**Smart City Website**

**Cloud Nine**

The website is being developed in cloudnine. This is an online environment. It has a linux terminal used to execute commands. This is a free website. The link is: <https://ide.c9.io/miguelkulisic/smartcity_webpage>

**Project Structure**

The project is divided into many folders. The in the top folder there are two very important files. App.js calls joins all the files together and handles the backend. Package.json holds important information about the program like the dependencies. The views directory holds the ejs files. This file is like an html. They need to be kept in this directory, because express expects them to be there. The node\_modules directory is where all the downloaded packages are saved. Inside the public directory are all css or javascript files. The name “public” can be changed, but it is convention to call it public.

**Pothole Markers**

The different colored markers are images that I found on the internet. The colors used for these markers do not match a severity scale very good. It would be good to use photoshop to alter their colors and then go and change the colors in the colors\_three and colors\_ten arrays. The images are in the CustomMarkers directory inside the public directory.

**Styling**

The file css file responsible for some of the styling can be found in the public directory. This file only compliments the styling that the libraries provide. There are two styling libraries which both get linked in the head of the ejs file (via cdn). The first one is Bootstrap and this one provides the styling for the top part of the page (drop down menus and title). It also creates a grid for positioning of the elements when handling different screen sizes. This would be to hard and annoying to do without bootstrap, since css is notorious for being bad with positioning. The second cdn is for Font Awesome and this library has a lot of icons we can use. This icons are better than pictures, because they act as letters so you can style them and resize them however needed.

**Functionality**

The project works thanks to node.js. This technology is serverside javascript. This makes it possible to program most of the website purely on javascript. This also creates the project structure for me. The second thing that gives the website structure is the package express. This package pieces the different parts of the project together. The code in express is not likely to change.

To run the program go into the linux terminal and navigate to the main folder. This is where app.js is. Type in the command “node app.js” in the terminal. This will initialize the program. If it starts successfully it will print a success message. To then view the website. Click on preview (to right) and hit “Preview running application”. The code is now running on a server provided by cloudnine. **THIS SERVER IS JUST FOR TESTING AND NOT FOR TOO MANY PEOPLE TO CONNECT TO IT.**

**Expected Data Structure**

This code works under the assumption that it will receive an array that has objects in its elements. Example :

Potholes = [{

month: “06”, //name of the month would also work

year : “2017”, //could be a string or a number

latlong: "40.420462, -86.910372", //could be two separate fields

sev: 1,

}]

As indicated in the comments this data can be received in many formats and changing from one format to the other is not hard. The only important thing is to keep whoever is still programming informed about the new data structure. The data should still be saved in a JSON file in the server and it should maintain the structure of an array with objects as elements. The keys in the objects could be changed easily.

**App.js**

The first line calls express. The require function is part of server side javascript. I pass this function over to the variable app. This name can be changed, but is common to call it app. The name app.js is also a common name for this file. App.use tells the express to look at the “public” directory when searching for css or js files. App.get is telling the server what to send when it gets a request to the desired location. The first input symbolizes the path. “/” means the main page. For example this would be the response you would get if you type [www.google.com](http://www.google.com). The code then sends back the file “smart\_city.ejs”. The last line says where this code is going to run. The first to inputs have to be changed when the code runs in our server. Those two inputs tell cloud nine to run it in it’s server.

**Smart\_city.ejs**

At this moment this file contains mostly html code and some scripts. It is a ejs file, because it might come in handy later on in development and there is no real reason not to. Ejs is just like html, but it can take in variables like a javascript file. The data may be passed through the request to the ejs file and from there it can be passed to the javascript file. In the head of the file the css and some js files are linked. They don’t need to have an absolute address, because we already told express where to look for them. There are three dropdown menus. This three may seem empty at first, but they get their contents from the data. This code for this is in another file. At the bottom of the page is the script that contains the google maps api. The src field will have to be changed when you get a professional google maps api key.

