

## SETUP

```
<!--Google Fonts-->
<link rel="preconnect" href="https://fonts.gstatic.com" />
<link
  href="https://fonts.googleapis.com/css2?family=Open+Sans&display=swap"
  rel="stylesheet"
/>
<link
  href="https://fonts.googleapis.com/css2?family=Open+Sans:wght@400;700&display=swap"
  rel="stylesheet"
/>
<title>Xtreme Weather Events</title>
```

The **dependencies** that we will be installing are as follows. It is worth mentioning that the user will also need to install **yarn** globally. This is because the penultimate **dependency** is only available via **yarn**. Plus, the first two **dependencies** are actually interdependent on each other.

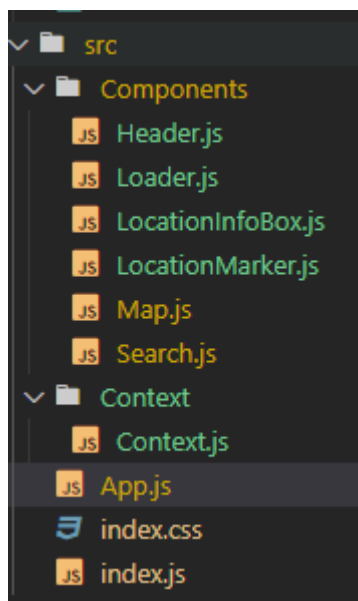
```
yarn add --dev @iconify/icons-emojione
yarn add --dev @iconify/react
yarn add google-map-react
yarn add supercluster use-supercluster
yarn add dotenv
```

We will also need to show of the NASA ‘EONET’ website to the user & the ‘iconify’ but remember that the icons we are using will be limited to the “emojione” selection, so make sure that is mentioned to the user.

**BUT DO NOT FORGET** the Google Map API KEY that we need. We will store it in a ‘dotenv’ file which is why we installed the **dependency** earlier on.

## REFERENCE

The finished FOLDER STRUCTURE looks like this.



## START

We will start with the “Context.js” file.

```
import React, {useContext, useState} from 'react';

const mainContext = React.createContext();

export function useMainContext() {
  return useContext(mainContext);
}
```

This allows children of the **contexts provider** to be able to access its value. They can import the hook, and so long as they are children of the **context provider**, they will have its value stored on the hook.

```
export function ContextProvider({children}) {
  //All of the data from NASA EONET
  const [eventData, setEventData] = useState([]);
  //Used to store the event that the user wants to go to
  const [selectedEvent, setSelectedEvent] = useState(null);
  //Need to re-render markers because user has changed filter option
  const [reRenderMarkers, setReRenderMarkers] = useState(null);

  const value = {
    eventData,
    setEventData,
    selectedEvent,
    setSelectedEvent,
    reRenderMarkers,
    setReRenderMarkers
  }

  return(
    <mainContext.Provider value={value}>
      {children}
    </mainContext.Provider>
  )
}
```

The functionality of these **states** will become clearer as I continue along with the application.

## MAP COMPONENT

This is where it will be rendered in the “App.js” file.

```

return (
  <div>
    <Header />
    {!loading ? <Map eventData={renderEvent} /> : <Loader />}
    {!loading && <Search />}
  </div>
);

```

At the moment though, none of this other LOGIC will be present. The only code available will be just the newly created MAP COMPONENT at the moment.

It will have these two **properties**.

```

function Map({center, eventData}) {

```

This “center” **property** will be assigned a default LONGITUDE & LATITUDE values.

```

Map.defaultProps = {
  center: {
    lat: 29.305561,
    lng: -3.981108
  }
}

```

This is the first **state** that we will be using. Out of many on the MAP COMPONENT.

```

const [zoom, setZoom] = useState(1);

```

We will then **import** all of these, some of which will be for future **reference**.

```

import GoogleMapReact from 'google-map-react';
import LocationMarker from './LocationMarker';
import useSuperCluster from 'use-supercluster';
import React, {useRef, useState, useEffect} from 'react';

```

We will then add the GOOGLEMAREACT COMPONENT to the **render statement**. As of present these are the **properties** that it will have.

