Meet app

Development process

August	**2023**

*All usernames, passwords or other similar information shown in this presentation is fictive and for illustrative purpose only

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

01

Setting up the app development environment

Skills used

Research

WHY WAS THIS STEP IMPORTANT

Organizing the development and the coding environment for efficient and well-oriented work from the beginning is one of the keys to ensure smooth workflow and limit avoidable time loss due to inefficient project initialization.

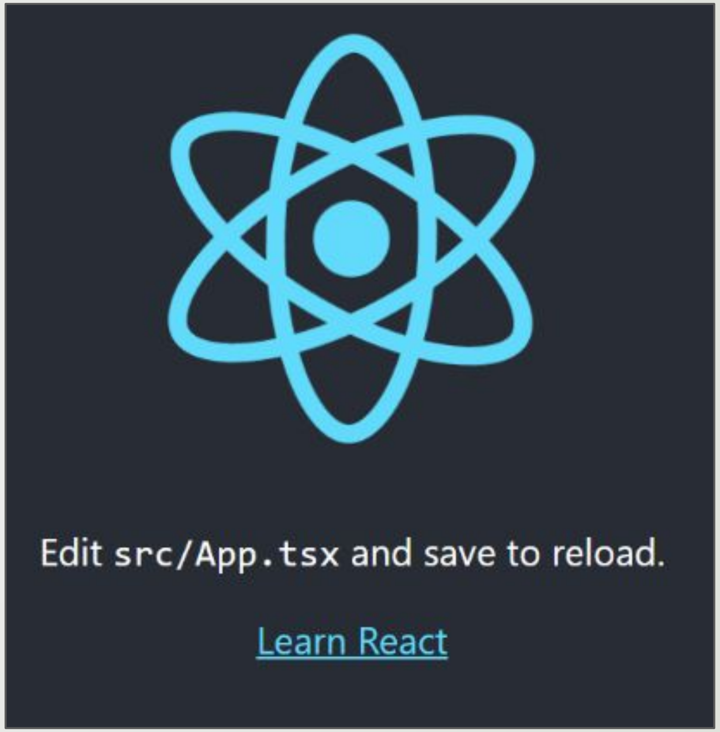
WHAT WAS THE GOAL

Creating a new Github repository for the project.

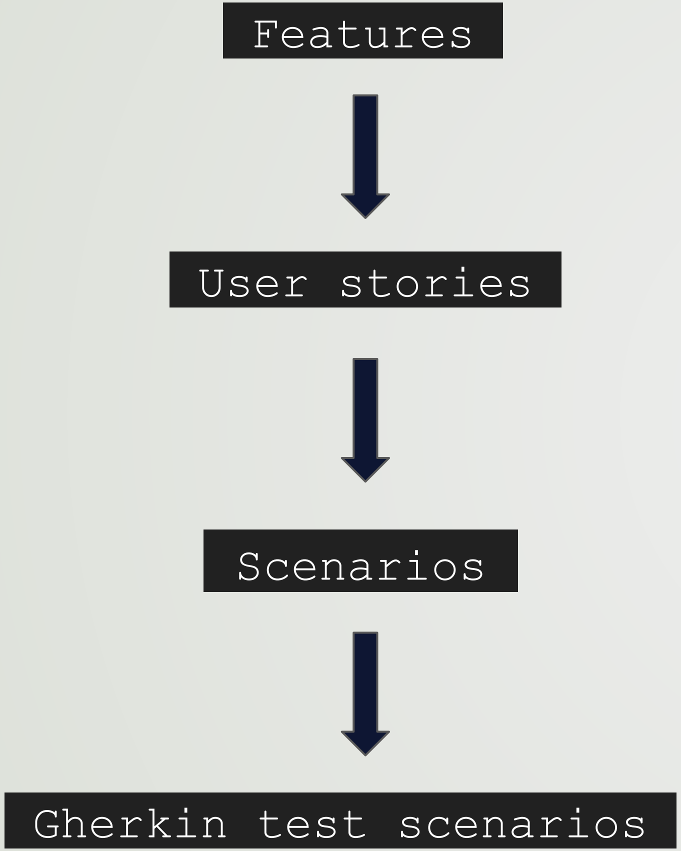
Setting up the project initial structure using Create-React-App (CRA), a tool that initiates React-based projects by automatically creating and configuring the build tools and first files needed (CRA allows to streamline the project creation process by skipping lot of manual work normally necessary at the beginning of a React project).

Ensuring proper project initialization by updating some elements in the newly created project files to allow its deployment (e.g.: add some elements in the package.json).

Pushing the created project to Github Pages.


Edit `src/App.tsx` and save to reload.

Learn React



02

Writing user stories and test scenarios

Skills used

Analytical thinking
Content writing

WHY WAS THIS STEP IMPORTANT

Features requirements for a product to develop normally give a good idea of what needs to be built, but it sometimes still remains relatively broad when it comes to identifying the exact necessary functionalities, hence the relevance of translating features into user stories to provide more information about what to implement exactly.

Yet, user stories, by nature, can also remain quite large. Breaking them down into multiple scenarios helps to better identify each individual function that needs to be implemented for each required feature.

These scenarios can in turn be translated into test scenarios using the Gherkin syntax, which allow to carry out several tests along the product development, and ensure all new logic implemented work while still contributing to the features that need to be coded.

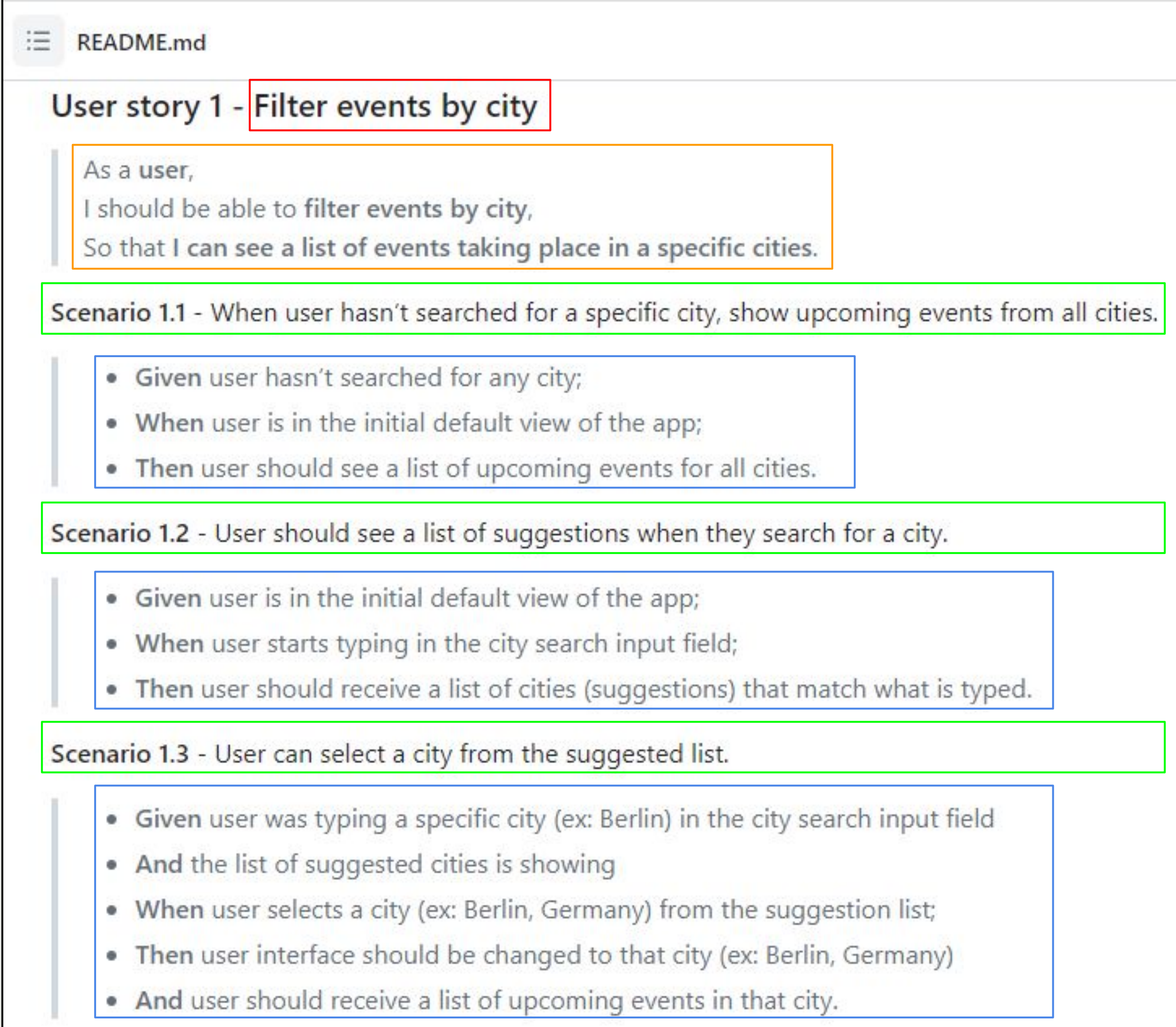
An example of this breakdown process, from high level features to low level Gherkin test scenarios is presented over the next slides.

WHAT WAS THE GOAL

Translating the six necessary key features from the project requirements for Meet app into user stories, then into scenarios and then into test scenarios using the Gherkin syntax.

Creating the README file for the project and updating it with the project general information and the user stories, scenarios and Gherkin test scenarios written (see an example over the next slides).

USER STORY, **SCENARIOS** AND **GHERKIN TEST SCENARIOS** FOR **FEATURE 1** REQUIRED IN THE PROJECT REQUIREMENTS (FILTER EVENTS BY CITY)


README.md

User story 1 - Filter events by city

> As a **user**,
> I should be able to **filter events by city**,
> So that **I can see a list of events taking place in a specific cities.**

Scenario 1.1 - When user hasn't searched for a specific city, show upcoming events from all cities.

> - **Given** user hasn't searched for any city;
> - **When** user is in the initial default view of the app;
> - **Then** user should see a list of upcoming events for all cities.

Scenario 1.2 - User should see a list of suggestions when they search for a city.

> - **Given** user is in the initial default view of the app;
> - **When** user starts typing in the city search input field;
> - **Then** user should receive a list of cities (suggestions) that match what is typed.

Scenario 1.3 - User can select a city from the suggested list.

> - **Given** user was typing a specific city (ex: Berlin) in the city search input field
> - **And** the list of suggested cities is showing
> - **When** user selects a city (ex: Berlin, Germany) from the suggestion list;
> - **Then** user interface should be changed to that city (ex: Berlin, Germany)
> - **And** user should receive a list of upcoming events in that city.

CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

Although I had already written user stories for different projects in the past, it was my first time writing test scenarios using the Gherkin syntax. It wasn't in itself difficult, but I had to think about it and iterate my initial Gherkin test scenarios a few times to make sure the following steps of this project would be based on a clear and workable frame.

03

Creating a Google OAuth consumer to use Google Calendar API and **setting up** AWS Lambda

Skills used

Research
Problem solving
Code writing
Debugging

WHY WAS THIS STEP IMPORTANT

The Google Calendar API used to retrieve event data for Meet app is a protected API, which means it can only be used by authenticated apps (apps that have a valid token issued by the API provider). Therefore, it was necessary to implement in the app a logic for an authentication process (using Google OAuth) so that users using the app for the first time can log in via their Google account and grants consent to the app, which then grant a token to it and allow the app to use that token (on behalf of the user) to access protected resources from the Google Calendar API.

Setting up AWS Lambda and eventually using it (over the next steps to host serverless functions) added an additional layer of security and control to the process of accessing the data from the Google API, thus justifying implementing it.

WHAT WAS THE GOAL

Creating an OAuth consumer for Meet app by:

- Creating my project into Google’s API console
- Enabling Google Calendar API
- Setting up my credentials
- Setting up the OAuth consent screen for Meet app
- Adding test users so that Meet app can be used for testing purposes while under development.

Setting up AWS Lambda (using the Serverless Toolkit):

- Installing Serverless framework globally on my computer using npm
- Creating a serverless service in my project directory
- Configuring and getting my credentials from the AWS Management console
- Configuring my AWS credentials in the Serverless framework previously installed on my system (and thus accessible by Meet app since it was installed globally).

CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

I was in my first experimentations with both Google OAuth and AWS. I therefore had to do some extra reading to understand the functioning behind those two new things at first, but I was able to use several solutions by myself to assimilate everything and eventually deliver the all expected requirements. Some of the solutions I used to face the challenges :

- Doing several researches on various platforms
- Read the official Google and AWS documentation
- Carrying tests locally to understand what works well and what does not work well, and adapt my codes consequently

04

Writing serverless functions for authorization process

Skills used

Research
Problem solving
Code writing

WHY WAS THIS STEP IMPORTANT

This step was important to complete the whole authorization process for Meet app and ensure that users would be able to connect to the Google Calendar API by way of an access token.

The creation of three different serverless functions in this step allowed the sound interaction between users, the app and Google OAuth for a simple and efficient authorization process.

WHAT WAS THE GOAL

Creating serverless functions using AWS Lambda and the Serverless Toolkit to finalize the authorization process, as well as setting up the backend infrastructure for Meet app.

More precisely, creating, deploying to AWS Lambda and testing a :

- **getAuthURL function** - when this function is invoked, an authorization URL is returned by Google OAuth, and Google displays a consent screen to users so they can authorize / grant consent to Meet app via an authorization code.
- **getAccessToken function** - invoked if users approves the consent screen. The authorization code received from Google OAuth after users grant consent to Meet app is passed to getAccessToken, which then get a temporary access token.
- **getCalendarEvents function** - function used by Meet app to request calendar events (by attaching the access token received previously to the request - and Google returning the requested data if it determines that the request and the token are valid).

05

Creating unit tests (TDD) and inserting mock-data in the project

Skills used

Research
Problem solving
Code writing
Debugging

WHY WAS THIS STEP IMPORTANT

Creating and writing unit tests is an important process to do at the beginnings of a project when using the TDD technique (needs to be done before implementing any logic), since it informs the way in which the code for each key feature of the product will be written afterwards.

Inserting mock-data was necessary as well to make sure some of the written tests could be carried out correctly.

WHAT WAS THE GOAL

Creating and writing unit tests for Meet app first three key features** scenarios and testing them out using Jest and React Testing Library.

**In Meet app, they were six key features to implement. The latter three features of the app included functionalities related to progressive web apps and data visualization, which were worked on later in the development process, hence why they haven't been considered in this step and the following one.

WHAT WAS THE GOAL (SUITE)

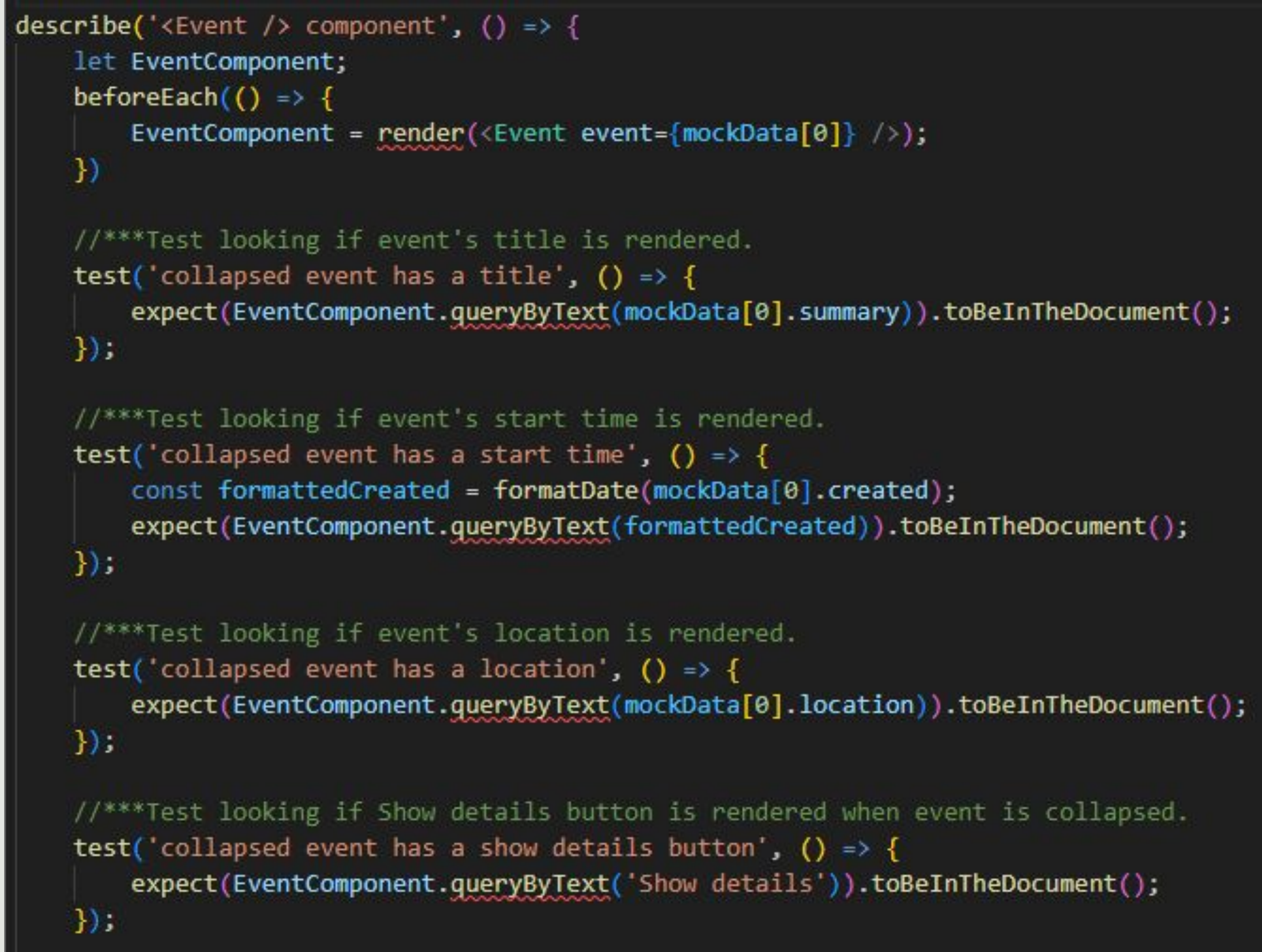
Creating a mock-data.js file in my project directory and inserting into it mock-data that mirrors the real data from the Google Calendar API to carry out my tests (using mock-data is a great way to simplify code during the testing phase rather than using real, full and complete data).

