Casestudy for Meet App

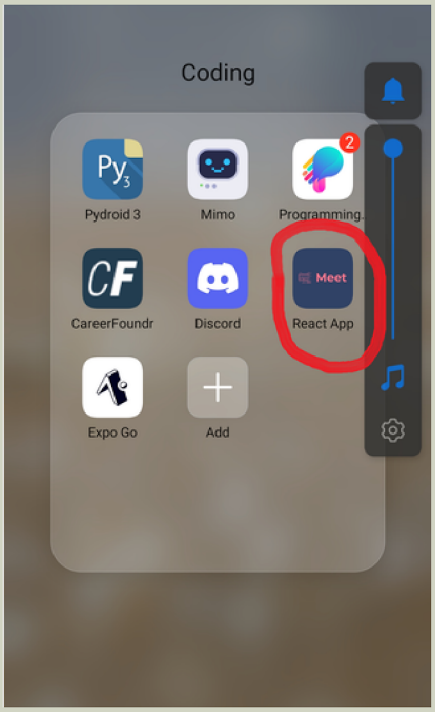
Meet App is a progressive web application where users can find details of upcoming events from all cities.

PREPARED BY SOPE KETIKU

2023

Overview

Meet App is a progressive web application where users can find details of upcoming events from all cities.



TOOLS USED:

React and *CSS*: to build the client-side UI
AWS Lambda: Serverless functions for coding authorization server
Gherkin: Test scenarios
Jest: Unit testing
Cucumber: User acceptance tests.
Puppeteer: End-to-end testing
Atatus: Performance monitoring

PURPOSE AND CONTEXT

Meet App is a personal project I built as part of my web development course at CareerFoundry to demonstrate my mastery of full-stack *JavaScript* development.

OBJECTIVE

The aim of the project was to build a serverless, progressive web application (PWA) with *React* using Google CalendarAPI to fetch data as part of my professional portfolio. The problem I wanted to solve was to achieve this using a test-driven development (TDD) technique.

DURATION

The project was intended to be completed in 6 weeks. I was able to complete within the planned completion timeline.

ROLE

Web developer

Approach

This entailed exploring the use of test-driven development to build a progressive web application using *JavaScript* and data visualization.

SERVER-SIDE

I set up my environment to prepare for writing serverless functions, by creating a *Google OAuth* consumer and setting up *AWS Lambda* to enable coding the authorization server.
I then had to write serverless functions using *AWS Lambda* to build an authorization server for issuing *OAuth2* access tokens which will be used to authorize requests to the Meet app. The Google Calendar API was then integrated into the Meet app.

```
const getToken = async (code) => {
    try {
        const encodeCode = encodeURIComponent(code);

        const response = await fetch(
            "https://9p662stctg.execute-api.eu-central-1.amazonaws.com/dev/api/token" +
                "/" +
                encodeCode
        );
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const { access_token } = await response.json();
        access_token && localStorage.setItem("access_token", access_token);
        return access_token;
    } catch (error) {
        error.json();
    }
};

/**
 *
 * This function will fetch the list of all events
 */
export const getEvents = async () => {
    if (window.location.href.startsWith("http://localhost")) {
        return mockData;
    }

    if (!navigator.online) {
```

CLIENT SIDE

The application has a main view where all events are displayed. Users can filter cities for events or number of events that will be displayed. Data visualization was integrated into the main view using a charting library called *Rechart* to transform the calendar API data into a visual format. Progressive functionality was implemented into the app, so that the app can be used offline or in slow network conditions with the help of a service worker and installed on desktop and mobile home screen.

```
Scenario: When user hasn't entered a set number of events to display, show all upcoming events from all cities
 Given the user hasn't searched for any city
 When the user opens the app
 Then the user should see a list of upcoming events

Scenario: User can set number of events to display when selecting a city from the suggested list
 Given the main page is open
 When  the user enters a number in the filter number of events to display text box
 Then the user should see a list of events matching the number entered and the city in the entered in the city text box
```

Automated Testing

It was important to test the key features of the Meet App using automated testing to safeguard the code whenever future changes are made.

Taking details from the project brief, each feature was broken down to separate scenarios which was then further broken down using *Gherkin's* "Given-When-Then" syntax.

This was used to create unit tests using the *React testing Library* which is more suitable for testing React components.

The next step was integration testing to test when the components interact with each other. *Cucumber* was used as a testing framework to transform the scenarios defined earlier into *JavaScript* tests using *Gherkin's* "Given, When, Then" syntax.

The creation of automated tests were completed by building user acceptance tests and end-to-end tests using *Puppeteer*, an automated testing framework which simulates user actions in the browser, eliminating the need for manual testers to perform the actions themselves.

Challenge

I had some challenges during the user acceptance testing stage when using an Application Performance Monitoring tool *Atatus* to detect errors and track the performance of the Meet app.

Solution

It was noticed after evaluating the app that events didn't filter locations properly when the number of events were less than the default 32. This was noticed as a bug in the code and had to do with the fetchData function in the *App.js* file. The bugs were later fixed and re-tested and worked properly.