Make note of the container DIV that it is inside of. We will be **styling** this so the MAP can have some **height** to it. Allowing us to actually see it.

```

return (
  <div className="map-container">
    <GoogleMapReact
      bootstrapURLKeys={{key: process.env.REACT_APP_GOOGLE_API_KEY}}
      center={center}
      zoom={zoom}
    />
  </div>
);

```

We begin the **styling** with the following then.

```
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
  font-family: "Open Sans", sans-serif;
}

body {
  background-color: #f2f2f2;
  overflow-x: hidden;
}
```

After we have added the basic “bread and butter” styling, we can style the MAP CONTAINER DIV.

```
/*Position Relative is important for location-info*/
.map-container {
  position: relative;
  width: 100vw;
  height: 600px;
}
```

## HEADER

Once we have gotten our MAP actually rendered to the screen, we want to give it a header so that the application looks more personalized.

The HEADER COMPONENT will be **rendered** in the “App.js” **component**, just above where the MAP COMPONENT is rendered.

```
return (
  <div>
    <Header />
  </div>
)
```

```
import {Icon} from '@iconify/react';
import fireIcon from '@iconify/icons-emojione/fire';

function Header(props) {
  return (
    <div className="header-bar">
      <Icon icon={fireIcon} /> Xtreme Weather Events
    </div>
  );
}

export default Header;
```

```
.header-bar {
  width: 100vw;
  background-color: firebrick;
  color: white;
  text-align: center;
  padding: 10px;
  font-size: 20px;
}
```

## LOCATION MARKER

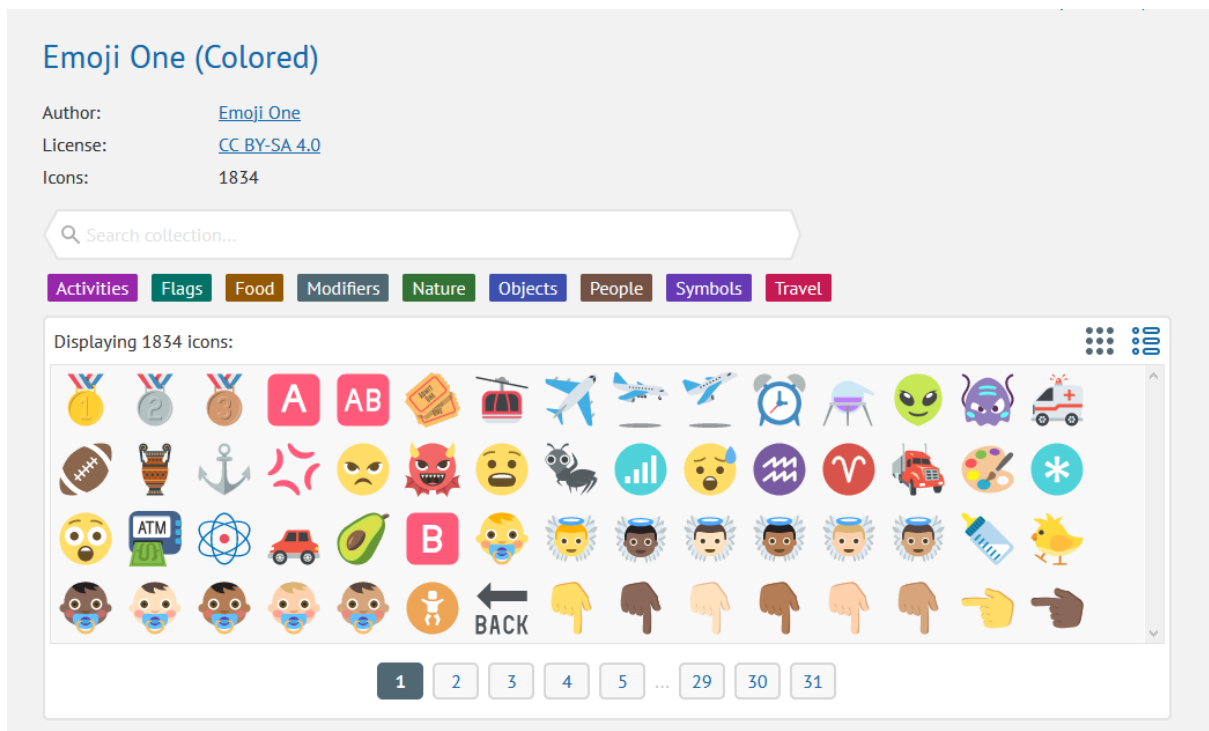
```
components > @iconify > LocationMarker
✓ import React from 'react';
  import {Icon} from '@iconify/react';
  import fireIcon from '@iconify/icons-emojione/fire';
  import volcanoIcon from '@iconify/icons-emojione/volcano';
  import stormIcon from '@iconify/icons-emojione/cloud-with-lightning-and-rain';
  import iceIcon from '@iconify/icons-emojione/snowflake';

✓ function LocationMarker({lat, lng, onClick, id}) {
  let renderIcon = null;
  ✓ if(id === 8){
    renderIcon = fireIcon
  ✓ }else if (id === 10){
    renderIcon = stormIcon
  ✓ }else if (id === 12){
    renderIcon = volcanoIcon
  ✓ }else if (id === 15){
    renderIcon = iceIcon
  }

  return (
  ✓ <div onClick={onClick}>
    <Icon icon={renderIcon} className="location-icon" />
    </div>
  );
}

export default LocationMarker;
```

