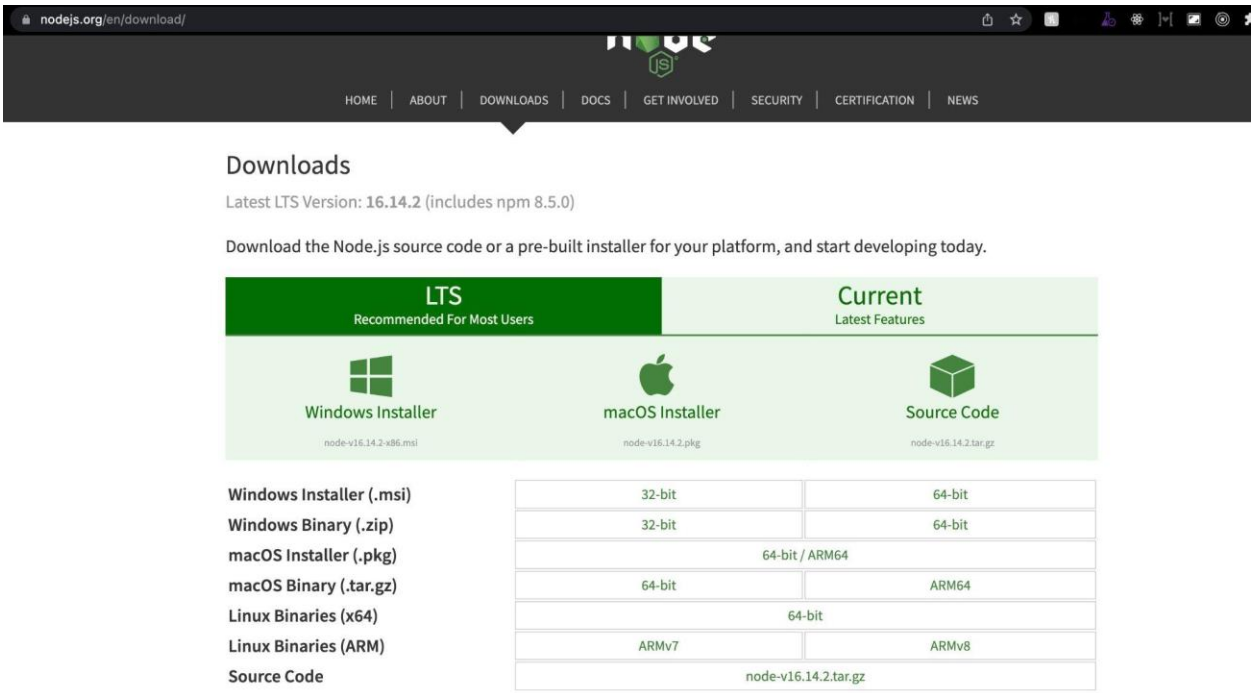
Detailed Implementation

React

There are many front-end frameworks but our team decided to go with ReactJS and it benefits in many ways. React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes and since it is component based and also called SPA(single-web-page). This makes your code easier to view and also easier to debug.

There are two ways to implement the component which are class based and function based. Follow our design pattern, functional based component is the design pattern, our team follow and it simplifies the reactjs lifecycle enough for our developers to handle.

Creating a ReactJS app requires Nodejs version that >= 14.0.0 and npm version >=5.6. We first download the Nodejs and install it on our local machine.

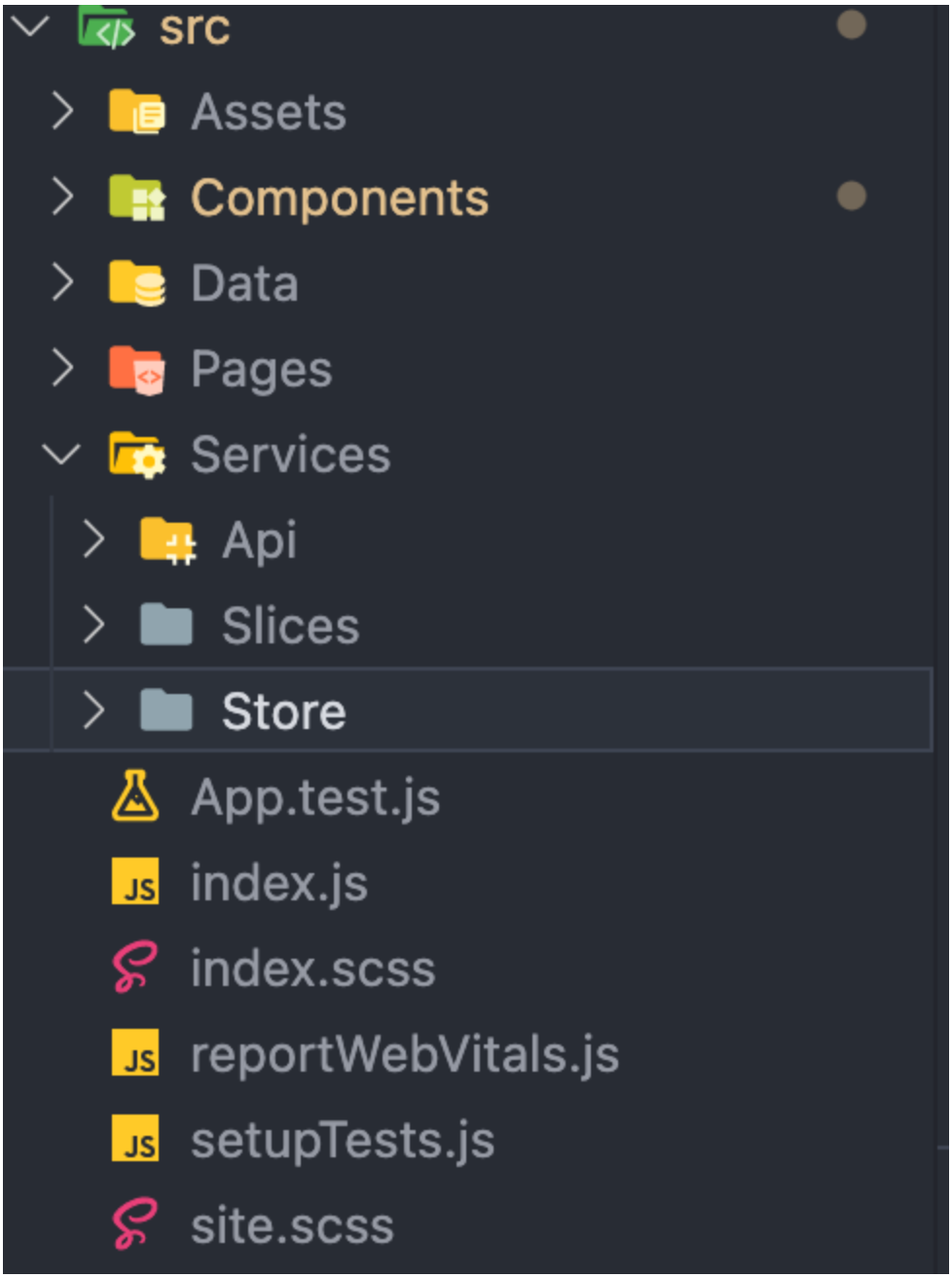


After installing nodejs, we are ready to create a new react application. We need to open the command prompt or terminal to execute the create react command.

```
npx create-react-app my-app
cd my-app
npm start
```

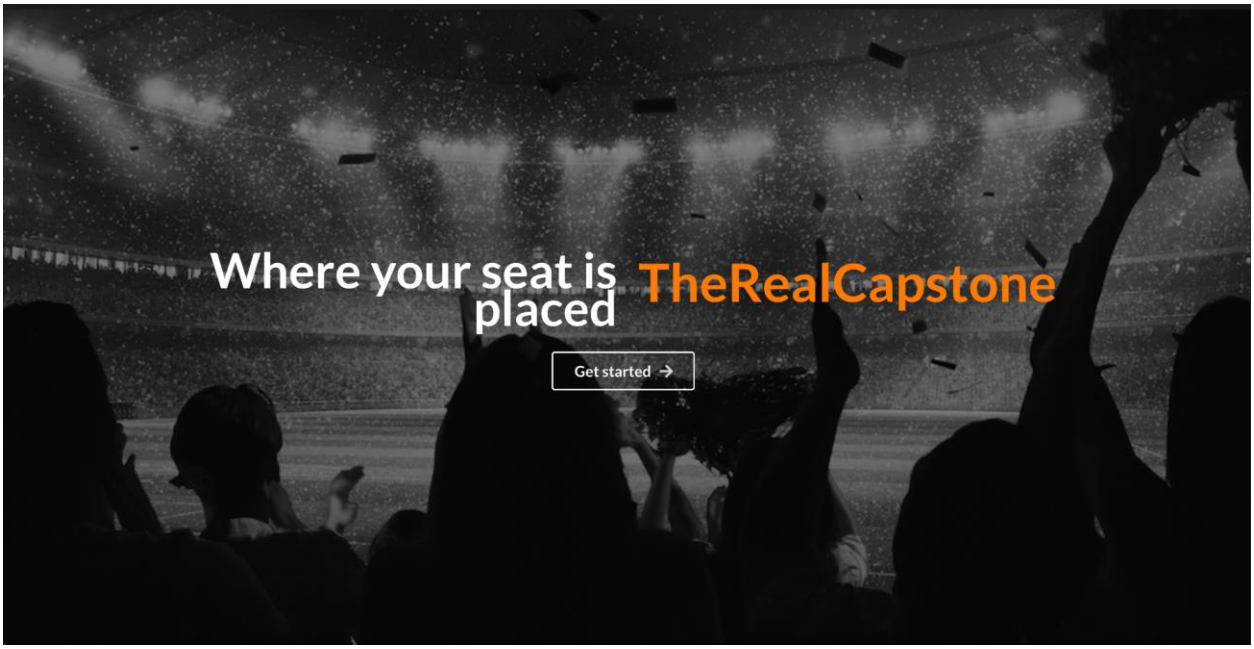
The React App doesn’t handle backend logic or databases; it just creates a frontend build pipeline. After creating the react app, folder organizing is also the key for our developers to speed up the process of debugging and organizing the code. We create a Components folder, Data folder, Pages folder,Assets folder and Service folder.

Since our react app is a SPA(single-page-application) and it uses components. All of our components go inside the Components folder, Pages folder contains the main routes components. Assets folder contains image files. Service folder contains our api call service and redux store which we implement later on.

