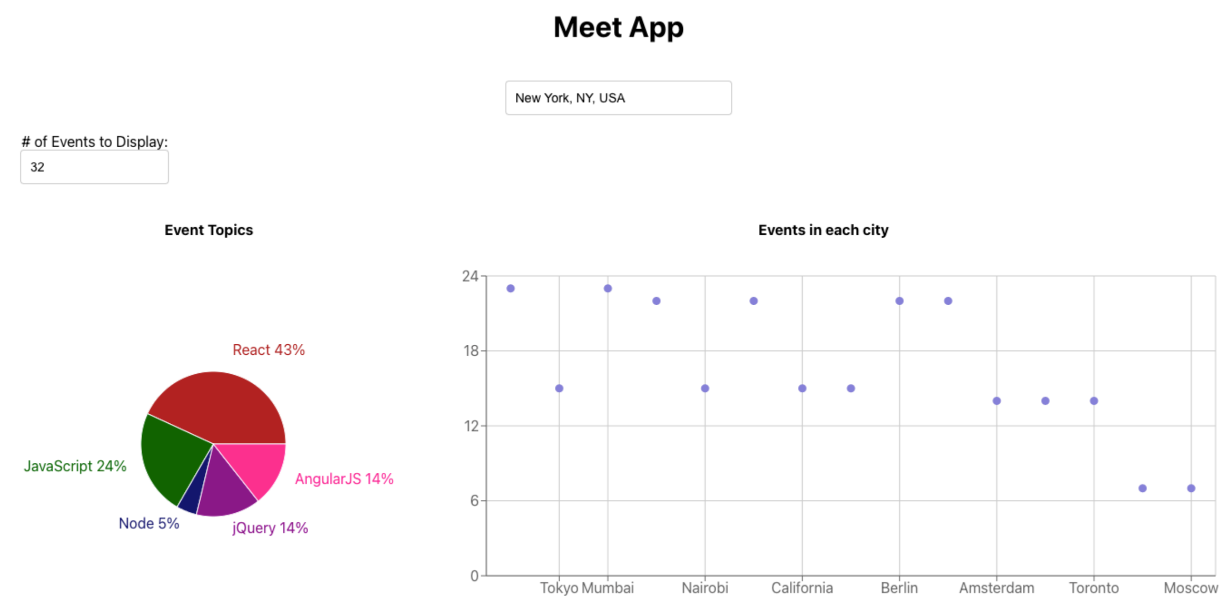
Meet App

A Test of Tests

Meet App is a Progressive, Serverless Web App built with React. It allows the user to see events happening in cities around the world and add them to the user's personal calendar.

Meet App is a project I completed in my Full Stack Immersion course at CareerFoundry. It was my first exposure to Test Driven Development (TDD) and allowed me to hone my burgeoning React skills. It was also my first time using AWS as the backend of an app, and my first time building a Progressive Web App (PWA).



Data Visualization by Recharts

I was the lead developer on all aspects of the project, with the invaluable support of my mentor Ted Walther and tutor Jay Quach. This project took about five weeks. My daughter had just been born the month prior, so progress was slower than normal. Despite this, I was able to get it done and learned a great deal, while also keeping the little one fed and happy.

The Process

Building the app involved many steps, which I will touch on briefly here. This will be followed by a more in depth discussion of some of the more interesting problems I encountered, in the "Project Highlights" section.

1. Write **User Stories**, using the Given-When-Then syntax, to map out the app's features.

2. Set up **Amazon Web Service (AWS)** to act as a serverless backend for the app. This allows the user to be authorized and authenticated to access the **Google Calendar API**.

3. Write Unit and Integration tests for the app's components with **Jest** and **Enzyme**, letting the tests drive the generation of the app's code (see the next section for more details).

4. Write Acceptance tests with **Cucumber** and an End-to-End test with **Puppeteer**. Run these tests to see if app functions properly.

5. Set up **Atatus** to monitor app performance for Continuous Integration and Delivery. Test app on friends and family and analyze performance stats.

6. Implement an alert function using **JavaScript** Object Oriented Programming (OOP).

7. Convert App into a Progressive Web App, using **Lighthouse** on Chrome DevTools to evaluate. Enable service worker and caching for offline use and customize "manifest.json" file with new app icon.

8. Add data visualization with **Recharts**. Generate a pie chart of the different event topics and a scatter chart showing event locations.

Project Highlights

Meet App was a complex project with many moving parts, and I would like to highlight three aspects that I found most illuminating and I hope will be interesting to the reader.

