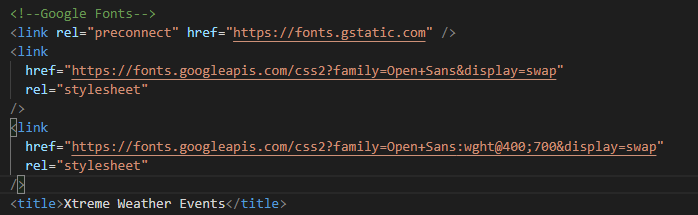
**SETUP**



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

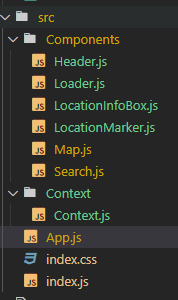
yarn add dotenv

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.

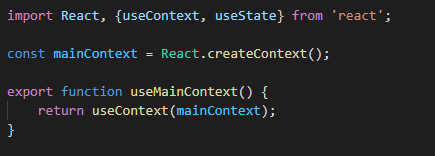
**REFERENCE**

The finished FOLDER STRUCTURE looks like this.

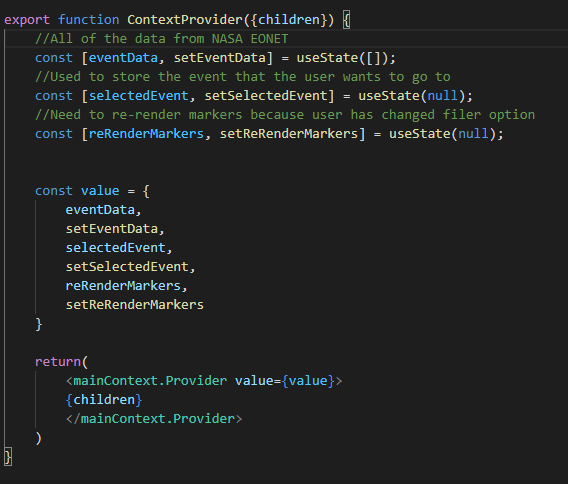


**START**

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



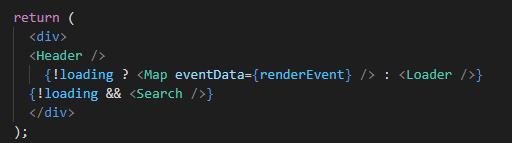
*This allows children of the* ***contexts provider*** *to be able to access its value. They can import the hook, and so long as they are children of the* ***context provider,*** *they will have its value stored on the hook.*



*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.

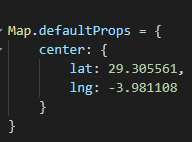


At the moment though, none of this other LOGIC will be present. The only code available will be just the newly created MAP COMPONENT at the moment.

It will have these two **properties.**



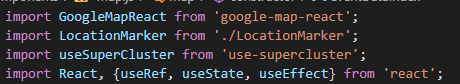
This “center” **property** will be assigned a default LONGITUDE & LATITUDE values.



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

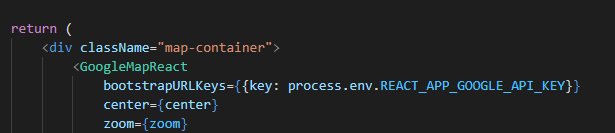


We will then **import** all of these, some of which will be for future **reference**.

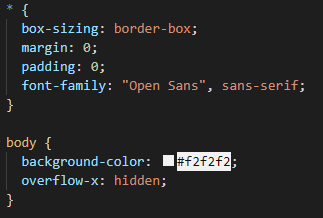


We will then add the GOOGLEMAPREACT COMPONENT to the **render statement.** As of present these are the **properties** that it will have.

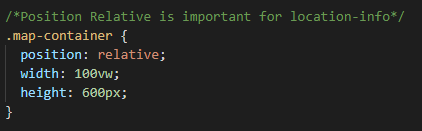
Make note of the container DIV that it is inside of. We will be **styling** this so the MAP can have some **height** to it. Allowing us to actually see it.