- To do so, I had to know how the Google Calendar API events data was structured, so I went to the official Google Calendar API documentation. I then extracted some mock-data provided on the website and copied and pasted those data into my newly created mock-data.js file to use them later for my tests.

Writing the actual implementation code for each of the first three app's key features scenarios being tested (to make their tests pass).

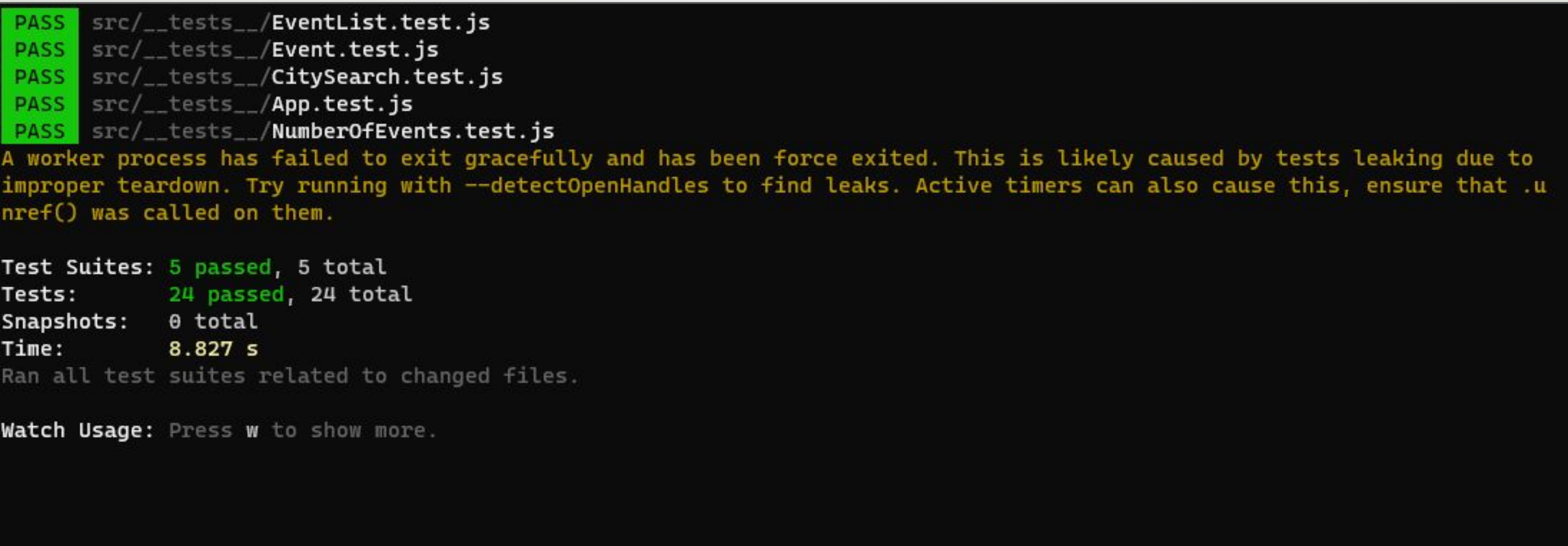
- At the end of this step, the first three app's features and related scenarios were tested and coded using the test-driven development approach. This ensured that each individual piece of these features tested worked the way it should independently and on its own, thus allowing to later create more integrated tests that involve interaction and relation between the features to check how they work together (rather than only on their own). Those are integration tests and are presented in the next step.

EXAMPLES OF UNIT TESTS



```
describe('<Event /> component', () => {
    let EventComponent;
    beforeEach(() => {
        EventComponent = render(<Event event={mockData[0]} />);
    })

    //***Test looking if event's title is rendered.
    test('collapsed event has a title', () => {
        expect(EventComponent.queryByText(mockData[0].summary)).toBeInTheDocument();
    });

    //***Test looking if event's start time is rendered.
    test('collapsed event has a start time', () => {
        const formattedCreated = formatDate(mockData[0].created);
        expect(EventComponent.queryByText(formattedCreated)).toBeInTheDocument();
    });

    //***Test looking if event's location is rendered.
    test('collapsed event has a location', () => {
        expect(EventComponent.queryByText(mockData[0].location)).toBeInTheDocument();
    });

    //***Test looking if Show details button is rendered when event is collapsed.
    test('collapsed event has a show details button', () => {
        expect(EventComponent.queryByText('Show details')).toBeInTheDocument();
    });
```

EXAMPLES OF UNIT TESTS ALL PASSING



```
 PASS  src/__tests__/EventList.test.js
 PASS  src/__tests__/Event.test.js
 PASS  src/__tests__/CitySearch.test.js
 PASS  src/__tests__/App.test.js
 PASS  src/__tests__/NumberOfEvents.test.js
A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests leaking due to
improper teardown. Try running with --detectOpenHandles to find leaks. Active timers can also cause this, ensure that .u
nref() was called on them.

Test Suites: 5 passed, 5 total
Tests:       24 passed, 24 total
Snapshots:   0 total
Time:        8.827 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

06

Creating integration tests (TDD), **connecting** the app to the real data from the Google Calendar API and **styling** the app

Skills used

Creative thinking
Research
Code writing

WHY WAS THIS STEP IMPORTANT

Creating integration tests was important to ensure that each of the app's individual parts (previously tested with unit tests) continues to behave as expected and work well when interacting with every other part (in React, the parts that need to be tested are the components, which means that integration testing here involved creating tests and writing code for cross-component functionalities).

Also, since most of the first key features were finished to be developed at this point, it was a good moment to integrate the Google Calendar API. This step was essential since without it (and by remaining only with mock-data), the app would have been useless.

WHAT WAS THE GOAL

Creating and writing integration tests for Meet app first three features scenarios and testing them out (and expect them to fail at first since no code has been written to make them pass).

Writing the actual implementation code for the features scenarios being tested to make their tests pass.

Integrating Google Calendar API into the app so that it can request and display real data (rather than the mock-data used up until this point).

Applying CSS on the app to give it a responsive, appealing and professional UI visual for the best user experience possible.

Deploying the app to Github Pages to see the results of the work done since its first deployment:

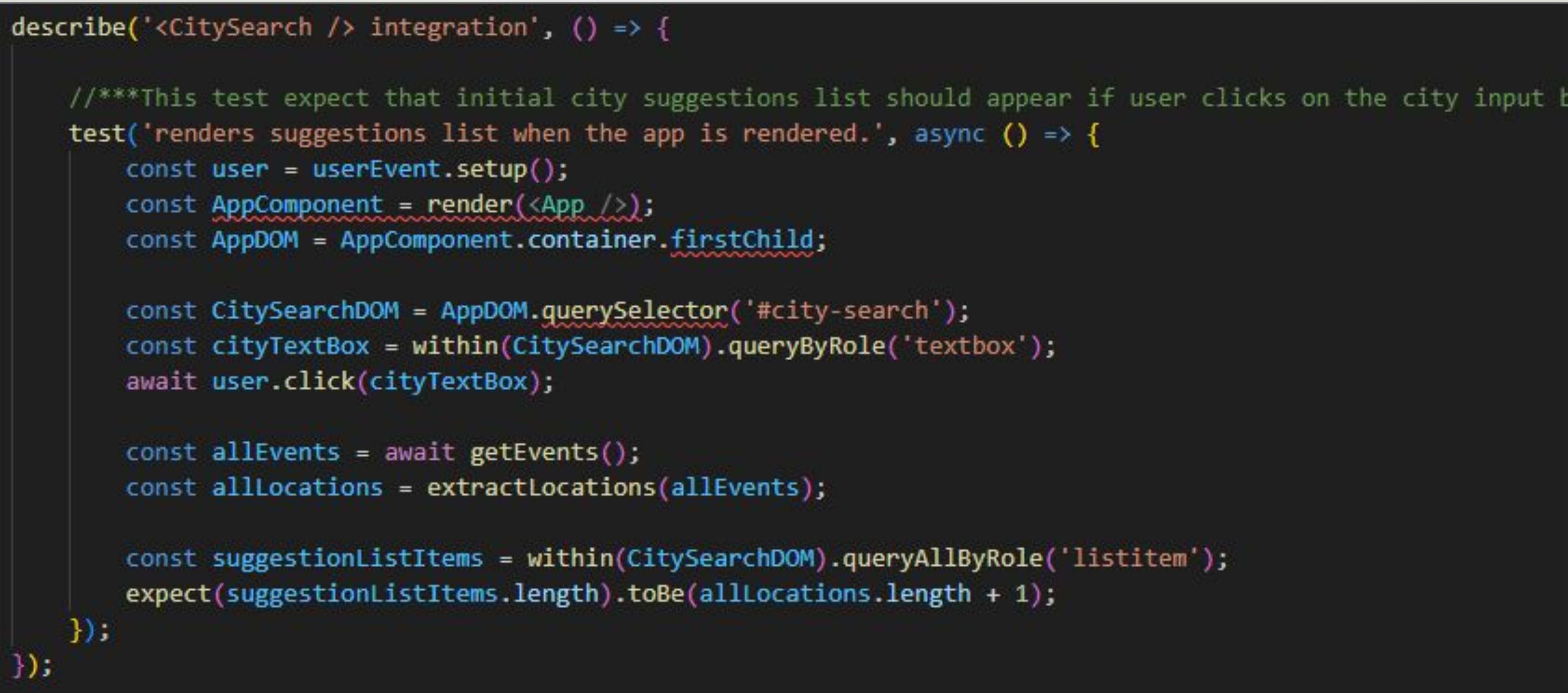
- Data from the Google Calendar API are now showing up on the newly updated UI visual
- Possible to start making more realistic user interactions on the UI (e.g.: searching for a specific city and see the results rendered on the screen accordingly).

✓ github-pages

DECISIONS MADE

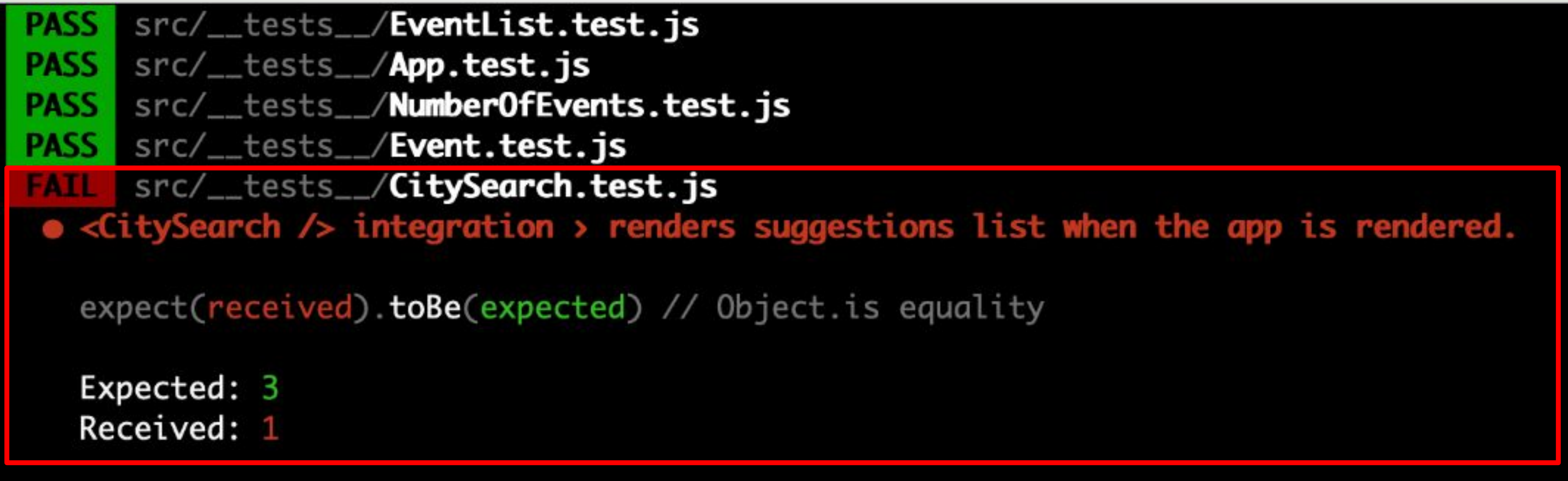
I choose how to design the visuals and the content structure on the UI. I made all my decisions based on how I thought users would like to interact with Meet app. I've implemented, inspired by other well-designed web apps, a consistent styling across each view for a professional looking app.

EXAMPLES OF INTEGRATION TEST



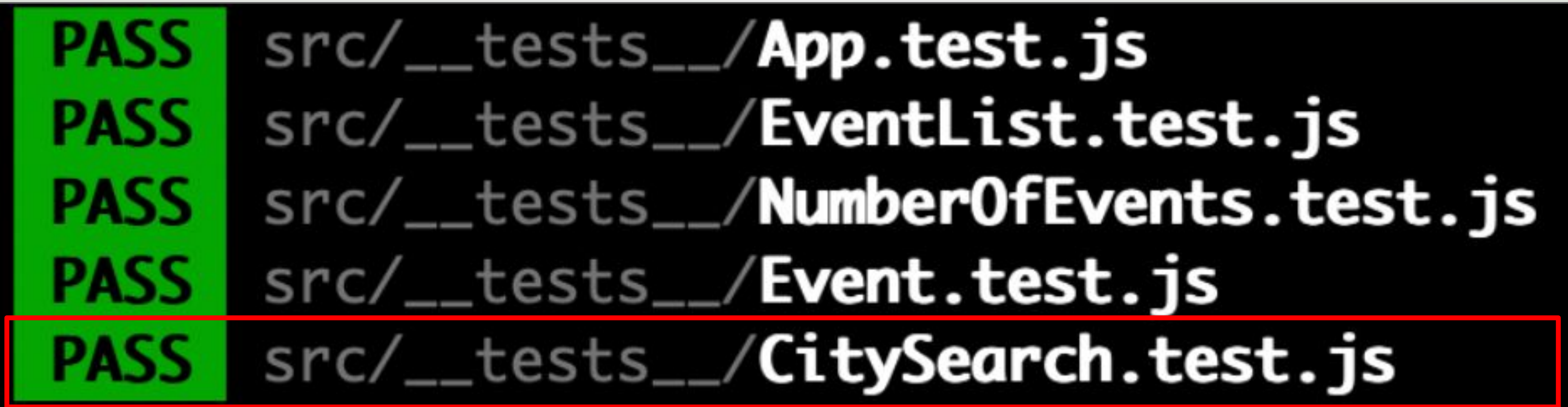
```
describe('<CitySearch /> integration', () => {

    //***This test expect that initial city suggestions list should appear if user clicks on the city input b
    test('renders suggestions list when the app is rendered.', async () => {
        const user = userEvent.setup();
        const AppComponent = render(<App />);
        const AppDOM = AppComponent.container.firstChild;

        const CitySearchDOM = AppDOM.querySelector('#city-search');
        const cityTextBox = within(CitySearchDOM).queryByRole('textbox');
        await user.click(cityTextBox);

        const allEvents = await getEvents();
        const allLocations = extractLocations(allEvents);

        const suggestionListItems = within(CitySearchDOM).queryAllByRole('listitem');
        expect(suggestionListItems.length).toBe(allLocations.length + 1);
    });
});
```

INTEGRATION TEST FAILED AT FIRST (EXPECTED SINCE CODE HAVE NOT BEEN UPDATED TO MAKE IT WORK UP TO THIS POINT)



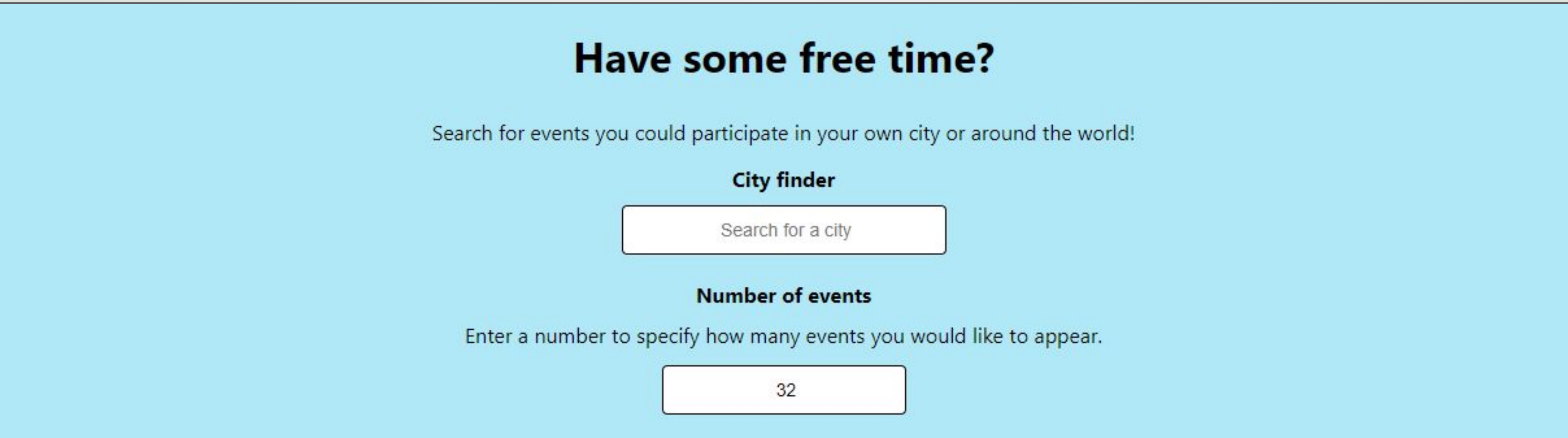
```
PASS  src/__tests__/EventList.test.js
PASS  src/__tests__/App.test.js
PASS  src/__tests__/NumberOfEvents.test.js
PASS  src/__tests__/Event.test.js
FAIL  src/__tests__/CitySearch.test.js
  ● <CitySearch /> integration › renders suggestions list when the app is rendered.

    expect(received).toBe(expected) // Object.is equality

    Expected: 3
    Received: 1
```

INTEGRATION TEST PASSED (EXPECTED SINCE CODE HAVE BEEN UPDATED TO MAKE IT WORK UP TO THIS POINT)