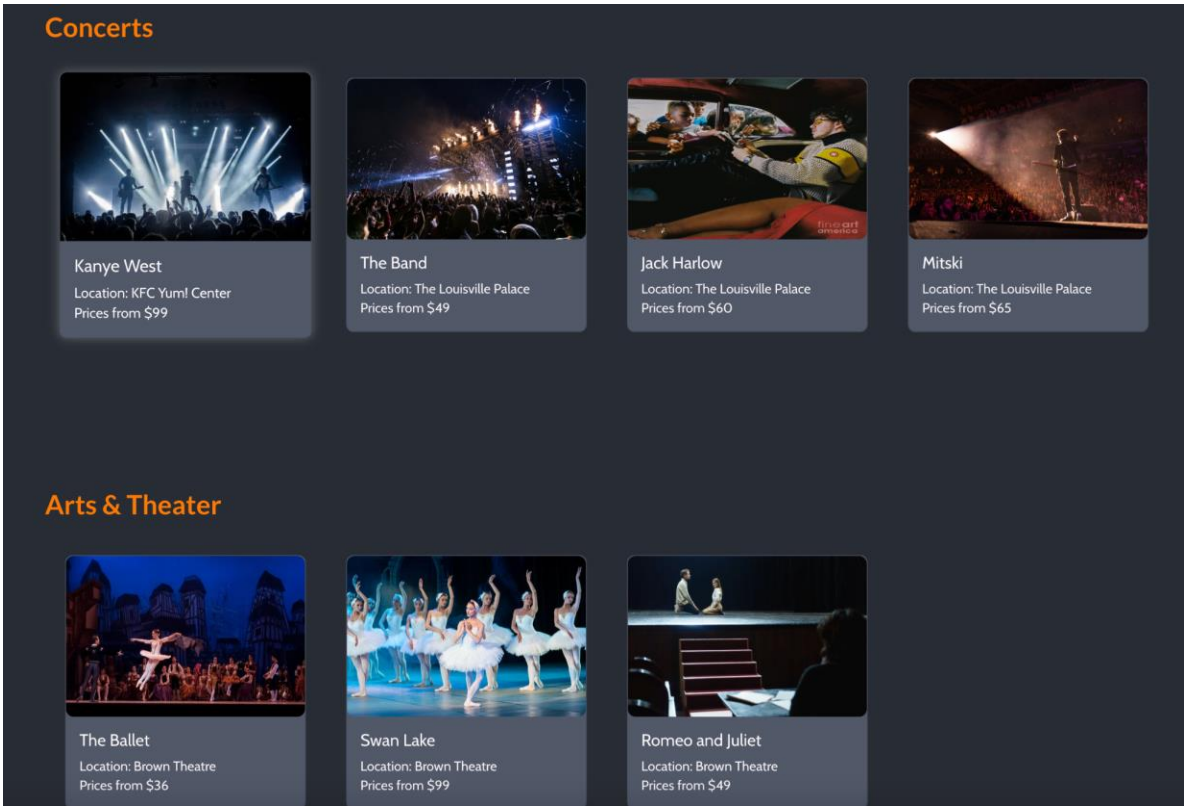


In our react app, we have 2 main pages that we need to show our clients which are the main page and the events page. Main page is mainly about the overall view of this project and

styling and user selections for the events page.



In our event page, the main purpose is to show clients the events and make it available for clients to purchase any provided events.



These are two main routes we need and these are also implemented in function based component.

React-Router

In react to manage routing and expose front-end end points to users we need to install and use a package name react-router. Command prompt or terminal is required to be opened in order to install the package.

```
npm install react-router-dom@6
```

After installing the package into the project dependencies, We keep the architecture of exposing endpoints to users in the app.js. We need to wrap all of our endpoint within the BrowserRouter tag

```
<BrowserRouter>
  <App />
</BrowserRouter>,
```

The BrowserRouter tag allows us to use React Router props. And within the BrowserRouter tag we need to use extra extension to fully use the router for react

```
<BrowserRouter>
  <Navbar />
  <Routes>
    <Route path="/" element={<HomePage />} />
    <Route path="/disclaimer" element={<Disclaimer />} />
    <Route path="/events" element={<Events />} />
    <Route path="/createevent" element={<EventForm />} />
    <Route path="*" element={<NotFound />} />
  </Routes>
  <Footer />
</BrowserRouter>
```

The react-router already handles the base URI for our app. To declare a new route we need to put the Route tag inside the Routes and also the Route tag needs a path which contains the whole url and element which is rendered when the route is hit. The rendered components in the Routes tag are kept in the Pages folder. This keeps our structure easy to find and easy to debug. We create a page for NotFound error. The path with "*" is reached after the end point input from users doesn't match with any of our ends. This helps users navigate back after going to wrong routes. Create event route is used for admin users to create new events and event route is where users can see and purchase events that they are interested in.

Redux and Redux Toolkit

React is component based web application and it allows developers to pass parameters from a component to the others. The process of passing parameters can be very tedious and sometimes can cause unpredictable behavior. In the real world this is not ideal to pass parameters through many(more than 2). This is when Redux comes in handy. For a short description of Redux "Redux is a predictable state container for JavaScript apps". Redux can be used with any

javascript application, but since we develop React app, we need to use React Redux. To install React Redux, we need to open a terminal on Mac and on Windows command prompt is needed to install the package.

```
# If you use npm:
npm install react-redux

# Or if you use Yarn:
yarn add react-redux
```

Copy

Using Redux we need to familiar ourselves with the concept of redux store , reducer and selector. There are many descriptions for each term but according to Redux official website “A store holds the whole state tree of your application. The only way to change the state inside it is to dispatch an action on it. A store is not a class. It's just an object with a few methods on it.“. Redux reducer is the action where we can modify the current state of the store. And the Redux Selector and the method we use to get the state(values) of the current store that we want to retrieve the values.

To start using redux in our react app we need to configure the store. The first store that we want is the shopping cart store which has a list of events, total quantity of events and total amount of currency. Also the shopping cart store needs two actions(reducers) that allows users to add and remove the events from their cart. One tricky thing about redux is when we modify the cart, we are not allowed to modify the state which means we need to create a copy of the previous value and change the copied values then replace the store with copied state. This can take a lot of time to develop the application. So this is when Redux-Toolkit comes into place.

To install Redux-Toolkit we need to use the npm install which is required to open the terminal.

```
# NPM
npm install @reduxjs/toolkit
```

With redux toolkit, it comes in a few convenience function that allow us to create stores and reducers faster.

Redux Toolkit includes these APIs:

- `configureStore()`: wraps `createStore` to provide simplified configuration options and good defaults. It can automatically combine your slice reducers, adds whatever Redux middleware you supply, includes `redux-thunk` by default, and enables use of the Redux DevTools Extension.
- `createReducer()`: that lets you supply a lookup table of action types to case reducer functions, rather than writing switch statements. In addition, it automatically uses the `immer` library to let you write simpler immutable updates with normal mutative code, like `state.todos[3].completed = true`.
- `createAction()`: generates an action creator function for the given action type string. The function itself has `toString()` defined, so that it can be used in place of the type constant.
- `createSlice()`: accepts an object of reducer functions, a slice name, and an initial state value, and automatically generates a slice reducer with corresponding action creators and action types.
- `createAsyncThunk`: accepts an action type string and a function that returns a promise, and generates a thunk that dispatches `pending/fulfilled/rejected` action types based on that promise
- `createEntityAdapter`: generates a set of reusable reducers and selectors to manage normalized data in the store
- The `createSelector` utility from the Reselect library, re-exported for ease of use.

To create cart store now can be very easy and short. We need to give it initial value

```
const initialState = {
    cartEvents: [],
    cartTotalQuantity: 0,
    cartTotalAmount: 0
}
```

Then create a slice which will create a store for the cart that has a name, initial value and reducers

```
const cart = createSlice({
    name: 'cart',
    initialState ,
    reducers: {
        addToCart(state, action) {
            console.log(action.payload);
            state.cartEvents.push(action.payload);
            state.cartTotalQuantity += 1;
            state.cartTotalAmount += parseFloat(action.payload.event_pricesFrom);
        },
        removeFromCart(state, action) {
            state.cartTotalAmount -= parseFloat(state.cartEvents[action.payload].event_pricesFrom);
            state.cartEvents.splice(action.payload, 1)
            state.cartTotalQuantity -= 1;
        }
    }
})

export const { addToCart, removeFromCart } = cart.actions
```

With this toolkit reducers helper already helps us with state mutation so there won't be any unpredictable state. And to consume the cart store we need to configure store and the best place to do is our app.js.

```
export default function configureStore(){
    const store= createStore(rootReducer,composeWithDevTools(applyMiddleware(thunk)))


    return store;
}
```

This function will help us create a store with thunk middleware which allows us to write asynchronous code in react-redux. And since we have multiple stores, it is best to have a root store that is a container for all stores.

The React-Redux requires us to wrap all the components that use redux to call the store in the Provider tag and in the Provider tag we need to give it a store parameter.

```
const store = configureStore();

return (
  <Provider store={store}>
    <BrowserRouter>
      <Navbar />
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/disclaimer" element={<Disclaimer />} />
        <Route path="/events" element={<Events />} />
        <Route path="/createevent" element={<EventForm />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
      <Footer />
    </BrowserRouter>
  </Provider>
```

Now we are able to consume the stores from redux. To get the current state, we need to import useSelector from Redux. And with that we are able to get the state of cart store.

```
const cart = useSelector((state) => state.cart);
```

Using reducers of the store, we need to export our reducers from store slice. Redux also requires us to put reducers inside of useDispatch.

```
export const { addToCart, removeFromCart } = cart.actions
```

```
const dispatch = useDispatch();
```

```
dispatch(addToCart(product));
```

If dispatch is called in any component and state is changed, all components that use selector that point to the modified store will be rerendered with the new state.

Git/GitHub

During the project's development phase, the team required a manner to host and store the code in order for simultaneous team work. The ability to utilize Git to push and pull code through the platform GitHub is incredibly useful and efficient. Through GitHub, the team could store, track and update their work done on the project, thus allowing remote collaboration among team members.