\*Show the audience the “iconify” “emojione” selection



\*Mention to the user that we are not going to assign any logic to the “lat” & “lng” **properties** ourselves. We have added them to represent that their presence on the COMPONENT, however the logic will be handled by the “google-map-react” **dependency** and not us.

```
.location-icon {
  font-size: 2rem;
  position: relative;
}
```

Now inside of the GOOGLEMAREACT **component** we will add a single MARKER COMPONENT.

```
return <LocationMarker
  lat={latitude}
  lng={longitude}
  id={clusterId}
```

Keep in mind that these **values** will not currently be used since they do not exist as of yet. Instead we will use the necessary **properties** of the “center” PROPERTY on the MAP COMPONENT. As for the **id** it does not matter which number we use, so long as we use a number that it does accept.

This MARKER is simply there to show the user what one looks like.

## EONET NASA API

Back in the “App.js” file we are going to create these 2 **states**.

```
const [loading, setLoading] = useState(false);
//Event to render
const [renderEvent, setRenderEvent] = useState([]);
```

We will also import the following two things from the **global context**. The first one is a METHOD and the second one is currently **null**.

```
const { setData, reRenderMarkers } = useMainContext();
```

Below this we will create a **USEEFFECT** hook where we use the **fetch** API to send a GET request to the EONET API.

```
useEffect(() => {  
  const fetchEvents = async () => {  
    setLoading(true);  
    const res = await fetch("https://eonet.sci.gsfc.nasa.gov/api/v2.1/events");  
    //Extract the Array contained in the 'events' field.  
    const {events} = await res.json();  
    //Event data is globally accessible. But 'renderEvent' is just to render out the MAP with the markers  
    setData(events);  
    reRenderMarkers(events);  
    setLoading(false);  
  }  
  fetchEvents();  
}, [])
```

\*Explain to the user that there is some data in the JSON OBJECT retrieved from the API which we do not want. We want all of the data from the “events” Array hence the OBJECT DESTRUCTURING.