```
PASS src/__tests__/App.test.js
PASS src/__tests__/EventList.test.js
PASS src/__tests__/NumberOfEvents.test.js
PASS src/__tests__/Event.test.js
PASS src/__tests__/CitySearch.test.js
```

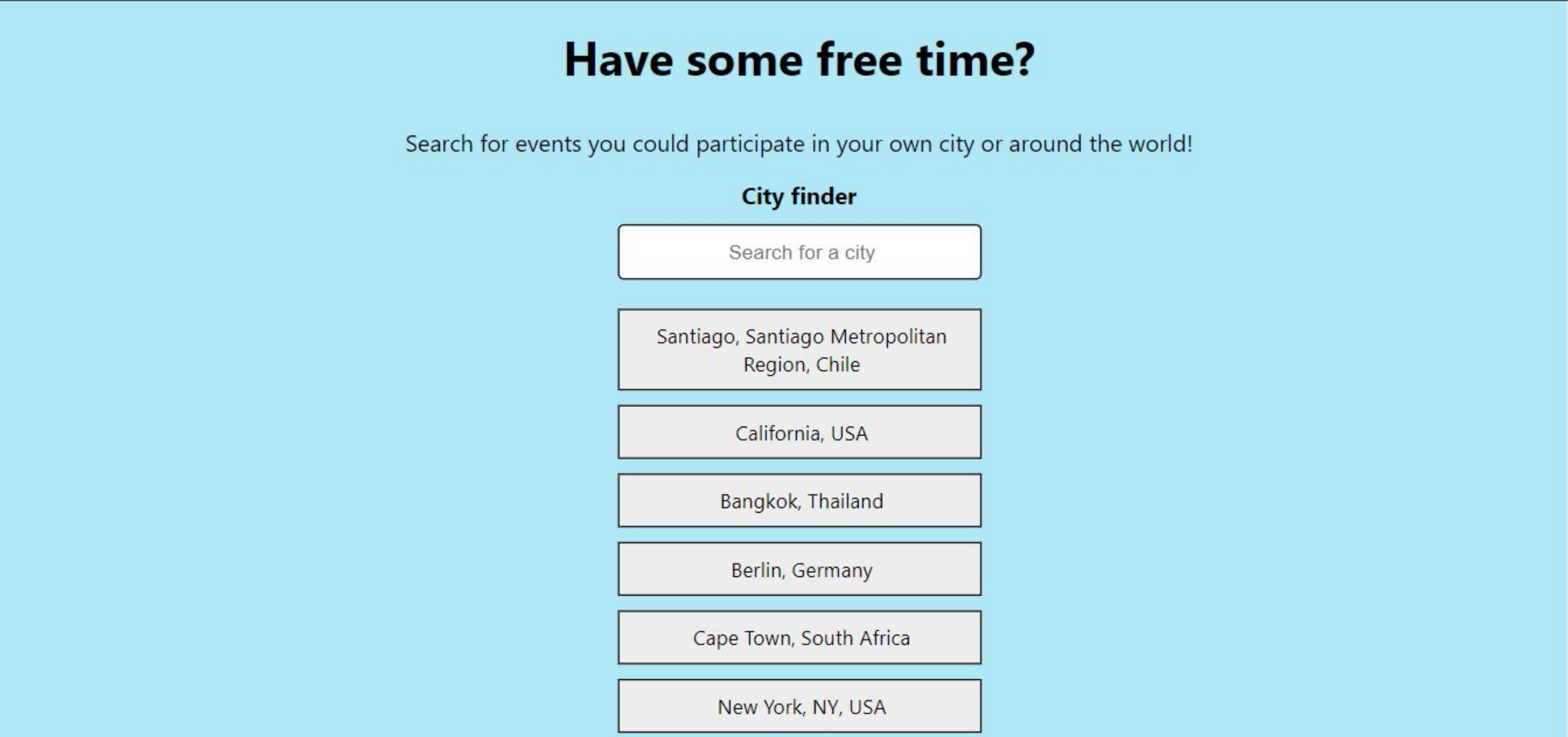
EXAMPLE OF STYLING USING CSS (TOP SECTION FOR USER INPUTS)


Have some free time?

Search for events you could participate in your own city or around the world!

Enter a number to specify how many events you would like to appear.

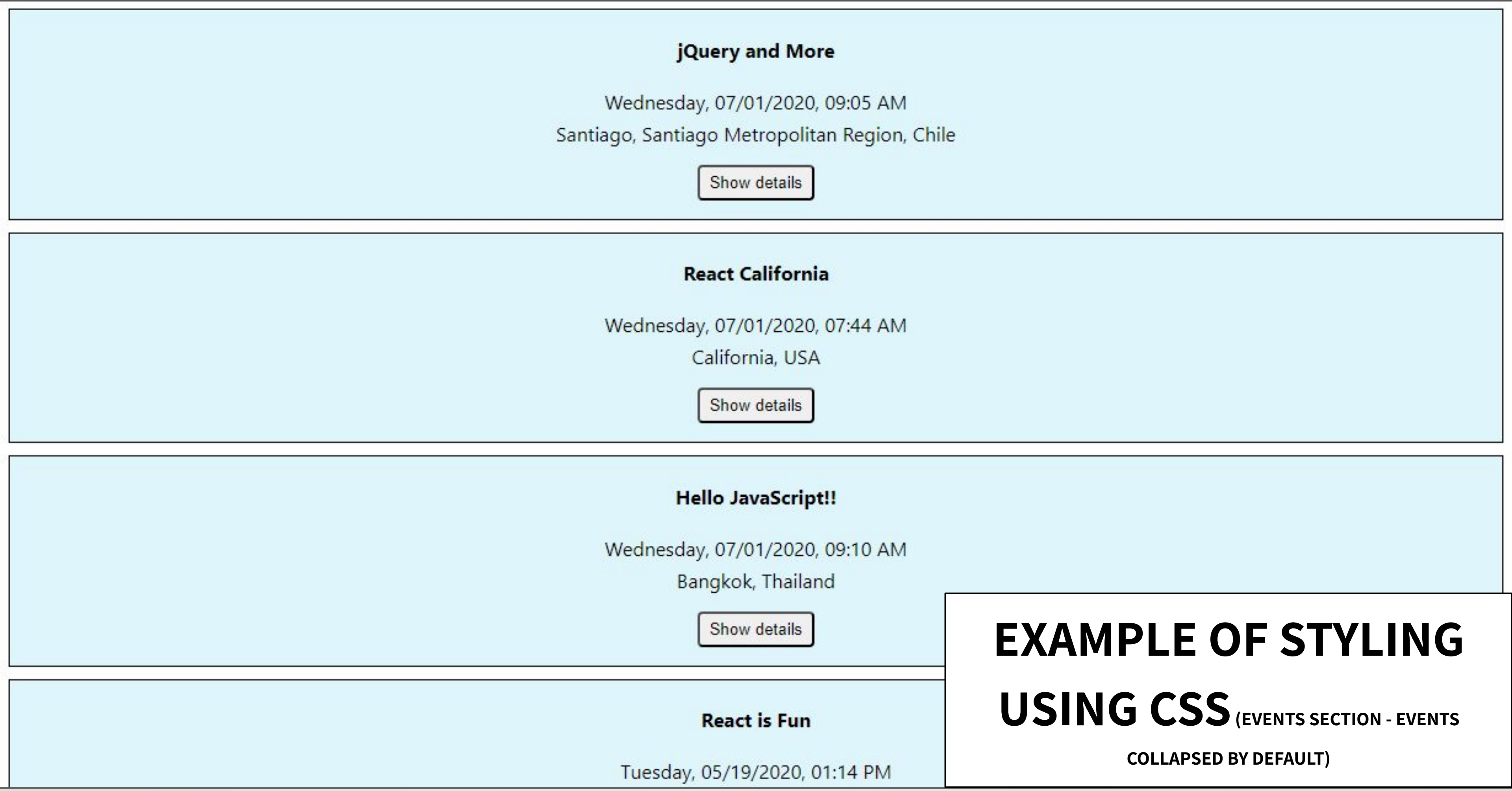
EXAMPLE OF STYLING USING CSS (TOP SECTION FOR USERS INPUTS)



Have some free time?

Search for events you could participate in your own city or around the world!

City finder



jQuery and More

Wednesday, 07/01/2020, 09:05 AM

Santiago, Santiago Metropolitan Region, Chile

React California

Wednesday, 07/01/2020, 07:44 AM

California, USA

Hello JavaScript!!

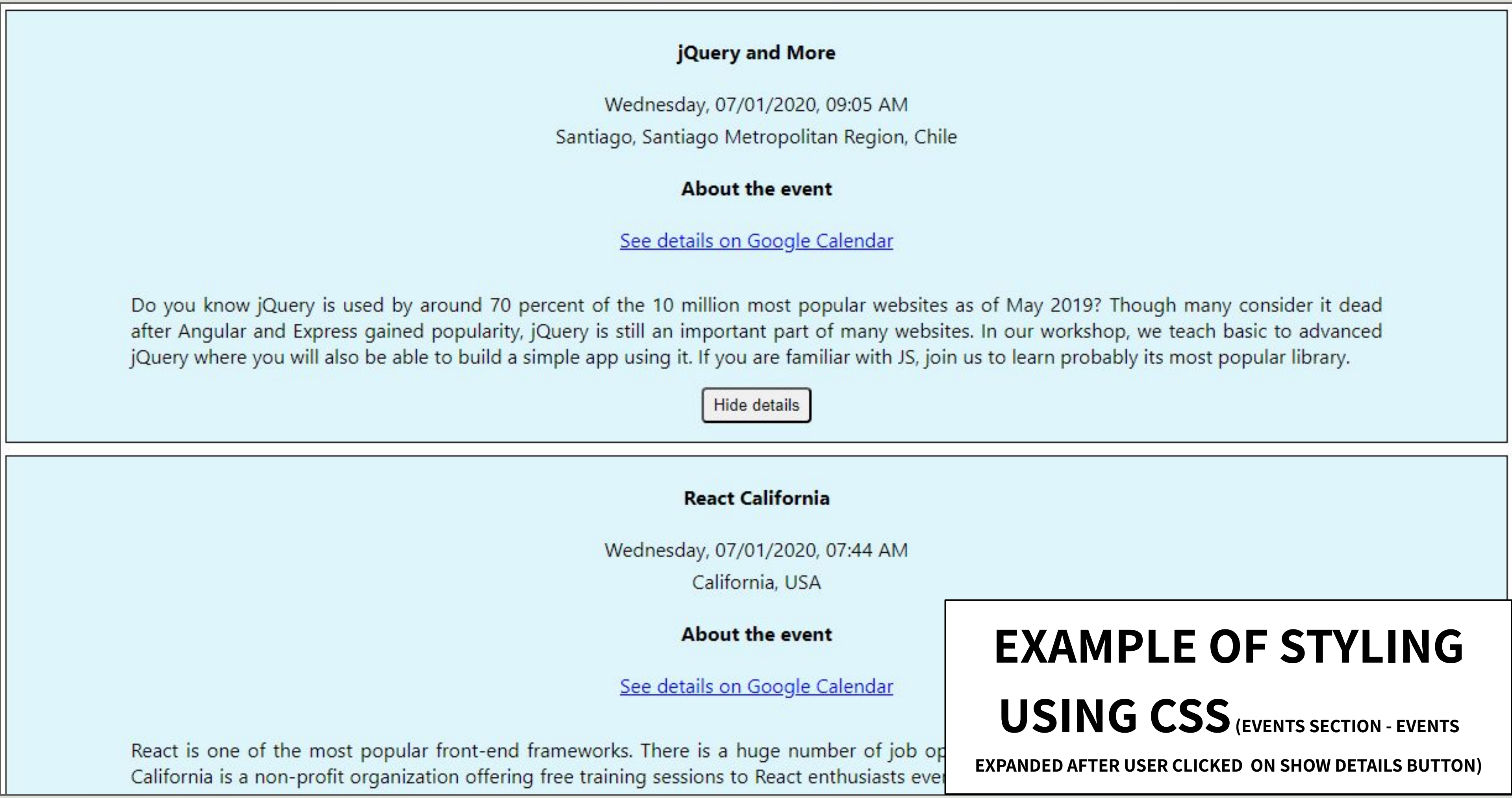
Wednesday, 07/01/2020, 09:10 AM

Bangkok, Thailand

React is Fun

Tuesday, 05/19/2020, 01:14 PM

EXAMPLE OF STYLING USING CSS (EVENTS SECTION - EVENTS COLLAPSED BY DEFAULT)



jQuery and More

Wednesday, 07/01/2020, 09:05 AM

Santiago, Santiago Metropolitan Region, Chile

About the event

See details on Google Calendar

Do you know jQuery is used by around 70 percent of the 10 million most popular websites as of May 2019? Though many consider it dead after Angular and Express gained popularity, jQuery is still an important part of many websites. In our workshop, we teach basic to advanced jQuery where you will also be able to build a simple app using it. If you are familiar with JS, join us to learn probably its most popular library.

React California

Wednesday, 07/01/2020, 07:44 AM

California, USA

About the event

See details on Google Calendar

React is one of the most popular front-end frameworks. There is a huge number of job op
California is a non-profit organization offering free training sessions to React enthusiasts eve

EXAMPLE OF STYLING USING CSS (EVENTS SECTION - EVENTS EXPANDED AFTER USER CLICKED ON SHOW DETAILS BUTTON)