GitHub consists of many development tools that allow developers to create repositories. These repositories are made up of branches which themselves hold code. The purpose of branches is to permit different developers to push their changes to the repository. Each developer on the team has their own unique branch. Once a developer has completed their task for a sprint, they then add and commit the changed files and push to their remote branch on the repository. GitHub now takes notice of the changes done on their branch and runs a production build test. If the production build was successful, then a pull request is recommended to proceed. A pull request is a request to merge the contents of the developer's newly modified branch into another branch. This other branch is usually a main branch or as the team conducted, a pre-main branch. Pre-main branches, stg in the case of the team, serve the purpose of preventing breaking errors in the software upon merging. This is the method utilized by the team in order to stay up to date on the current code for the project.

Git is an open source software that is utilized to maintain track of files in a directory. Through Git, developers are able to have control over project systems. Git is used by the team to maintain version control at max efficiency. Through Git, individual developers can host local repositories and as a team, developers can host a remote repository. The differences among a local and remote repository are only there to maintain security and functionality. A local repository is only available within the developers computer system, yet the repository is only

created within the desired directory. A remote repository is a repository that is hosted online, the team utilized GitHub as stated.

Git commands are the syntax commands of Git that the user inputs to execute desired tasks. The main commands used are; git clone, git branch, git checkout, git status, git add, git commit, git push, git pull, git revert and git merge. Most of the functionality provided by Git is offered through these ten commands. Git clone is a command that is used to execute the task of downloading code that is already hosted within a remote repository. This task in essence copies the existing code within the repo along with its branches. The team utilized this task in order to initially make a copy of the project's software. Git branch is conducted to execute the creation of a new branch for the repository. As can be inferred, this is a vital task for the team since it will create the individual developers branch. Git checkout, a command that is used to switch to another branch of the repository. Team developers often have to switch into other branches to pull code from the newly switched branch in order to continue development. Git checkout is for the most part the task that proceeds git branch. Once a developer has switched into a branch, they then need to checkout the branch's files and commits. Git status command is one that allows the developers to view all of the data for a particular branch. Oftentimes a team developer will utilize this command to view whether they are on the correct branch and to tell if they are on the most recent version of the branch. Git add, this is one of the most essential commands for any project development, it offers the ability for Git to add the desired file changes for the next commit. Adding is a crucial aspect for developers to constantly update their work. Git commit is a command that allows the developer to commit changes added to the repository. Similar to git add, git commit is of high importance and usage by developers. The commit command has a variation that allows developers to write a message in order to maintain record of the changes

made for that particular commit. Git push is usually the follow up to a commit, it pushes the changes made by the commit to a remote server. In the case of the team, push was used to send code to the GitHub repository. Git pull, similar properties to the push command, however, a pull is used to get the updates from a remote repository. A developer utilizes git pull to combine the functionalities of git fetch and git merge. Git revert is not a command that is meant to be used frequently since it is executed to undo previous changes. To prevent or undo unwanted errors, a developer uses git revert to undo changes from the local or remote repository. Finally there is git merge. Once development is completed on a developers branch, they now need to merge their branch to another. As stated previously, the team set up a pre-main branch, stg, this was the branch that was used to merge individual branches into.

To simplify the process and increase efficiency, the team utilized GitHub Actions to conduct automation of the software workflow. Continuous integration (CI) and continuous delivery (CD) features are easily set up within GitHub Actions. Building, testing and deploying are the actions that were set up to be tested for every commit. The manner in which these actions are tested, is called a workflow. In essence a workflow is used to configure automated processes that will run the set up jobs. Jobs are sets of steps that perform in a set up order, their purpose is to execute and test the code. GitHub offers what is known as a runner for Actions. A runner is a server that runs the workflow when triggered to execute.

The team set up a workflow file that defines the sequence of jobs and their order of execution upon every commit to the repository on all branches.

```yaml
name: Production Build
on: push

env:
  STRIPE_PUBLISHABLE_KEY: ${{ secrets.TEST_STRIPE_PUBLISHABLE_KEY }}
  STRIPE_SECRET_KEY: ${{ secrets.TEST_STRIPE_SECRET_KEY }}

jobs:
  src:
    name: build src files
    if: always()
    runs-on: ubuntu-latest
    timeout-minutes: 20
    steps:
    - name: Check out repository
      uses: actions/checkout@v2

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: us-east-1

    - name: Map Branch to DEPLOYMENT
      id: check_branch
      run: |
        echo "DEPLOYMENT=$(if [[ ${GITHUB_REF##*/} = main ]];
            then echo "PRD";
        elif [[ ${GITHUB_REF##*/} = stg ]];
            then echo "STG";
        elif [[ ${GITHUB_REF##*/} = caty* ]];
            then echo "CATY";
        elif [[ ${GITHUB_REF##*/} = abu* ]];
            then echo "ABU";
        elif [[ ${GITHUB_REF##*/} = jose* ]];
            then echo "JOSE";
        elif [[ ${GITHUB_REF##*/} = steven* ]];
            then echo "STEVEN";
        elif [[ ${GITHUB_REF##*/} = lam* ]];
            then echo "LAM";
        elif [[ ${GITHUB_REF##*/} = elijah* ]];
            then echo "ELIJAH";
        elif [[ ${GITHUB_REF##*/} = * ]];
            then echo "DEV"; fi)" >> $GITHUB_ENV

    - name: Map Branch to Domain ENV VAR
      run: |
        echo "DOMAIN=$(if [[ $DEPLOYMENT = PRD ]];
          then echo "www.therealcapstone.com";
        else
          echo "${DEPLOYMENT,,}.therealcapstone.com"; fi)" >> $GITHUB_ENV
    - name: Yarn Install
      run: |
        yarn install
    - name: Production Build
      run: yarn build

    - name: Deploy to s3
      run: |
        aws s3 sync ./build s3://$DOMAIN --acl public-read --delete
```

The team has designed the workflow file, ci.yml, to execute upon a push of code, then it sets the environment for Stripe.

```yaml
name: Production Build
on: push

env:
  STRIPE_PUBLISHABLE_KEY: ${{ secrets.TEST_STRIPE_PUBLISHABLE_KEY }}
  STRIPE_SECRET_KEY: ${{ secrets.TEST_STRIPE_SECRET_KEY }}

```

From there, the workflow sequence leads on to the execution of jobs. The team made sure the order of execution of the jobs was logical. The first job is named, “build src files’, as can be inferred, this job creates the source files always, and the time for execution is limited to only 20 minutes.

```yaml
jobs:
  src:
    name: build src files
    if: always()
    runs-on: ubuntu-latest
    timeout-minutes: 20
    steps:
    - name: Check out repository
      uses: actions/checkout@v2
```

Upon completion of the first job, the workflow continues onto the next. Configuring the AWS credentials, this job was set up to maintain track and automate the updating of the AWS credentials. The job is named “Configure AWS credentials”, it will revise the credentials according to the access key ID, the secret access key and the region.

```
18      - name: Configure AWS credentials
19        uses: aws-actions/configure-aws-credentials@v1
20        with:
21          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
22          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
23          aws-region: us-east-1
24
```

We move on to the third job in the workflow. This job is done to map each developer's branch to their appropriate bucket based on the revision and checking of the keyword ID. The manner in which it can go through and appropriately assign the developers bucket, is through the implementation of a loop. This loop is initialized by echoing the deployment, then goes through all of the keywords, if a keyword matches according to a developers branch name it will execute appropriately, if none match then it will return as failed. The keywords for the team were set up as the members first names.

```
25      - name: Map Branch to DEPLOYMENT
26        id: check_branch
27        run: |
28          echo "DEPLOYMENT=$(if [[ ${GITHUB_REF##*/} = main ]];
29              then echo "PRD";
30          elif [[ ${GITHUB_REF##*/} = stg ]];
31              then echo "STG";
32          elif [[ ${GITHUB_REF##*/} = caty* ]];
33              then echo "CATY";
34          elif [[ ${GITHUB_REF##*/} = abu* ]];
35              then echo "ABU";
36          elif [[ ${GITHUB_REF##*/} = jose* ]];
37              then echo "JOSE";
38          elif [[ ${GITHUB_REF##*/} = steven* ]];
39              then echo "STEVEN";
40          elif [[ ${GITHUB_REF##*/} = lam* ]];
41              then echo "LAM";
42          elif [[ ${GITHUB_REF##*/} = elijah* ]];
43              then echo "ELIJAH";
44          elif [[ ${GITHUB_REF##*/} = * ]];
45              then echo "DEV"; fi)" >> $GITHUB_ENV
46
```

The following jobs are basic in comparison since they only require minimal steps to execute, so they will be discussed together. “Map Branch to Domain ENV VAR” is the next job in the sequence. The priority of this job is to branch the domain to whichever branch was found from the previous job. Installation of the yarn and building of the yarn are done through the following jobs, “Yarn Install” and “Production Build” in that order respectively. Finally, the last job “Deploy to S3” executes to the S3 bucket. A bucket is used to maintain and see changes in real time, while remaining serverless. S3 buckets offer other opportunities thanks to the compatibility of AWS services.

```
47      - name: Map Branch to Domain ENV VAR
48        run: |
49          echo "DOMAIN=$(if [[ $DEPLOYMENT = PRD ]];
50            then echo "www.therealcapstone.com";
51          else
52            echo "${DEPLOYMENT,,}.therealcapstone.com"; fi)" >> $GITHUB_ENV
53      - name: Yarn Install
54        run: |
55          yarn install
56      - name: Production Build
57        run: yarn build
58
59      - name: Deploy to s3
60        run: |
61          aws s3 sync ./build s3://$DOMAIN --acl public-read --delete
```

Applications/Packages

Throughout the project development phase, the team utilized software applications.. An application is a collection of modules that work with one another to achieve their intended functionality design. Particularly for the team, the applications utilized consisted mainly of a direct relationship with what was to be achieved. This then excludes unrelated applications and packages that might have been used but are of no direct relation.

Among the packages used, the most commonly used was npm. Packages are an essential aspect of JavaScript. The functionality of npm, is to manage all of the various packages that the team is working with. It has to be stated that npm is specific to the JavaScript runtime environment Node.js, which is the environment used by the team. Among development, there are various modules that need to be placed and seeked in order to operate accordingly, the npm application does this through command line prompting. Developers enter tasks commands within the project's runtime terminal. The most essential commands are `npm i` and `npm start`. The first command, `npm i`, initiates the application. The latter command, `npm start`, starts the application; all of the modules are now seeked and ran accordingly.

The team utilized and relied heavily on AWS tools for project creation. Boto3 is the AWS SDK for Python. Boto3 is a software development kit that provides the team with the required features to build according to desired functionalities. Among these key features there is, resource API's, up-to-date and consistent interface, Python support, waiters and service-specific high-level features.

