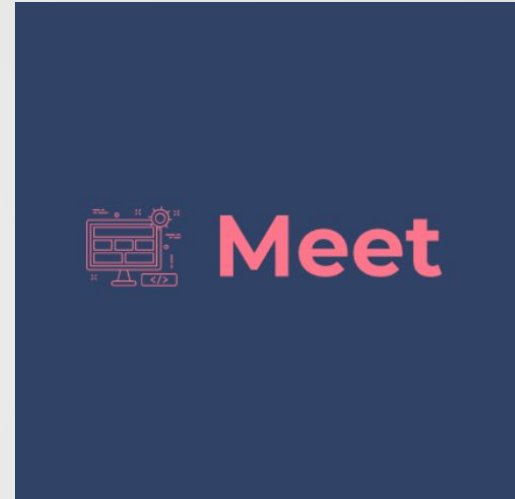


# Meet web app

## Development process



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

# TABLE OF CONTENTS

---

**01**

---

Setting up the development environment

---

**02**

---

Writing user stories and test scenarios

---

**03**

---

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

---

**04**

---

Writing serverless functions for authorization process

---

**05**

---

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

---

**06**

---

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

# TABLE OF CONTENTS

---

**07**

---

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

---

**08**

---

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

---

**09**

---

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

---

**10**

---

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

---

**11**

---

Transforming the Google API data into visuals using a visualization library

---

**12**

---

Finalizing code revision and refactoring

# TABLE OF CONTENTS

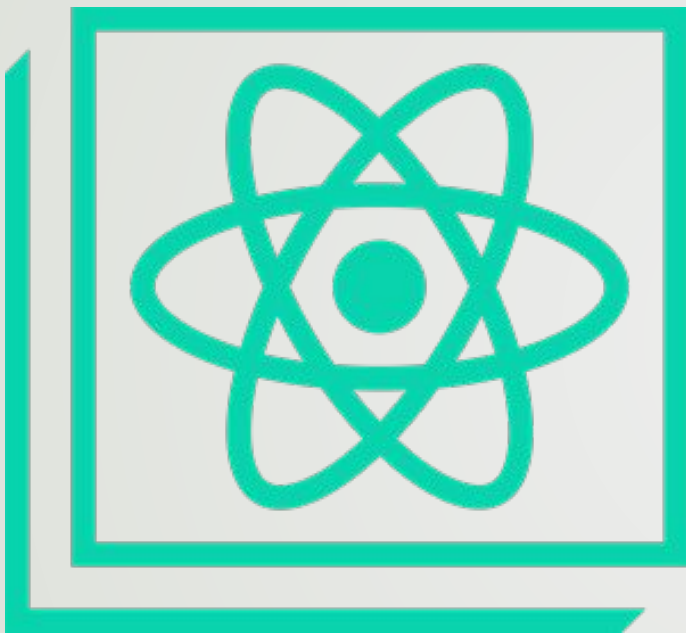
---

**13**

---

Completing the README document

# 01



**Setting up** the development environment

Skills used

Research

# **WHY** WAS THIS STEP IMPORTANT

Organizing the development environment for efficient and well-oriented work from the beginning is not only necessary, but is also 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 command 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 a 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.: addition of some elements in the *package.json* file).

Pushing the created project to Github Pages.



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

[Learn React](#)

---

# 02

---

Features



User stories



Scenarios



Gherkin test scenarios

**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 carrying out several tests along the product development, and ensure all new logic implemented work. 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 key features from the project requirements for Meet web 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 (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.



**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 the web 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 web app a logic for an authentication process (using Google OAuth), so that users using Meet web app for the first time can log in via their Google account and grants consent to the app. This would then grant a token to it and allow Meet web 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.

# WHAT WAS THE GOAL

Creating an OAuth consumer for the web app by:

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

Setting up AWS Lambda (using the Serverless Toolkit):

- Installing the Serverless framework globally on my computer
- Creating a serverless service in the 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 web app since it was installed globally).

# CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

I was in my first experimentation with both Google OAuth and AWS. I therefore had to do some extra reading to understand the mechanic behind those two new elements at first, but I was able to use several solutions by myself to assimilate everything and eventually deliver all expected requirements. Some solutions 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 doesn't, and adapt my codes consequently





## **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 the web 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 web 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 the web 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 web 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 the web app is passed to getAccessToken, which then get a temporary access token.
- **getCalendarEvents function** - function used by Meet web 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).