```
filterEventsByCity.feature
filterEventsByCity.test

showHideAnEventsDetails.feature
showHideAnEventsDetails.test.js

specifyNumberOfEvents.feature
specifyNumberOfEvents.test.js

EndToEnd.test.js
```

Writing acceptance tests and end-to-end tests (BDD)

Skills used

Research
Problem solving
Code writing
Debugging

WHY WAS THIS STEP IMPORTANT

While TDD takes a developer-based approach, BDD takes a non-developer approach. Tests written following a BDD approach are easier to understand for all the other / non-developers stakeholders in a project. This thus ensures that everyone involved can verify whether the final app matches what was initially expected.

It was therefore useful and relevant to use TDD and BDD in sync because all these types of testing play a different role in the development process, and grant for a better and more complete testing approach overall.

WHAT WAS THE GOAL

Creating and writing acceptance tests for the first three app's features by way of acceptance criteria (which are descriptions of expected behavior to be tested) using Cucumber (a BDD-style acceptance testing JavaScript framework).

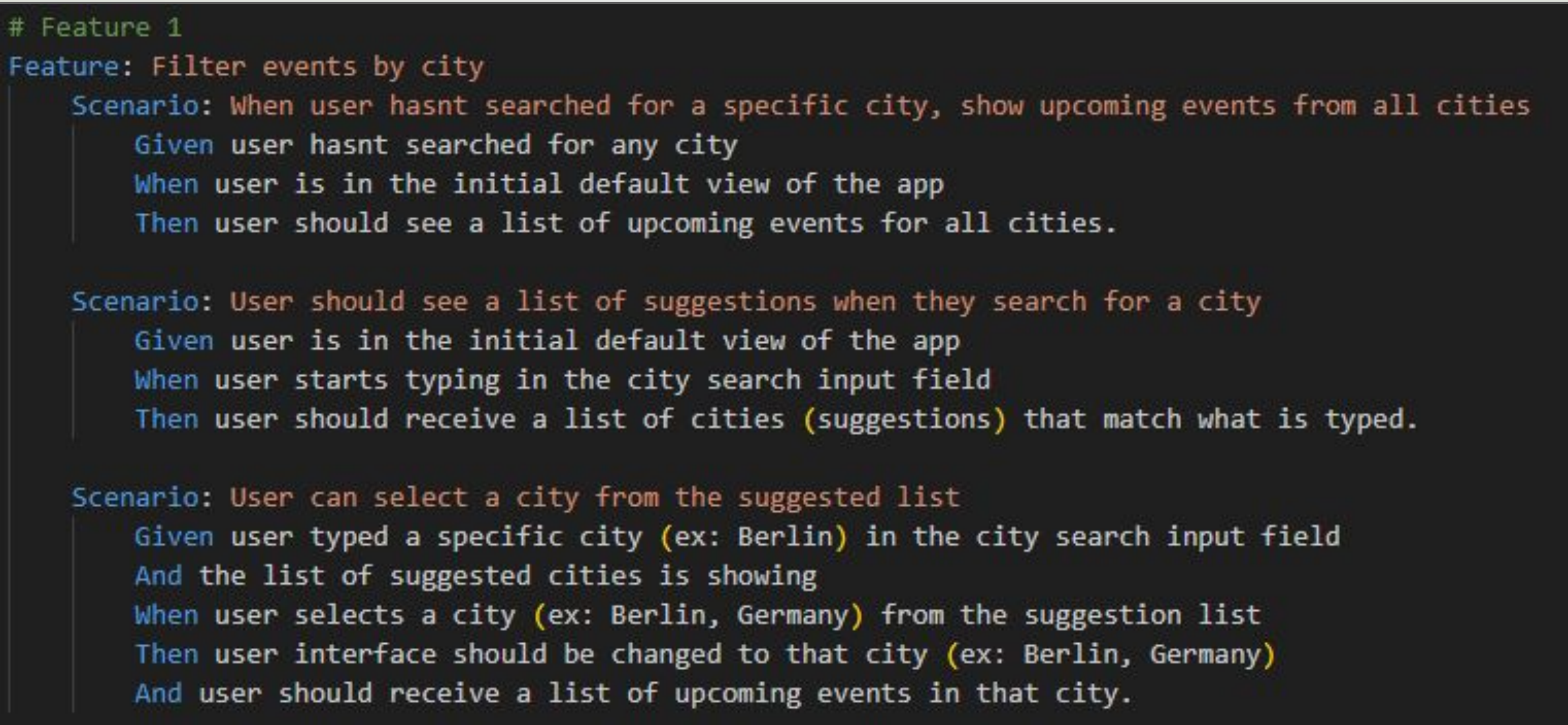
- Tests in Cucumber need to be written in Gherkin's Given-When-Then syntax. This syntax had already been written back in step 2 for all the app's features scenarios.

- I therefore created a new *features* folder in my project directory to hold each of my new Gherkin file (e.g.: filterEventByCity.feature) for each of the first three app's features and related scenarios. I then pasted into each Gherkin file a specific feature, his related scenarios and the related Gherkin syntax for each scenario (see example on the next slide).

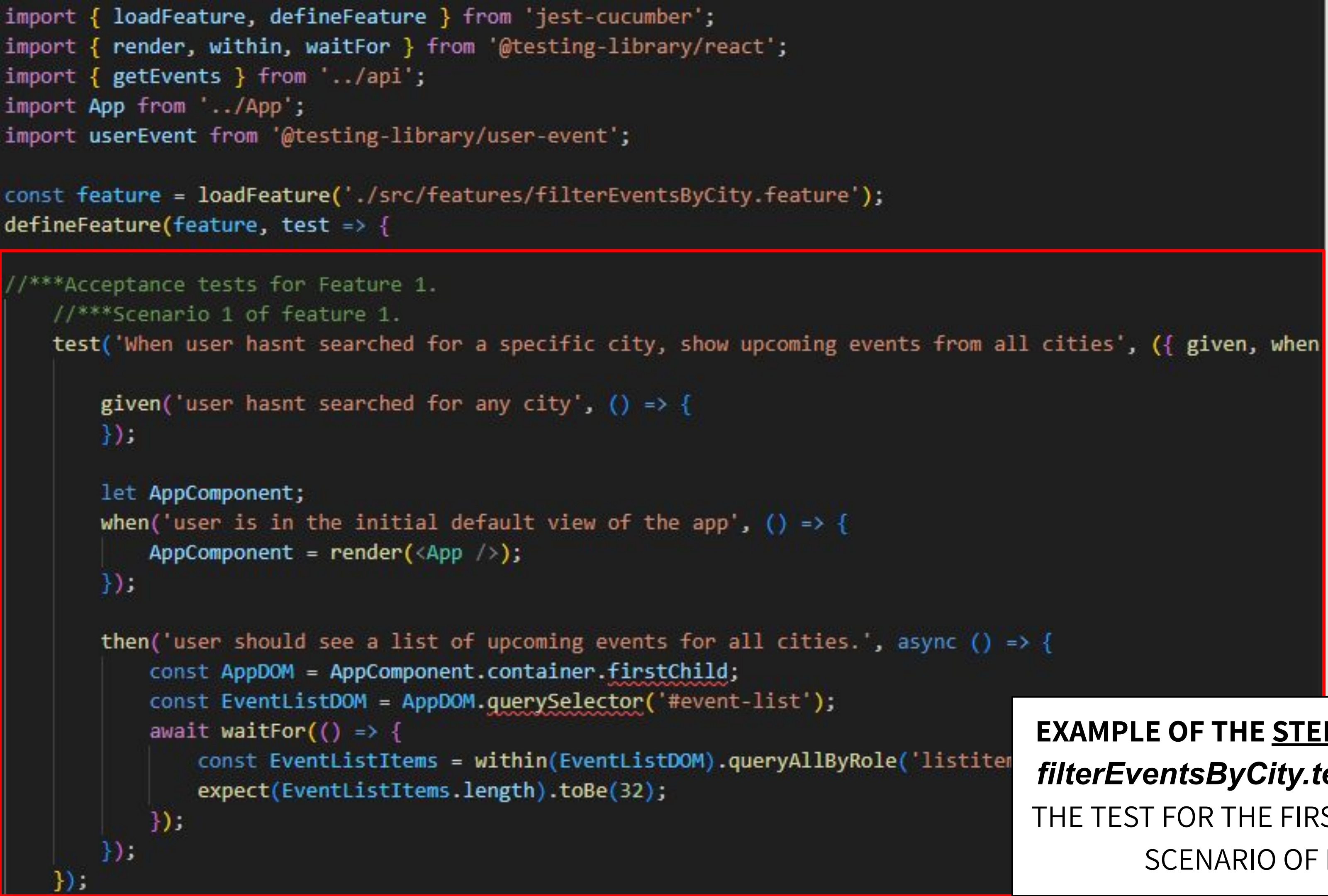
- I also created the necessary / complementary step definitions files (e.g: filterEventsByCity.test.js) to connect each piece of the Gherkin-based scenarios from the Gherkin files to the actual codes that would test each Gherkin syntax from each scenario (see example over the next slides). Step definition files are written in JavaScript and run by Jest.

Writing the code for the acceptance tests to make them pass.

EXAMPLE OF THE <u>GHERKIN FILE</u> *filterEventsByCity.feature* CONTAINING ONE FEATURE (FIRST FEATURE OF THE APP), THE 3 RELATED SCENARIOS AND THE GHERKIN SYNTAX FOR EACH SCENARIO)

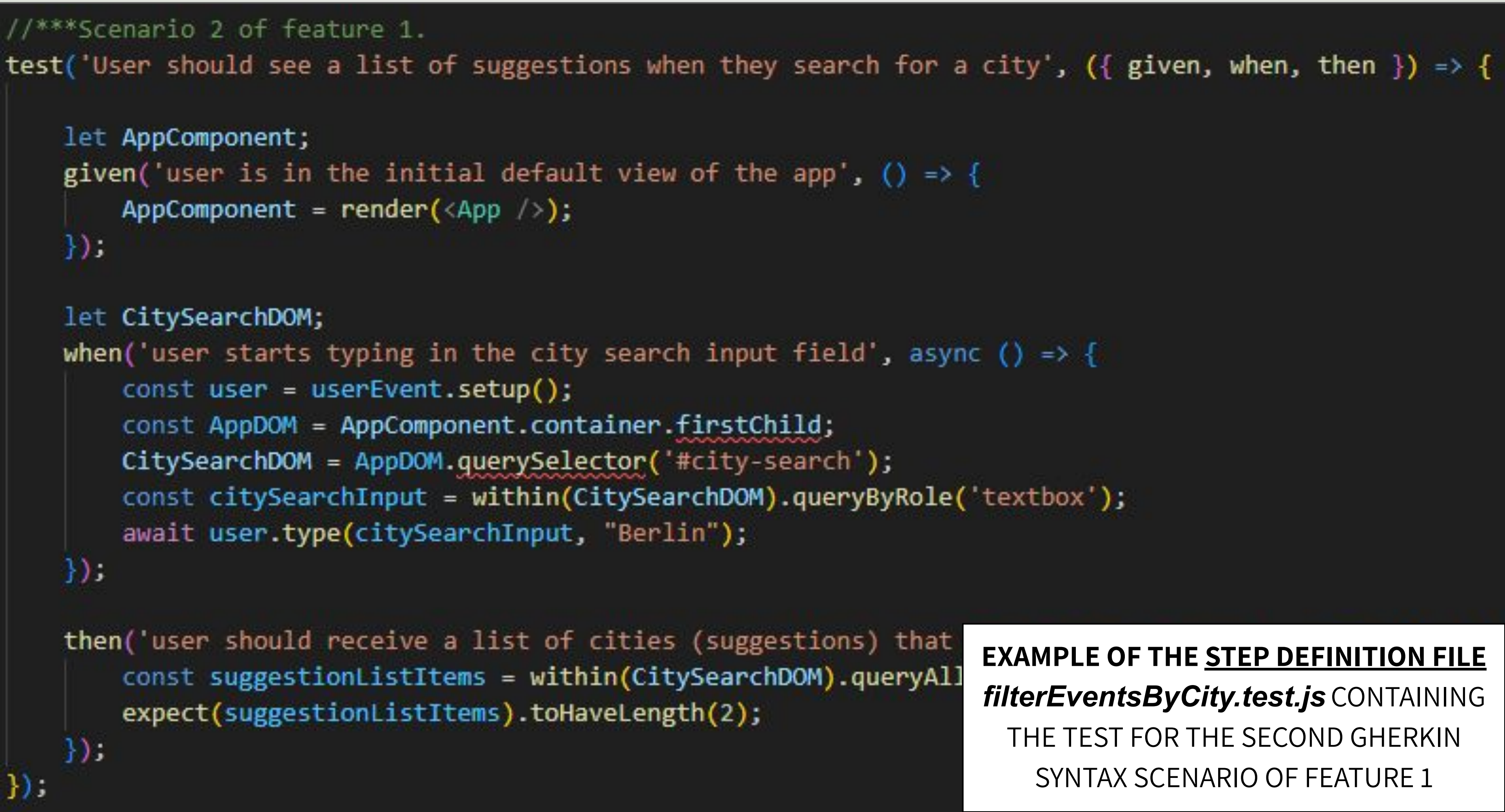


```
# Feature 1
Feature: Filter events by city
    Scenario: When user hasnt searched for a specific city, show upcoming events from all cities
        Given user hasnt searched for any city
        When user is in the initial default view of the app
        Then user should see a list of upcoming events for all cities.

    Scenario: User should see a list of suggestions when they search for a city
        Given user is in the initial default view of the app
        When user starts typing in the city search input field
        Then user should receive a list of cities (suggestions) that match what is typed.

    Scenario: User can select a city from the suggested list
        Given user typed a specific city (ex: Berlin) in the city search input field
        And the list of suggested cities is showing
        When user selects a city (ex: Berlin, Germany) from the suggestion list
        Then user interface should be changed to that city (ex: Berlin, Germany)
        And user should receive a list of upcoming events in that city.
```



```
import { loadFeature, defineFeature } from 'jest-cucumber';
import { render, within, waitFor } from '@testing-library/react';
import { getEvents } from '../api';
import App from '../App';
import userEvent from '@testing-library/user-event';

const feature = loadFeature('./src/features/filterEventsByCity.feature');
defineFeature(feature, test => {

//***Acceptance tests for Feature 1.
    //***Scenario 1 of feature 1.
    test('When user hasnt searched for a specific city, show upcoming events from all cities', ({ given, when

        given('user hasnt searched for any city', () => {
        });

        let AppComponent;
        when('user is in the initial default view of the app', () => {
            AppComponent = render(<App />);
        });

        then('user should see a list of upcoming events for all cities.', async () => {
            const AppDOM = AppComponent.container.firstChild;
            const EventListDOM = AppDOM.querySelector('#event-list');
            await waitFor(() => {
                const EventListItems = within(EventListDOM).queryAllByRole('listitem
                expect(EventListItems.length).toBe(32);
            });
        });
    });
```

EXAMPLE OF THE <u>STEP DEFINITION FILE</u>
filterEventsByCity.test.js CONTAINING
THE TEST FOR THE FIRST GHERKIN SYNTAX
SCENARIO OF FEATURE 1



```
//***Scenario 2 of feature 1.
test('User should see a list of suggestions when they search for a city', ({ given, when, then }) => {

    let AppComponent;
    given('user is in the initial default view of the app', () => {
        AppComponent = render(<App />);
    });

    let CitySearchDOM;
    when('user starts typing in the city search input field', async () => {
        const user = userEvent.setup();
        const AppDOM = AppComponent.container.firstChild;
        CitySearchDOM = AppDOM.querySelector('#city-search');
        const citySearchInput = within(CitySearchDOM).queryByRole('textbox');
        await user.type(citySearchInput, "Berlin");
    });

    then('user should receive a list of cities (suggestions) that
        const suggestionListItems = within(CitySearchDOM).queryAll
        expect(suggestionListItems).toHaveLength(2);
    });
});
```

EXAMPLE OF THE <u>STEP DEFINITION FILE</u> ***filterEventsByCity.test.js*** CONTAINING THE TEST FOR THE SECOND GHERKIN SYNTAX SCENARIO OF FEATURE 1

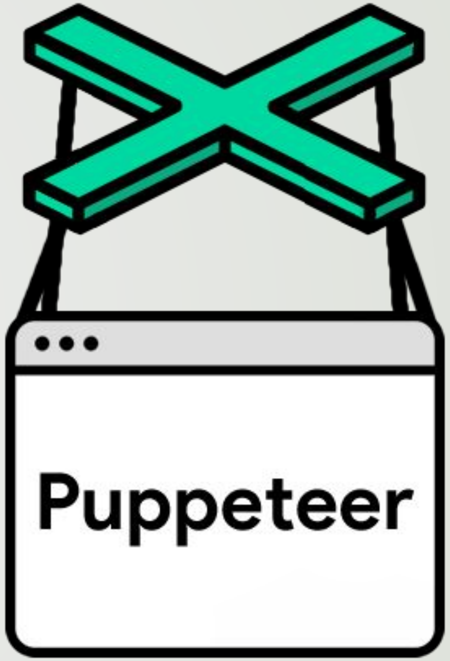
WHAT WAS THE GOAL (SUITE)

Creating and writing end-to-end tests for the first three app's features using Puppeteer, an automated testing framework allowing to simulate user interaction in a browser and see how the app is behaving accordingly.

- This testing approach is quite useful as it tests the entire app, ensuring that it performs as it's supposed from the beginning when the user first opens the app and then interacts with different UI elements (textboxes, buttons, etc.), to the end of the utilization process.

- Using this approach allowed to confirm that each of the UI interactions necessary to navigate and use Meet app worked properly and as expected.

Writing the code for the end-to-end tests to make them pass.



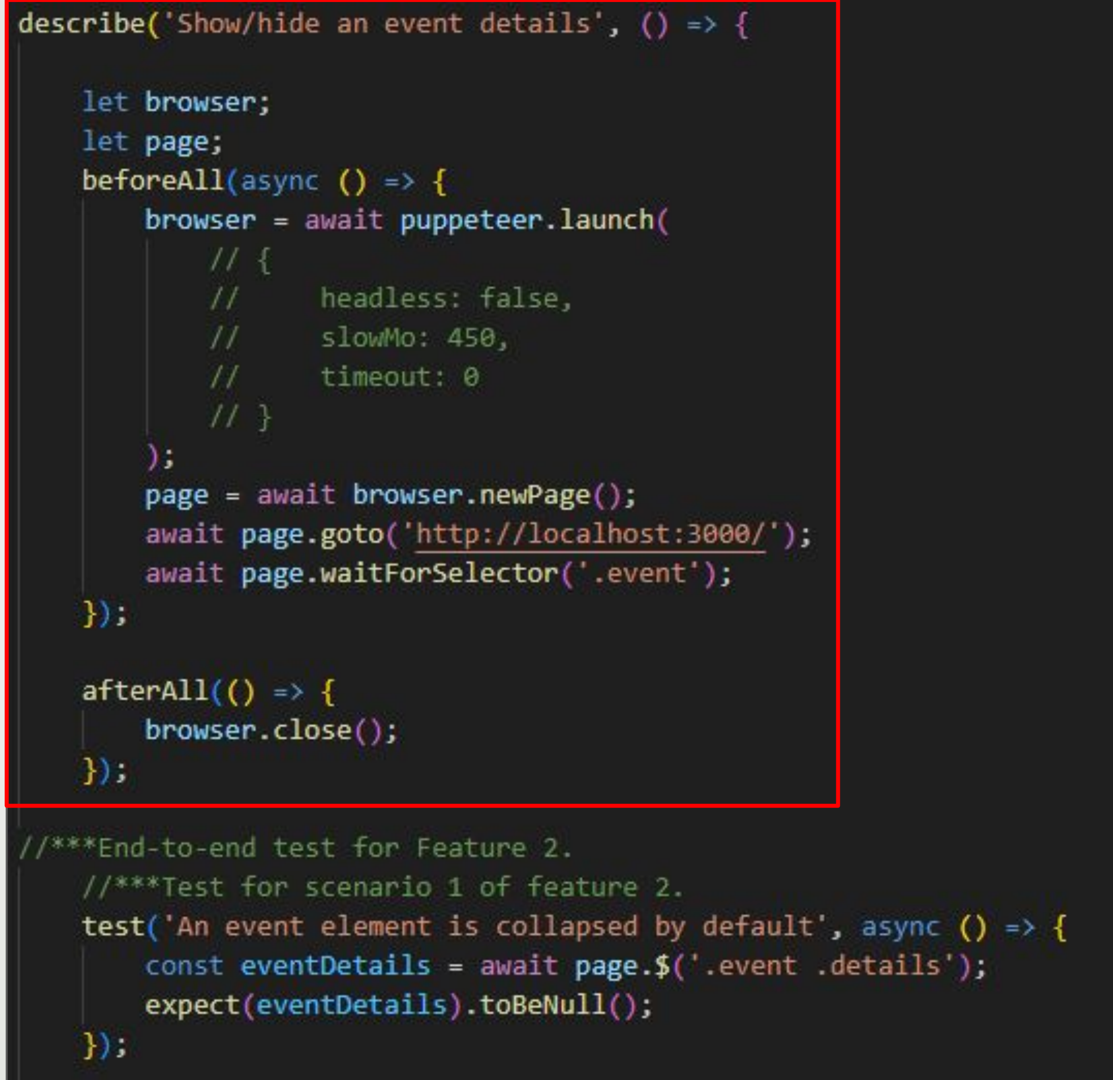
INITIAL LOGIC RUNNING BEFORE ALL END-TO-END TESTS AND AFTER THEY'VE ALL BEEN EXECUTED

This initial logic targeting all tests allows to avoid repeating the same instructions for each end-to-end tests.

Before all end-to-end tests (beforeAll), this code launch a browser environment using Puppeteer, navigates to a web page and waits for a specific element with the class "event" to appear.

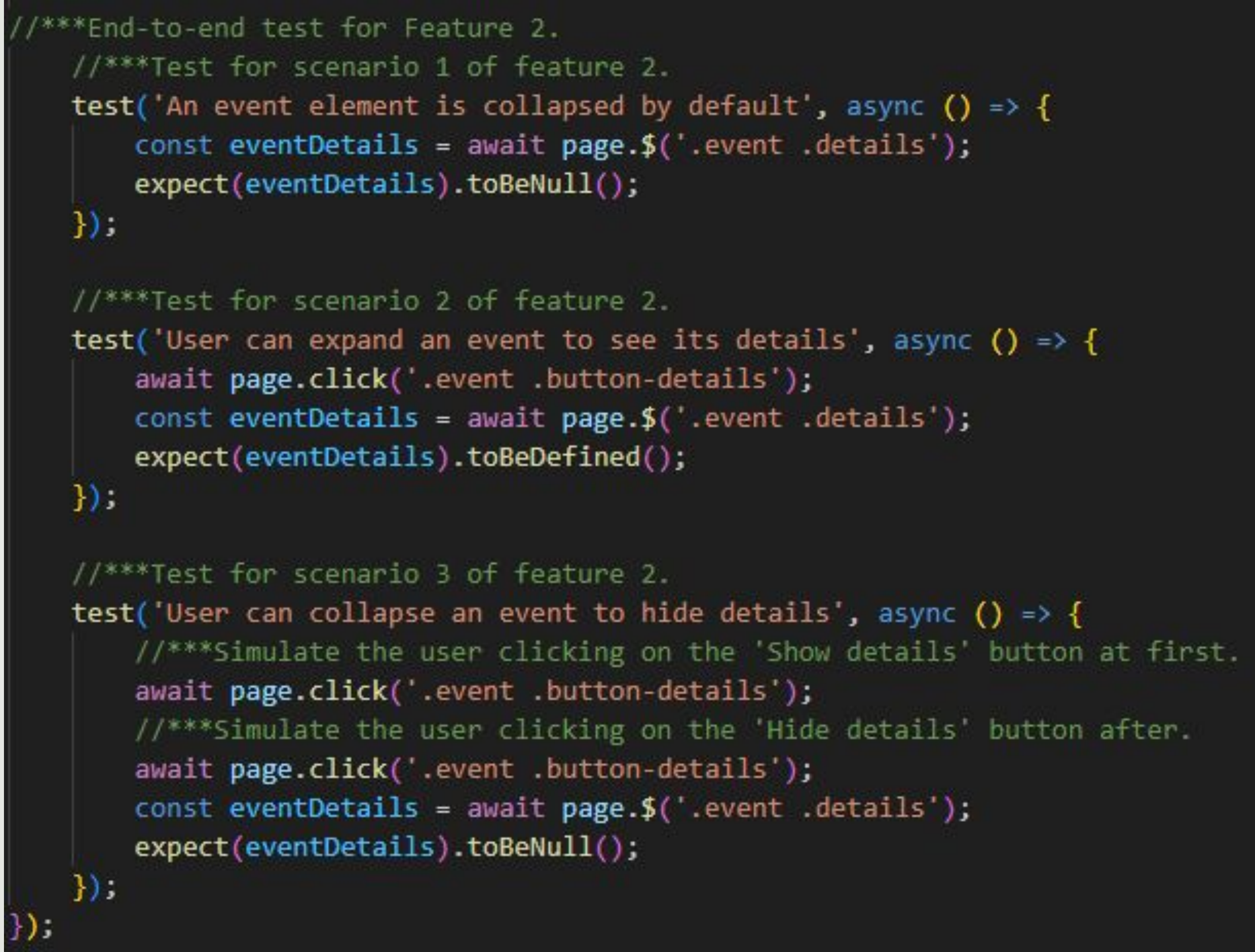
The written end-to-end tests below are then executed.

After all test have been executed (afterAll), the browser is closed.



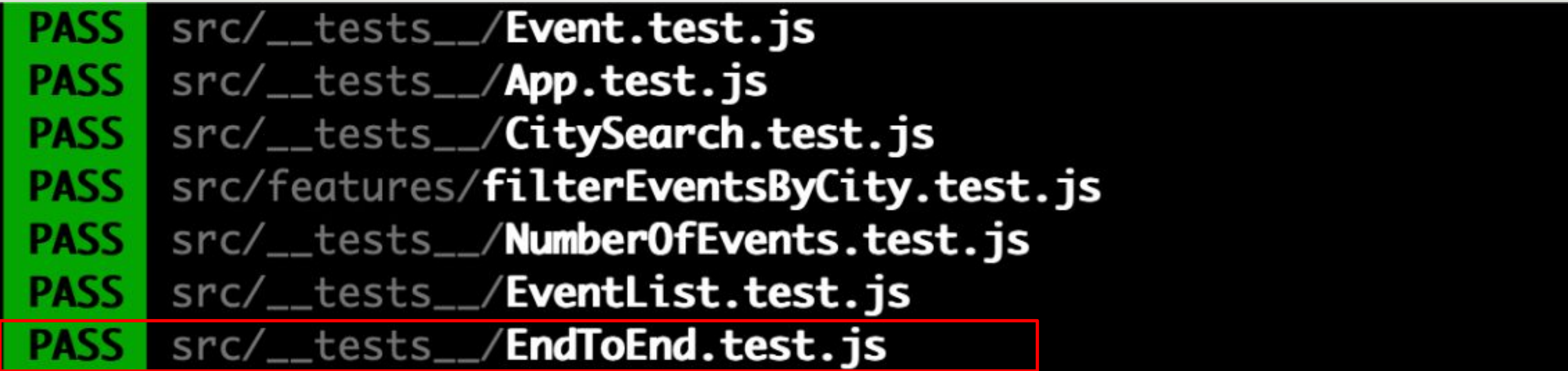
```
describe('Show/hide an event details', () => {