Real time transaction and processing, is among the main desired features when dealing with an online merchant website. The team decided the most adequate method to implement this feature is through Stripe API. Stripe is a well established, payment processing software that developers implement through API calls. Stripe communicates as a middleman between merchant websites and customers, when transactions occur. Stripe is an audited and certified PCI Level 1 service provider. This suffices to establish Stripe as a safe to use application. The team implemented and revised Stripe by creating a test account. The test account serves to process transactions and verify proper functionalities of site features.

These are among the applications and packages used by the team throughout the project development. They serve high importance to simplify and help accomplish the desired features for every sprint as assigned by the project sponsor.

Axios

Axios is a very popular npm package for http requests. According to official website "Axios is a *promise-based* HTTP Client for node.js and the browser. It is *isomorphic* (= it can run in the browser and nodejs with the same codebase). On the server-side it uses the native node.js http module, while on the client (browser) it uses XMLHttpRequests." This allows us to make http calls to our back-end to get data or the pose the data.

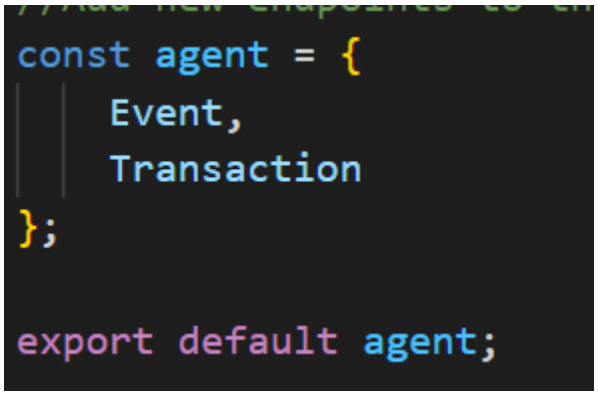
To install axios, we need to open the terminal and navigate to react app folder and run "npm install axios". To keep the design clean and easy to debug, we create a file that contains all the logic for axios requests. Since we call our end-point that is a REST API which has basic 4 api methods (GET, POST,PUT,DELETE), we need to create these methods with axios functions. From that concept, we want to create an object that contains 4 properties(get,post,put,delete). First for the get property, it need to take one parameter which is the url string and execute axios.get(). Since this is an async function, we need to get the value with function then(). We have our first property ready. Next we need to create another property for post. The post request takes 2 arguments, url and body and it triggers axios.post() then catches the response after making the post request. Method put has the same logic as post but instead of axios post we want to replace it with axios.put. The delete takes 1 argument for the url and calls axios.delete and passes the url to the param.

```
const requests = {
  get: (url) => axios.get(url).then(responseBody),
  post: (url, body) => axios.post(url, body).then(responseBody),
  put: (url, body) => axios.put(url, body).then(responseBody),
  delete: (url) => axios.delete(url).then(responseBody),
};
```

To make our api call more centralized we want to create an object that has the actual end-points url to be ready to be consumed by other components. Since we have events data we want to fetch from our back-end, we can create an Event object that contains all api call logic. First we want to get all the events, we can create a property named list, and all it does is get the list of events so we use requests.get and put the url inside the parameter. For event details, we create details property and it takes id as parameter and executes requests.get with an url that is injected with id. We want to give our user permission to create events as well. Create property is created for that with an event as a parameter then execute requests.post with url and values passed through the arguments. One extension we want to do is give users permission to query the event by event type, so we create listType that takes type as parameter and make request.get with an url that is injected with event type.

```
const Event = {
  list: () => requests.get("events"),
  details: (id) => requests.get(`event/${id}`),
  create:(values) =>requests.post("create_event",values),
  listType: (type) => requests.get(`events?type=${type}`),
};
```

These event end-points are ready to be consumed by other components but we have to export the object. To make things easier we can bind all end-point objects to a root object and export the root object.



React Extensions

In react dealing with form is a lot of code. To make experience with form in React better, There is a third react library called Formik. To give a better overview of Formik, according to the website

> Let's face it, forms are really verbose in React. To make matters worse, most form helpers do wayyyy too much magic and often have a significant performance cost associated with them. Formik is a small library that helps you with the 3 most annoying parts:
>
> 1. Getting values in and out of form state
> 2. Validation and error messages
> 3. Handling form submission
>
> By colocating all of the above in one place, Formik will keep things organized--making testing, refactoring, and reasoning about your forms a breeze.

To install the package we need to open a terminal or a command prompt on Windows and run "npm install formik --save". Formik already handles validation and 2 way binding variables with form submission. This can cut down the time to develop dramatically. We need to use Formik inside our Create Event page. Initially, we need to create initial values for all for form's fields. Formik already handles the 2 way binding for us so all we have to do is give the input a name that matches with the variable that we prefer to.

```
const initialValue =  {
  EventID:uuid(),
  EventTitle: "",
  EventPrice: 0,
  EventLocation:"",
  EventType: "",
  img: "",
};
```

```
<MyTextInput name="EventTitle" placeholder="Event Title" />
<MyTextInput name="EventLocation" placeholder="Event Location" />
<MyTextInput name="EventType" placeholder="Event Type" />
<MyTextInput name="img" placeholder="Event Image URL" />
<MyTextInput name="EventPrice" type="number" placeholder="Event Price" min="0" />
```

Formik gives us the tool to perform form validation. To use the schema validation, we have to use an extension named Yup.Yup is a JavaScript schema builder for value parsing and validation. Define a schema, transform a value to match, validate the shape of an existing value, or both. Yup schemas are extremely expressive and allow modeling complex, interdependent validations, or value transformations.

To install Yup we need to use the npm install package. Creating the validation schema is quite simple with Yup. With each property in the value object, we have to specify the data type for that variable and extend the object with validation rule(required for example), and to make it more customized we can give it a validation message for each property.

```
const validationSchema = Yup.object({
  EventTitle: Yup.string().required("You must provide a title"),
  EventLocation: Yup.string().required("You must provide a Location"),
  EventType: Yup.string().required("You must provide a EventType"),
  img: Yup.string().required("You must provide a image URL"),
});
```

Put Yup and Formik together we have a fully functional form that have 2 way binding and form validation

```
<Formik
  initialValues={initialValue}
  validationSchema={validationSchema}
  onSubmit={ async(values) =>await  handleSubmit(values)}
>
  {({ values,isValid,dirty}) => (

    <Form className="ui form">
      <Header content="Event Details" color="teal" />

      <MyTextInput name="EventTitle" placeholder="Event Title" />
      <MyTextInput name="EventLocation" placeholder="Event Location" />
      <MyTextInput name="EventType" placeholder="Event Type" />
      <MyTextInput name="img" placeholder="Event Image URL" />
      <MyTextInput name="EventPrice" type="number" placeholder="Event Price" min="0" />
```

Developing UI components can take a lot of developer's time. We use the third party UI components to shorten the development time. First package we use is Bootstrap-react and to install the bootstrap-react package we need to use npm install.

```
npm install react-bootstrap bootstrap
```

For the UI component we will have to include the script and also the css tag in the index.html so that the component can be rendered as expected.

```
<script src="https://unpkg.com/react/umd/react.production.min.js" crossorigin></script>

<script
  src="https://unpkg.com/react-dom/umd/react-dom.production.min.js"
  crossorigin></script>

<script
  src="https://unpkg.com/react-bootstrap@next/dist/react-bootstrap.min.js"
  crossorigin></script>
```

```
<link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
  integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
  crossorigin="anonymous"
/>
```

With these lines added to index.html, we can use bootstrap components now for our react-app

Another library that we use is Semantic-UI, to install the Semantic-UI:

```
npm install semantic-ui-react semantic-ui-css
```

To use semantic-ui we also need to import some scripts and css tag in the index.html

```
import 'semantic-ui-css/semantic.min.css'
```

index.html

```
<body>
  <!-- ... other HTML ... -->
  <!-- ... Load React ... -->

  <link
    async
    rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/semantic-ui@2/dist/semantic.min.css"
  />
  <script src="https://cdn.jsdelivr.net/npm/semantic-ui-react/dist/umd/semantic-ui-react.min.js"></scri

  <!-- Load our React component. -->
  <script src="like_button.js"></script>
</body>
```

These third party UI components have many useful components that apply in almost every website UI today such as Grid,Button,Form,Dialog, Breadcrum, Pagination,... Instead of

writing plain html and css, now we only need to handle the business logic for our component to render exact information that we provide for our clients.

Styling Languages

In this project the styling language we used was CSS (Cascading style sheets). CSS is the most popular language used for styling Web Pages. In css you can customize the size and position of individual elements, and apply specific style patterns throughout your application simply. In our project we utilized CSS to make the website mobile friendly, desktop friendly and with more work you could add accessibility features directly with css. Especially for visually impaired users.

Programming Languages

The team used javascript, python, and css within our project. The Java script is used to create react elements and functions to be used throughout the application. The CSS is used to style the elements and create device compatibility throughout the site.