**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 unit tests is an important process to do at the beginning 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 afterward.

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

# WHAT WAS THE GOAL

Creating and writing unit tests for the web app first three key features\* and testing them using Jest and React Testing Library.

\*In Meet web app, they were six key features to implement from the project requirements. The latter three features 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 and inserting into it mock-data that mirrors the real data from the Google Calendar API to carry out the tests (using mock-data is sometimes a great way to simplify code during the testing phase rather than using real data).

- To do so, I had to know how the Google Calendar API 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-pasted those data into my newly created *mock-data.js* file to use them later for my tests.

Writing the actual implementation codes for each of the first three app key features (to make their tests pass).

- At the end of this step, the first three app features and related scenarios were tested and coded using the Test-Driven Development approach. This ensured that each individual piece of these features worked the way it should independently and on their own, thus allowing to later create more integrated tests that involve interaction and relation between those 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 `.unref()` 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.
```



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

---

## Skills used

---

Creative thinking  
Research  
Code writing



# WHY WAS THIS STEP IMPORTANT

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

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

# WHAT WAS THE GOAL

Creating integration tests for the web app first three features, testing them, and expect them to fail at first since no code has been written to make them pass at this point.

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

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

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

Deploying the web 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 chose 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 web app. I've implemented, inspired by other well-designed web apps, a consistent styling for a professional looking web 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 IMPLEMENTED 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 IMPLEMENTED 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!

### City finder

### Number of events

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

Search for a city

Santiago, Santiago Metropolitan  
Region, Chile

California, USA

Bangkok, Thailand

Berlin, Germany

Cape Town, South Africa

New York, NY, USA



### **jQuery and More**

Wednesday, 07/01/2020, 09:05 AM  
Santiago, Santiago Metropolitan Region, Chile

[Show details](#)

### **React California**

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

[Show details](#)

### **Hello JavaScript!!**

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

[Show details](#)

### **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.

Hide details

### 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 opportunities in the market. React California is a non-profit organization offering free training sessions to React enthusiasts everywhere.

## EXAMPLE OF STYLING USING CSS

(EVENTS SECTION - EVENTS OPEN 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 product 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 acceptance tests for the first three web app 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 web app features scenarios.
- I therefore created a new *features* folder in the project directory to hold each of the new Gherkin files (e.g.: filterEventByCity.feature) for each of the first three web app features. I then pasted into each Gherkin file a specific web app 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 them (see example over the next slides). Step definition files are written in JavaScript and run by Jest.