**Smart\_city.js**

This file is in the public directory. The first thing you’ll see is the global variable map. If possible this should be the only global variable (easier to keep track of your program). It contains the contents of the map and needs to be used by many function.

**refresh(x)**

This function is responsible for calling Pothole\_object\_generator (Pothole\_data) and draw routes. Also on load of the page it will also call the function that generate the two dropdown menus. This is handled by the input “x”. At this moment this function holds the test data and passes it to the other function. In the future the data will be collected from a different function and stored in global variable. This means that the “Pothole\_data” input in many of the function may not be necessary in the future.

**Global variables**

Global variables should be avoided as much as possible to avoid making functions confusing. Still, they are still useful if used carefully. This code is using 6 of them. The first one is called “map” and it holds the information for the map. Many functions call this variable so it is smart to have it a global variable. The next to are previous\_lat and previous\_lng. These two store the value of the current center of the map. They are required in case some filtering used that has no potholes. If there are no potholes there will be no center calculated, so we need to remember the previous center. The other three global variables are click\_count, click\_count1 and click\_count2. They are used to count the click in the select menus. There should be no worries about them, they are necessary for the event handlers. The final possible global variable is the data retrieved from the server. The variable Global\_Function\_Count is used to track how many times we have gone into the object generating function. This function needs to act different depending on whether the data is from app or from the data analysis team. If geocoding works, this variable and all conditionals that depend on it should be removed.

**draw\_routes(Potholes, length)**

This is a recursive function that draws the lines in the map. It does this by calling the directions object from google maps and then sending the pothole data over to the directions renderer. If the server gets too many responses then it won’t draw the line. This should not be an issue when a professional google key is bought. The color of the line is chosen by the severity of the pothole. The Potholes input is an array which contains objects. Each object represents a different street. Each waypoint, starting location and final location is a different pothole. If there is only one pothole in the street then only the marker will appear and this should not appear in the pothole object. The markers are drawn separately to reduce the amount of requests to the server.

**Pothole\_object\_generator (Pothole\_data)**

This function is responsible for generating the Potholes objects that go inside the array fed into draw\_routes. This function will call others and use the value of the drop down menus to filter the data to know what data is to be used. There are three of these function. Function called from here:

* **date\_filter(Pothole\_data)**
  + Returns an array containing the location of all the potholes that meet the desired date. This is based on month and year
* **street\_filter(Pothole\_data)**
  + The first one is date\_filter, which. The second one is street filter. This function will return three arrays inside another array. The first one is an array containing the location of all the potholes that will be used during by this filter. The second array's length represents the amount of streets being looked up and the contents of each elements is the amount of potholes per street (This information is necessary in order to know when to set a waypoint and when to send a starting point or finishing point. The third array contains the names of the street. The order in this array is the same as the order in the second array. Street\_name[0] has Street\_count[0] number of potholes. If the street only has one pothole it will be deleted. It does not need to draw a route so it will only appear as a marker.
* **severity\_filter(Pothole\_data, Street\_names)**
  + The final filter is a severity filter. This function is used after all the severity fields in the potholes have been averaged. It removes elements from the pothole array that don’t fit the desired severity. It will also remove the corresponding element in the Street\_names.
* **render\_map(location)**
  + This function redraws the map on screen to clear previous data. It centers map on the required location and the zoom is picked depending on weather we are looking at all streets or just one.
* **center\_location(Pothole\_data, Date\_index, Street\_index, Street\_names)**
  + Return the average of the latitude and longitude of all the potholes that are going to be drawn on map. This latitude and longitude are used to center the map.
* **marker\_creator(Pothole\_data, Pothole\_data.length, Date\_index, Street\_index, Street\_names)**
  + Draw all the markers that meet the filters chosen by the user. This is a recursive function. The images for markers are picked based on severity and they can be found in the public directory. This function must be called after render\_map().

**function street\_dropdown\_creator(Pothole\_data)**

This function creates the elements in the “Streets” dropdown menu. It does this by first finding out the all the street names (takes out all duplicates) and then using DOM (document object manipulation) to create new elements that have the name of the street as a value and as text content. This value is important because it is later used for the filtering. A hardcopy of the input is necessary to avoid deleting our main data.