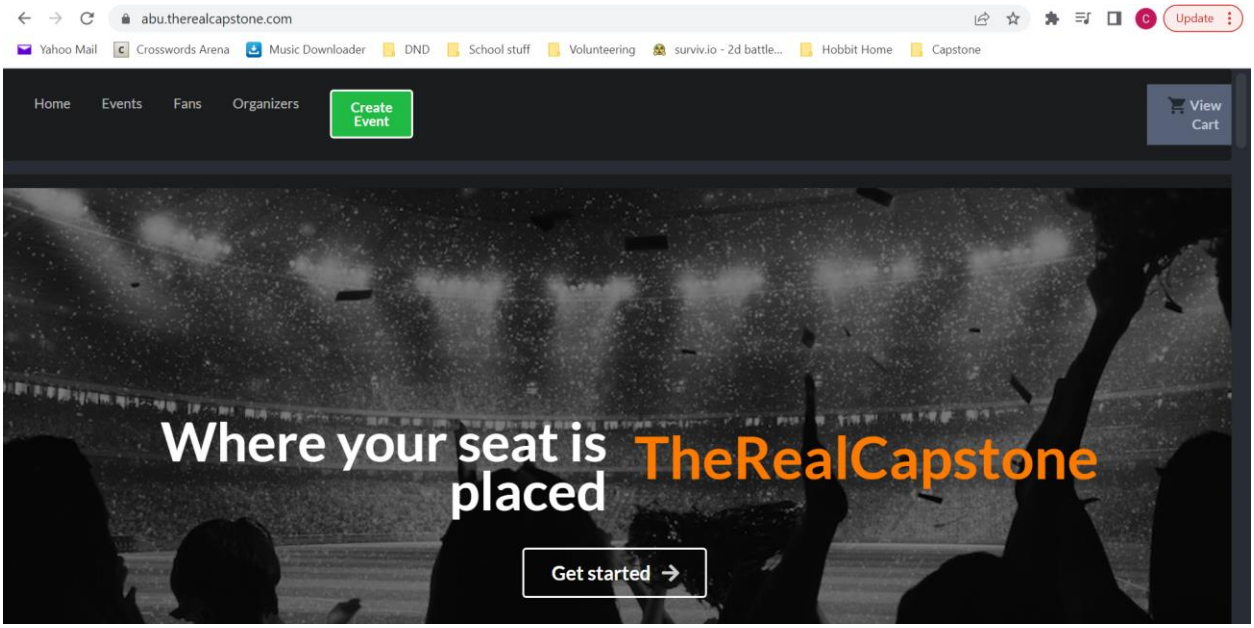
AWS Overview

Amazon Web Services (AWS) is an on-demand cloud computing service that allows users to serverless deploy websites and provides services for back-end set up. TheRealCapstone used AWS for storage, databases, creating a serverless application, connecting to APIs, running lambdas, and downloading content.

S3

AWS Simple Storage Service, or S3, is a scalable object storage service. All code deployed by our developers goes through our specific bucket and displays on a website with the same name as the bucket. For example, any code deployed by Abu would go through the S3 bucket abu.therealcapstone.com and going to that link would show the website as deployed.

○	abu.therealcapstone.com	US East (N. Virginia) us-east-1	Public	January 24, 2022, 18:31:53 (UTC-05:00)



Each developer's bucket follows the same naming scheme as above, with [name].therealcapstone.com.

○	caty.therealcapstone.com	US East (N. Virginia) us-east-1	Public	January 23, 2022, 11:51:12 (UTC-05:00)
○	elijah.therealcapstone.com	US East (N. Virginia) us-east-1	Public	January 28, 2022, 07:51:18 (UTC-05:00)
○	jose.therealcapstone.com	US East (N. Virginia) us-east-1	Public	January 24, 2022, 14:45:01 (UTC-05:00)

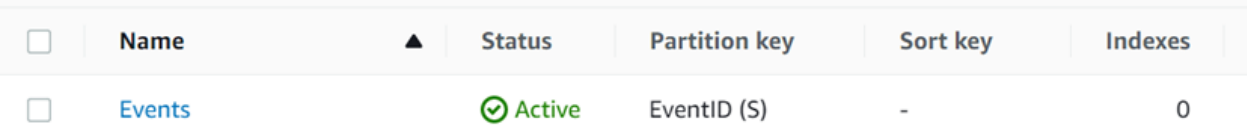
○	lam.therealcapstone.com	US East (N. Virginia) us-east-1	Public	January 26, 2022, 15:08:45 (UTC-05:00)
○	steven.therealcapstone.com	US East (N. Virginia) us-east-1	Public	January 25, 2022, 14:31:16 (UTC-05:00)

Each developer will send code to their bucket for testing before sending it on to stage and production. Both the stage and production environments have their own buckets. The stage bucket, stg.therealcapstone.com, holds the combined code from all developers to allow for testing to determine if the different code pushes work together before deploying the code to our production environment, www.therealcapstone.com. The production environment is our final product and code is only deployed to it after tests to make sure that any new code will not break our current product.

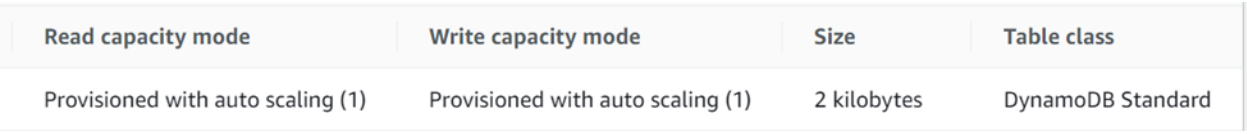
○	stg.therealcapstone.com	US East (N. Virginia) us-east-1	⚠ Public	February 18, 2022, 12:20:25 (UTC-05:00)
○	www.therealcapstone.com	US East (N. Virginia) us-east-1	⚠ Public	January 16, 2022, 20:09:34 (UTC-05:00)

DynamoDB

AWS DyanmoDB is a NoSQL database service where items can be added manually or dynamically. TheRealCapstone used DynamoDB for two databases, Events and User_Transactions. The Events database holds the information for every event posted to the website.



	Name ▲	Status	Partition key	Sort key	Indexes
☐	Events	✓ Active	EventID (S)	-	0

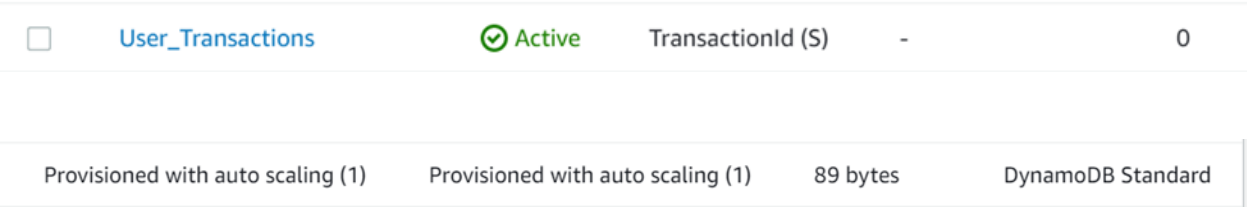


Read capacity mode	Write capacity mode	Size	Table class
Provisioned with auto scaling (1)	Provisioned with auto scaling (1)	2 kilobytes	DynamoDB Standard

EventID holds a unique key generated for each event posted, EventImg holds a URL to a picture associated with the event, EventType holds the type of event it is, EventLocation describes the location/venue of the event, EventTitle holds the name of the event, and EventPrice holds the cost of a ticket to the event. An example of an item in the database is shown below.


```
{
  "EventID": {
    "S": "53fedd97-79d8-4174-97a6-ba986785a379"
  },
  "EventImg": {
    "S": "https://images.pexels.com/photos/268882/pexels-photo-268882.jpeg?auto=compress&cs=tinysrgb&dpr=1&w=500"
  },
  "EventType": {
    "S": "Sports"
  },
  "EventLocation": {
    "S": "KFC Yum! Center"
  },
  "EventTitle": {
    "S": "JV BasketBall"
  },
  "EventPrice": {
    "S": "49"
  }
}
```


The User_Transactions database holds the information for every transaction a user makes.



TransactionID holds a unique key generated for each user transaction posted, TotalPrice holds the total cost of all tickets purchased by the user, and CreatedOn holds when the transaction was completed. An example of an item in the database is shown below.

To send the data to the databases, different lambdas were created. Two for the Event database to add new events to the database and to pull all events from the database to show on the website, and one for the User_Transaction database to add transactions to the database.

SAM CLI

AWS provides a friendly and easy to use console for creating all the resources that you need to create your website. The console is nice for when you have a small project but for large deployments, or if you want to deploy all of your infrastructure to another region it is better to use the SAM CLI. The CLI allows you to define all of the resources that you need into a yaml file. Once the yaml file is created you can deploy that yaml file to AWS using SAM. AWS will receive the yaml file and generate all the resources defined and deploy them. Alongside the yaml file, SAM requires or creates a ‘samconfig’ file that holds the values for all the parameters to be

passed with the yaml. These parameters can be used to determine what S3 bucket to use or what domain should be used for the CloudFront Distribution.

Like I mentioned earlier this is mostly useful for distributing massive systems all at once. It takes out the manual work of creating everything in the console for every region that you want to deploy to. Once the yaml file is completed, you can reuse this same file for any project that you want to create. All that is required is that you change the parameters to fit your needs.

API Gateway

Considering that the project was an interactive web application that fetches and modifies data, a layer of APIs needed to be implemented to allow each piece of software to communicate and handle requests. Since AWS offers a service called Amazon API Gateway to create and maintain APIs at scale, that is what was moved forward with. There are two main API Types that are offered with Amazon API Gateway, RESTful APIs and websocket APIs. Since HTTP responses were expected in this web application, stateless and standard operations, fast, and reliable, it was what was used in all our APIs that were implemented.

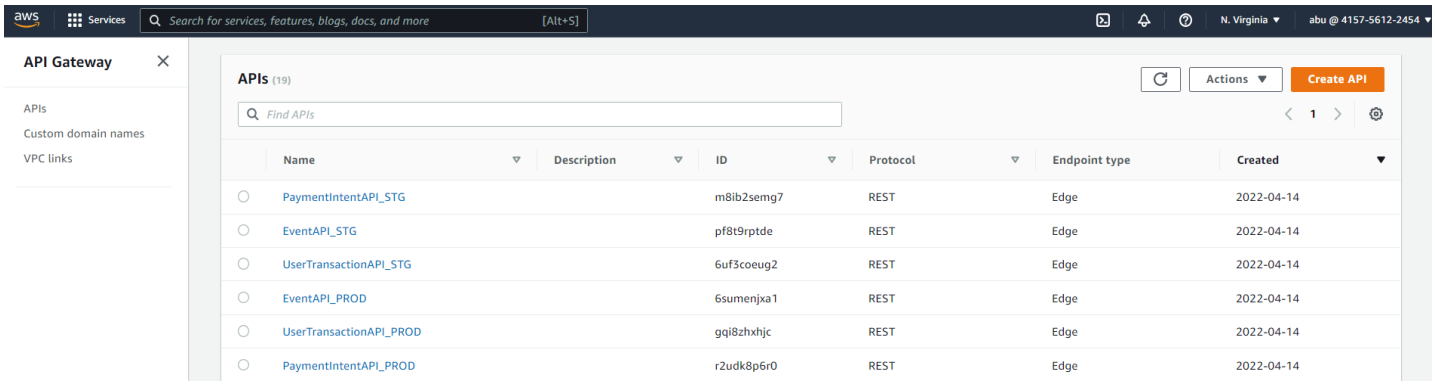
There were three main items our web application was going to handle: events items, user transactions, and stripe payment intents. Since we were following the microservice architecture pattern that structures an application as a loosely coupled and collaborative set of services, we defined API gateways for events, user transactions, and stripe payment intents. This was done using AWS SAM, the excerpt of that section can be found below.

```yaml
EventAPI:
  Type: AWS::Serverless::Api
  Properties:
    Name: !Sub "EventAPI_${ENVIRONMENT}"
    StageName: Prod
    Cors:
      AllowMethods: "'*'"
      AllowHeaders: "'*'"
      AllowOrigin: "'*'"

UserTransactionAPI:
  Type: AWS::Serverless::Api
  Properties:
    Name: !Sub "UserTransactionAPI_${ENVIRONMENT}"
    StageName: Prod
    Cors:
      AllowMethods: "'*'"
      AllowHeaders: "'*'"
      AllowOrigin: "'*'"

PaymentIntentAPI:
  Type: AWS::Serverless::Api
  Properties:
    Name: !Sub "PaymentIntentAPI_${ENVIRONMENT}"
    StageName: Prod
    Cors:
      AllowMethods: "'*'"
      AllowHeaders: "'*'"
      AllowOrigin: "'*'"
```