Refer back to the API.

```
title: "EONET Events"  
description: "Natural events from EONET."  
link: "https://eonet.sci.gsfc.nasa.gov/api/v2.1/events"  
▼ events:  
  ▼ 0:  
    id: "EONET_5300"  
    title: "Tropical Storm Surigae"  
    description: ""  
    ▼ link: "https://eonet.sci.gsfc.nasa.gov/api/v2.1/events/EONET_5300"  
    ▼ categories:  
      ▼ 0:  
        id: 10  
        title: "Severe Storms"  
    ▼ sources:  
      ▼ 0:  
        id: "GMAES"
```

\*Also explain the difference between the **eventData** STATE and the **renderEvent** STATE. The **eventData** is a global store of this “events” field Array which remains constant throughout the application. The **renderEvent** STATE is not GLOBAL and it will change throughout the application to reflect the necessary markers.

### Loader Component

We will now create the **Loader** component which is what we set up the **loading** STATE for.

```
import React from 'react';

function Loader(props) {
  return (
    <div className="loader-container">
      <div className="loader"></div>
    </div>
  );
}


export default Loader;
```

```
.loader-container {
  position: absolute;
  top: 0;
  left: 0;
  display: flex;
  justify-content: center;
  align-items: center;
  width: 100%;
  height: 100%;
}

.loader {
  width: 100px;
  height: 100px;
  border-radius: 50%;
  border: 10px solid #e6e6e6;
  border-top-color: #808080;
  animation: spin 1s linear infinite;
}

@keyframes spin {
  0% {
    transform: rotate(0deg);
  }
  100% {
    transform: rotate(360deg);
  }
}
```

Now we can do the following in our App.js file. In the **render** statement of course.

```
header ? 
{!loading ? <Map eventData={renderEvent} /> : <Loader />}
13 |   }
    |   ^
```



Back in the “Map.js” file we will remove the LOCATION MARKER COMPONENT from the **render** statement. We no longer need it as it is time to render the **markers** dynamically based of the EVENTDATA **property** and the audience has already seen an example of the **marker**.

```
return <LocationMarker  
  { ...props } />
```

Below the **state** VARIABLES we will declare the following **object**. Explain that it is basically an INDEX for deciphering the meaning of the ID's of the EVENT'S from the API.

```
//Index for reference  
const eventDataIndex = {  
  8: "Wildfires",  
  10: "Severe Storms",  
  12: "Volcanoes",  
  15: "Sea and Lake Ice"  
}
```

Below this, we will create an Array consisting of the KEYS of the **object** so in this case it will just be the numbers. Except that the method returns the KEYS as **strings** which is why we use the MAP method so that each item is a number.

```
//Create an Array of its keys  
let eventDataIndexNum = Object.keys(eventDataIndex);  
eventDataIndexNum = eventDataIndexNum.map(index => Number(index));
```

We will now turn each item in the **eventData** Array **property** into a “GEO FEATURE” **object**. This is merely an object that follows a certain format and is used for the CLUSTERING later on.

*Explain that the “cluster” Boolean is important because it will be set to TRUE if the **Markers** are close together hence it will be a **cluster**. If it is just a **Marker** on its own, not close to any others then the Boolean will be FALSE and it will not be a **cluster** it will just be an individual **Marker**.*

```
//Set up the geo-features. Do not need them anymore.  
const points = eventData.map(event => ({  
  "type": "Feature",  
  "properties": {  
    "cluster": false,  
    "eventKey": event.id,  
    "eventTitle": event.title,  
    "eventType": event.categories[0].id  
  },  
  "geometry": { "type": "Point", "coordinates": [event.geometries[0].coordinates[0], event.geometries[0].coordinates[1]] }  
}));
```

Adjacent to where we declared the “zoom” **state** we will declare the “bounds” **state** and the “map” **ref** which will give us the Google Map as a JavaScript function.

```
const mapRef = useRef();  
const [zoom, setZoom] = useState(1);  
const [bounds, setBounds] = useState(null);
```

Now we can use the “useSuperCluster” **hook** to extract both the “supercluster” and the “clusters” using OBJECT DESTRUCTURING.