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



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

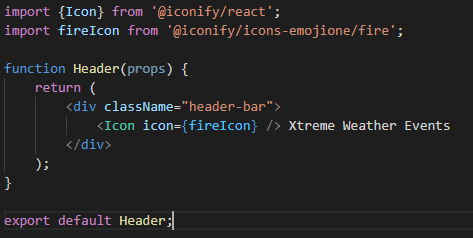


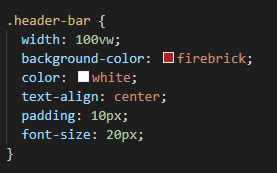
**HEADER**

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.

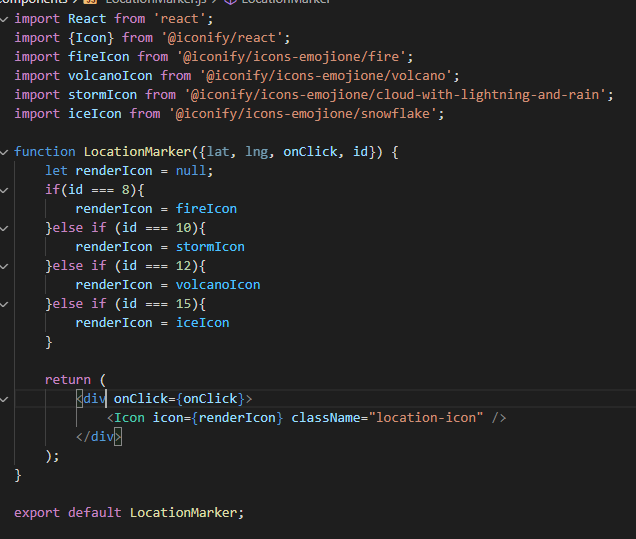
The HEADER COMPONENT will be **rendered** in the “App.js” **component**, just above where the MAP COMPONENT is rendered.



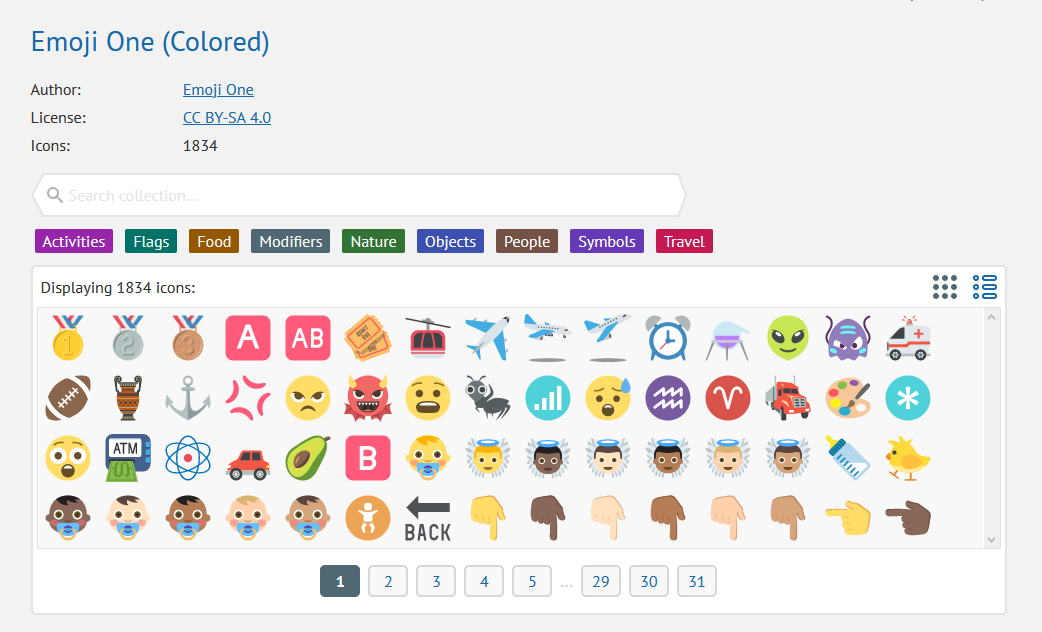




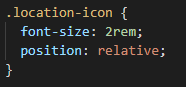
**LOCATION MARKER**



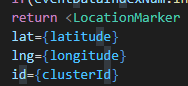
\*Show the audience the “iconify” “emojione” selection



\*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.