In the code excerpt you can notice the consistent implementation pattern that was followed since the main requirements of each API did not deviate at all. First comes the resource name such as “EventAPI” (which will also be referenced by other AWS resources mentioned later), then the obvious “type” of this resource. The following section is the properties such as “Name”, which needed to be modified based on which environment was deploying the template so each environment can have their own resource instance for their separate development purposes. Also included in the properties section is the stage name, which we always kept as prod since staging would be done on a separate domain instance, and “Cors”, which manages cross-origin resource sharing for the API and in this case we pass the wildcard operator for each configuration for testing. Thus when deployed for staging and production we can see the below creations in the AWS API Gateway console.



Since there were two deployments for both production and staging, as well as three APIs for events, user transactions, and payment intents. A total of six API instances were created. Once the APIs were set up, a “front door” was established for other AWS applications to interface with such as the next resources that will be discussed, including AWS Lambda. API Gateway enables a two way communication that can be demonstrated when any of the endpoints are reached.

Lambda

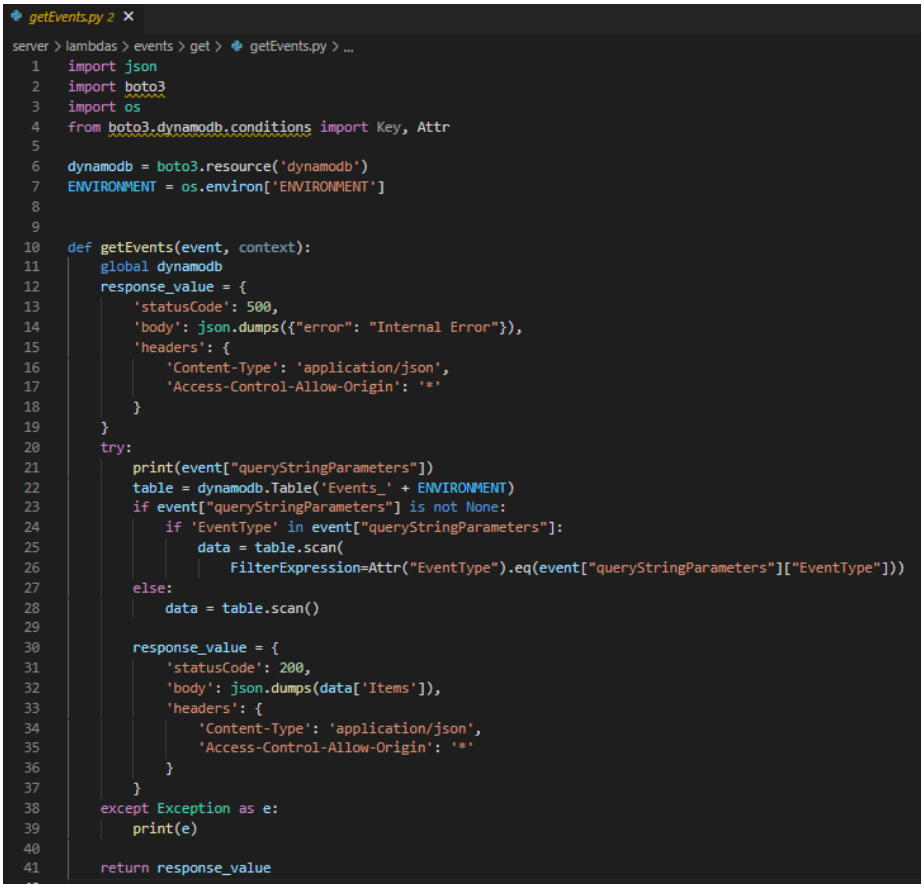
AWS Lambda, a serverless and event-driven service that allows us to run code for our backend service, was connected with API Gateway to handle all our requests to our database as well as handling payments with Stripe. Combining API Gateway and AWS Lambda is the perfect combination as it allows the ability to release new features faster and at scale thanks to the AWS environment.

The lambdas that were created were all made in Python and used a breadth of packages such as the Boto3 SDK, which makes it much easier to integrate AWS services such as DynamoDB with our python application. Some of the lambdas included: The get and post requests for events, as well as the posts requests for both user transactions and stripe payments. This follows the microservice architecture mentioned before and was all deployed using AWS

SAM. Once deployed these endpoints can be found in the AWS Lambda function console, where we can find and test the API endpoints that are triggering the Lambda functions. During testing the original API endpoints would be used which didn't route to the custom domain names and instead had their own unique API ids such as:

https://pdtwnb8fu5.execute-api.us-east-1.amazonaws.com/Prod/api/events

Accessing this endpoint would execute the get request of the EventsAPI and return the Json body of all the event items stored inside of the events table (the one deployed in the "abu" developer environment in this case"). The code of the lambda can be observed below.



The getEvents lambda uses the Boto3 SDK to scan the entire table that belongs to the current environment if there are no query parameters (in case only a certain event type was expected to

be filtered). The response body would then be updated upon success which would then be returned.

CloudFront

CloudFront is AWS's CDN which allows you to register a SSL certificate and securely share content. In the case of our website we are only sharing the html, css, and images associated with the page. For us the biggest feature that we utilized was the SSL certificate. This allows us to channel our API through CloudFront and allow secure HTTP requests.

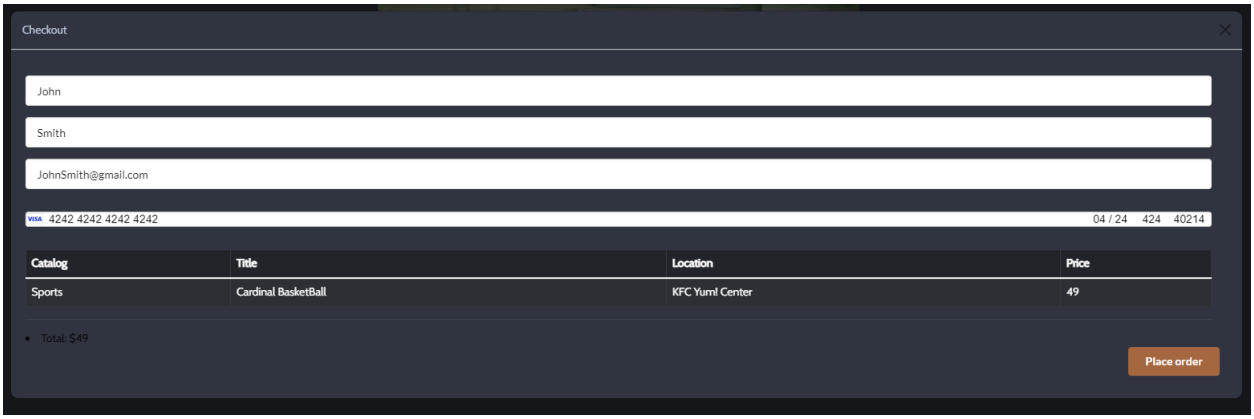
Seeing that CloudFront only distributes the content, it piggybacks off of S3 which is the file bucket that holds all of our webpage. Originally we were hosting our website using S3 using only HTTP rather than HTTPS. This worked fine and we were able to take the URL for that bucket and redirect it to our designated URL, therealcapstone.com, but the issue was that it wasn't secure. Seeing we were making an ECommerce site, it doesn't make sense to have an unsecure connection. It also allowed us to secure our API Gateways by routing our request through CloudFront. This is a huge plus for obvious security reasons.

CloudFront also allows you to set up your CORS settings. With these settings you can determine what request to serve and which to throw out based on the origin of the request and based on what kind of request is being sent. This allows us to disable HTTP requests from any other source than our live product. Nobody outside of our testing environment should be able to query or edit our data.

Stripe

As the project is an event ticketing platform, an online payment processing service needed to be implemented in the software, in this case Stripe was chosen. Stripe offers many powerful and easy to use APIs to allow payment transactions on your application. Stripe also

offers a thin wrapper around its Stripe Elements called React Stripe.js. These payment elements was used to create our checkout form which can be seen in the figure below:



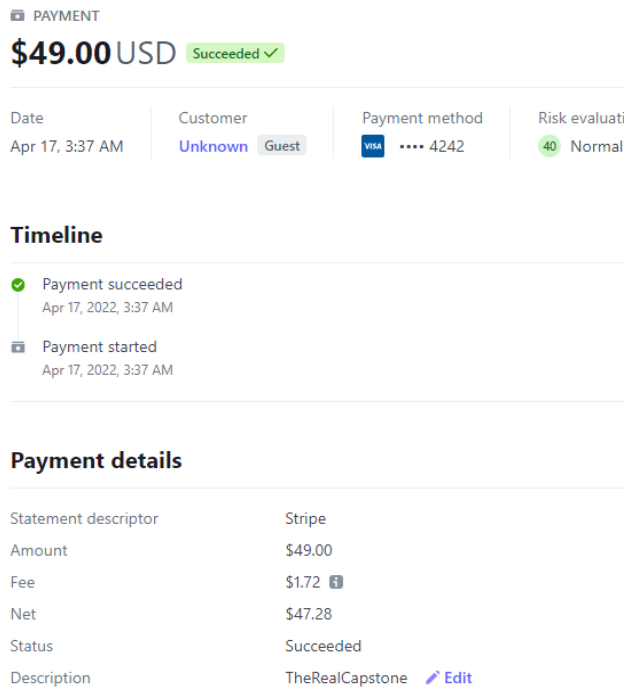
To complete a payment transaction in the checkout form with Stripe, a reference to the Stripe instance that was passed is set to variable. Then once a submission is made the card element is used to create a payment method with the passed card information, then a payment intent is created by calling the post request to the PaymentIntentsAPI that was set up using API Gateway, which makes the request to Stripe. The code can be also found below as well as the successful transaction that can be found in the Stripe dashboard.

```
const CheckoutForm = ({ show, handleClose }) => {
  // const [Success,setSuccess] = useState(false);
  const stripe = useStripe();
  const elements = useElements();
  const cart = useSelector((state) => state.cart);
  const initialValue = {
    FirstName: "",
    LastName: "",
    Email: "",
  };
  const validationSchema = Yup.object({
    FirstName: Yup.string().required("You must enter your first name"),
    LastName: Yup.string().required("You must enter your last name"),
    Email: Yup.string().required("You must enter your email address"),
  });

  const handleSubmit = async (value) => {
    console.log(value);
    const { error, paymentMethod } = await stripe.createPaymentMethod({
      type: "card",
      card: elements.getElement(CardElement),
    });
    if (!error) {
      try {
        const { id } = paymentMethod;

        const response = await agent.PaymentIntent.create({
          amount: cart.cartTotalAmount * 100,
          currency: "usd",
          description: "TheRealCapstone",
          payment_method: id,
        });

        //id containers payment method
        console.log(response);
        if (response.data.success) {
          console.log("Successful payment");
          // setSuccess(true);
        }
      } catch (error) {
        console.log(error);
      }
    }
  };
```



Environment Configuration

In our capstone we configured different environments for testing. There were 3 levels: personal, development, and production. Each environment had its own copies of API and DBs that could be configured to the developers' liking. In the case of our capstone all the members developed our features in our personal environments and then at the end of the week we all merged into the development environment. Sometimes we would have to update Tables in development so that there was data for new features. Lastly before the project was submitted we updated the production environment and made the website publicly accessible.

Testing Procedure and Analysis of Results

The team's testing procedure for our capstone was weekly throughout the development of our application. Every week we had tests to verify that our application was working as intended and each individual was responsible for their verification. The deployment tools we were using had features to verify that the input for our functions were formatted properly and that the correct user had access to the right functions.

When our capstone was finished we tested our application and made sure that every function in the final product was working properly. And verified the security of our application that no new code could easily be deployed. As of writing our application does all the correct functions like purchasing an event ticket, creating an event, sending the payment info to stripe and updating a table for transactions and events.

Societal Impact of Project Including Legal and Ethical Considerations

Among the various responsibilities that were designated throughout the project's lifespan, the team had to take careful consideration in dealing with those of ethical and legal nature. The resulting impacts of the project have an effect throughout various areas of reach, whether that be local, nation wide, or even global. As professional engineers, it was top priority to develop work that represents integrity and honesty. A standard in computing practice, decisions were made with informed judgment and careful consideration of the engineering impacts throughout various contexts. The team proceeded with task integrations only after impacts were measured and agreed upon as ethically and legally secured.

Due to the nature of the project, socially-driven events that center around various forms of entertainment to the likings of the individual, the impact placed is very important. Particularly in recent times, large social gatherings are seen as unnatural. The impact that the global pandemic has caused takes time for people to become accustomed to what once was seen as natural. Through an economic standpoint, festivities like those presented on the project's site are positive for the economy. However, through an ethical standpoint, those same events are viewed as inconsiderate and acquisitive.

As engineers, the team expanded further on this impact to review how to proceed with the best intentions for all those involved. The project's sponsor and TheRealSeat CEO, Dan Ross-Li, explained how the intentions of the company are to begin operating once large gatherings were deemed safe. Ultimately, the decision was made that even if the current situation seemed

unethical due to health conditions that globally impacted humanity, the parent company would only publicly operate once it was legally allowed, therefore the effects wouldn't be negative for society. Through careful measurement and understanding, the team carried on with the responsibility.

In terms of legality involved in the project, there was minimal. The engineering frameworks surrounding the practice of engineering were clearly understood by all team members. Dan and the university had all of the team members sign a non-disclosure agreement, for security reasons in order to protect the sponsor company. It was fully known that any sharing of confidential information with those outside of project involvement was strictly and legally prohibited. This confidentiality obligation was fully respected by the team and trust was maintained among all involved. Code of conduct was thoroughly practiced and acknowledged when dealing with all forms of legal responsibilities pertaining to the project.

Currently in our modern internet society, there are corporations that resemble that of a monopoly due to their vast market share. These companies have direct connections with venues and event organizers, together they conduct business to reap max profits of ticket selling. The issue with this form of practice is that it is highly unethical and one could even question the legality. The ticket selling platform sells the tickets at an extravagant price due to their deemed terms of secondary market elimination, but others see it as greed. They drain fans economically because there are no other options.

TheRealSeat seeks to eliminate this corporate greed by providing exclusion of unnecessary fees. By incorporating dynamic pricing, allowing the pricing of tickets to be set by the organizer rather than by the ticket selling platform. Utilizing this method leads to an overall increase in societal benefitting. Fans have the ability to gather with friends, and form deeper

relationships built upon positive experiences. As has been researched, social gatherings and the forming of interpersonal relationships leads to better overall mental health. Others can meet new people with common interests. Networking increases individuals' possibilities to succeed in various areas of life. All of these are forms of benefits that social interaction brings and can be directly provided by TheRealSeat.

The following are examples and explanations of the fees charged for every ticket by Ticketmaster, the largest market share holder. Face price - an established price set by the client as a minimum earning. Service fee and order processing fee - A fee set based on servicing and processing of each transaction. TM + resale service fee - a set fee for the reselling of each ticket, priced according to ticket price. Delivery price - a fee that is based on delivery of each ticket. Facility charge - a fee for the host venue/facility where the event will take place. Taxes - standard tax fee pertaining to each state. Once all is accumulated, the fees can at times even be more than the base price of the ticket. This practice has discouraged many customers and even has led to some questioning the integrity of the company. As of recent, many view this and can statistically be proven as corporate greed. Because of these actions, legal actions have been requested to place a review on this corporation.

This new system that is to be implemented by TheRealSeat onto the e-commerce platform project, benefits customers not only positively in a social aspect but ethically as well. Rather than having the system be abused to where the vendor can take advantage of customers economically by setting any price wanted, customers would pay far less in comparison. The legality of this absurd and unethical system that is currently implemented seems to be in resemblance to that of scalping, but instead of individuals charging these prices, they come directly from distributors.

Relating to the topic of reselling done by corporations, the team took consideration of reselling from an individual's perspective as well. Greed leads individuals to seek an opportunity to profit from the misfortune of others. There are those who seek to purchase as many possible tickets from platforms utilizing automated software. Considering the professional engineers code of conduct, the team formed a plan to prevent this from taking place on the project's site. Utilizing an API provided by Stripe, Inc. customers can only handle one participation per instance. Stripe provides and handles all of the transaction process, in a professional and legal manner. Implementing this feature onto the site was a great ethical accomplishment by the team.

Not only will the project website now allow multiple transactions from the same user, but there is also no implementation of reselling functionality hosted directly on the site. Other ticket merchants justify the high price in fees as a prevention mechanism to disencourage resellers. However, they still give those same resellers that they seeked to prevent the option of reselling directly within their platform. Absurdity and unprofessionalism is what customers currently have to deal with when conducting business provided by these corporations. As professional engineers, we seek integrity and justification of our actions. It is hypocritical to be in the contrary of something figuratively, yet still profit and even simplify those same actions due to it being a feature on your platform. The team unanimously decided that the societal effects allowing hosting reselling within the project were completely unethical and did not incorporate it.

Throughout the project's development, the team constantly gathered to review whether certain tasks were professional as it pertains to ethics and legality. Being a professional engineer goes beyond that of meeting requirements for technical work, it also pertains to critical thinking on how the work created will affect society. At times there were some complications and components of the project where boundaries became unfamiliar. To achieve success the team

came together and reviewed the engineering standards to seek out the best course of action. Current trends in the business pertaining to the team project seem to be opposing, however that never discouraged the team to continue with what is ethical and legal.

Contribution to Society

Online ticketing systems are popular on the market nowadays. Customers can find many ways to buy their tickets even through many kinds of means such as venues, and provider websites. But almost every means that exists nowadays sells tickets from their own venues and that can cause different prices of the tickets to happen. And if customers go to any physical venues to buy tickets, there can be a line and there is a chance the ticket can be sought out when it gets to their turn in the line. This can cause unpleasant experiences when buying tickets online. TheRealCapstone shows the idea of how people can sell tickets not just only venues to customers, but from customers to customers. It is a good concept of providing a better experience for clients to choose their best options when purchasing online tickets.

Regarding the technology aspect, most existing companies have had their technology implemented for a really long time now. This makes it very hard to integrate new technology into the existing system. TheRealCapstone project takes a step further in implementing the technology of both front-end,back-end and databases. With React-js and Redux for front-end this technology is long supported which can guarantee future versions of the software with responsive UI. The back-end integrates with the newest cloud technology(AWS) and DynamoDB which is a non-relational database and it can boost up the time for querying the database. The back-end is hosted with Amazon Cloud which has very high up running time. This all together provides a robust system that can handle commercial scale amounts of users. Through technology and concept aspects, TheRealCapstone contributes to society a better view of new ways to offer better online ticketing experience for both users and developers.

Engineering Standards, Constraints, and Security

IEEE Std 12207 - Software Life Cycle Process

In our project we continued to discuss progress with our sponsor throughout the entire semester. We made sure he was updated on all progress that was made, and he decided what work should be done on what sprints.

To keep track of our progress we used Notion, a ticket noteboard that allowed us to block off tasks and to assign them to members. We used this to track progress and to get feedback from our sponsor on the tasks that were done.

IEEE Std 730 - Software Quality Assurance Plans

We used GitHub actions to implement CI/CD. This allowed us to continually progress without having to redeploy manually. This allowed us to push changes to our source control repository and for our website to automatically update. Once the changes had been deployed our sponsor could take a look and tell us his feedback.

IEEE Std 1012 - Software Verification and Validation

This section is similar to 730, our CI/CD ran test to ensure that the code was warning and error free. Once these checks were done it would then push to our website.

We did manual tests during development on our local machines. This allowed us to make sure our changes were accurate before attempting to push.