Explain that the “bounds” is so the clusters know whether the **markers** are concentrated or sparse so that the **super cluster** can determine whether to return a **cluster** or not.

```
//Get clusters
const [{clusters, supercluster}] = useSuperCluster({
  points,
  bounds,
  zoom,
  options: {radius: 75, maxZoom: 20}
});
```

Next, we add the following **properties** to the GOOGLEMAREACT COMPONENT.

```
yesIWantToUseGoogleMapApiInternals
onGoogleApiLoaded=(({map}) => {
  mapRef.current = map;
})
onChange=(({zoom, bounds}) => {
  setZoom(zoom);
  setBounds([
    bounds.nw.lng,
    bounds.se.lat,
    bounds.se.lng,
    bounds.nw.lat
  ]);
})
```

Inside of the GOOGLEMAREACT COMPONENT we will use the MAP METHOD on the **clusters**.

```
{clusters.map(cluster => {
  const [longitude, latitude] = cluster.geometry.coordinates;
  const {cluster: isCluster, point_count: pointCount} = cluster.properties;
  //Used for icon type
  const clusterId = cluster.properties.eventType;
  if (isCluster){
```

These are the values that we DESTRUCT. Show the user the GEO FEATURE **object** and where they come from.

Inside the IF block we will render out our own JSX COMPONENTS which will represent the **clusters**. We know that it is a **cluster** because the Boolean was true.

(Note that the closing DIV is not shown here because the ONCLICK **attribute** is below which we do not want to create as of yet.)

```
const clusterId = cluster.properties.eventType;
if (isCluster){
  let changeSize = Math.round(pointCount / points.length * 100);
  //Can't exceed 40 px
  let addSize = Math.min(changeSize * 10, 40);
  return (
    <section key={cluster.id} lat={latitude} lng={longitude}>
      <div className="cluster-marker" style={{
        width: `${addSize + changeSize}px`,
        height: `${addSize + changeSize}px`
      }}
    </div>
  </section>
  )
}
```

It will look like this though with the “pointCount” **number** stored inside.

```
    <div className="cluster-marker" style={{
      width: `${addSize + changeSize}px`,
      height: `${addSize + changeSize}px`
    }}
  </div>
</section>
```

Here is the **styling**.

```
.cluster-marker {
  color: ■ #fff;
  background: ■ firebrick;
  border-radius: 50%;
  padding: 10px;
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Below this & still inside of the MAP METHOD we will catch the scenario where the Boolean is set to false and the **cluster** is returning an individual Marker which needs to be rendered as one & not a **cluster**.

The code in the IF parenthesis is merely for error checking to filter out any MARKERS that might have missing data.

```
//Not a cluster. Just a single point
if(eventDataIndexNum.indexOf(clusterId) !== -1 && cluster.geometry.coordinates.length === 2){
  return <LocationMarker
    lat={latitude}
    lng={longitude}
    id={clusterId}
    key={cluster.properties.eventKey}
  />
}
```

It is time now to create that ONCLICK **attribute** on the cluster component.

The **super cluster** HOOK calculates the zoom level for us automatically and we set the PAN of the map in relation to the zoom so that the MAP does not zoom in on a random point but on where the **cluster** is located.

```

    <section key={cluster.id} lat={latitude} lng={longitude}>
      <div className="cluster-marker" style={{
        width: `${addSize + changeSize}px`,
        height: `${addSize + changeSize}px`
      }}
      onClick={() => {
        const expansionZoom = Math.min(
          supercluster.getClusterExpansionZoom(cluster.id),
          20
        );
        mapRef.current.setZoom(expansionZoom);
        mapRef.current.panTo({lat: latitude, lng: longitude});
      }}>

```

## LOCATION INFO BOX

We want some information to be shown to the user when they click on a MARKER.

The “LocationInfoBox.js” file will look as follows.