**function date\_dropdown\_creator(Pothole\_data)**

This function creates the elements in the “Dates” drop down menu. It finds all the unique dates in the data (takes out the duplicates) and then uses DOM to create this elements and append them to the “select” in the ejs file. The value of this elements is later used for date filtering. The first two digits of this should be the month and then the last four should be year. The filtering function is expecting this format so do not change it. A hardcopy of the input is necessary to avoid deleting our main data by accident.

**Event Listeners**

There are three event listeners at the bottom of the js file. One for each dropdown menu. They me way. They listen for clicks in the select boxes. When the count reaches 2. They call the functions to refresh the map. If the user clicks on a select box and then anywhere else in the page, the count gets reset to 0.It is important that the event propagation is turned of to make sure that the count gets properly reset when the body es clicked. This can be done more easily with jquery, but there is no real need for jquery in this project, so I will just code it manually.

**Mode Select**

The data can be seen in three modes. App Data, Data analysis Data and All Data. To keep track of these modes there are three radio buttons on the top right of the page. By default the page will load in the App Data mode. The page will not allow the user to unselect all three modes. Each one of these radio buttons comes with it’s own event listener and in the many filtering functions their values (true or false) are used to determined how to filter the data. This is pretty similar to the use of global variables.

**Reverse Geocoding for the Data Analysis Data**

Using reverse geocoding to get the street names on Data Analysis Data would be ideal. Unfortunately, the geocoding object does not seem to be responding fast enough. Did some test and out of 13 requests it only answered about 6 times. Incomplete data is not useful for this project, so for now the website will handle the data analysis data only as separate markers. Street filtering will not affect this data. When using mixed data, the street filtering should only affect the data from the app. I don’t know if this will be fixed by the time a professional key is purchased. If I have time, I will write a proper function that does this for later testing with the professional key.

**Note:** Minor functions are not mentioned here. To see a description ready the codes comments.

**Note for street\_filter function:** It is necessary to make a deep copy of the data being inputted into this function. The filtering function will delete some members of the data to properly sort it. If you just delete the data inputted then it will delete it in every stack frame. This variable acts a pointers, so this means that just copying its contents will not work (you will only be copying the address and not the content).

**Idea for Reverse Geocoding**

First try at geocoding I attempted to use the same method I use for all function where I need to use a certain google maps objects for an undefined amount of times. I used recursion. This did not work as the reverse geocoding was not fast enough and did not respond to all requests. For when a professional key is purchased, it might be smarter to make the function that does the reverse geocoding to only retrieve one address at a time and for the caller function to handle the necessary looping. If this option does not respond properly (probably won’t) then I delay could be used to give the server some time. This delay is not really a big deal since it will only affect the initial loading time of the page. This function should only be called once right after the data is retrieved from the server. My function is still in the code, but it is commented out.

**Still needs to be done (yellow = important, blue = solved, white: optional):**

* Set up server.
* Implement function to make request to the server
* Organize pothole data based on street and latlong. So that the two furthest points are used as origin and destinations. (Optional, will make the drawing look slightly better).
* Fixing type error. Weird type error when using the waypoints. It is not affecting functionality so for now i’m leaving it alone.
* Second option for viewing “special” data gathered from the data analysis team. This data is more specific. The severity goes from 1 to 10 instead of 1 to 3. Thinking about only using this data when it comes to the markers, street painting could still be averaged.
* Code is getting big, might have to create 2 js files.
* Tweak street filter so that it always returns true for the potholes data analysis data.
* Function to determine the zoom that needs to be used for a given set of filtered data. Zoom based on the maximum difference between two potholes in the array. Don’t know if this is possible, because I don’t know the equation for the zoom.
* Fix event listeners on the dropdown menus so they work from mobile devices.
* Use Bootstrap.js to make all dropdowns merge into navbar instead of just stacking.