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



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**

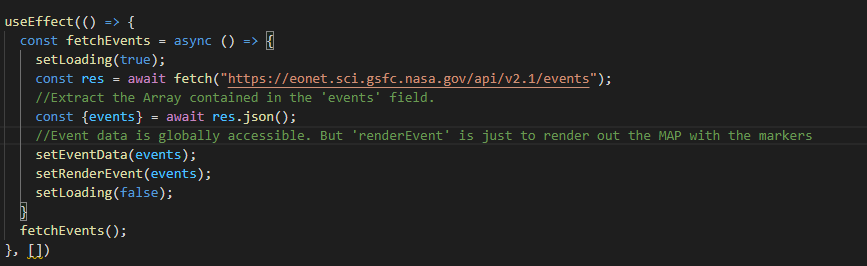
Back in the “App.js” file we are going to create these 2 **states**.



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**.

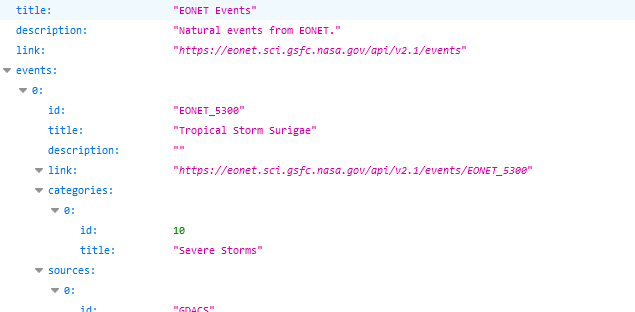


Below this we will create a USEEFFECT **hook** where we use the **fetch** API to send a GET request to the EONET API.



\*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.

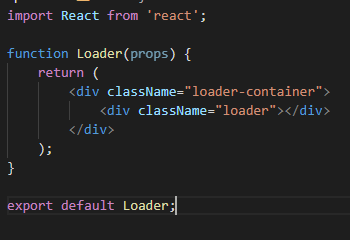
Refer back to the API.



\*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.





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

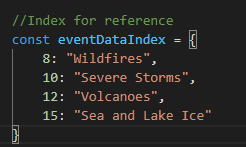


**CLUSTERS**

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**.



Below the **state** VARIABLESwe 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.

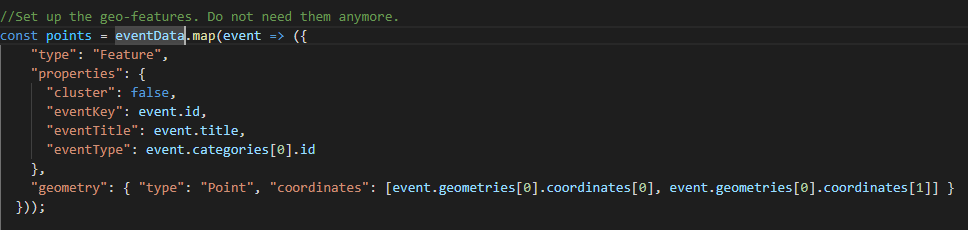


Below this, we will create an Array consisting of the KEYS of the **object** so in this case it will just be the numbers. Except that the method returns the KEYS as **strings** which is why we use the MAP method so that each item is a number.

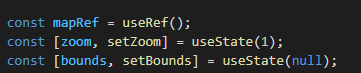


We will now turn each item in the **eventData** Array **property** into a “GEO FEATURE” **object**. This is merely an object that follows a certain format and is used for the CLUSTERING later on.