IEEE Std 1233 - System Requirement Specification

The specifications were set out by our sponsor directly, so all we had to do was follow these specifications.

IEEE Std 1016 - Software Design Document

This section continually changed based on the dependencies that were added throughout the semester. When we moved into a new section our sponsor would ask for a new feature to be implemented and we would commonly have to add a library to help us do so.

IEEE Std 829 - Software Test Document

We simply followed the testing procedure that was set out by our sponsor and set up the repository to check our code before deploying the changes.

Conclusions

In conclusion a full stack event ticketing web application was successfully developed by identifying, formulating, and applying principles of computer engineering and science. The original goal that was set forth in the initial planning phase of the capstone was done by analyzing the given computing problem and applying principles of engineering to implement a solution using a modern platform that consists of React JS, AWS, and following proper testing procedures. The solutions and overall outcome of the project as a whole met the specified needs with important considerations such as data privacy and scaling costs, as well as other factors which include but are not limited to: global, cultural, and environmental factors.

The designed and implemented solution described in the software detailed implementation, followed a given set of requirements that can be observed in the system description to satisfy the described audience. Furthermore, throughout development appropriate experimentation was conducted using testing procedures in GitHub Actions which was analyzed and interpreted using engineering judgements to come to the observed conclusions. Ethical and professional responsibilities were recognized as well and informed judgements during design and development considerations were made which considered the impacts of a number of engineering solutions.The foundation that was made for a serverless eventing ticketing platform was complete and further user features can be implemented using the provided documentations and design. Developing the project has grown many technical and professional skills that will be applied in future projects such as the understanding and development in different phases of the software development life cycle and a number of IEEE standards.

Recommendations for Future Work

TheRealCapstone is a project based around showing users different events and allowing them to purchase tickets to the events they are interested in. The project in its current state performs well, but there are still many different implementations that could be added to continue improving the site.

The current interface for showing events is a page where a user can scroll through and see events separated by broad categories such as “Sports” and “Concerts.” If there are a small number of events listed, this isn’t a bad way for a user to find an event, but as more events get listed, it can be hard for a user to find what they are looking for. To combat this, a search bar could be implemented so that users can search for specific events. The search bar could be created to allow users to search by category, venue, state/city, and name.

Further implementations for the event page would be to add in event recommendations. This could be separated into a few different categories. First, an event spotlight could be implemented. This would be a recommendation that shows up to all users. Events in the spotlight would be larger, more popular events like the Super Bowl or Coachella. Since these are generally more popular than other events may be, spotlighting them would allow users to more easily find them and more tickets would be sold.

Another way to add in event recommendations would be to use user location data. By asking the user to allow TheRealCapstone to use their location, events can be recommended to the user within a chosen radius. If the user were accessing the site from Louisville, then events from Louisville, all around Kentucky, or even in neighboring states could all be recommended. This can be implemented so that the user does not have to give their current location and can

instead input their chosen location and radius and search for events in that specified area. This would work similarly to searching up cities and states, but would allow the recommended events to be from a broader area.

To continue on the idea of using user data for event recommendation, it could be implemented that users get events recommended for them based on past purchases and searches. User accounts could be created to allow for purchase and search data to be stored and then later called upon for recommendation. To keep the recommendations up to date, it would be best to keep and use only data from the past twenty to thirty purchases. Doing this would allow a user who had previously bought tickets to a basketball game to be shown more sporting events. To improve this recommendation further, every event could get a sub-category since “Sports” is a broad category especially if a user is only interested in basketball.

Along with user accounts, event coordinators could have specific accounts to allow them to submit events and get ticket data back. An example of data they could receive is the seats that are the most popular, what areas in the venue are more likely to be purchased. Seat maps could be submitted by event coordinators that would be shown to users so that they knew what tickets they were purchasing. These maps can be expanded upon to allow users to be able to click on their desired seats straight from the map.

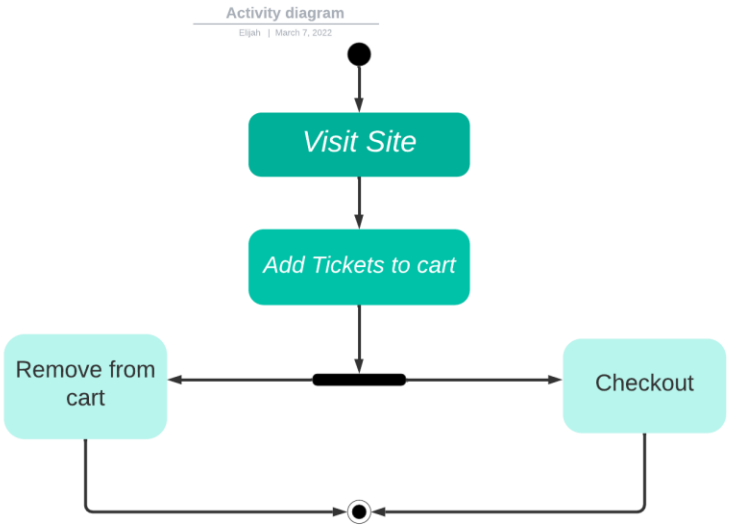
Appendices

Customer Contact Information: drossli@therealseat.com

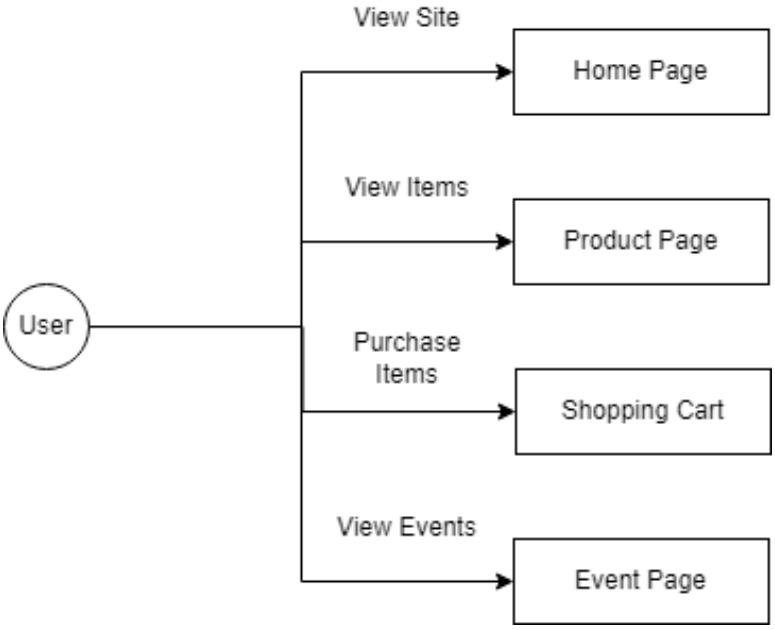
Data Sheets: N/A

Additional Drawings and Diagrams:

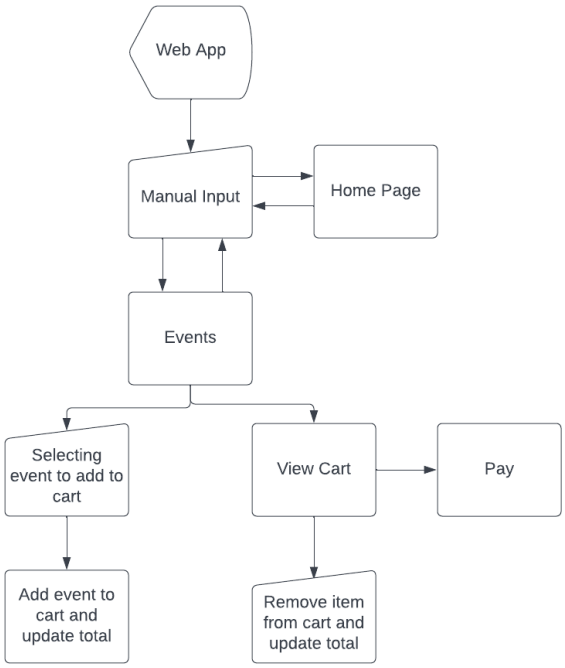
Activity Diagram



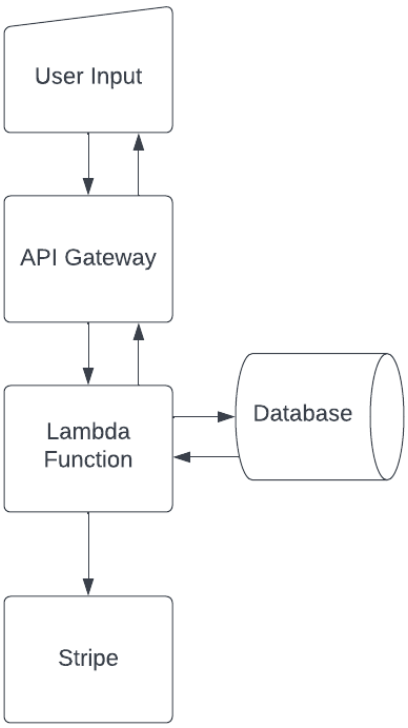
Use Case Diagram:



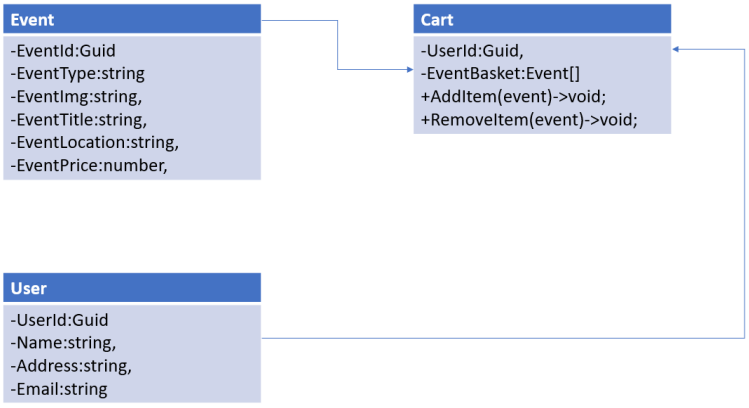
Interaction Overview Diagram:



Communication Diagram:



Class Diagram:



Source Code:

GitHub Repository: https://github.com/TheRealSeat/Capstone

Website: TheRealCapstone.com

Experimental Results: (See GitHub actions test workflows in repo)

Software Installation Instructions:

1. Pull Repository and run react app

Users Manuals: Repo ReadMe

Quotes: N/A

White Papers: N/A