    let browser;
    let page;
    beforeAll(async () => {
        browser = await puppeteer.launch(
            // {
            //      headless: false,
            //      slowMo: 450,
            //      timeout: 0
            // }
        );
        page = await browser.newPage();
        await page.goto('http://localhost:3000/');
        await page.waitForSelector('.event');
    });

    afterAll(() => {
        browser.close();
    });

//***End-to-end test for Feature 2.
    //***Test for scenario 1 of feature 2.
    test('An event element is collapsed by default', async () => {
        const eventDetails = await page.$('.event .details');
        expect(eventDetails).toBeNull();
    });
```

END-TO-END TESTS FOR FEATURE 2, SCENARIO 1, 2 AND 3



```
//***End-to-end test for Feature 2.
    //***Test for scenario 1 of feature 2.
    test('An event element is collapsed by default', async () => {
        const eventDetails = await page.$('.event .details');
        expect(eventDetails).toBeNull();
    });

    //***Test for scenario 2 of feature 2.
    test('User can expand an event to see its details', async () => {
        await page.click('.event .button-details');
        const eventDetails = await page.$('.event .details');
        expect(eventDetails).toBeDefined();
    });

    //***Test for scenario 3 of feature 2.
    test('User can collapse an event to hide details', async () => {
        //***Simulate the user clicking on the 'Show details' button at first.
        await page.click('.event .button-details');
        //***Simulate the user clicking on the 'Hide details' button after.
        await page.click('.event .button-details');
        const eventDetails = await page.$('.event .details');
        expect(eventDetails).toBeNull();
    });
});
```

END-TO-END TESTS PASSED