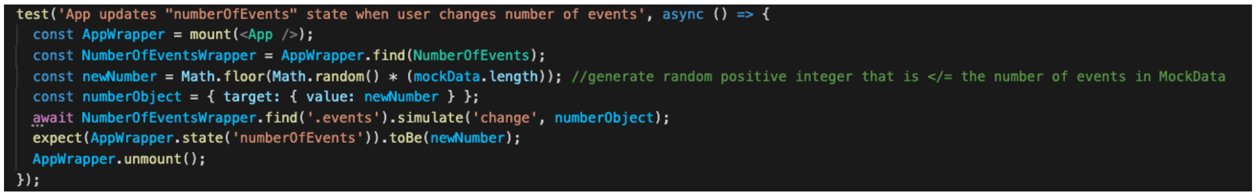
How I created Unit and Integration tests

This was my first time doing Unit and Integration tests, and, well, it was a doozy. The idea of Test Driven Development (TDD) is to

1. Figure out what you want the app to do,
2. Write tests to see whether it does those things
3. Write the code

In essence, you write the tests first, and then write the program. This has its advantages, but for me, the young developer, it felt a bit like dancing in the dark.

One challenge is that when you're writing tests for the front end, you are testing features that depend on user behavior, like pushing a button or entering text. However, you are testing these features with a computer program, which does not behave like a human. For example, computers are sometimes just too fast, so the test software might push all the buttons and fill in all the text fields before the site gets a chance to respond properly. This creates all sorts of *race conditions* where things are basically happening too fast and happen out of the intended order. To combat this, you have to use commands like "async await" to make the computer program chill out and give the site a few milliseconds to do its thing before proceeding to the next step.



```
test('App updates "numberOfEvents" state when user changes number of events', async () => {
  const AppWrapper = mount(<App />);
  const NumberOfEventsWrapper = AppWrapper.find(NumberOfEvents);
  const newNumber = Math.floor(Math.random() * (mockData.length)); //generate random positive integer that is </= the number of events in MockData
  const numberObject = { target: { value: newNumber } };
  await NumberOfEventsWrapper.find('.events').simulate('change', numberObject);
  expect(AppWrapper.state('numberOfEvents')).toBe(newNumber);
  AppWrapper.unmount();
});
```

Above is an example of a test I wrote that solved some of these issues. This one is testing whether the app knows when the user has typed a number of events into the number field on the site. To do this, I generate a random number and tell the program to simulate entering the number into the “Number of Events” field on the site. At this point, I use the “await” command to make sure the site has the time to respond. Once we’re all set, the test asks if the Meet App has the same number saved in its “state.” If so, we pass!

I like to use a random number generator in my tests instead of just picking a number myself. That way, every time I run the test I’m making sure that it doesn’t just work for some numbers and not for others. If there is a problem, it should eventually show up if I’m using the random number generator. It reflects the randomness of the world.

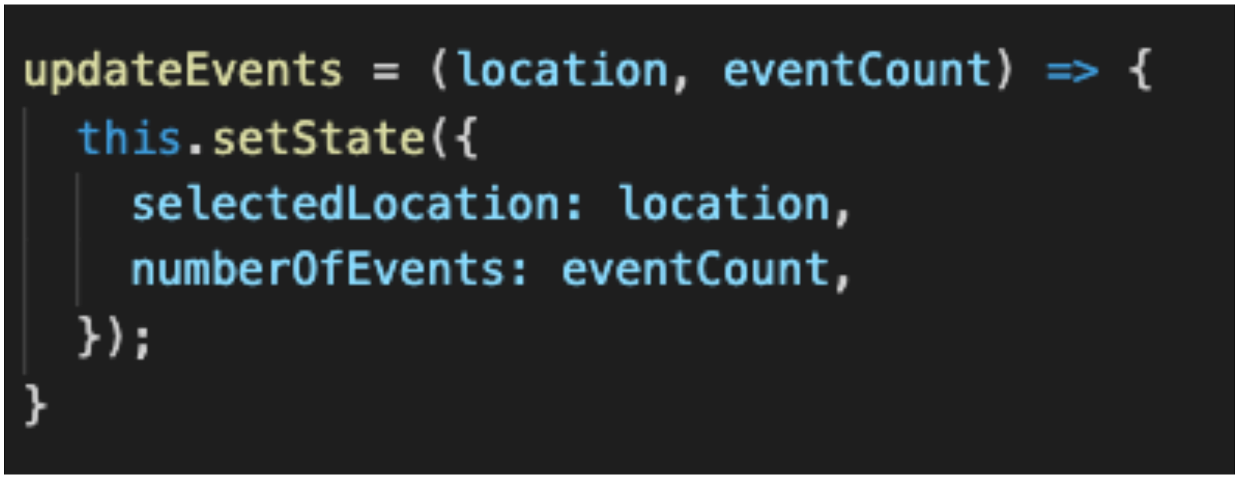
You also may notice a reference to MockData in the code. When we run tests involving data from an external source, we usually use “fake” data that we generate instead of constantly going to the external source for more data every time we test. I like to use bigger chunks of mock data for my tests to be sure that the test is dealing with a diverse range of information.

