

Overview:

Meet is an application that allows users to login and authenticate themselves with their gmail accounts where they can then view details about multiple programming events taking place in various cities and countries. A search functionality is also a part of the Meet app, so a user is able to search for a city and get a list of events in that city. This project did present several challenges based on it's requirements which are listed as follows:

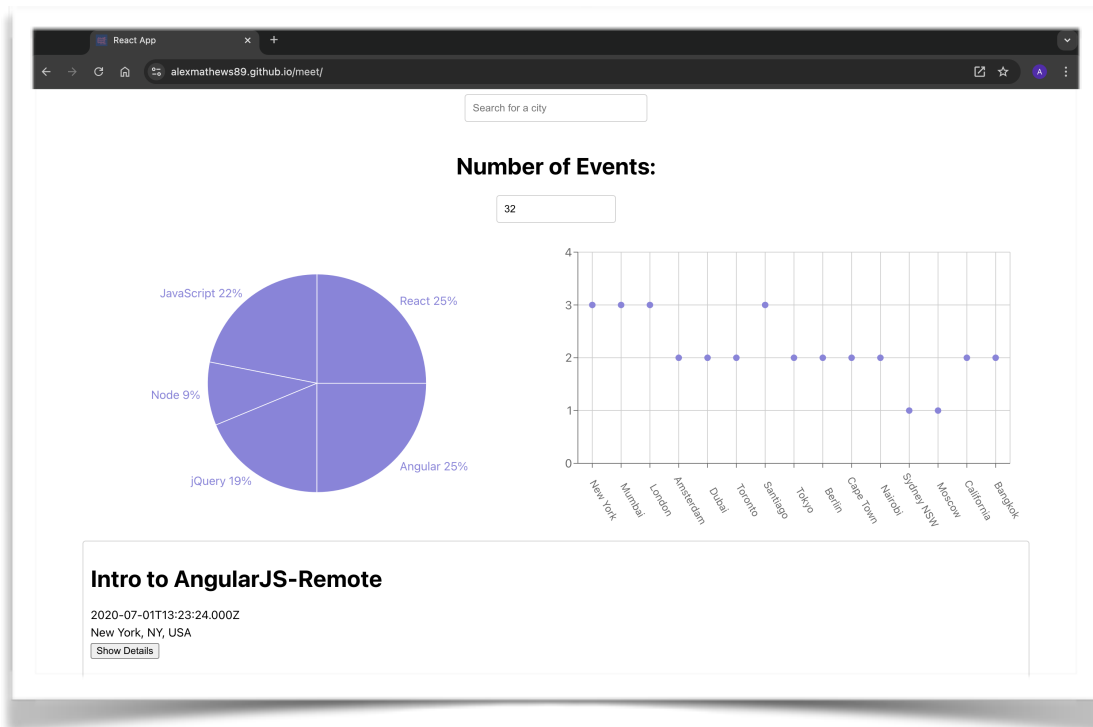
- The app must be built using a test - driven development(TDD) technique.
- The app must use the Google Calendar API and OAuth2 authentication flow.
- The app must serve as a progressive web application(PWA) meaning it must work offline and have the ability to be installed and added as a shortcut to a mobile device or computer.
- The app must use serverless functions for the authorization server.
- The app must display charts visualizing event details

Purpose and Context:

The Meet app was a project I built for my full-stack web development course at CareerFoundry.

Tools and Methodologies:

This is a serverless application, so there is no backend maintenance and is easy to scale. It uses the FaaS(Function as a Service) cloud service and the Google Calendar API. Users are authenticated with OAuth2 authentication flow and access tokens are provided with AWS Lambda. This project was developed with React.