```
PASS  src/__tests__/Event.test.js
PASS  src/__tests__/App.test.js
PASS  src/__tests__/CitySearch.test.js
PASS  src/features/filterEventsByCity.test.js
PASS  src/__tests__/NumberOfEvents.test.js
PASS  src/__tests__/EventList.test.js
PASS  src/__tests__/EndToEnd.test.js
```

DECISIONS MADE

To get more practice and ensure even more my app's codes were working and running as expected, I've decided to create and write more acceptance and end-to-end tests than initially planned. This enhances my app verification process, as well as giving me a chance to better master these testing techniques.

08

Getting familiar with CI and CD development practices and **integrating** an application performance monitoring (APM) tool for Meet app

Skills used

Critical thinking
Analytical thinking
Synthetic writing

WHY WAS THIS STEP IMPORTANT

Getting familiar with continuous integration (CI) and continuous development (CD) was important since these two practices go hand to hand and allow for faster, up-to-date, efficient, less-resources consuming and easier multi-environments testing (compared to doing tests without these techniques for example). Both methods bring great advantages and are particularly fit for agile environments frequently observed in software / web development.

As for the APM, it allows to track the app's behavior in production mode, making it easier to pinpoint and proactively resolve any issues as they arise, hence highlighting the importance of using such tool.

WHAT WAS THE GOAL

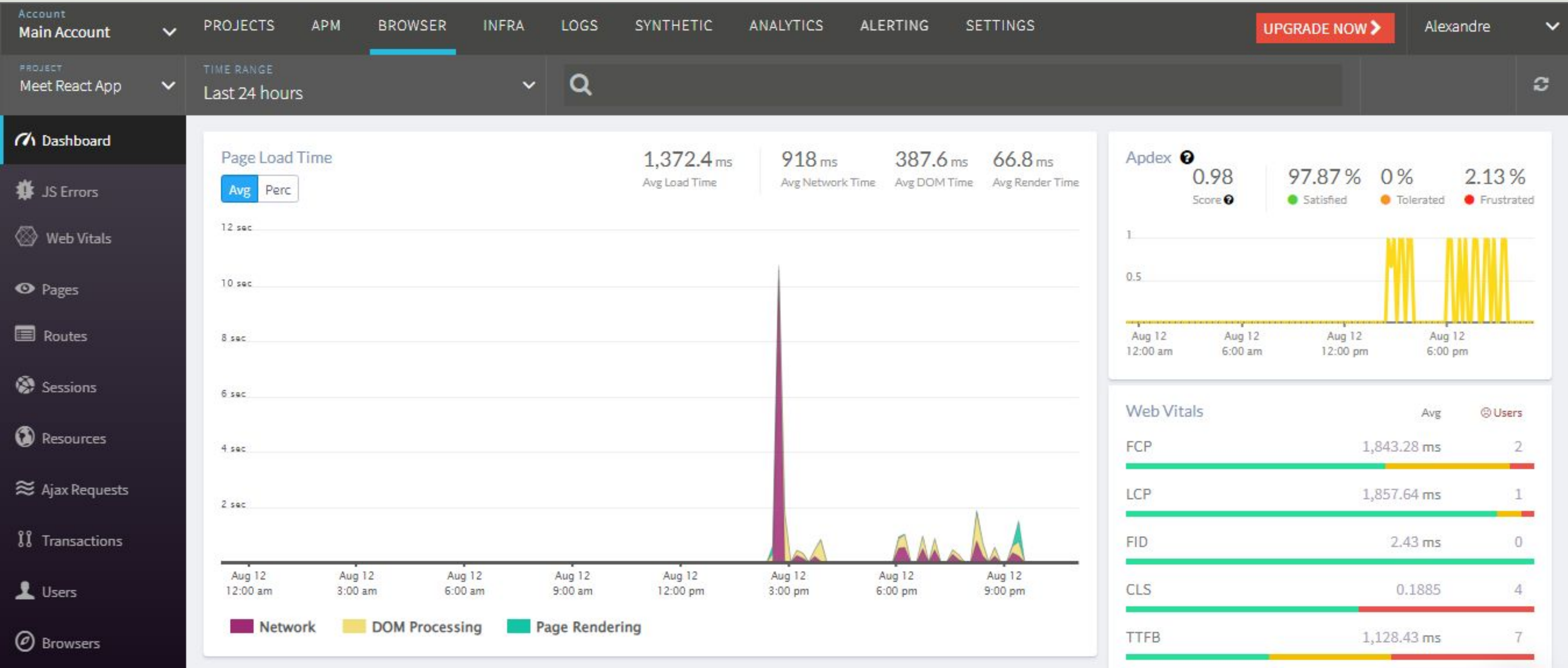
Setting up my project in Atatus (APM tool) to track Meet app's performance. To do so, I asked external testers to test Meet app to generate activities into my Atatus dashboard, which allowed me to better monitor how my app's behaving (performance, errors, page loading time, etc.).

- The next slide show some activities reported in my Atatus dashboard after Meet app have been used by different testers.

Summarizing what CI and CD are and explaining how such practices could be beneficial to the Meet app project.

ATATUS MAIN DASHBOARD VIEW

Indicators showing that Meet app perform very well (page load time has an average of about 1,4 second and the Apdex -a way to measure user satisfaction depending on frontend application response time- has a strong score of 0.98)



PROJECT: Meet React App

TIME RANGE: Last 24 hours

Page Load Time

1,372.4 ms Avg Load Time | 918 ms Avg Network Time | 387.6 ms Avg DOM Time | 66.8 ms Avg Render Time

Apdex

0.98 Score | 97.87 % Satisfied | 0 % Tolerated | 2.13 % Frustrated

Web Vitals

	Avg	Users
FCP	1,843.28 ms	2
LCP	1,857.64 ms	1
FID	2.43 ms	0
CLS	0.1885	4
TTFB	1,128.43 ms	7

09

InfoAlert

ErrorAlert

Implementing alerts using Object-Oriented Programming (OOP) and React class components

Skills used

Code writing

WHY WAS THIS STEP IMPORTANT

Understanding OOP and learning how it can be used (in React for instance) was important since this programming paradigm is common in different programming languages in web development. Knowing about OOP was therefore interesting as it could make it easier to learn and read code written in other programming languages such as Java or Typescript, and provide a deeper understanding of how some React components work.

WHAT WAS THE GOAL

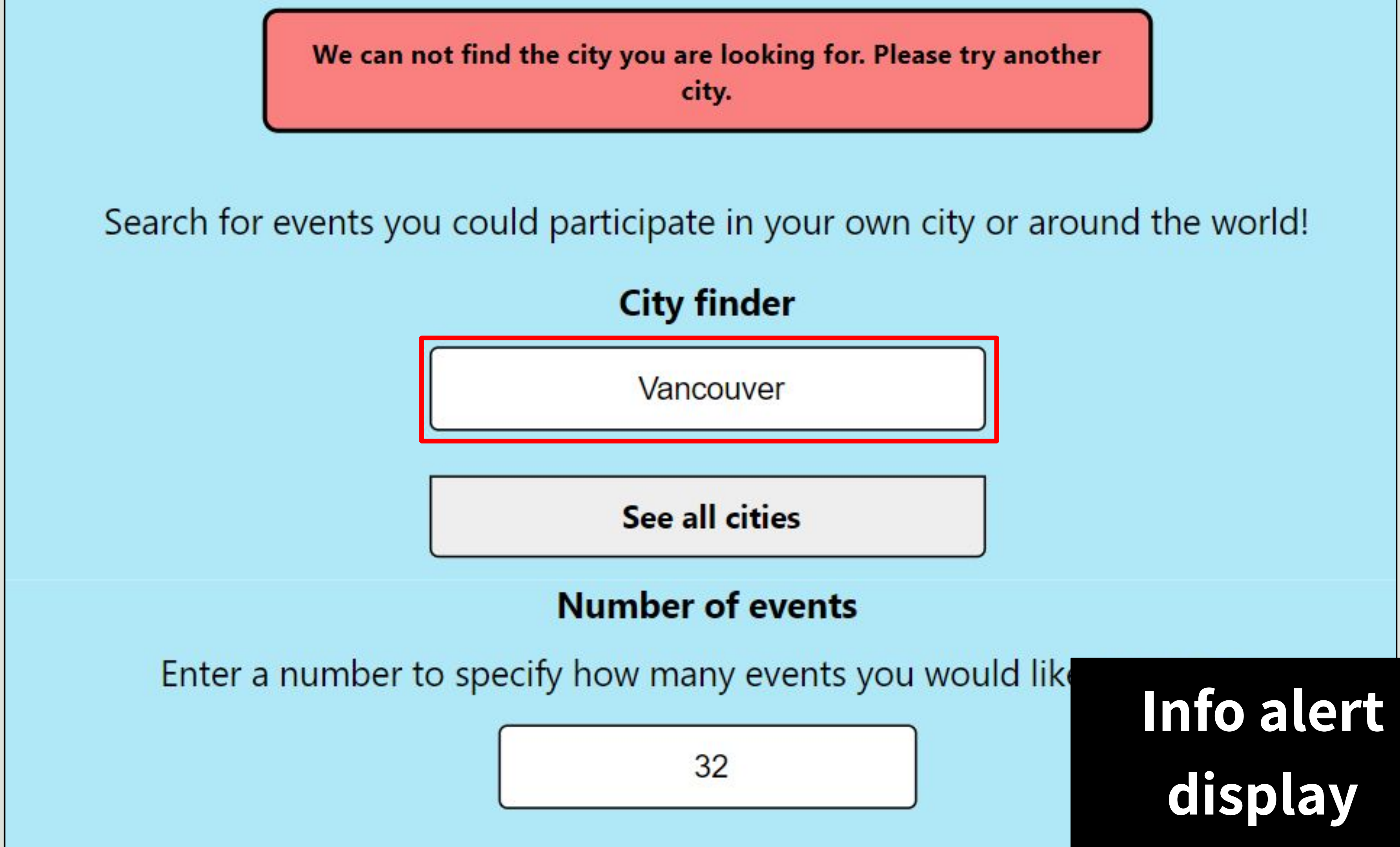
Setting up alerts in Meet app using OOP (see examples on next slides). More precisely:

- An info alert that tells users when they've entered an incorrect search term in the city search input field (e.g.: entered an unknown city, a number or a special character)
- An error alert that tells users when they've entered an incorrect number in the number of events input field (e.g.: entered a letter, a negative number or a special character)

Testing the new implemented alerts to ensure their proper functioning.

Ensuring that all previous tests written before are still all passing (and make the updates when necessary to fix those who weren't passing anymore due to updated files for the new alerts logic).

Deploying the app to Github Pages to see the implemented changes in the publicly available version of the app.



We can not find the city you are looking for. Please try another city.

Search for events you could participate in your own city or around the world!

City finder

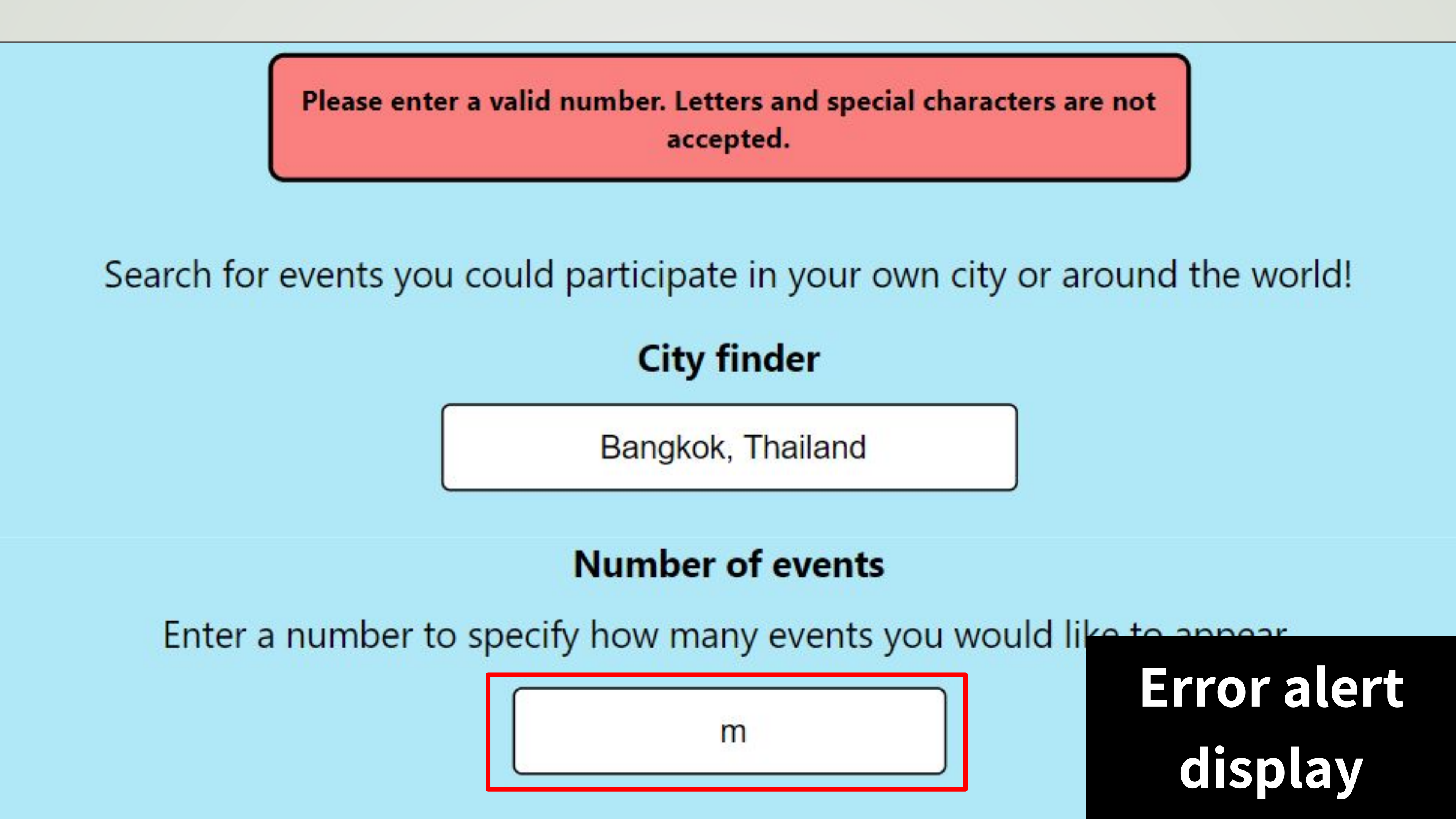
Vancouver

Number of events

Enter a number to specify how many events you would lik

32

Info alert display


Please enter a valid number. Letters and special characters are not accepted.

Search for events you could participate in your own city or around the world!

City finder

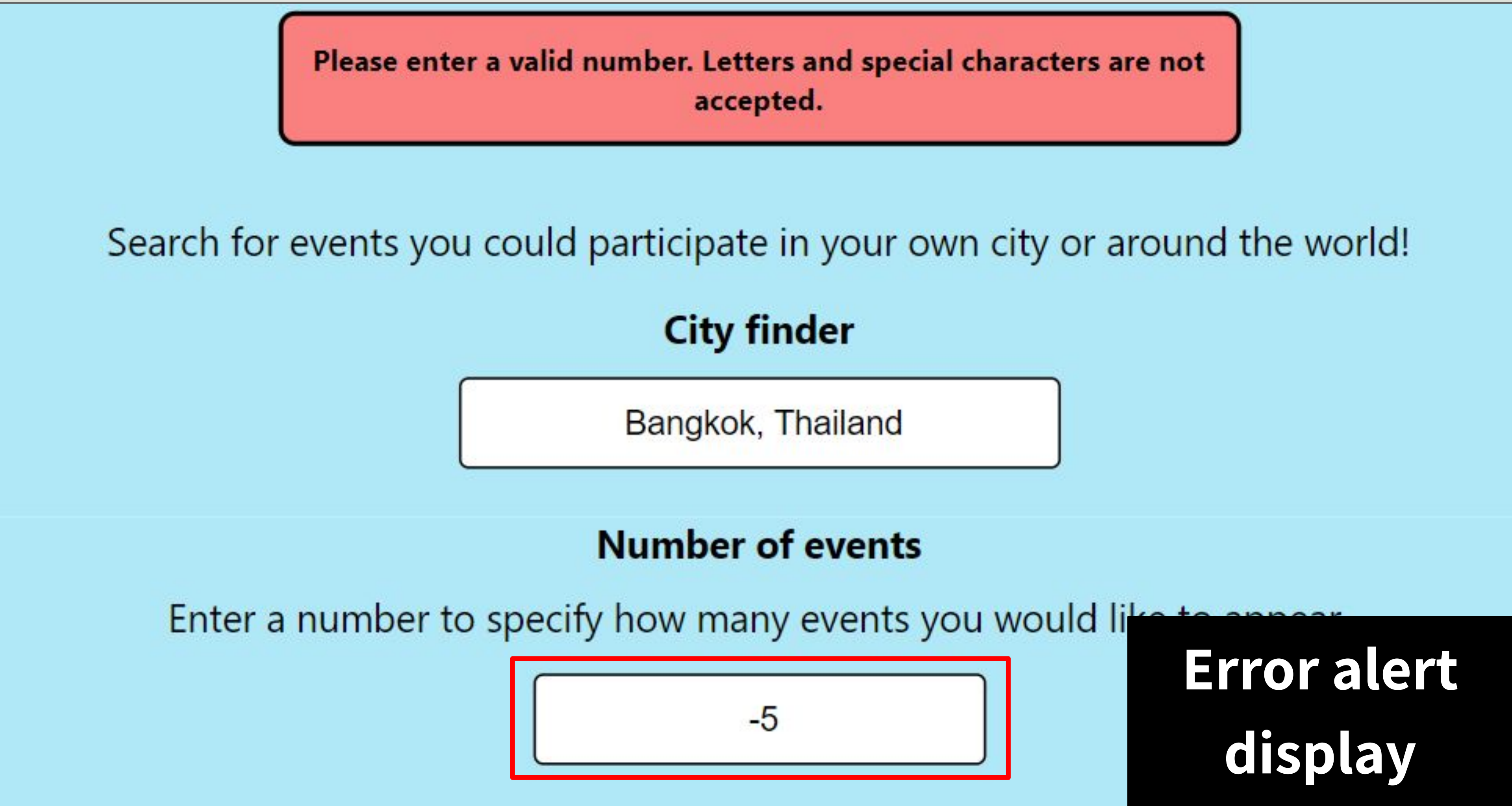
Bangkok, Thailand

Number of events

Enter a number to specify how many events you would li

m

Error alert display



Please enter a valid number. Letters and special characters are not accepted.

Search for events you could participate in your own city or around the world!

City finder

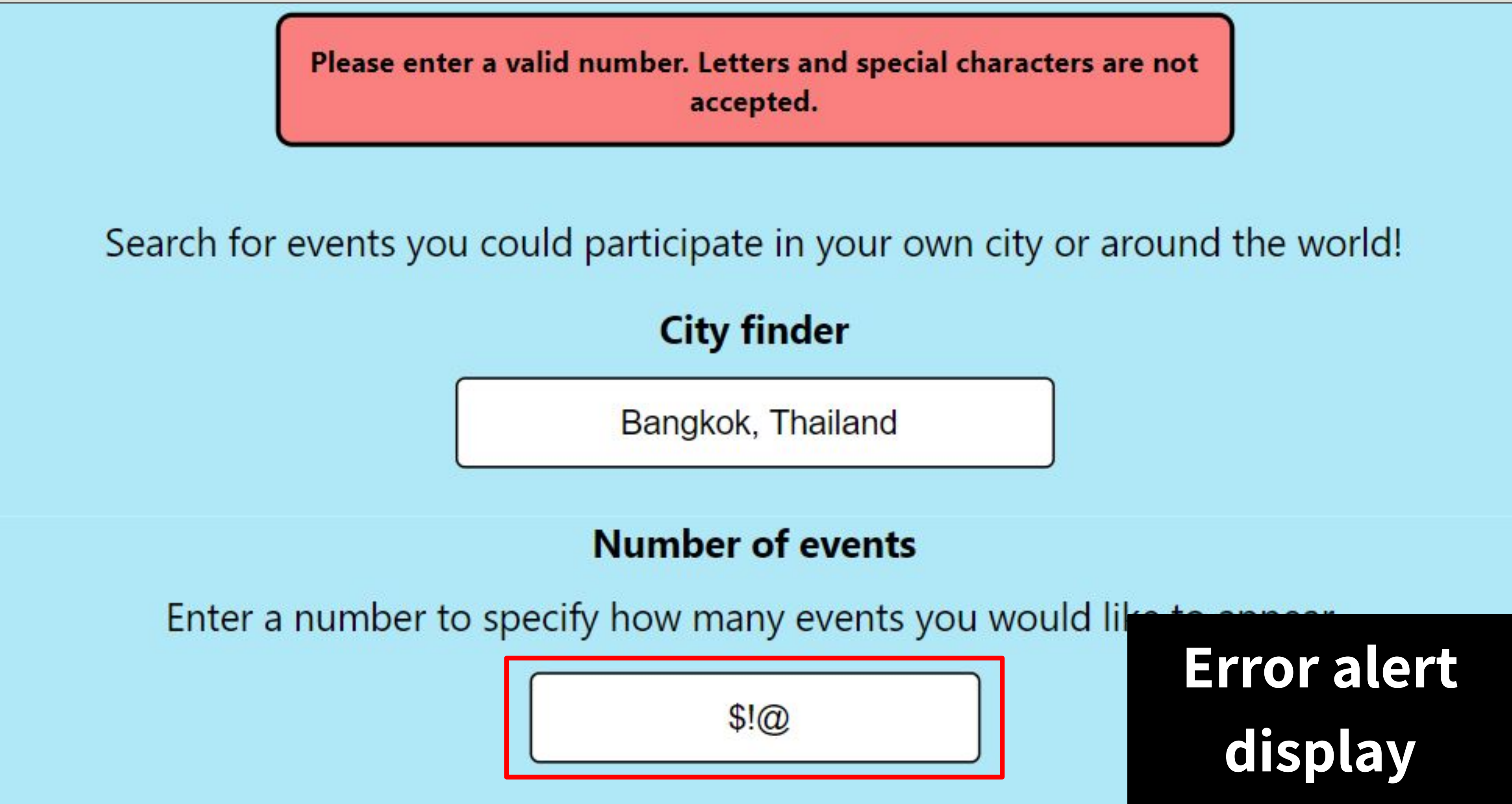
Bangkok, Thailand

Number of events

Enter a number to specify how many events you would li

-5

Error alert display



Please enter a valid number. Letters and special characters are not accepted.

Search for events you could participate in your own city or around the world!

City finder

Bangkok, Thailand

Number of events

Enter a number to specify how many events you would lik

$!@

Error alert display

10

Implementing progressive functionalities so that the app can be used offline and be added to a user's home screen (desktop and mobile)

Skills used

Code writing

WHY WAS THIS STEP IMPORTANT

Setting up Meet app to be a progressive web app was important since it allowed it to be installed on both mobile devices and computers (just like a regular apps) and to work offline, two things that are quite useful and that greatly enhance the user experience.

WHAT WAS THE GOAL

Converting Meet app into a progressive web app. To do so, I:

- Analyzed where my app already stands against the PWA criteria using the Lighthouse tool from the browser Google DevTools.
- Configured the web app manifest (which allows the app to be installed on desktop or mobile) by adding a new custom icon for Meet app.