*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****.*

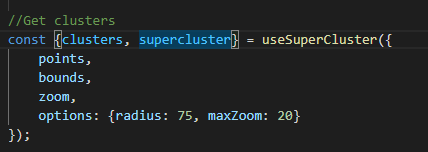


Adjacent to where we declared the “zoom” **state** we will declare the “bounds” **state** and the “map” **ref** which will give us the Google Map as a JavaScript function.

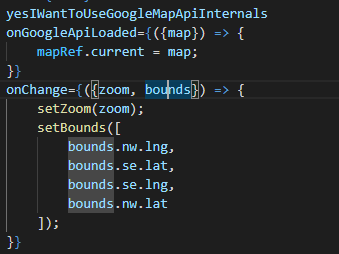


Now we can use the “useSuperCluster” **hook** to extract both the “supercluster” and the “clusters” using OBJECT DESTRUCTURING.

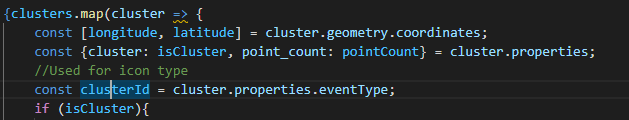
*Explain that the “bounds” is so the clusters know whether the* ***markers*** *are concentrated or sparse so that the* ***super cluster*** *can determine whether to return a* ***cluster*** *or not.*



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



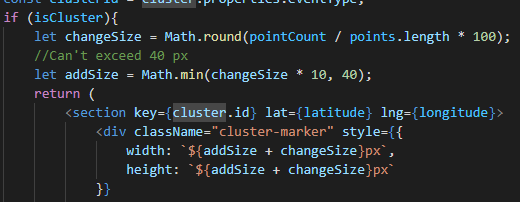
Inside of the GOOGLEMAPREACT COMPONENT we will use the MAP METHOD on the **clusters**.



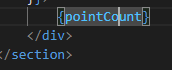
These are the values that we DESTRUCT. *Show the user the GEO FEATURE* ***object*** *and where they come from*.

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.

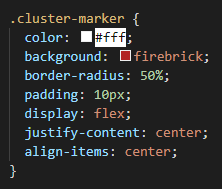
(Note that the closing DIV is not shown here because the ONCLICK **attribute** is below which we do not want to create as of yet.)



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

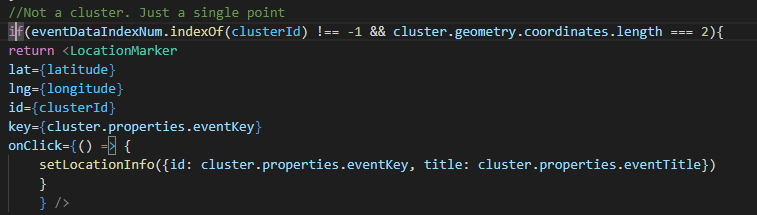


Here is the **styling**.



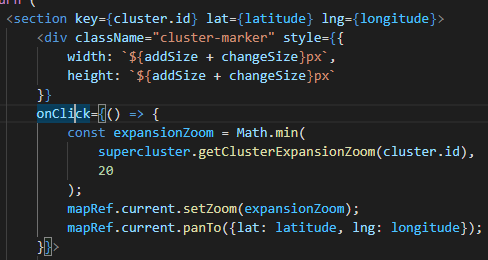
Below this & still inside of the MAP METHOD we will catch the scenario where the Boolean is set to false and the **cluster** is returning an individual Marker which needs to be rendered as one & not a **cluster.**

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



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

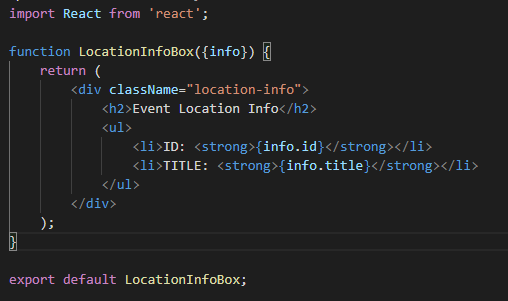
The **super cluster** HOOK calculates the zoom level for us automatically and we set the PAN of the map in relation to the zoom so that the MAP does not zoom in on a random point but on where the **cluster** is located.