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

**EXAMPLE OF THE GHERKIN FILE *filterEventsByCity.feature*** CONTAINING ONE FEATURE (FIRST FEATURE OF THE WEB 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 STEP DEFINITION FILE**  
***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 STEP DEFINITION FILE**  
***filterEventsByCity.test.js*** CONTAINING  
 THE TEST FOR THE SECOND GHERKIN  
 SYNTAX SCENARIO OF FEATURE 1

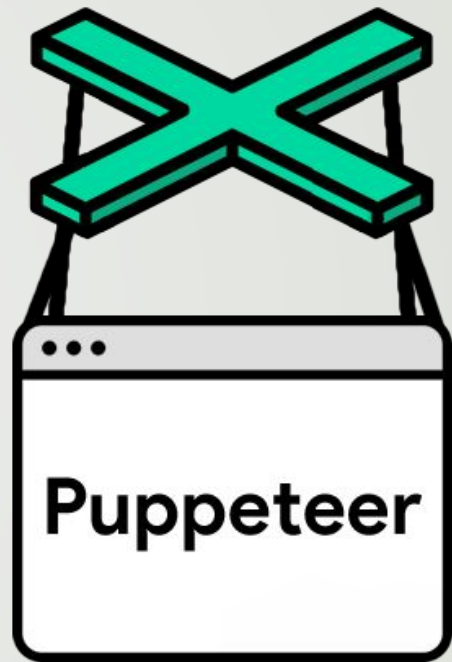


# WHAT WAS THE GOAL (SUITE)

Creating end-to-end tests for the first three web app 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 (text boxes, buttons, etc.), to the end of the utilization process.
- Using this approach allowed to confirm that each of the necessary UI interactions to navigate and use Meet web app worked properly and as expected.

Writing the codes 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 tests 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 increase the code reliability and ensure the web app was working and running as expected, I've decided to create and write more acceptance and end-to-end tests than initially planned. This improved the web app verification process.



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

---

## 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 in 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 tracking the web app behavior in production mode, making it easier to pinpoint and proactively resolve any issues that may arise, hence highlighting the importance of using such tool.

# WHAT WAS THE GOAL

Setting up the project in Atatus (APM tool) to track Meet web app performance.

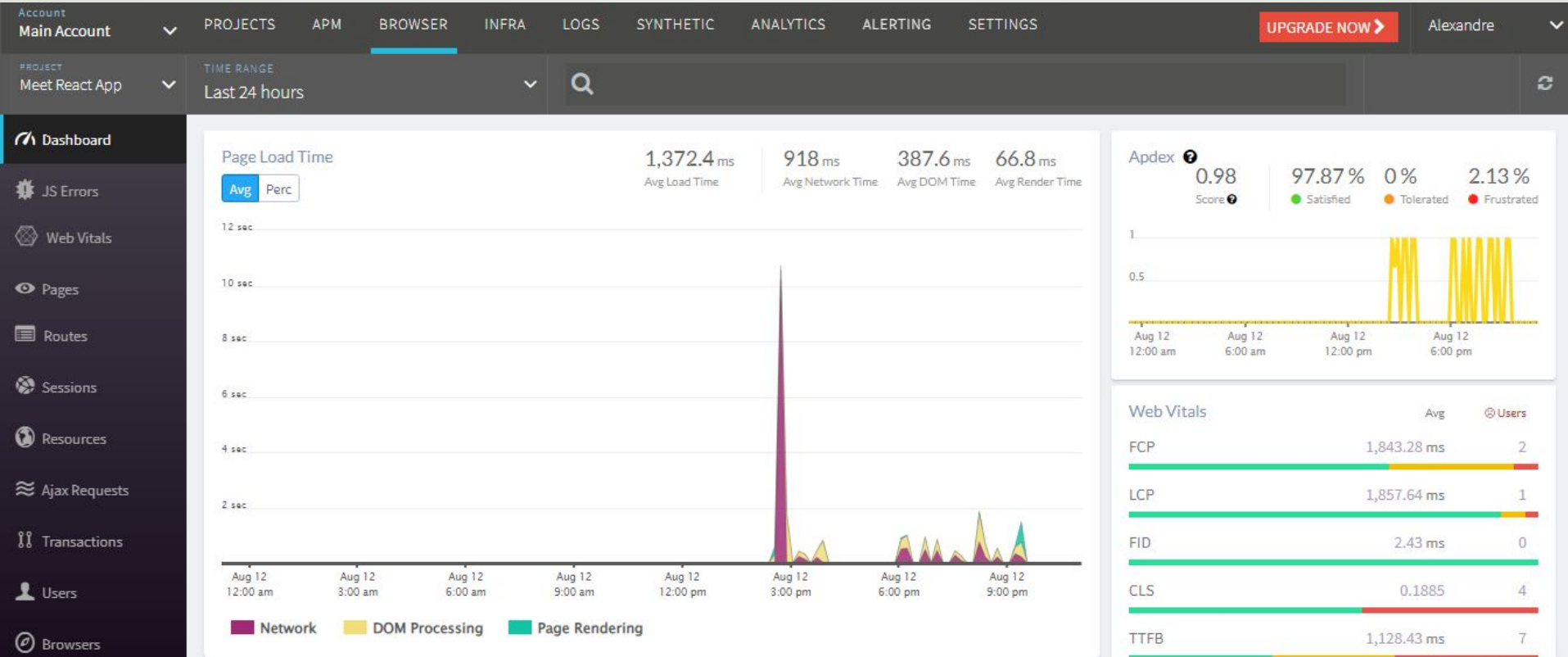
- To do so, I asked external testers to test the web app in order to generate activities into the Atatus dashboard, which allowed me to see how the web app was behaving (performance, errors, page loading time, etc)