- Registered / enabled the service worker (which allows the app to work while offline).
- Implemented the app's offline logic so that when a user is online, the list of events is loaded from the API and then saved to localStorage, which allows the list of events to be loaded from this localStorage when the user is offline and the API isn't available.

WHAT WAS THE GOAL (SUITE)

- Implemented a warning alert to show a notification that tells offline users that the displayed list of events they see has been loaded from the cache (rather than the API).

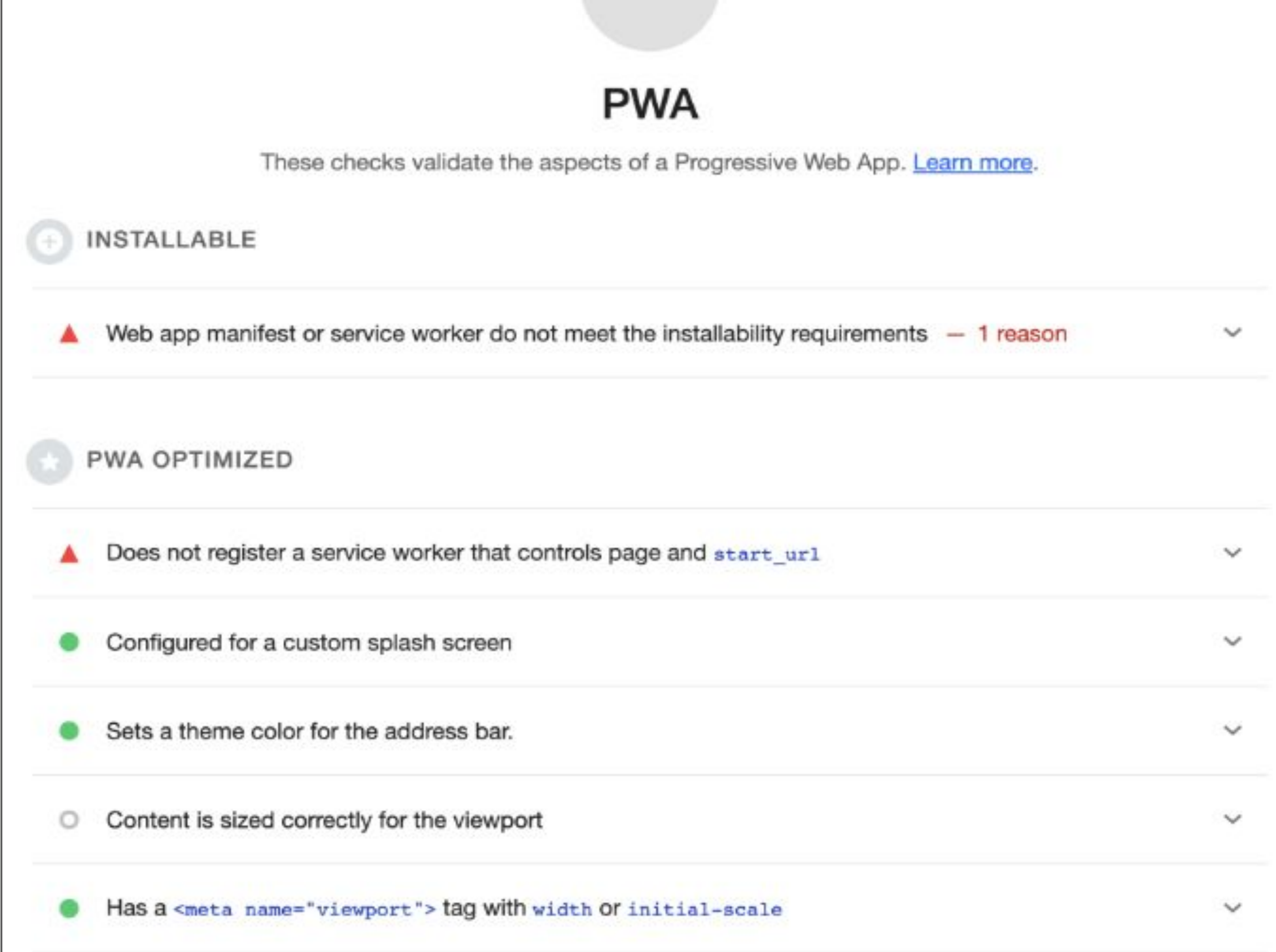
- Deployed my app again on Github Pages with the new updates.

- Re-ran the Lighthouse tool audit to ensure my app has been converted correctly into a PWA (which was the case).

- Tested my new PWA features by installing my app on my computer and mobile phone, as well as using it offline by using the Service Workers section in the Application tab of the DevTools to simulate no internet connection, and by simply turning off of my internet connection manually.

FIRST LIGHTHOUSE ANALYSIS - APP IS NOT A PWA

(EXPECTED SINCE IT WASN'T CONFIGURED FOR IT AT THIS POINT)



PWA

These checks validate the aspects of a Progressive Web App. Learn more.

INSTALLABLE

▲ Web app manifest or service worker do not meet the installability requirements — 1 reason

PWA OPTIMIZED

▲ Does not register a service worker that controls page and `start_url`

● Configured for a custom splash screen

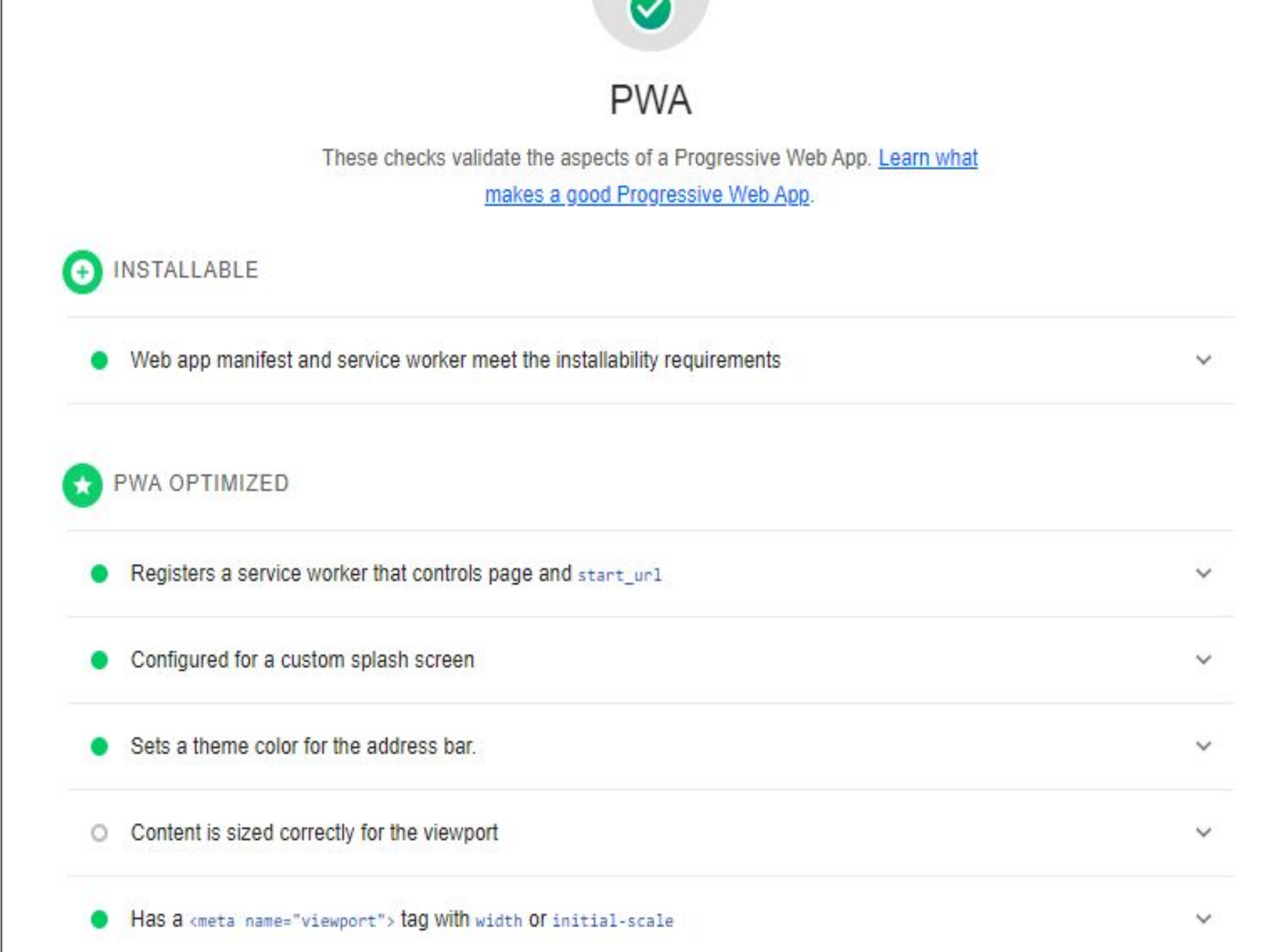
● Sets a theme color for the address bar.

○ Content is sized correctly for the viewport

● Has a `<meta name="viewport">` tag with `width` or `initial-scale`

SECOND LIGHTHOUSE ANALYSIS - APP IS A PWA

(EXPECTED SINCE IT HAS BEEN CONFIGURED FOR IT AT THIS POINT)



PWA

These checks validate the aspects of a Progressive Web App. Learn what makes a good Progressive Web App.

INSTALLABLE

● Web app manifest and service worker meet the installability requirements

PWA OPTIMIZED

● Registers a service worker that controls page and `start_url`

● Configured for a custom splash screen

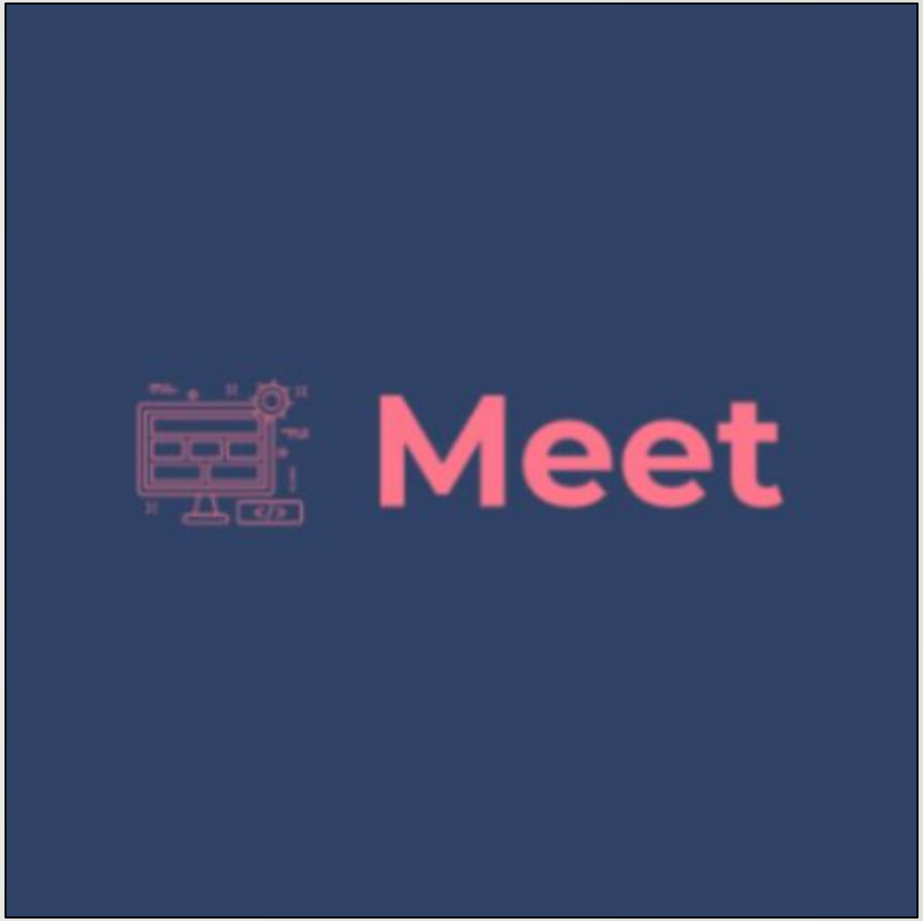
● Sets a theme color for the address bar.

○ Content is sized correctly for the viewport

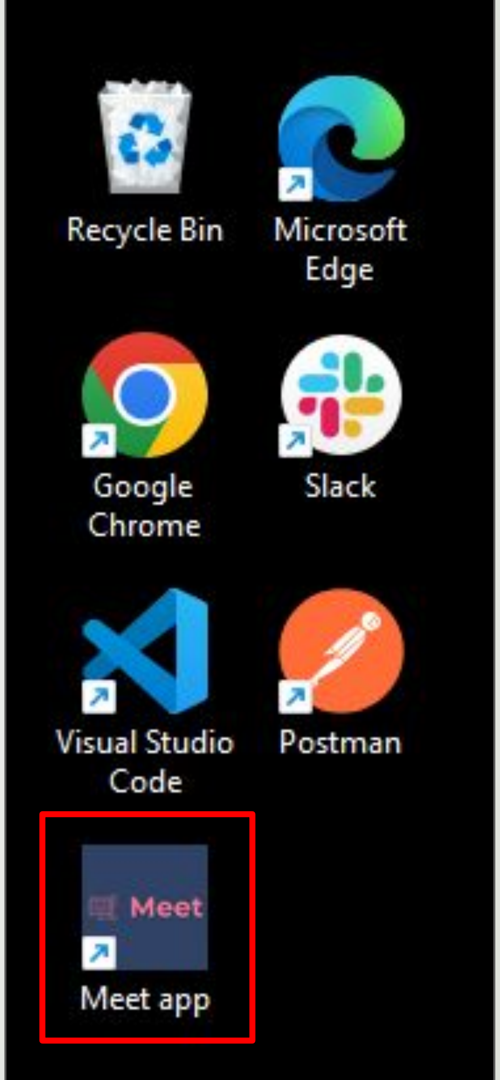
● Has a `<meta name="viewport">` tag with `width` or `initial-scale`

NEW ICON ADDED IN MY WEB APP MANIFEST

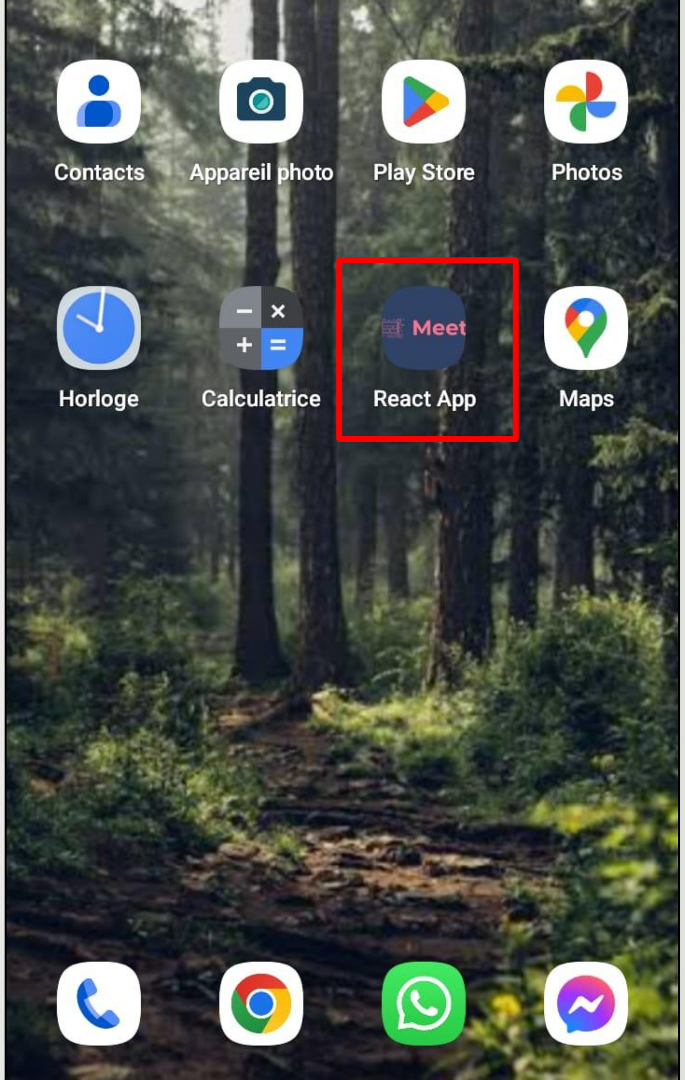
(THIS IS THE MEET APP ICON USERS INSTALLING THE APP ON THEIR DESKTOP OR MOBILE SEE)

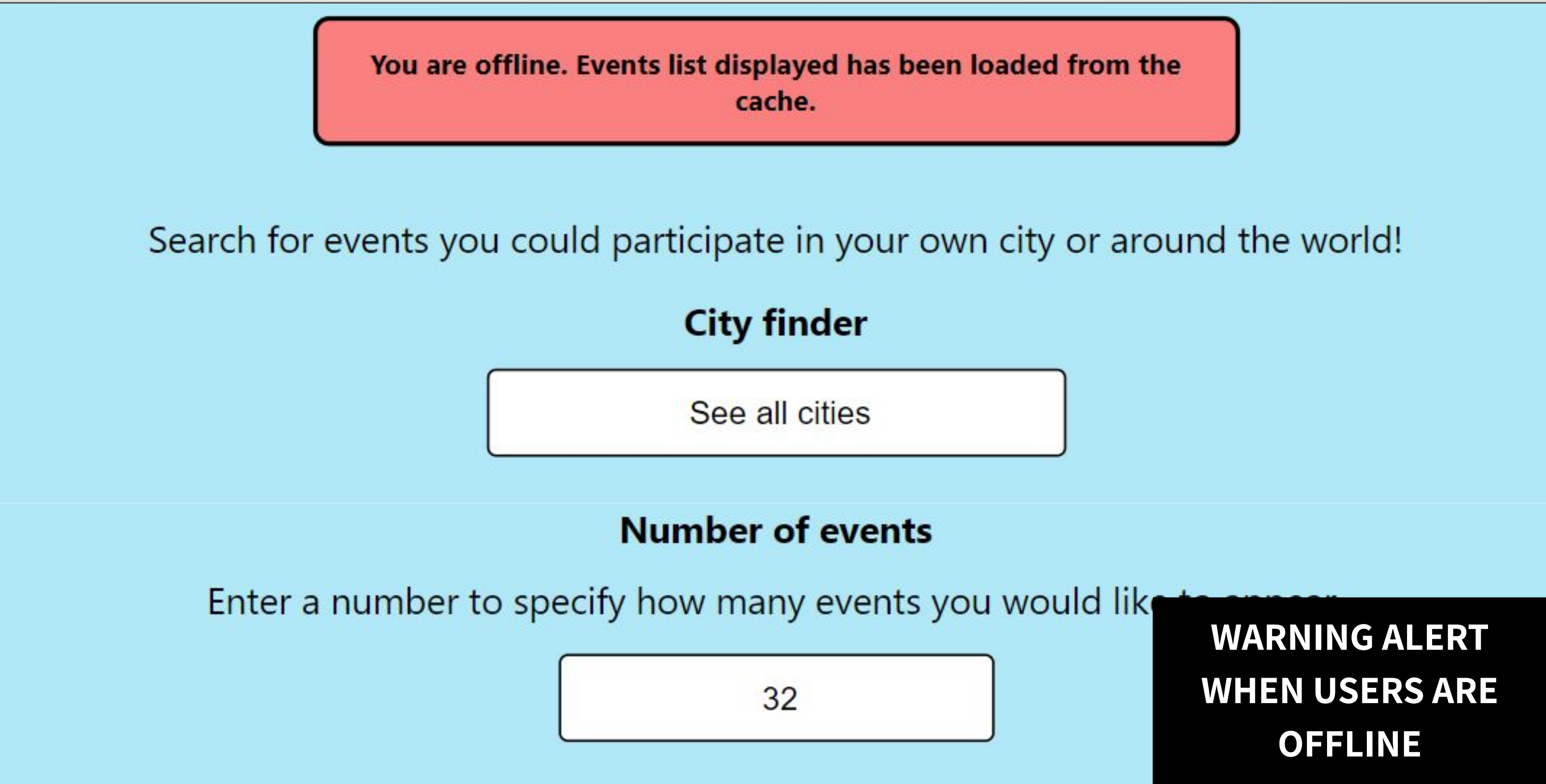


MEET APP INSTALLED ON DESKTOP



MEET APP INSTALLED ON MOBILE PHONE





You are offline. Events list displayed has been loaded from the cache.

Search for events you could participate in your own city or around the world!

City finder

See all cities

Number of events

Enter a number to specify how many events you would lik

32

WARNING ALERT WHEN USERS ARE OFFLINE

11

Recharts

Transforming the Google API data into visuals / app UIs using a visualization library

Skills used

Research
Code writing
Debugging

WHY WAS THIS STEP IMPORTANT

Displaying information and data in visual format makes it easier for insights and trends to be recognized and understood. It also helps users engage more easily with the information displayed in the app. Images and visual representations often speak more than just plain text, hence the relevance of adding visual features into Meet app.

WHAT WAS THE GOAL

Using Recharts (a React-specific charting library) to add different visuals in Meet app, as well as styling those visuals for a clean, responsive, professional and easy-to-understand design. Two types of visual have been implemented:

- A scatter plot graph displaying the number of events by city
- A chart pie displaying the percentage of events by genre

CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

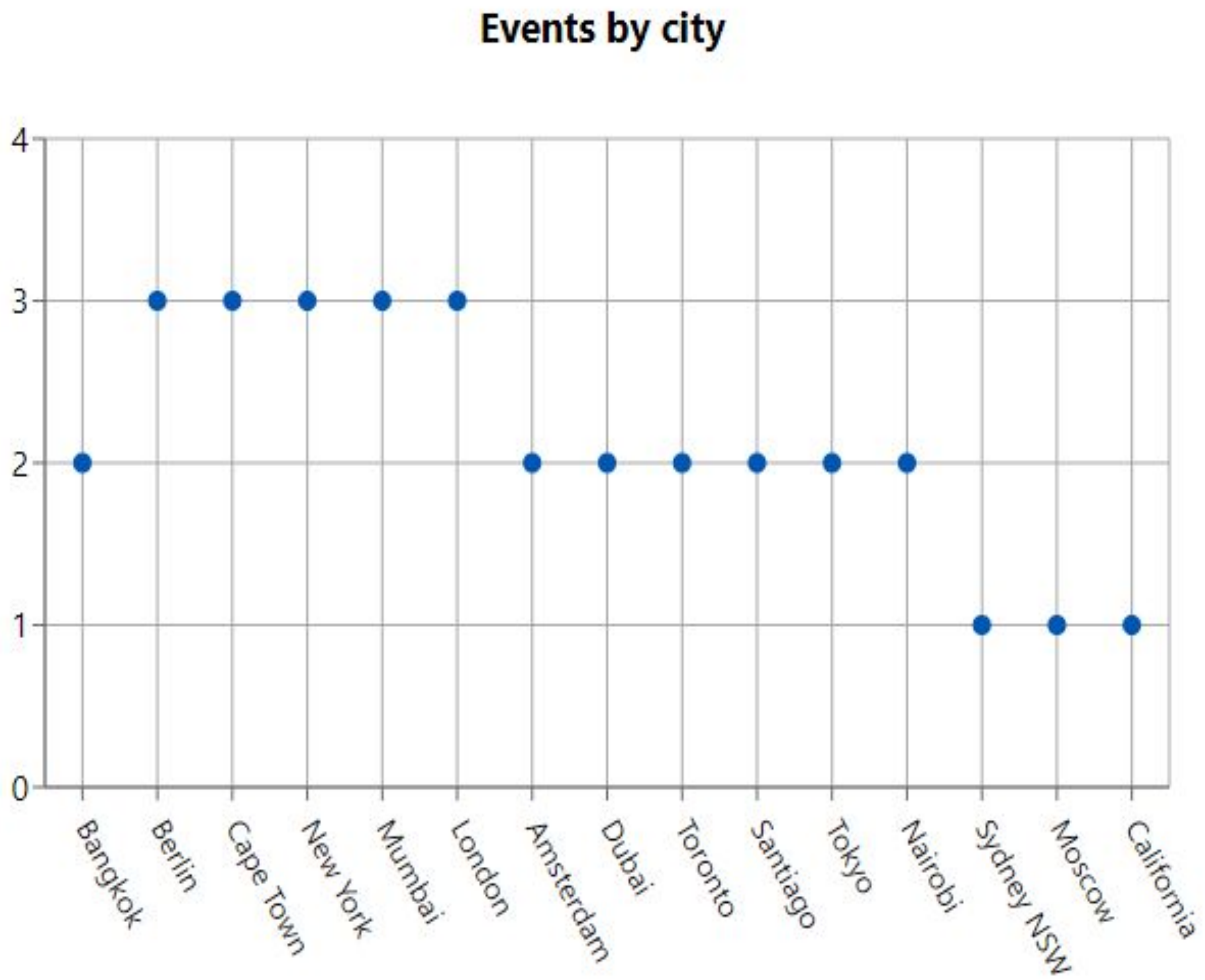
I was in my first experimentation with using Recharts. I therefore had to do some extra reading to understand the way this library works at first, but I was able to use several solutions by myself to assimilate everything and eventually deliver the all expected requirements. Some of the solutions I used to face the challenges:

- Read the official Reacharts documentation
- Carrying several tests to understand what works well and what does not work when implementing graphs and charts, and adapt my codes consequently

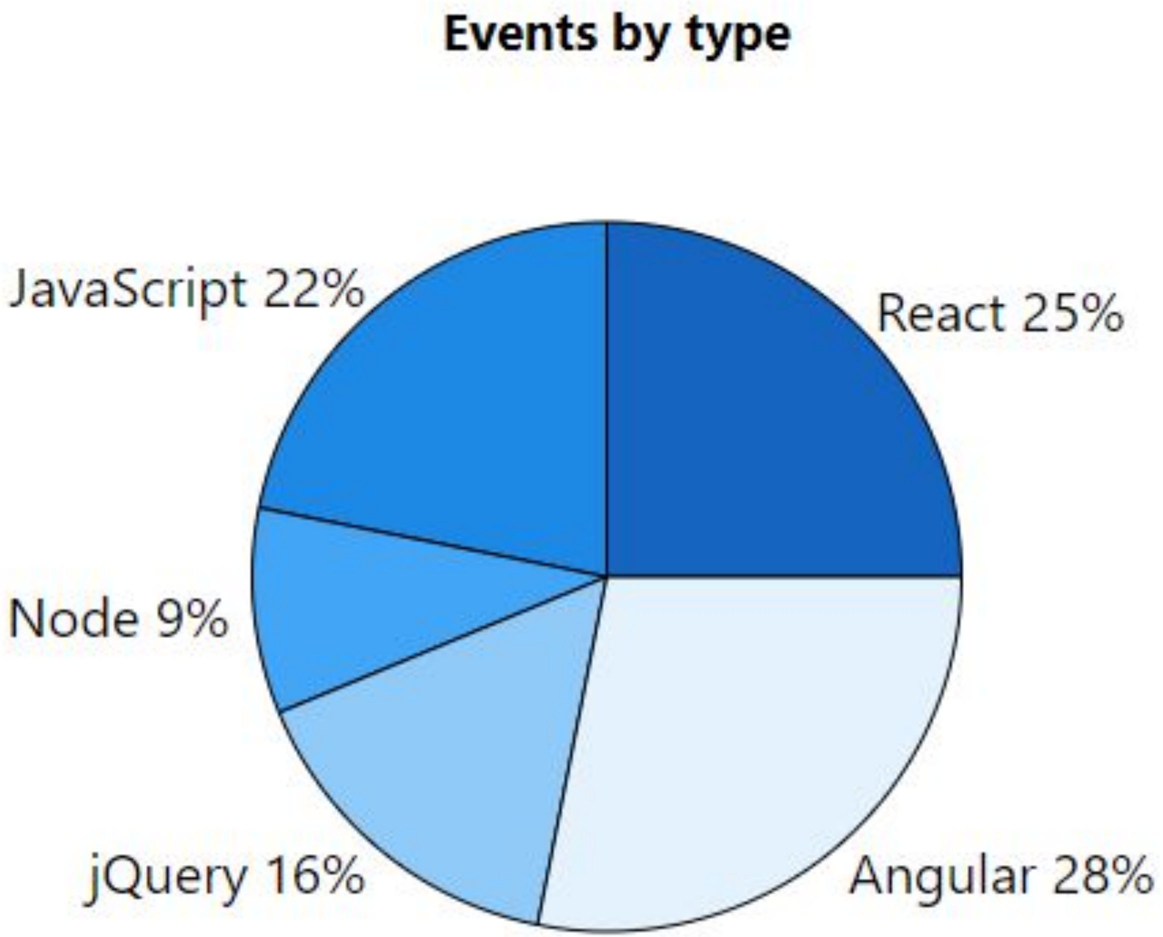
DECISIONS MADE

I configured the visuals. I also decided on the layout to have for these visuals depending on the screen size, ensuring that they are responsive for all devices and remain well displayed on any screen sizes.

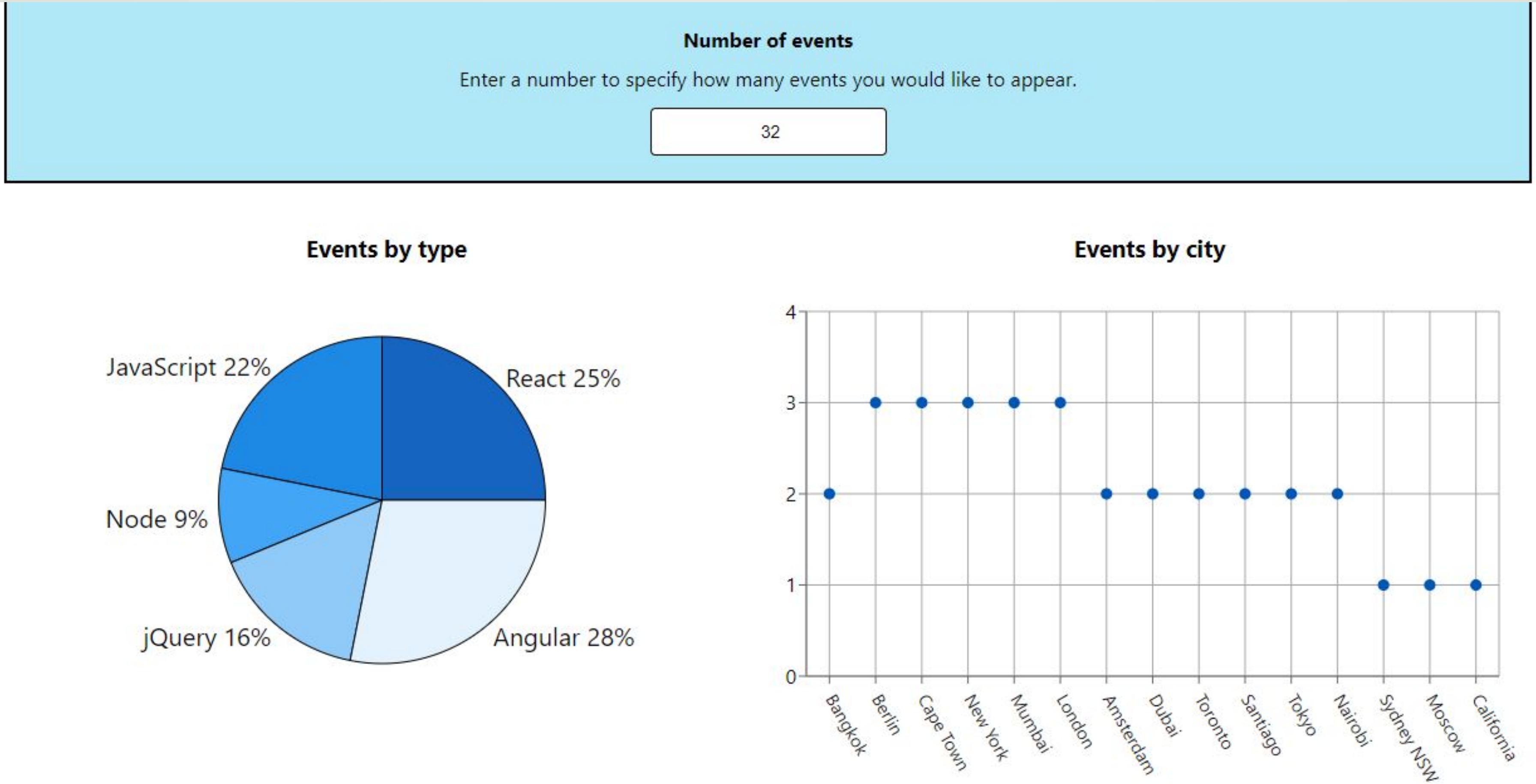
SCATTER PLOT REPRESENTING THE NUMBER OF EVENTS SHOWN BY CITY



PIE CHART REPRESENTING THE EVENT'S GENRE (AND THE % FOR EACH EVENT'S GENRE SHOWN)



COMPLETE VIEW ON LARGE SCREENS (COMPUTERS AND TABLETS)

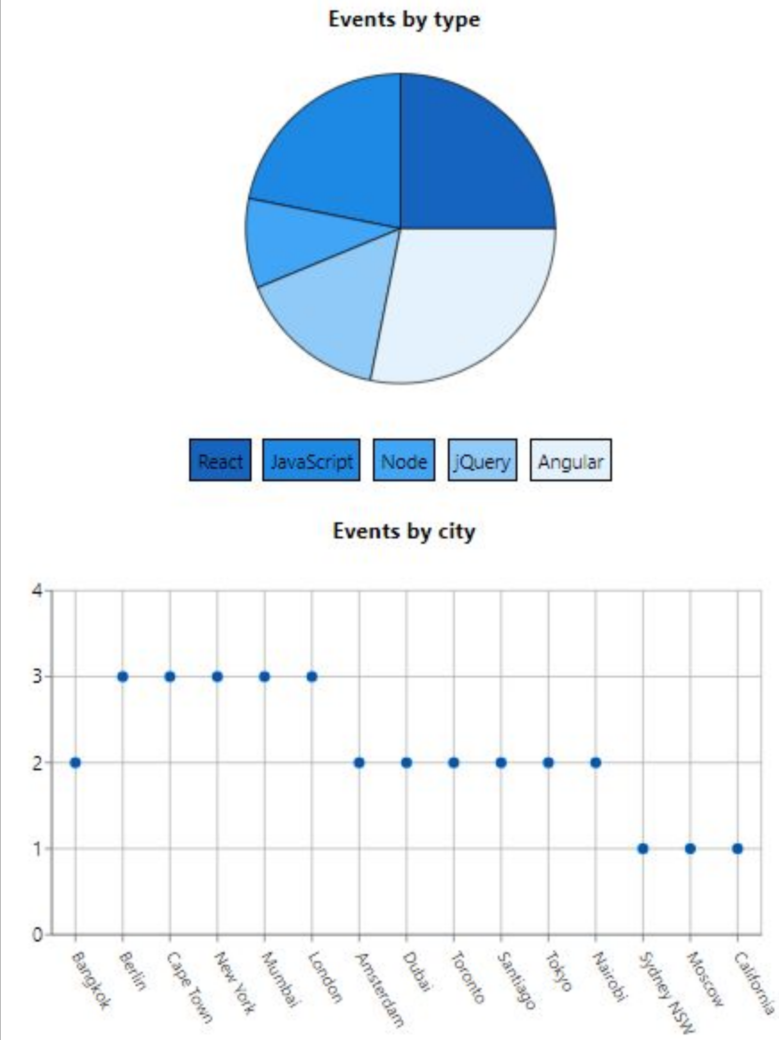

Number of events

Enter a number to specify how many events you would like to appear.

32

COMPLETE VIEW ON SMALL SCREENS (MOBILE PHONE)

*The current photo is not as clear and precise as the view in reality due to the zoom applied to capture the whole elements for illustration purpose here. To get a better view, visit Meet app in live.



11

Finalizing code revision and refactoring

Skills used

Critical thinking

WHY WAS THIS STEP IMPORTANT

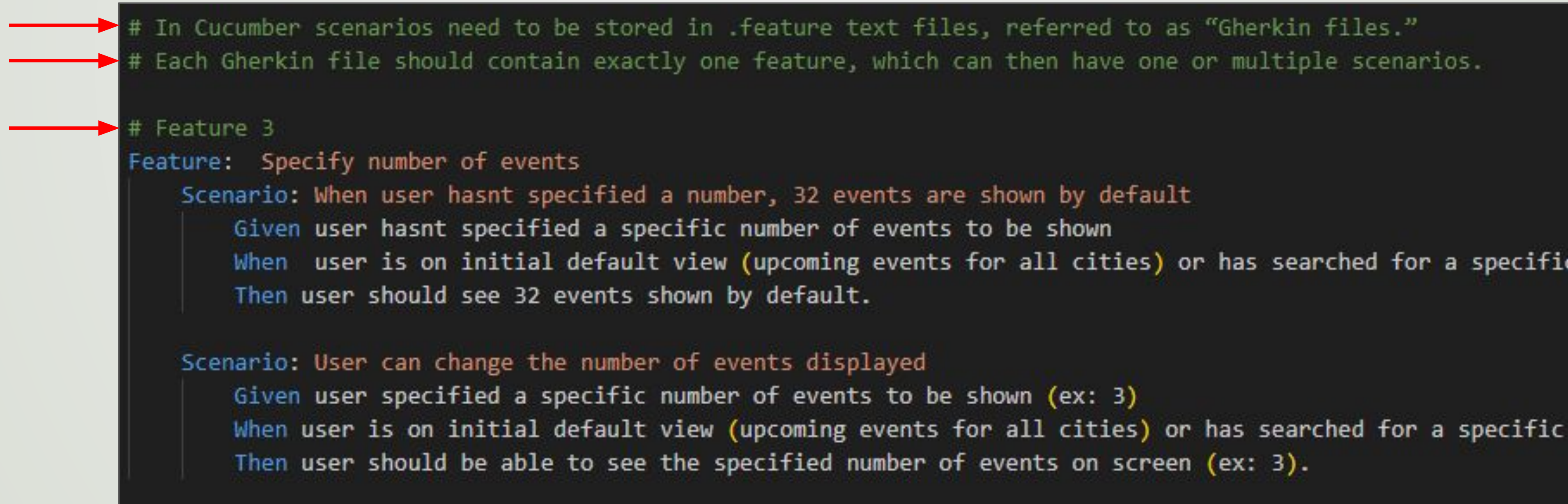
Ensuring that the codes are optimized to facilitate possible appropriation by other developers in the future is useful and could possibly save time. It can also facilitate any future adjustments to the codes.

WHAT WAS THE GOAL

Reviewing each code to make sure everything was optimized as much as possible in order to facilitate future modification, addition or adjustment in the future.

Adding comments and clarifications points in the code where important for the benefit and better understanding of anyone else who may work on this project later.

EXAMPLE OF CODE COMMENTS TO FACILITATE UNDERSTANDING OF THE FILE AND POSSIBLE FUTURE UPDATES AND WORK