Project Setup

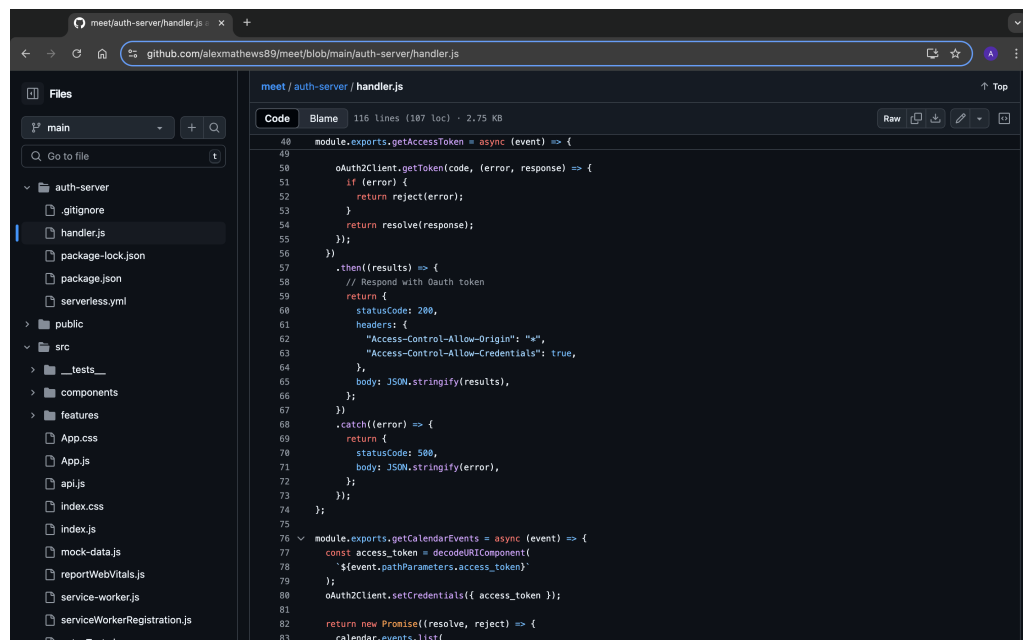
The initial setup of this application was done using a CRA(Create-React-App) tool which makes this project unique in that all of my other React applications were built from scratch.

- This type of setup created the necessary test files to go about its development using a TDD approach which was one of its primary purposes.
- This setup allowed me to dive into the project's source code right away.
- The CRA setup takes care of the build process and runs the app with a simple “npm run start” command without having to use a third party tool such as Parcel.

APIs and Authorization

As mentioned earlier, the Meet app is a serverless application. All events that are accessed through this application are public calendar events, so it wouldn't make sense to build and maintain an entire server just to provide access tokens, this is why setting up the app to use the Google Calendar API and a function hosted on AWS Lambda to handle authentication requests was the best option for this step.

- Google API Console is where the client_ID and client_secret credentials were generated which were necessary for user authorization for this project.
- This step is where the app was connected to the Google Calendar API.
- AWS is what generated two very important files when it comes to serverless applications and the FaaS cloud service: the handler.js file and the serverless.yml file.
- AWS Lambda is where both the access key and secret access key were generated which were needed to configure AWS for serverless, one problem to look out for here was to make sure that neither of these credentials ended up on GitHub, so I had to make sure that when I ran the “serverless config credentials” command in the terminal, that neither of these credentials were stored in the auth-server directory or anywhere else in the project before pushing anything to GitHub.
- This step is where I added the necessary code to the handler.js and serverless.yml files such as functions like getAuthURL, getAccessToken, and getCalendarEvents. These functions are what defined the steps to allow users to login with their gmail account and view a list of calendar events.