How I got my app to run at Super Speed

Creative Commons License

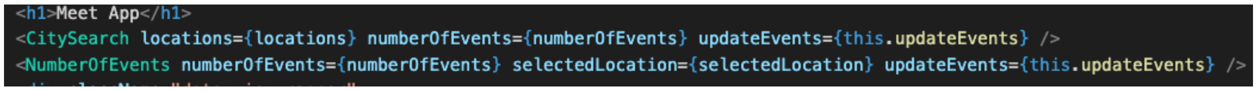
When I coded the app according to the guidelines of the course, it worked. As the user, I could select my city and number of events I wanted to see. However, the process was SUPER SLOW! The reason was that every time I entered or deleted even a single character in the “Number of Events” box, the “onChange” function was calling the external API to check for updates. So if I wanted to type the number “25” in the box, I would have to type “2”, wait about 2 seconds, type “5”, and then wait again.

In order to fix this, I rethought the flow of the entire app. Instead of constantly changing the list of events in the App state and updating every time the user moved a muscle, the app would save all the events just once, when the app first loaded. After that, the child components, “Number of Events” and “City Search” would handle filtering the events to fit the user’s specification, using JavaScript methods like “filter” and “slice”. All the while, the events would sit in the parent component’s state unchanged.



```
updateEvents = (location, eventCount) => {
  this.setState({
    selectedLocation: location,
    numberOfEvents: eventCount,
  });
}
```

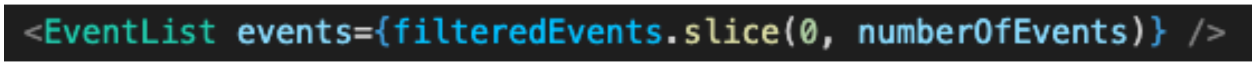
The updateEvents function was key to putting this app in the fast lane.



```
<h1>Meet App</h1>
<CitySearch locations={locations} numberOfEvents={numberOfEvents} updateEvents={this.updateEvents} />
<NumberOfEvents numberOfEvents={numberOfEvents} selectedLocation={selectedLocation} updateEvents={this.updateEvents} />
```

The updateEvents function is passed to the CitySearch and NumberofEvents components as a prop, so those components can change the app’s state.

To my considerable delight and relief, this made the app lightning fast! Because the app was basically operating offline after the initial load, anything the user wanted to do could be accomplished in a split second. Of course, a possible disadvantage is that the user is no longer getting updated event data every few seconds. However, since the nature of the data is not particularly time sensitive (at least not on a minute-by-minute basis) and the Google Calendar did not seem to be updated more than once a day at most, it seemed to me that speed was preferable to constant updates. After all, if you, the user, wanted “fresh out of the oven” event info, you could refresh the page at any time.



```
<EventList events={filteredEvents.slice(0, numberOfEvents)} />
```

When the events are passed to the EventList to be displayed, they are edited down to the selected number of events with the “slice” method. However, the complete list of events is left untouched in the App state so that the user can retrieve them again without going back to the API.

And one fun little feature

I always like to challenge myself to add a little something of my own that isn't required but is kind of fun and hopefully useful. In the app the user gets to pick how many events are displayed. However, if the user chooses 30 events and "London" but London only has 22 events scheduled, the user only sees 22 events. I wanted to make this clear with an alert that appears at the end of the list of events.

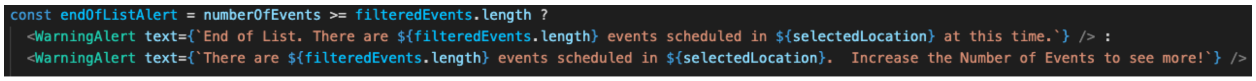
Depending on the number of events the user has chosen, it either reads something like:

End of List. There are 22 events scheduled in London, UK at this time.

OR (if the user has chosen to see fewer than 22 events)

There are 22 events scheduled in London, UK. Increase the Number of Events to see more!

This was achieved with a simple ternary expression. Nothing fancy, but I thought it added a little clarity to the user experience and encouraged the user to explore the app a little more.



```
const endOfListAlert = numberOfEvents >= filteredEvents.length ?
  <WarningAlert text={`End of List. There are ${filteredEvents.length} events scheduled in ${selectedLocation} at this time.`} /> :
  <WarningAlert text={`There are ${filteredEvents.length} events scheduled in ${selectedLocation}.  Increase the Number of Events to see more!`} />
```

Here's how I set up the end-of-list alert

In Summary…

I found this to be one of the more challenging projects, because the Test Driven Development process shook up my understanding and made me turn everything I knew upside down. I was happy to get through to the other side and implement some features of my own into the app. With more time, I would like to punch up the look of the app and find ways to make it even more efficient.

Technologies Used

- React
- Google Calendar API
- AWS Lambda
- Jest (testing)
- Enzyme (testing)
- Cucumber (testing)
- Puppeteer (testing)
- Atatus (performance evaluation)
- Lighthouse (Chrome DevTools)
- Recharts (data visualization)