- The next slide shows some activities reported in the Atatus dashboard after different testers have used Meet web app

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



# ATATUS MAIN DASHBOARD VIEW

Indicators showing that Meet web 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)



InfoAlert

ErrorAlert

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

---

Skills used

---

Research  
Code writing  
Debugging

# 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 web 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 newly implemented alerts to ensure their proper functioning.

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

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

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

See all cities

### Number of events

Enter a number to specify how many events you would like

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 like to appear

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 like to appear

-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 like to appear

\$!@

**Error alert  
display**





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

---

## Skills used

---

Research  
Code writing

# WHY WAS THIS STEP IMPORTANT

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



# WHAT WAS THE GOAL

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

- Analyzed where the app stands against the PWA criteria at first, using the Lighthouse tool from Google DevTools
- Configured the web app manifest (which allows the app to be installed on desktop or mobile) by adding a new custom icon
- 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, his list of events is loaded from the API and then saved to localStorage, which then allows the list of events to be loaded from this localStorage when the user is later offline and the API isn't available
- (suite on next slide)

# 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 the web app again on Github Pages with the new updates
- Re-ran the Lighthouse tool audit to ensure the app has been converted correctly into a PWA (which was the case)
- Tested the new PWA features by installing Meet web app on a computer and mobile phone, as well as testing it offline by using the Service Workers in the Application tab of DevTools to simulate no internet connection, and by simply turning off 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 THE WEB APP MANIFEST

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



# Meet

# MEET WEB APP INSTALLED ON DESKTOP



Recycle Bin



Microsoft  
Edge



Google  
Chrome



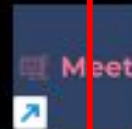
Slack



Visual Studio  
Code



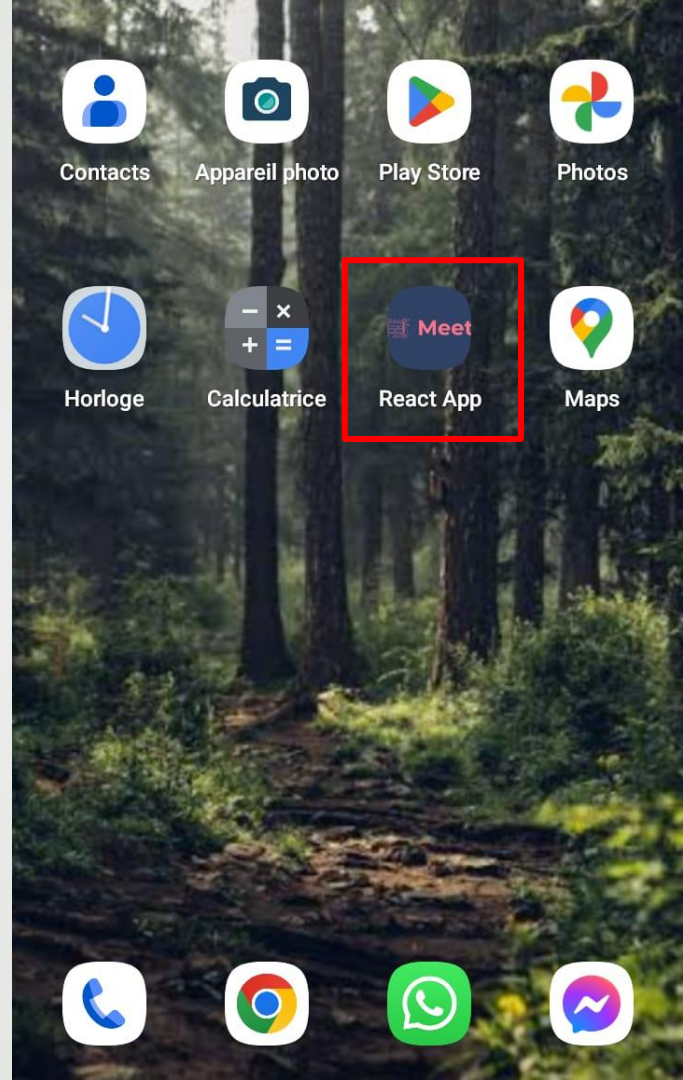
Postman



Meet app



# MEET WEB APP INSTALLED ON DESKTOP



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 like to connect

32

**WARNING ALERT  
WHEN USERS ARE  
OFFLINE**

# Recharts

**Transforming** the Google API  
data into visuals 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. Images and visual representations often speak more than just plain text, hence the relevance of adding visual features into Meet web app.

# WHAT WAS THE GOAL

Using Recharts (a React-specific charting library) to add different visuals in the web 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 displaying the number of events by city
- A pie chart displaying the percentage of events by genre

# CHALLENGES OR SPECIAL POINTS OF CONSIDERATION

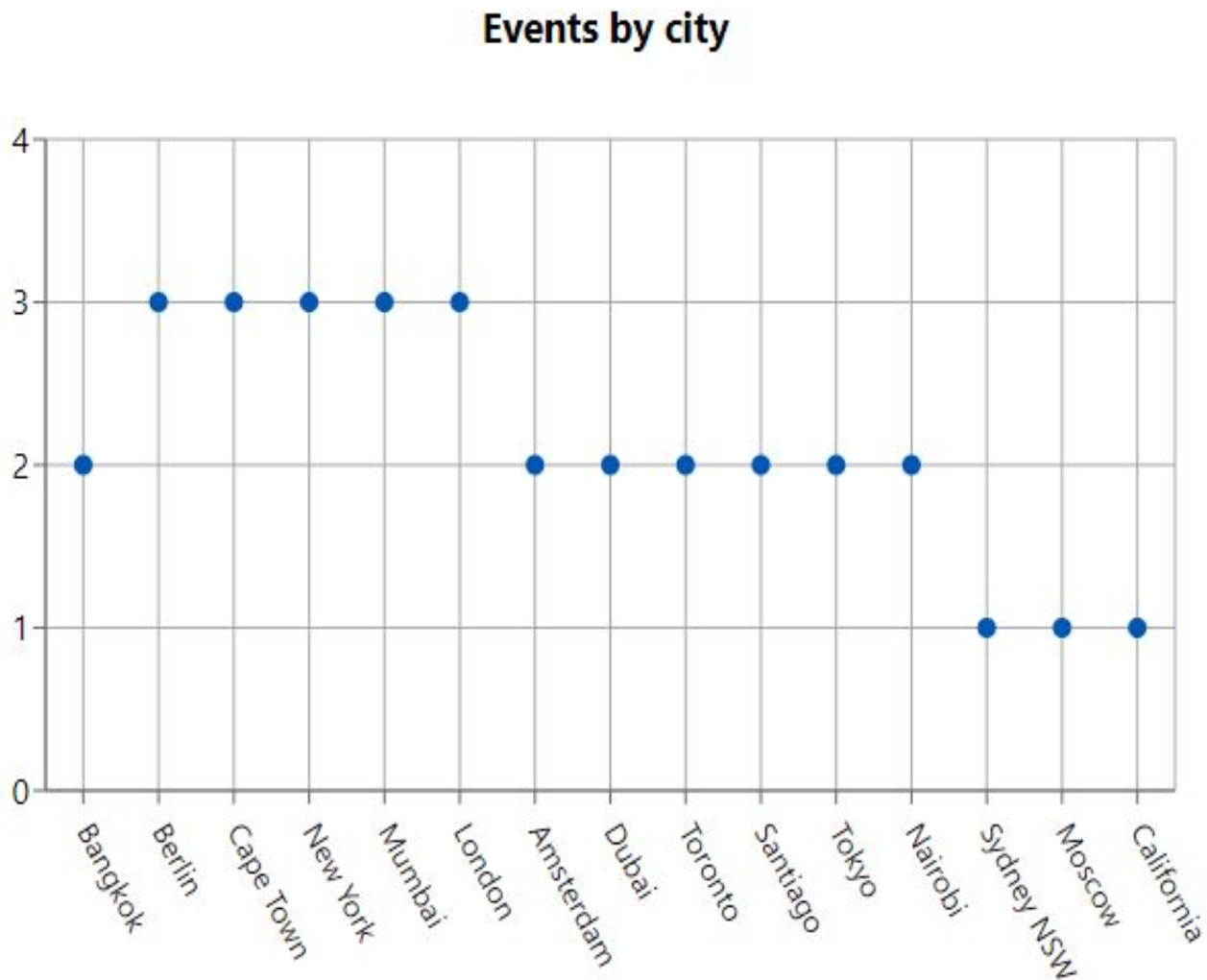
I was in my first experimentation with Recharts. I therefore had to do some extra reading to understand the way this library works at first, but I was able to use several solutions by myself to assimilate everything and eventually deliver all expected requirements. Some solutions used to face the challenges:

- Read the official Recharts documentation
- Carrying several tests to understand what works well and what doesn't when implementing graphs and charts, and adapt the codes consequently

# DECISIONS MADE

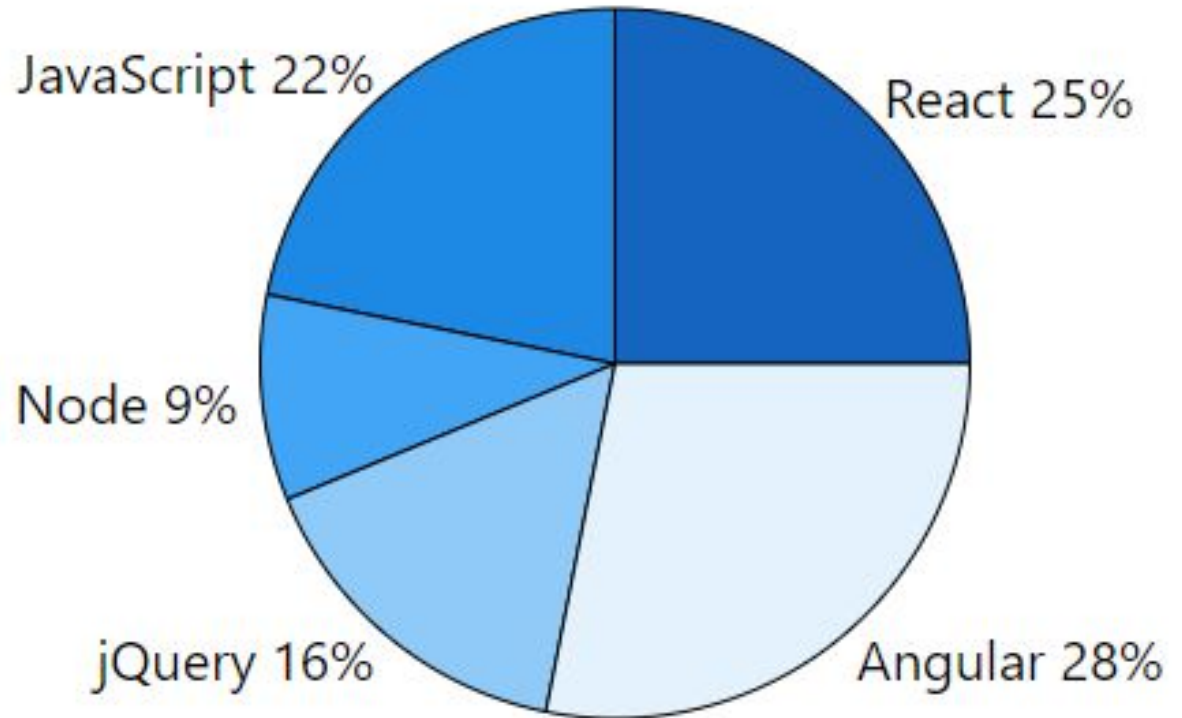