The screenshot shows a web browser displaying a GitHub repository. The address bar shows the URL: `github.com/alexmathews89/meet/blob/main/auth-server/handler.js`. On the left, a file explorer sidebar shows the repository structure, with the `auth-server` directory selected. The main area displays the content of `handler.js`, which is 116 lines long. The code is written in JavaScript and uses ES6 arrow functions and async/await. It defines two main functions: `getAccessToken` and `getCalendarEvents`. `getAccessToken` takes an event object and returns an access token by calling `oAuth2Client.getToken`. `getCalendarEvents` takes an event object and returns a list of calendar events by calling `calendar.events.list`. The code includes error handling and sets appropriate status codes and headers for the responses.

```
40 module.exports.getAccessToken = async (event) => {
41
42   oAuth2Client.getToken(code, (error, response) => {
43     if (error) {
44       return reject(error);
45     }
46     return resolve(response);
47   });
48   .then((results) => {
49     // Respond with OAuth token
50     return {
51       statusCode: 200,
52       headers: {
53         "Access-Control-Allow-Origin": "*",
54         "Access-Control-Allow-Credentials": true,
55       },
56       body: JSON.stringify(results),
57     };
58   })
59   .catch((error) => {
60     return {
61       statusCode: 500,
62       body: JSON.stringify(error),
63     };
64   });
65 }
66
67 module.exports.getCalendarEvents = async (event) => {
68   const access_token = decodeURIComponent(
69     `${event.pathParameters.access_token}`
70   );
71   oAuth2Client.setCredentials({ access_token });
72   return new Promise((resolve, reject) => {
73     calendar.events.list(
74
75
```

Implemented Unit Testing

One of the primary purposes to this project was to go about the programming the application from a test-driven-development(TDD) approach. In this step, I implemented the unit tests for the project.

- A unit test is designed to test a single unit of code such as a function, so this is where I not only wrote the unit tests, but also declared many of the the different components of the app.
- Some of the primary components developed here were the event list, events, the city search text box.
- Not only were the unit tests written in this step, but also enough code to get the unit tests to pass, which also implemented the initial functionality of the app, such as programming the text box to render city suggestions based on what the user types.
- Because unit testing is only meant to verify that each component is functioning independently, mock data provided by CareerFoundry was used for this step rather than data from the Google Calendar API for the final project.

Implemented Integration Testing

All of the unit tests written in the previous step were not enough to make sure this app functioned in its entirety, to do this, I also had to write integration tests to make sure each individual component or part of the app worked with each other.

- This is where the overall structure of the app's components in which were tested and designed in the previous step were set up also through testing. An example would be writing a test to access the event list through the app component, confirming the event list is a child component of the app component, and setting that test up to "expect" the number of events to be equal to the number of events specified in the event list component.
- Because only mock data was used in the previous step, some of the data used to get the prior unit tests to pass were hardcoded into the component. The integration testing implemented here involved setting up the components to reference data from each other rather than the mock data used for unit testing.

Implemented Progressive Functionality

The Meet app is also a progressive web application(PWA) which means it has the ability to work offline and can be installed on both mobile devices and computers.

- As mentioned in the very first step, a CRA tool was used to initially set up this app, so a good portion of the PWA work had already been done during the project setup.
- The CRA tool also generates a manifest.json file in the project which includes customizable properties such as icons. This is where I set up the icons that would display as a shortcut if a user were to install the app on their mobile device or computer.