```

import React from 'react';

function LocationInfoBox({info}) {
  return (
    <div className="location-info">
      <h2>Event Location Info</h2>
      <ul>
        <li>ID: <strong>{info.id}</strong></li>
        <li>TITLE: <strong>{info.title}</strong></li>
      </ul>
    </div>
  );
}

export default LocationInfoBox;

```

```

.location-info {
  position: absolute;
  top: 40px;
  right: 50px;
  width: 400px;
  min-height: 200px;
  padding: 20px;
  background-color: rgba(0, 0, 0, 0.6);
  border-radius: 10px;
  font-size: 18px;
  color: #fff;
}

.location-info ul {
  list-style: none;
  padding: 0;
}

.location-info li {
  padding: 5px 0;
}

```

We will now add one more **state** variable to our “Map.js” file.

```

//Info Box
const [locationInfo, setLocationInfo] = useState(null);

```

Add the following to the **render** statement.

```

</GoogleMapReact>
{locationInfo && <LocationInfoBox info={locationInfo} />}
iv>

```

Add the following to the ONCLICK **attribute** of the “LocationMarker” COMPONENT.

```

onClick={() => {
  setLocationInfo({id: cluster.properties.eventKey, title: cluster.properties.eventTitle})
}}

```

Add the following to the “GoogleMapReact” COMPONENT so that it will disappear again when the user clicks of the MARKER.

```

onClick={() => {setLocationInfo(null)}}
onDrag={() => setLocationInfo(null)}

```

## SEARCH COMPONENT

In the “Search.js” file we will create two **references**.

```

const searchBox = useRef();
const optionBox = useRef();

```

Then we will add the following to the **render** statement.

```
<section className="search-container">
  <p>Search:</p>
  <input type="text"
  // ...
  </input>
  // ...
  </section>
```

Below, but still inside the **render** statement.

```
<table className="search-table">
  <tr>
    <th style={{width: "60%"}}>Title</th>
    <th>Type</th>
    <th>Location</th>
  </tr>
  [REDACTED]
</table>
```

```

/*Font Styling for browser compatibility*/

button,
input,
select,
✓ textarea {
  font-family: inherit;
  font-size: 100%;
}

✓ .search-container input[type="text"] {
  font-family: inherit;
  outline: none;
  width: 100%;
  max-width: 500px;
  margin: 8px 20px 0 0;
  padding: 0 0 0 10px;
}

✓ .search-table {
  margin: 0 auto;
  table-layout: fixed;
  border-collapse: collapse;
  width: 100%;
  max-width: 1000px;
  text-align: left;
}

✓ .search-table th {
  background-color: ■ firebrick;
  color: ■ white;
}

✓ .search-table tr:nth-child(even) {
  background-color: ■ #cccccc;
}

✓ .search-table th,
td {
  padding: 10px;
}

```

We will now extract the following from the **global context**.

```
const {eventData, setSelectedEvent, setReRenderMarkers} = useMainContext();
```

Also create this **state**.

```
//Matching results
const [matchEvent, setMatchEvent] = useState(eventData);
//HandleKeyUpEvent
```

This method will be called on the ONKEYUP **property** on the text input.

```
const userSearch = (searchQuery, eventData) => {
  let eventMatch = [];
  //Redacted line
  if(searchQuery.length > 0 && eventData ){
    for(const event in filteredEventData){
      let eventTitle = filteredEventData[event].title.toLowerCase();
      if(eventTitle.indexOf(searchQuery) !== -1){
        eventMatch.push(filteredEventData[event]);
      }
    }
    //If they have typed in something but it didn't match
    if(eventMatch.length === 0){
      eventMatch = [{title: "No Results!", categories: [{title:""}]}]
    }
    setMatchEvent(eventMatch);
  }else{
    setMatchEvent(filteredEventData);
  }
}
```

As I said before we will add this. The twist is it will only be called if the user has not changed the text **inputs** value in .3 of a second. This gives the **database** some breather space, so that it is not changing its size in rapid succession.