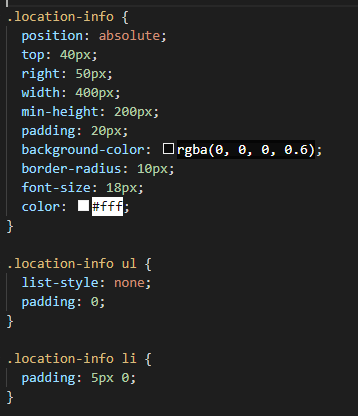


**LOCATION INFO BOX**

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

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





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



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



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



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

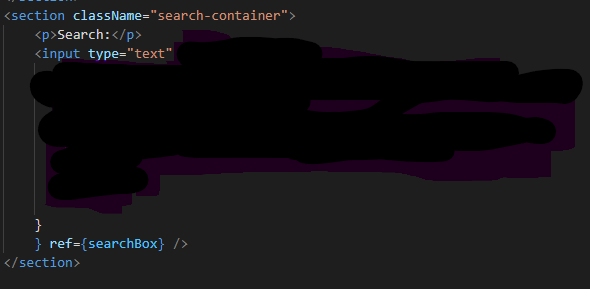


**SEARCH COMPONENT**

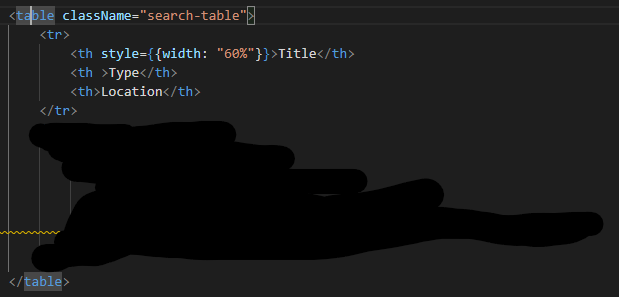
In the “Search.js” file we will create two **references**.

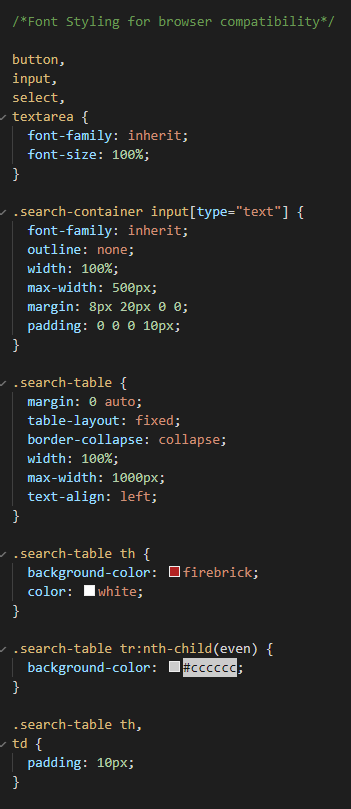


Then we will add the following to the **render** statement.



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





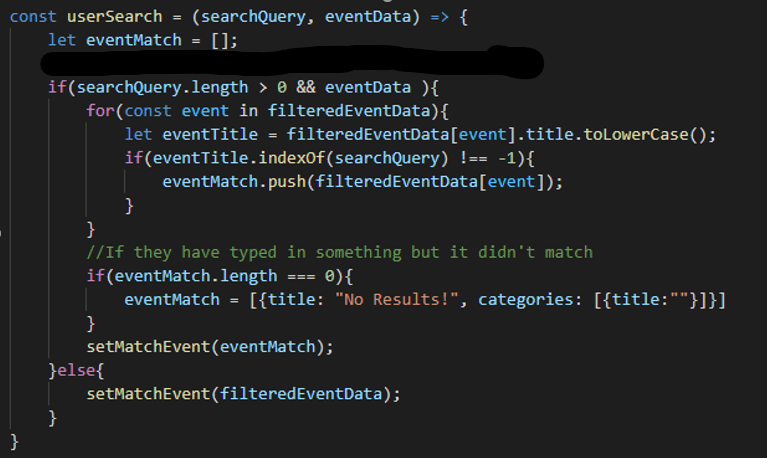
We will now extract the following from the **global context**.