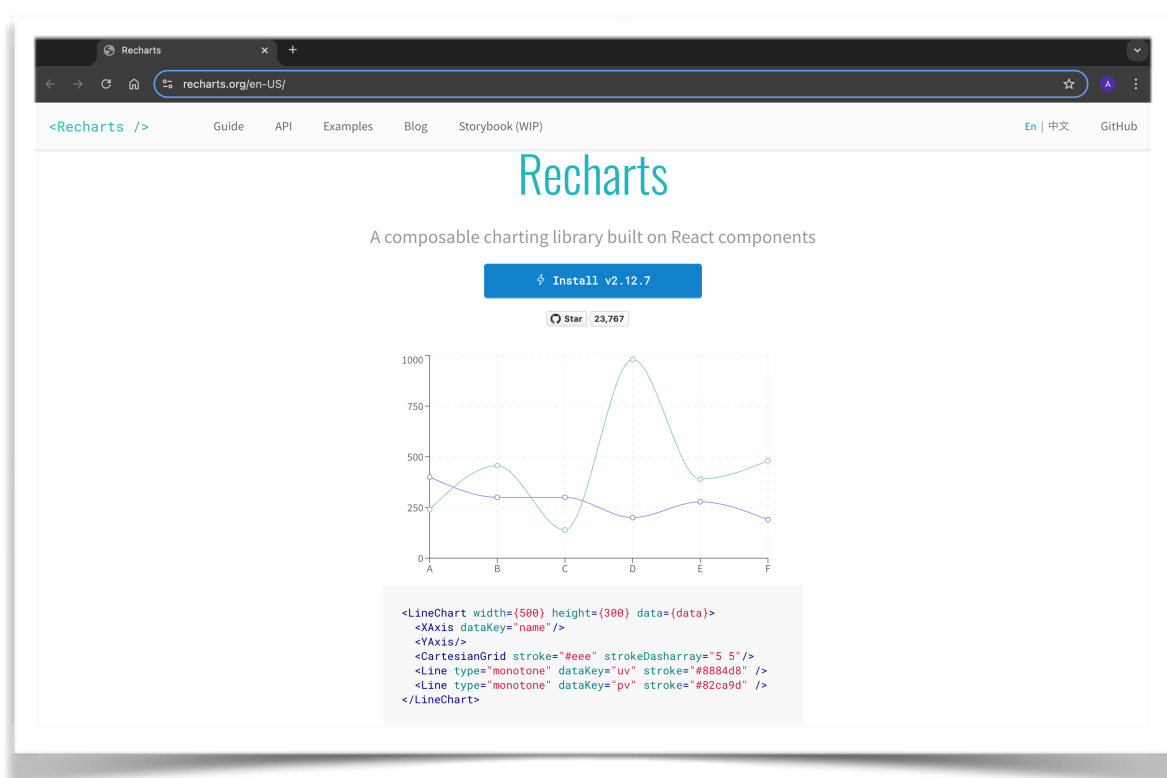


- A service worker was also required for the app to load while offline and cache resources in the browser.
- Very similarly to setting up the icons that would serve as a shortcut to the app when downloaded, CRA had already created most of what I needed to allow the app to load while offline - a service worker. In fact, it had already created a service worker, all that I had to do was register it by changing one line in the CRA generated index.js file while making sure another CRA generated file, serviceWorkerRegistration.js was imported into the index.js file. (Pic of SW code here?)

Data Visualization & Charts

Graphs were also added to this app to make it more visually appealing. In total, two charts were added, one that shows how many events take place in each location which is shown with a scatterplot and another that shows the popularity of event genres which uses a pie chart.

- Recharts, a React charting library, is used in this step to create both the scatterplot and pie chart. Recharts was chosen because it supports a wide variety of charts such as area charts, pie charts, line charts, composed charts, pie charts, radar charts, etc. Recharts is also one of the most popular React data visualization libraries based on GitHub stars and is also well-documented.
- The code required to create the charts components was provided by the Recharts library, however, the charts needed to display the application's event data, so this step required changing the Recharts provided data with the app's event data.
- One problem I encountered here was that some of the plots on the scatterplot would disappear when I adjusted the size of the browser window to a smaller screen size. To fix this, I had to adjust a few of the properties in the X-axis element of the chart component which were the angle, interval and tick property.



Summary:

With the help of React's Create-React-App(CRA) tool, Google Oauth2, AWS Lambda and React's Recharts library, I was able to create a fully functioning Progressive Web Application in which a user can gather details about programming events taking place in different cities and countries all over the world. A surprising aspect of the project was the use of the CRA tool, I initially thought that using a tool that auto generated configuration files would put certain restrictions on the project, but in the end, it actually turned out to be very useful when it came to making sure the app served as a PWA. This really helped in completing all the required steps such as following a test-driven development approach and contributed to the final result of the project turning out just it was outlined in the original project brief.