I configured the visuals. I also decided on the layout to have for these visuals depending on the screen size, ensuring that they are responsive for all devices and remain well displayed.

**SCATTER PLOT  
REPRESENTING  
THE NUMBER OF  
EVENTS BY CITY**



**PIE CHART  
REPRESENTING  
EVENT GENRES**  
(AND THE % FOR  
EACH EVENT  
GENRE)

**Events by type**



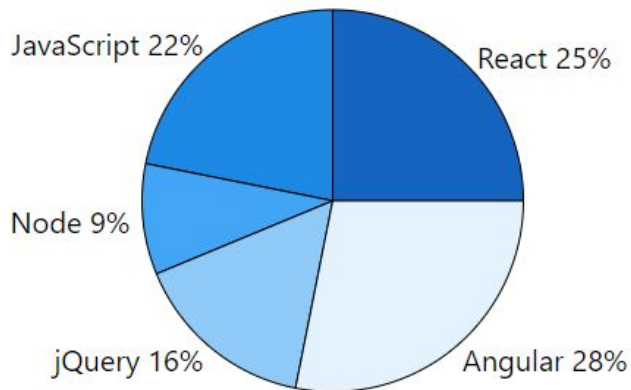


# COMPLETE VIEW ON LARGE SCREENS (COMPUTER AND TABLET)

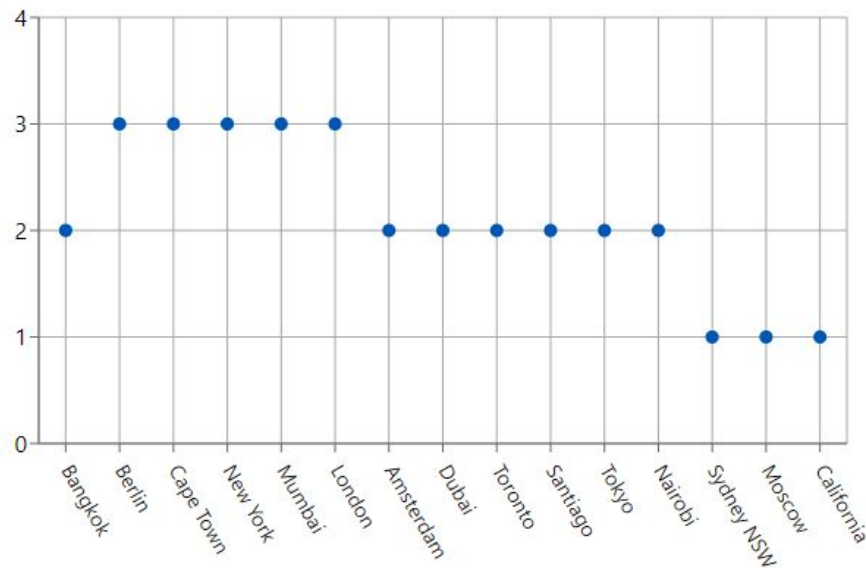
## Number of events

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

Events by type



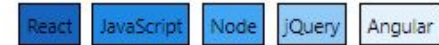
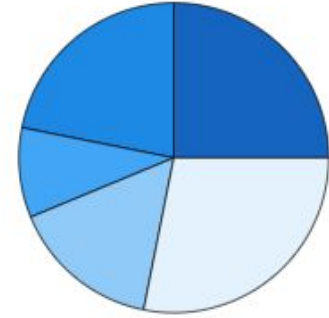
Events by city



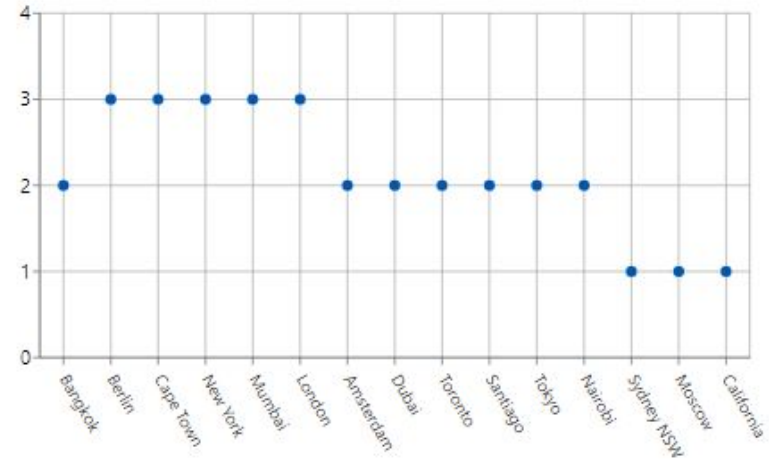
## COMPLETE VIEW ON SMALL SCREENS (MOBILE PHONE)

\*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 web app in live.

Events by type



Events by city





**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 ensure everything was optimized as much as possible in order to facilitate future modifications, additions, or adjustments.

Adding comments and clarification 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

```
# 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 specific  
    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 this project. How can I make this project and these codes as clear as possible to promote their 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 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

---

### Skills used

---

Communication  
Content writing

# WHY WAS THIS STEP IMPORTANT

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

# WHAT WAS THE GOAL

Updating and completing the README file located in the Meet web app Github repository. The goal was to ensure that all relevant information regarding Meet web 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 at times. I also drew inspiration from other READMEs I've consulted for similar projects 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