```
<input type="text" onKeyUp={() => {
  let searchQuery = searchBox.current.value.toLowerCase();
  //Want to wait for the user to finish typing before sending method
  setTimeout(() => {
    if(searchQuery === searchBox.current.value.toLowerCase()){
      userSearch(searchQuery, eventData);
    }
  }, 300)
},
  ref={searchBox} />
```

Then we will add the following to the TABLE inside the **render** statement.



```

    </tr>
    {matchEvent.map(ev => {
      return(<tr key={ev.id}>
        <td>{ev.title}</td>
        <td>{ev.categories[0].title}</td>
        {ev.categories[0].title ? <td><a href="#"
        onClick={() => {setSelectedEvent(ev)}}>Click Here</a></td> : <td></td>}
      </tr>)
    })}
  </table>

```

Now we want to go to the MARKER that they have just clicked on in the **table**.

We go to our “Map.js” file and add the following from the **global context**.

```

const {selectedEvent} = useMainContext();

```

Then we create this USEEFFECT hook.

```

//User has clicked on searched link. They want to go to it
useEffect(() => {
  if(selectedEvent !== null){
    let longitude = selectedEvent.geometries[0].coordinates[0];
    let latitude = selectedEvent.geometries[0].coordinates[1];
    mapRef.current.panTo({lat: latitude, lng: longitude});
    mapRef.current.setZoom(10);
  }
}, [selectedEvent])

```

Back in our “Search.js” file we will give the user the option to filter out **search** results based on the category of **event**.

We create this **state**.

```

//Handle dropDown
const [storeSelection, setStoreSelection] = useState("All");

```

This will be the very first JSX content in the **return** statement now.

```

return (
  <>
    <section className="option-container">
      <p>Type:</p>
      <select ref={optionBox}
        onChange={() => {setStoreSelection(optionBox.current.value)}}>
        <option value="All">All</option>
        <option value="Wildfires">Wildfires</option>
        <option value="Severe Storms">Severe Storms</option>
        <option value="Volcanoes">Volcanoes</option>
        <option value="Sea and Lake Ice">Sea and Lake Ice</option>
      </select>
    </section>
  </>
)

```

The final styling.

```

.option-container {
  display: flex;
  justify-content: center;
  margin-top: 20px;
}

.option-container p {
  margin-right: 10px;
}

```

We will create the method inside the **SEARCH component**. It will allow us to filter out the **events** based on their category.

```

//Filter eventData
const filterEventData = eventData => {
  //Spread operator so we don't overwrite Reference data
  let filteredEventData = [...eventData];
  if(storeSelection !== "All"){
    filteredEventData = filteredEventData.filter(event => event.categories[0].title === storeSelection);
  }
  return filteredEventData;
}

```

Then we will add this method to the **event handler** for when the user releases their key on the text input.

```

const userSearch = (searchQuery, eventData) => {
  let eventMatch = [];
  let filteredEventData = filterEventData(eventData);
}

```

Finally, we will create the following **USEEFFECT hook** that will be called when the user changes the “storeSelection” **state** which will happen when they click on a new “option” in the “select” element.

```
//They have changed their filter option. We want the markers to change aswell.
useEffect(() => {
  //First we want to sort out the Markers
  let filteredEventData = filterEventData(eventData);
  setReRenderMarkers(filteredEventData);
  //Now we want to sort out the search results
  userSearch(searchBox.current.value.toLowerCase(), filteredEventData);
}, [storeSelection])
```

The “setReRenderMarkers” method from the **global state** is used because we want to render out the actual MARKERS on the MAP to reflect this filter.

So back in the “App.js” file we **import** the necessary data from the **global context**. And use it to create this USEEFFECT **hook** that will be called when the user changes the “reRenderMarkers” **global state**.

```
const { setData, reRenderMarkers } = useMainContext();
```

```
//Need to re-render markers to reflect filter option
useEffect(() => {
  if(reRenderMarkers !== null){
    setRenderEvent(reRenderMarkers);
  }
}, [reRenderMarkers])
```