```
# In Cucumber scenarios need to be stored in .feature text files, referred to as “Gherkin files.”
# Each Gherkin file should contain exactly one feature, which can then have one or multiple scenarios.

# Feature 3
Feature:  Specify number of events
    Scenario: When user hasnt specified a number, 32 events are shown by default
        Given user hasnt specified a specific number of events to be shown
        When  user is on initial default view (upcoming events for all cities) or has searched for a specifi
        Then user should see 32 events shown by default.

    Scenario: User can change the number of events displayed
        Given user specified a specific number of events to be shown (ex: 3)
        When user is on initial default view (upcoming events for all cities) or has searched for a specific
        Then user should be able to see the specified number of events on screen (ex: 3).
```

CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

I positioned myself from the point of view of future colleagues who could work on my project. How can I make my project and my codes as clear as possible to promote its easy appropriation? I reviewed each file to bring improvements in certain places and add comments where I thought it could be useful.

DECISIONS MADE

This step was done on my own initiative and was not required in the project requirements. I made decisions regarding the improvement of certain parts of my codes, and the addition of comments where necessary in order to set my mind to work in a collaborative environment already.

README.
md

Completing the README document for Meet app

Skills used

Communication
Content writing

WHY WAS THIS STEP IMPORTANT

Ensure Meet app is well documented and easily accessible by anyone interested.

WHAT WAS THE GOAL

Updating and completing the README file located in my Meet app Github repository. The goal was to ensure that all relevant information regarding Meet app is accessible under these five categories:

- Project description
- User interface
- User stories and scenarios
- Technical aspects
- App dependencies

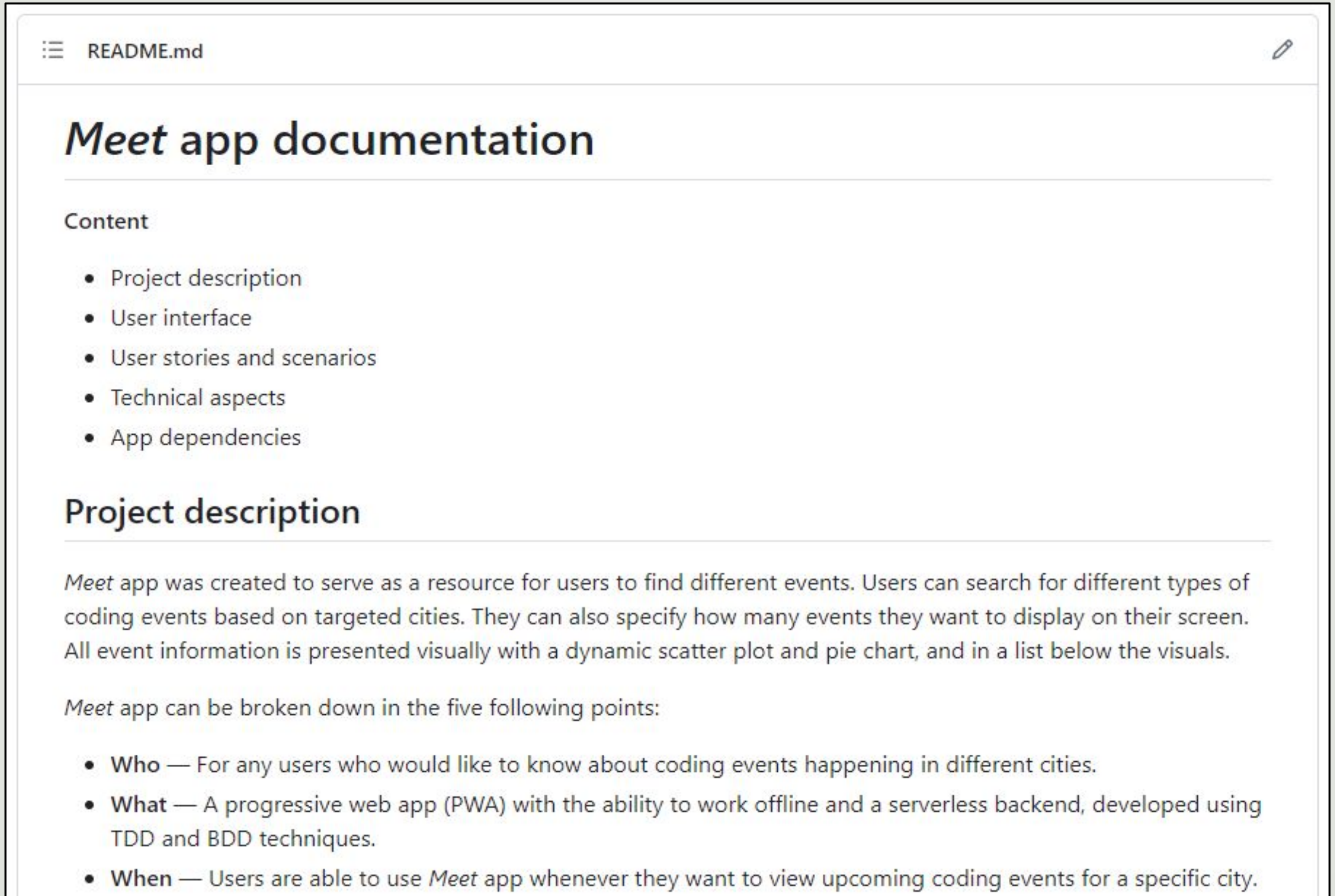
CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

Finding the right balance between giving the right level of information, while remaining as synthetic as possible. To help me, I made a first draft, which I then modified several times. I get inspired by other READMEs I’ve consulted and for which I found that the information presented was relevant.

DECISIONS MADE

I wrote the README documentation from A to Z, in terms of content, presentation and structure.

README SAMPLE - FULL VERSION ON GITHUB



README.md

Meet app documentation

Content

- Project description
- User interface
- User stories and scenarios
- Technical aspects
- App dependencies

Project description

Meet app was created to serve as a resource for users to find different events. Users can search for different types of coding events based on targeted cities. They can also specify how many events they want to display on their screen. All event information is presented visually with a dynamic scatter plot and pie chart, and in a list below the visuals.

Meet app can be broken down in the five following points:

- **Who** — For any users who would like to know about coding events happening in different cities.
- **What** — A progressive web app (PWA) with the ability to work offline and a serverless backend, developed using TDD and BDD techniques.
- **When** — Users are able to use *Meet* app whenever they want to view upcoming coding events for a specific city.