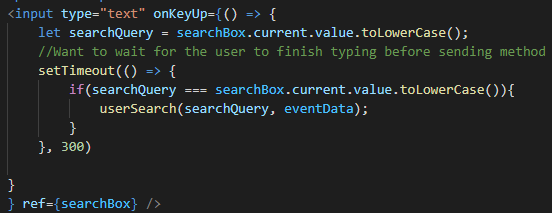
Also create this **state**.



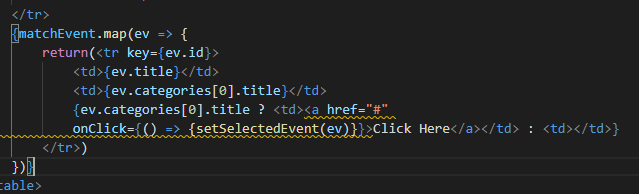
This method will be called on the ONKEYUP **property** on the text input.



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.



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

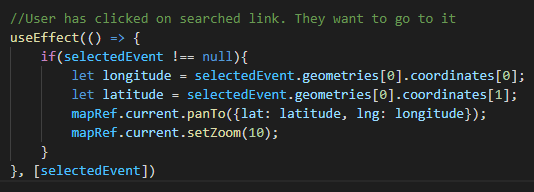


Now we want to go to the MARKER that they have just clicked on in the **table**.

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



Then we create this USEEFFECT **hook**.

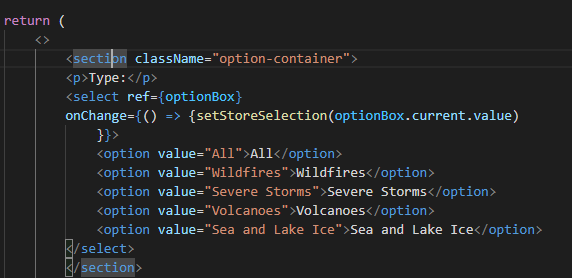


Back in our “Search.js” file we will give the user the option to filer out **search** results based on the category of **event**.

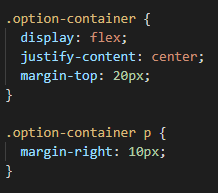
We create this **state**.



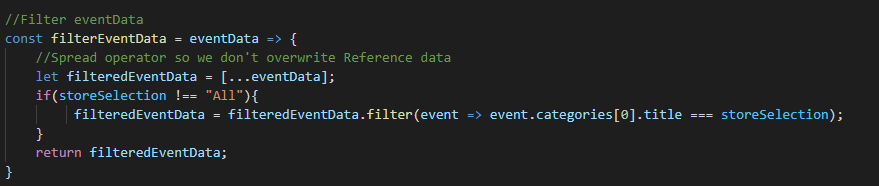
This will be the very first JSX content in the **return** statement now.



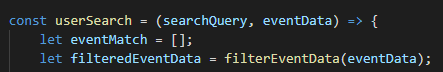
The final **styling**.



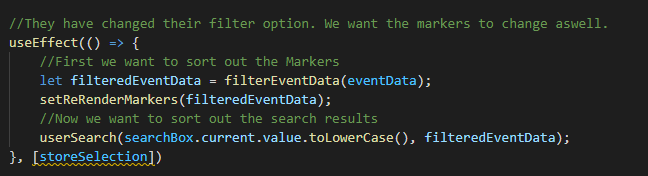
We will create the method inside the SEARCH **component**. It will allow us to filter out the **events** based on their category.



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



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.



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**.



